GAP
Groups, Algorithms and Programming
version 3.4.4 distribution gap3-jm

Martin Schönert
with
Hans Ulrich Besche, Thomas Breuer, Frank Celler, Bettina Eick
Volkmar Felsch, Alexander Hulpke, Jürgen Mnich, Werner Nickel
Götz Pfeiffer, Udo Polis, Heiko Theißen
Lehrstuhl D für Mathematik, RWTH Aachen
Alice Niemeyer
Department of Mathematics, University of Western Australia

Jean Michel
Department of Mathematics, University Paris-Diderot, France

GAP 3.4.4 of 20 Dec. 1995, gap3-jm of 19 Feb. 2018

Preface

Welcome to the first release of GAP3 from St Andrews. In the two years since the release of GAP3 3.4.3, most of the efforts of the GAP3 team in Aachen have been devoted to the forthcoming major release, `GAP4.1`, which will feature a re-engineered kernel with many extra facilities, a completely new scheme for structuring the library, many new and enhanced algorithms and algorithms for new structures such as algebras and semigroups.

While this was going on, however, our users were not idle, and a number of bugs and blemishes in the system were found, while a substantial number of new or improved share packages have been submitted and accepted. Once it was decided that the computational algebra group at St Andrews would take over GAP3 development, we agreed, as a learning exercise, to release a new upgrade of GAP3 3.4, incorporating the bug fixes and new packages.

Assembling the release has indeed been a learning experience, and has, of course, taken much longer than we hoped. The release incorporates fixes to all known bugs in the library and kernel. In addition, there are two large new data libraries:of transitivie permutation groups up to degree 23; and of all groups of order up to 1000, except those of order 512 or 768 and some others have been extended. This release includes a number of share packages that are new since 3.4.3:

`autag`
: for computing the automorphism groups of soluble groups;

`CHEVIE`
: for computing with finite Coxeter groups, Hecke algebras, Chevalley groups and related structures;

`CrystGap`
: for computing with crystallographic groups;

`glissando`
: for comnputing with near-rings and semigroups;

`grim`
: for computing with rational and integer matrix groups;

`kbmag`
: linking to Knuth-Bendix package for monoids and groups;

`matrix`
: for analysing matrix groups over finite fields, replacing `smash` and `classic`;

`pcqa`
: linking to a polycyclic quotient program;

`specht`
: for computing the representation theory of the symmetric group and related structures; and

`xmod`
: for computing with crossed modules.

A number of other share packages have also been updated. Full details of all of these can be found in the updated manual, which is now also supplied in an HTML version.

Despite the tribulations of this release, we are looking forward to taking over a central role in GAP3 development in the future, and to working with the users and contributors who are so essential a part of making GAP3 what it is.

St Andrews, April 18.,1997, Steve Linton.

In the distribution `gap3-jm`, there are the following additional packages:

`anupq`
: The p-quotient algorithm, to work with p-groups.

`anusq`
: The soluble quotient algorithm.

`arep`
: Constructive representation theory.

`cohomolo`
: Cohomology and extensions of finite groups.

`dce`
: Double coset enumeration.

`grape`
: Computing with graphs and group.

`guava`
: Coding theory algorithms.

`meataxe`
: Splitting modular representations.

`monoid`
: Computing with monoids and semigroups.

`nq`
: The nilpotent quotient algorithm.

`sisyphos`
: Modular group algebras of p-groups.

`ve`
: Vector enumeration, for representations of finitely presented algebras.

`algebra`
: Finite-dimensional algebras.

`vkcurve`
: Fundamental group of the complement of a complex hypersurface. Also provides multivariate polynomials and rational fractions.

GAP3 stands for **Groups, Algorithms and Programming**. The name was chosen to reflect the aim of the system, which is introduced in this manual.

Until well into the eighties the interest of pure mathematicians in computational group theory was stirred by, but in most cases also confined to the information that was produced by group theoretical software for their special research problems – and hampered by the uneasy feeling that one was using black boxes of uncontrollable reliability. However the last years have seen a rapid spread of interest in the understanding, design and even implementation of group theoretical algorithms. These are gradually becoming accepted both as standard tools for a working group theoretician, like certain methods of proof, and as worthwhile objects of study, like connections between notions expressed in theorems.

GAP3 was started as an attempt to meet this interest. Therefore a primary design goal has been to give its user full access to algorithms and the data structures used by them, thus allowing critical study as well as modification of existing methods. We also intend to relieve the user from unwanted technical chores and to assist him in the programming, thus supporting invention and implementation of new algorithms as well as experimentation with them.

We have tried to achieve these goals by a design which in addition makes GAP3 easily portable, even to computers such as Atari ST and Amiga, and at the same time facilitates the maintenance of GAP3 with the limited resources of an academic environment.

While I had felt for some time rather strongly the wish for such a truly **open** system for computational group theory, the concrete idea of GAP3 was born when, together with a larger group of students, among whom were Johannes Meier, Werner Nickel, Alice Niemeyer, and Martin Schönert who eventually wrote the first version of GAP3, I had my first contact with the Maple system at the EUROCAL meeting in Linz/Austria in 1985. Maple demonstrated to us the feasibility of a strong and efficient computer algebra system built from a small kernel, with an interpreted library of routines written in a problem-adapted language. The discussion of the plan of a system for computational group theory organized in a similar way started in the fall of 1985, programming only in the second half of 1986. A first version of GAP3 was operational by the end of 1986. The system was first presented at the Oberwolfach meeting on computational group theory in May 1988. Version 2.4 was the first officially to be given away from Aachen starting in December 1988. The strong interest in this version, in spite of its still rather small collection of group theoretical routines, as well as constructive criticism by many colleagues, confirmed our belief in the general design principles of the system. Nevertheless over three years had passed until in April 1992 version 3.1 was released, which was followed in February 1993 by version 3.2, in November 1993 by version 3.3 and is now in June 1994 followed by version 3.4.

A main reason for the long time between versions 2.4 and 3.1 and the fact that there had not been intermediate releases was that we had found it advisable to make a number of changes to basic data structures until with version 3.1 we hoped to have reached a state where we could maintain upward compatibility over further releases, which were planned to follow much more frequently. Both goals have been achieved over the last two years. Of course the time has also been used to extend the scope of the methods implemented in GAP3. A rough estimate puts the size of the program library of version 3.4 at about sixteen times the size of that of version 2.4, while for version 3.1 the factor was about eight. Compared to GAP3 3.2, which was the last version with major additions, new features of GAP3 3.4 include the following:

- New data types (and extensions of methods) for algebras, modules and characters
- Further methods for working with finite presentations (IMD, a fast size function)
- Some "Almost linear" methods and (rational) conjugacy classes for permutation groups
- Methods based on "special AG systems" for finite soluble groups
- A package for the calculation of Galois groups and field extensions
- Extensions of the library of data (transitive permutation groups, crystallographic groups)
- An X-window based X-GAP3 for display of subgroup lattices
- Five further share libraries (ANU SQ, MEATAXE, SISYPHOS, VECTORENUMERATOR, SMASH)

Work on the extension of GAP3 is going on in Aachen as well as in an increasing number of other places. We hope to be able to have the next release of GAP3 after about 9 months again, that is in the first half of 1995.

The system that you are getting now consists of four parts:

1. A comparatively small **kernel**, written in C, which provides the user with:
   - automatic dynamic storage management, which the user needn't bother about in his programming;
   - a set of time-critical basic functions, e.g. "arithmetic" operations for integers, finite fields, permutations and words, as well as natural operations for lists and records;
   - an interpreter for the GAP3 language, which belongs to the Pascal family, but, while allowing additional types for group theoretical objects, does not require type declarations;
   - a set of programming tools for testing, debugging, and timing algorithms.
2. A much larger **library of GAP3 functions** that implement group theoretical and other algorithms. Since this is written entirely in the GAP3 language, in contrast to the situation in older group theoretical software, the GAP3 language is both the main implementation language and the user language of the system. Therefore the user can as easily as the original programmers investigate and vary algorithms of the library and add new ones to it, first for own use and eventually for the benefit of all GAP3 users. We hope that moreover the structuring of the library using the concept of **domains** and the techniques used for their handling that have been introduced into GAP3 3.1 by Martin Schönert will be further helpful in this respect.
3. A **library of group theoretical data** which already contains various libraries of groups (cf. chapter 38), large libraries of ordinary character tables, including all of the Cambridge **Atlas of Finite Groups** and modular tables (cf. chapter 53), and a **library of tables of marks**. We hope to extend this collection further with the help of colleagues who have undertaken larger classifications of groups.

4. The **documentation**. This is available as a file that can either be used for on-line help or be printed out to form this manual. Some advice for using this manual may be helpful. The first chapter **About GAP** is really an introduction to the use of the system, starting from scratch and, for the beginning, assuming neither much knowledge about group theory nor much versatility in using a computer. Some of the later sections of chapter 1 assume more, however. For instance section `About Character Tables` definitely assumes familiarity with representation theory of finite groups, while in particular sections `About the Implementation of Domains` to `About Defining New Group Elements` address more advanced users who want to extend the system to meet their special needs. The further chapters of the manual give then a full description of the functions presently available in GAP3.

Together with the system we distribute **GAP share libraries**, which are separate packages which have been written by various groups of people and remain under their responsibility. Some of these packages are written completely in the GAP3 language, others totally or in parts in C (or even other languages). However the functions in these packages can be called directly from GAP3 and results are returned to GAP3. At present there are 10 such share libraries (cf. chapter 57).

The policy for the further development of GAP3 is to keep the kernel as small as possible, extending the set of basic functions only by very selected ones that have proved to be time-critical and, wherever feasible, of general use. In the interest of the possibility of exchanging functions written in the GAP3 language the kernel has to be maintained in a single place which in the foreseeable future will be Aachen. On the other hand we hoped from the beginning that the design of GAP3 would allow the library of GAP3 functions and the library of data to grow not only by continued work in Aachen but, as does any other part of mathematics, by contributions from many sides, and these hopes have been fulfilled very well.

There are some other points to make on further policy:

- When we began work on GAP3 the typical user that we had in mind was the one wanting to implement his own algorithmic ideas. While we certainly hope that we still serve such users well it has become clear from the experience of the last years that there are even more users of two different species, on the one hand the established theorist, sometimes with little experience in the use of computers, who wants an easily understandable tool, on the other hand the student, often quite familiar with computers, who wants to get assistance in learning the theory by being able to do nontrivial examples. We think that in fact GAP3 can well be used by both, but we realize that for each a special introduction would be desirable. We apologize that we have not had the time yet to write such, however have learned (through the GAP3 forum) that in a couple of places work on the development of Laboratory Manuals for the use of GAP3 alongside with standard Algebra texts is undertaken.

- When we began work on GAP3, we designed it as a system for doing **group theory**. It has already turned out that in fact the design of the system is general enough, and some of its functions are also useful, for doing work in other neighbouring areas. For instance Leonard Soicher has used GAP3 to develop a system GRAPE for working with graphs, which meanwhile is available as a share library. We certainly enjoy seeing this happen, but we want to emphasize that in Aachen our primary interest is the development of a group theory system and that we do not plan to try to extend it beyond our abilities into a general computer algebra system.

- Rather we hope to provide tools for linking GAP3 to other systems that represent years of work and experience in areas such as commutative algebra, or to very efficient special purpose stand-alone programs. A link of this kind exists e.g. to the MOC system for the work with modular characters.

- We invite you to further extend GAP3. We are willing either to include such extensions into GAP3 or to make them available through the same channels as GAP3 in the form of the above mentioned **share libraries**. Of course, we will do this only if the extension can be distributed free of charge like GAP3. The copyright for such share libraries shall remain with you.

- Finally to answer an often asked question: The GAP3 language is in principle designed to be compilable. Work on a compiler is on the way, but this is not yet ready for inclusion with this release.

GAP3 is given away under the conditions that have always been in use between mathematicians, i.e. in particular **completely in source** and **free of charge**. We hope that the possibility offered by modern technology of depositing GAP3 on a number of computers to be fetched from them by ftp, will assist us in this policy. We want to emphasize, however, two points. GAP3 is **not** public domain software; we want to maintain a **copyright** that in particular forbids commercialization of GAP3. Further we ask that use of GAP3 be quoted in publications like the use of any other mathematical work, and we would be grateful if we could keep track of where GAP3 is implemented. Therefore we ask you to notify us if you have got GAP3, e.g., by sending a short e-mail message to gap@samson.math.rwth-aachen.de. The simple reason, on top of our curiosity, is that as anybody else in an academic environment we have from time to time to prove that we are doing meaningful work.

We have established a GAP3 forum, where interested users can discuss GAP3 related topics by e-mail. In particular this forum is for questions about GAP3, general comments, bug reports, and maybe bug fixes. We will read this forum and answer questions and comments, and distribute bug fixes. Of course others are also invited to answer questions, etc. We will also announce future releases of GAP3 in this forum.

To subscribe send an e-mail message to miles@samson.math.rwth-aachen.de containing the line subscribe gap-forum <your-name>, where ¡your-name¿ should be your full name, not your e-mail address. You will receive an acknowledgement, and from then on all e-mail messages sent to gap-forum@samson.math.rwth-aachen.de.

miles@samson.math.rwth-aachen.de also accepts the following requests. help for a short help on how to use miles, unsubscribe gap-forum to unsubscribe, recipients gap-forum to get a list of subscribers, and statistics gap-forum to see how many e-mail messages each subscriber has sent so far.

The reliability of large systems of computer programs is a well known general problem and, although over the past year the record of GAP3 in this respect has not been too bad, of course GAP3 is not exempt from this problem. We therefore feel that it is mandatory that we, but also other users, are warned of bugs that have been encountered in GAP3 or when doubts have arisen. We ask all users of GAP3 to use the GAP3 forum for issuing such warnings.

We have also established an e-mail address `gap-trouble` to which technical problems of a more local character such as installation problems can be sent. Together with some experienced GAP3 users abroad we try to give advice on such problems.

GAP3 was started as a joint Diplom project of four students whose names have already been mentioned. Since then many more finished Diplom projects have contributed to GAP3 as well as other members of Lehrstuhl D and colleagues from other institutes. Their individual contributions to the programs and to the manual are documented in the respective files. To all of them as well as to all who have helped proofreading and improving this manual I want to express my thanks for their engagement and enthusiasm as well as to many users of GAP3 who have helped us by pointing out deficiencies and suggesting improvements. Very special thanks however go to Martin Schönert. Not only does GAP3 owe many of its basic design features to his profound knowledge of computer languages and the techniques for their implementation, but in many long discussions he has in the name of future users always been the strongest defender of clarity of the design against my impatience and the temptation for "quick and dirty" solutions.

Since 1992 the development of GAP3 has been financially supported by the Deutsche Forschungsgemeinschaft in the context of the Forschungsschwerpunkt "Algorithmische Zahlentheorie und Algebra". This very important help is gratefully acknowledged.

As with the previous versions we send this version out hoping for further feedback of constructive criticism. Of course we ask to be notified about bugs, but moreover we shall appreciate any suggestion for the improvement of the basic system as well as of the algorithms in the library. Most of all, however, we hope that in spite of such criticism you will enjoy working with GAP3.

Aachen, June 1.,1994, Joachim Neubüser.

Contents

Contents

Chapter 1

About GAP

This chapter introduces you to the GAP3 system. It describes how to start GAP3 (you may have to ask your system administrator to install it correctly) and how to leave it. Then a step by step introduction should give you an impression of how the GAP3 system works. Further sections will give an overview about the features of GAP3. After reading this chapter the reader should know what kind of problems can be handled with GAP3 and how they can be handled.

There is some repetition in this chapter and much of the material is repeated in later chapters in a more compact and precise way. Yes, there are even some little inaccuracies in this chapter simplifying things for better understanding. It should be used as a tutorial introduction while later chapters form the reference manual.

GAP3 is an interactive system. It continuously executes a read–evaluate–print cycle. Each expression you type at the keyboard is read by GAP3, evaluated, and then the result is printed.

The interactive nature of GAP3 allows you to type an expression at the keyboard and see its value immediately. You can define a function and apply it to arguments to see how it works. You may even write whole programs containing lots of functions and test them without leaving the program.

When your program is large it will be more convenient to write it on a file and then read that file into GAP3. Preparing your functions in a file has several advantages. You can compose your functions more carefully in a file (with your favorite text editor), you can correct errors without retyping the whole function and you can keep a copy for later use. Moreover you can write lots of comments into the program text, which are ignored by GAP3, but are very useful for human readers of your program text.

GAP3 treats input from a file in the same way that it treats input from the keyboard.

The printed examples in this first chapter encourage you to try running GAP3 on your computer. This will support your feeling for GAP3 as a tool, which is the leading aim of this chapter. Do not believe any statement in this chapter so long as you cannot verify it for your own version of GAP3. You will learn to distinguish between small deviations of the behavior of your personal GAP3 from the printed examples and serious nonsense.

Since the printing routines of GAP3 are in some sense machine dependent you will for instance encounter a different layout of the printed objects in different environments. But the contents should always be the same.

In case you encounter serious nonsense it is highly recommended that you send a bug report to `gap-forum@samson.math.rwth-aachen.de`.

If you read this introduction on-line you should now enter `?>` to read the next section.

1.1 About Conventions

Throughout this manual both the input given to GAP3 and the output that GAP3 returns are printed in `typewriter` font just as if they were typed at the keyboard.

An *italic* font is used for keys that have no printed representation, such as e.g. the *newline* key and the *ctl* key. This font is also used for the formal parameters of functions that are described in later chapters.

A combination like *ctl*-`P` means pressing both keys, that is holding the control key *ctl* and pressing the key `P` while *ctl* is still pressed.

New terms are introduced in **bold face**.

In most places **whitespace** characters (i.e. *space*s, *tab*s and *newline*s) are insignificant for the meaning of GAP3 input. Identifiers and keywords must however not contain any whitespace. On the other hand, sometimes there must be whitespace around identifiers and keywords to separate them from each other and from numbers. We will use whitespace to format more complicated commands for better readability.

A **comment** in GAP3 starts with the symbol `#` and continues to the end of the line. Comments are treated like whitespace by GAP3.

Besides of such comments which are part of the input of a GAP3 session, we use additional comments which are part of the manual description, but not of the respective GAP3 session. In the printed version of this manual these comments will be printed in a normal font for better readability, hence they start with the symbol #.

The examples of GAP3 sessions given in any particular chapter of this manual have been run in one continuous session, starting with the two commands

```
SizeScreen( [ 72, ] );
LogTo( "erg.log" );
```

which are used to set the line length to 72 and to save a listing of the session on some file. If you choose any chapter and rerun its examples in the given order, you should be able to reproduce our results except of a few lines of output which we have edited a little bit with respect to blanks or line breaks in order to improve the readability. However, as soon as random processes are involved, you may get different results if you extract single examples and run them separately.

1.2 About Starting and Leaving GAP

If the program is correctly installed then you start GAP3 by simply typing `gap` at the prompt of your operating system followed by the *return* or the *newline* key.

```
$ gap
```

GAP3 answers your request with its beautiful banner (which you can surpress with the command line option -b) and then it shows its own prompt gap> asking you for further input.

```
gap>
```

The usual way to end a GAP3 session is to type quit; at the gap> prompt. Do not omit the semicolon!

```
gap> quit;
$
```

On some systems you may as well type *ctl*-D to yield the same effect. In any situation GAP3 is ended by typing *ctl*-C twice within a second.

1.3 About First Steps

A simple calculation with GAP3 is as easy as one can imagine. You type the problem just after the prompt, terminate it with a semicolon and then pass the problem to the program with the *return* key. For example, to multiply the difference between 9 and 7 by the sum of 5 and 6, that is to calculate $(9-7)*(5+6)$, you type exactly this last sequence of symbols followed by ; and *return*.

```
gap> (9 - 7) * (5 + 6);
22
gap>
```

Then GAP3 echoes the result 22 on the next line and shows with the prompt that it is ready for the next problem.

If you did omit the semicolon at the end of the line but have already typed *return*, then GAP3 has read everything you typed, but does not know that the command is complete. The program is waiting for further input and indicates this with a partial prompt >. This little problem is solved by simply typing the missing semicolon on the next line of input. Then the result is printed and the normal prompt returns.

```
gap> (9 - 7) * (5 + 6)
> ;
22
gap>
```

Whenever you see this partial prompt and you cannot decide what GAP3 is still waiting for, then you have to type semicolons until the normal prompt returns.

In every situation this is the exact meaning of the prompt gap> : the program is waiting for a new problem. In the following examples we will omit this prompt on the line after the result. Considering each example as a continuation of its predecessor this prompt occurs in the next example.

In this section you have seen how simple arithmetic problems can be solved by GAP3 by simply typing them in. You have seen that it doesn't matter whether you complete your input on one line. GAP3 reads your input line by line and starts evaluating if it has seen the terminating semicolon and *return*.

It is, however, also possible (and might be advisable for large amounts of input data) to write your input first into a file, and then read this into GAP3; see 3.23 and 3.12 for this.

Also in GAP3, there is the possibility to edit the input data, see 3.4.

1.4 About Help

The contents of the GAP3 manual is also available as on-line help, see 3.5–3.11. If you need information about a section of the manual, just enter a question mark followed by the header of the section. E.g., entering `?About Help` will print the section you are reading now.

`??`*topic* will print all entries in GAP3's index that contain the substring *topic*.

1.5 About Syntax Errors

Even if you mistyped the command you do not have to type it all again as GAP3 permits a lot of command line editing. Maybe you mistyped or forgot the last closing parenthesis. Then your command is syntactically incorrect and GAP3 will notice it, incapable of computing the desired result.

```
gap> (9 - 7) * (5 + 6;
Syntax error: ) expected
(9 - 7) * (5 + 6;
                ^
```

Instead of the result an error message occurs indicating the place where an unexpected symbol occurred with an arrow sign `^` under it. As a computer program cannot know what your intentions really were, this is only a hint. But in this case GAP3 is right by claiming that there should be a closing parenthesis before the semicolon. Now you can type *ctl*-`P` to recover the last line of input. It will be written after the prompt with the cursor in the first position. Type *ctl*-`E` to take the cursor to the end of the line, then *ctl*-`B` to move the cursor one character back. The cursor is now on the position of the semicolon. Enter the missing parenthesis by simply typing `)`. Now the line is correct and may be passed to GAP3 by hitting the *newline* key. Note that for this action it is not necessary to move the cursor past the last character of the input line.

Each line of commands you type is sent to GAP3 for evaluation by pressing *newline* regardless of the position of the cursor in that line. We will no longer mention the *newline* key from now on.

Sometimes a syntax error will cause GAP3 to enter a **break loop**. This is indicated by the special prompt `brk>`. You can leave the break loop by either typing `return;` or by hitting *ctl*-`D`. Then GAP3 will return to its normal state and show its normal prompt again.

In this section you learned that mistyped input will not lead to big confusion. If GAP3 detects a syntax error it will print an error message and return to its normal state. The command line editing allows you in a comfortable way to manipulate earlier input lines.

For the definition of the GAP3 syntax see chapter 2. A complete list of command line editing facilities is found in 3.4. The break loop is described in 3.2.

1.6 About Constants and Operators

In an expression like `(9 - 7) * (5 +  6)` the constants 5, 6, 7, and 9 are being composed by the operators `+`, `*` and `-` to result in a new value.

There are three kinds of operators in GAP3, arithmetical operators, comparison operators, and logical operators. You have already seen that it is possible to form the sum, the difference, and the product of two integer values. There are some more operators applicable to integers in GAP3. Of course integers may be divided by each other, possibly resulting in noninteger rational values.

```
gap> 12345/25;
2469/5
```

Note that the numerator and denominator are divided by their greatest common divisor and that the result is uniquely represented as a division instruction.

We haven't met negative numbers yet. So consider the following self–explanatory examples.

```
gap> -3; 17 - 23;
-3
-6
```

The exponentiation operator is written as `^`. This operation in particular might lead to very large numbers. This is no problem for GAP3 as it can handle numbers of (almost) arbitrary size.

```
gap> 3^132;
955004950796825236893190701774414011919935138974343129836853841
```

The `mod` operator allows you to compute one value modulo another.

```
gap> 17 mod 3;
2
```

Note that there must be whitespace around the keyword `mod` in this example since `17mod3` or `17mod` would be interpreted as identifiers.

GAP3 knows a precedence between operators that may be overridden by parentheses.

```
gap> (9 - 7) * 5 = 9 - 7  * 5;
false
```

Besides these arithmetical operators there are comparison operators in GAP3. A comparison results in a **boolean value** which is another kind of constant. Every two objects within GAP3 are comparable via `=`, `<>`, `<`, `<=`, `>` and `>=`, that is the tests for equality, inequality, less than, less than or equal, greater than and greater than or equal. There is an ordering defined on the set of all GAP3 objects that respects orders on subsets that one might expect. For example the integers are ordered in the usual way.

```
gap> 10^5 < 10^4;
false
```

The boolean values `true` and `false` can be manipulated via logical operators, i. e., the unary operator `not` and the binary operators `and` and `or`. Of course boolean values can be compared, too.

```
gap> not true; true and false; true or false;
false
false
true
gap> 10 > 0 and 10 < 100;
true
```

Another important type of constants in GAP3 are **permutations**. They are written in cycle notation and they can be multiplied.

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (1,2);
(2,3)
```

The inverse of the permutation `(1,2,3)` is denoted by `(1,2,3)^-1`. Moreover the caret operator `^` is used to determine the image of a point under a permutation and to conjugate one permutation by another.

```
gap> (1,2,3)^-1;
(1,3,2)
gap> 2^(1,2,3);
3
gap> (1,2,3)^(1,2);
(1,3,2)
```

The last type of constants we want to introduce here are the **characters**, which are simply objects in GAP3 that represent arbitrary characters from the character set of the operating system. Character literals can be entered in GAP3 by enclosing the character in **single-quotes** `'`.

```
gap> 'a';
'a'
gap> '*';
'*'
```

There are no operators defined for characters except that characters can be compared.

In this section you have seen that values may be preceded by unary operators and combined by binary operators placed between the operands. There are rules for precedence which may be overridden by parentheses. It is possible to compare any two objects. A comparison results in a boolean value. Boolean values are combined via logical operators. Moreover you have seen that GAP3 handles numbers of arbitrary size. Numbers and boolean values are constants. There are other types of constants in GAP3 like permutations. You are now in a position to use GAP3 as a simple desktop calculator.

Operators are explained in more detail in 2.9 and 2.10. Moreover there are sections about operators and comparisons for special types of objects in almost every chapter of this manual. You will find more information about boolean values in chapters 45 and 29. Permutations are described in chapter 20 and characters are described in chapter 30.

1.7 About Variables and Assignments

Values may be assigned to variables. A variable enables you to refer to an object via a name. The name of a variable is called an **identifier**. The assignment operator is `:=`. There must be no white space between the `:` and the `=`. Do not confuse the assignment operator `:=` with the single equality sign `=` which is in GAP3 only used for the test of equality.

```
gap> a:= (9 - 7) * (5 + 6);
22
gap> a;
22
gap> a * (a + 1);
```

```
506
gap> a:= 10;
10
gap> a * (a + 1);
110
```

After an assignment the assigned value is echoed on the next line. The printing of the value of a statement may be in every case prevented by typing a double semicolon.

```
gap> w:= 2;;
```

After the assignment the variable evaluates to that value if evaluated. Thus it is possible to refer to that value by the name of the variable in any situation.

This is in fact the whole secret of an assignment. An identifier is bound to a value and from this moment points to that value. Nothing more. This binding is changed by the next assignment to that identifier. An identifier does not denote a block of memory as in some other programming languages. It simply points to a value, which has been given its place in memory by the GAP3 storage manager. This place may change during a GAP3 session, but that doesn't bother the identifier.

The identifier points to the value, not to a place in the memory.

For the same reason it is not the identifier that has a type but the object. This means on the other hand that the identifier `a` which now is bound to an integer value may in the same session point to any other value regardless of its type.

Identifiers may be sequences of letters and digits containing at least one letter. For example `abc` and `a0bc1` are valid identifiers. But also `123a` is a valid identifier as it cannot be confused with any number. Just `1234` indicates the number 1234 and cannot be at the same time the name of a variable.

Since GAP3 distinguishes upper and lower case, `a1` and `A1` are different identifiers. Keywords such as `quit` must not be used as identifiers. You will see more keywords in the following sections.

In the remaining part of this manual we will ignore the difference between variables, their names (identifiers), and the values they point at. It may be useful to think from time to time about what is really meant by terms such as the integer `w`.

There are some predefined variables coming with GAP3. Many of them you will find in the remaining chapters of this manual, since functions are also referred to via identifiers.

This seems to be the right place to state the following rule.

The name of every function in the GAP3 library starts with a **capital letter.**

Thus if you choose only names starting with a small letter for your own variables you will not overwrite any predefined function.

But there are some further interesting variables one of which shall be introduced now.

Whenever GAP3 returns a value by printing it on the next line this value is assigned to the variable `last`. So if you computed

```
gap> (9 - 7) * (5 + 6);
22
```

and forgot to assign the value to the variable `a` for further use, you can still do it by the following assignment.

```
gap> a:= last;
22
```

Moreover there are variables `last2` and `last3`, guess their values.

In this section you have seen how to assign values to variables. These values can later be accessed through the name of the variable, its identifier. You have also encountered the useful concept of the `last` variables storing the latest returned values. And you have learned that a double semicolon prevents the result of a statement from being printed.

Variables and assignments are described in more detail in 2.7 and 2.12. A complete list of keywords is contained in 2.4.

1.8 About Functions

A program written in the GAP3 language is called a **function**. Functions are special GAP3 objects. Most of them behave like mathematical functions. They are applied to objects and will return a new object depending on the input. The function `Factorial`, for example, can be applied to an integer and will return the factorial of this integer.

```
gap> Factorial(17);
355687428096000
```

Applying a function to arguments means to write the arguments in parentheses following the function. Several arguments are separated by commas, as for the function `Gcd` which computes the greatest common divisor of two integers.

```
gap> Gcd(1234, 5678);
2
```

There are other functions that do not return a value but only produce a side effect. They change for example one of their arguments. These functions are sometimes called procedures. The function `Print` is only called for the side effect to print something on the screen.

```
gap> Print(1234, "\n");
1234
```

In order to be able to compose arbitrary text with `Print`, this function itself will not produce a line break after printing. Thus we had another newline character `"\n"` printed to start a new line.

Some functions will both change an argument and return a value such as the function `Sortex` that sorts a list and returns the permutation of the list elements that it has performed.

You will not understand right now what it means to change an object. We will return to this subject several times in the next sections.

A comfortable way to define a function is given by the **maps–to** operator `->` consisting of a minus sign and a greater sign with no whitespace between them. The function `cubed` which maps a number to its cube is defined on the following line.

```
gap> cubed:= x -> x^3;
function ( x ) ... end
```

After the function has been defined, it can now be applied.

```
gap> cubed(5);
125
```

Not every GAP3 function can be defined in this way. You will see how to write your own GAP3 functions in a later section.

In this section you have seen GAP3 objects of type function. You have learned how to apply a function to arguments. This yields as result a new object or a side effect. A side effect may change an argument of the function. Moreover you have seen an easy way to define a function in GAP3 with the maps-to operator.

Function calls are described in 2.8 and in 2.13. The functions of the GAP3 library are described in detail in the remaining chapters of this manual, the Reference Manual.

1.9 About Lists

A **list** is a collection of objects separated by commas and enclosed in brackets. Let us for example construct the list `primes` of the first 10 prime numbers.

```
gap> primes:= [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

The next two primes are 31 and 37. They may be appended to the existing list by the function `Append` which takes the existing list as its first and another list as a second argument. The second argument is appended to the list `primes` and no value is returned. Note that by appending another list the object `primes` is changed.

```
gap> Append(primes, [31, 37]);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 ]
```

You can as well add single new elements to existing lists by the function `Add` which takes the existing list as its first argument and a new element as its second argument. The new element is added to the list `primes` and again no value is returned but the list `primes` is changed.

```
gap> Add(primes, 41);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 ]
```

Single elements of a list are referred to by their position in the list. To get the value of the seventh prime, that is the seventh entry in our list `primes`, you simply type

```
gap> primes[7];
17
```

and you will get the value of the seventh prime. This value can be handled like any other value, for example multiplied by 2 or assigned to a variable. On the other hand this mechanism allows to assign a value to a position in a list. So the next prime 43 may be inserted in the list directly after the last occupied position of `primes`. This last occupied position is returned by the function `Length`.

```
gap> Length(primes);
13
gap> primes[14]:= 43;
43
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ]
```

Note that this operation again has changed the object `primes`. Not only the next position of a list is capable of taking a new value. If you know that 71 is the 20th prime, you can as well enter it right now in the 20th position of `primes`. This will result in a list with holes which is however still a list and has length 20 now.

```
gap> primes[20]:= 71;
71
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> Length(primes);
20
```

The list itself however must exist before a value can be assigned to a position of the list. This list may be the empty list `[ ]`.

```
gap> lll[1]:= 2;
Error, Variable: 'lll' must have a value
gap> lll:= [];
[  ]
gap> lll[1]:= 2;
2
```

Of course existing entries of a list can be changed by this mechanism, too. We will not do it here because `primes` then may no longer be a list of primes. Try for yourself to change the 17 in the list into a 9.

To get the position of 17 in the list `primes` use the function `Position` which takes the list as its first argument and the element as its second argument and returns the position of the first occurrence of the element 17 in the list `primes`. `Position` will return `false` if the element is not contained in the list.

```
gap> Position(primes, 17);
7
gap> Position(primes, 20);
false
```

In all of the above changes to the list `primes`, the list has been automatically resized. There is no need for you to tell GAP3 how big you want a list to be. This is all done dynamically.

It is not necessary for the objects collected in a list to be of the same type.

```
gap> lll:= [true, "This is a String",,, 3];
[ true, "This is a String",,, 3 ]
```

In the same way a list may be part of another list. A list may even be part of itself.

```
gap> lll[3]:= [4,5,6];; lll;
[ true, "This is a String", [ 4, 5, 6 ],, 3 ]
gap> lll[4]:= lll;
[ true, "This is a String", [ 4, 5, 6 ], ~, 3 ]
```

Now the tilde `~` in the fourth position of `lll` denotes the object that is currently printed. Note that the result of the last operation is the actual value of the object `lll` on the right hand side of the assignment. But in fact it is identical to the value of the whole list `lll` on the left hand side of the assignment.

A **string** is a very special type of list, which is printed in a different way. A **string** is simply a dense list of characters. Strings are used mainly in filenames and error messages. A string literal can either be entered simply as the list of characters or by writing the characters between **doublequotes** `"`. GAP will always output strings in the latter format.

```
gap> s1 := ['H','a','l','l','o',' ','w','o','r','l','d','.'];
"Hallo world."
gap> s2 := "Hallo world.";
"Hallo world."
gap> s1 := ['H','a','l','l','o',' ','w','o','r','l','d','.'];
"Hallo world."
gap> s1 = s2;
true
gap> s2[7];
'w'
```

Sublists of lists can easily be extracted and assigned using the operator `{ }`.

```
gap> sl := lll{ [ 1, 2, 3 ] };
[ true, "This is a String", [ 4, 5, 6 ] ]
gap> sl{ [ 2, 3 ] } := [ "New String", false ];
[ "New String", false ]
gap> sl;
[ true, "New String", false ]
```

This way you get a new list that contains at position i that element whose position is the ith entry of the argument of `{ }`.

In this long section you have encountered the fundamental concept of a list. You have seen how to construct lists, how to extend them and how to refer to single elements of a list. Moreover you have seen that lists may contain elements of different types, even holes (unbound entries). But this is still not all we have to tell you about lists.

You will find a discussion about identity and equality of lists in the next section. Moreover you will see special kinds of lists like sets (in 1.11), vectors and matrices (in 1.12) and ranges (in 1.14). Strings are described in chapter 30.

1.10 About Identical Lists

This second section about lists is dedicated to the subtle difference between equality and identity of lists. It is really important to understand this difference in order to understand how complex data structures are realized in GAP3. This section applies to all GAP3 objects that have subobjects, i. e., to lists and to records. After reading the section about records (1.13) you should return to this section and translate it into the record context.

Two lists are equal if all their entries are equal. This means that the equality operator `=` returns `true` for the comparison of two lists if and only if these two lists are of the same length and for each position the values in the respective lists are equal.

```
gap> numbers:= primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> numbers = primes;
true
```

We assigned the list `primes` to the variable `numbers` and, of course they are equal as they have both the same length and the same entries. Now we will change the third number to 4 and compare the result again with `primes`.

```
gap> numbers[3]:= 4;
4
gap> numbers = primes;
true
```

You see that `numbers` and `primes` are still equal, check this by printing the value of `primes`. The list `primes` is no longer a list of primes! What has happened? The truth is that the lists `primes` and `numbers` are not only equal but they are identical. `primes` and `numbers` are two variables pointing to the same list. If you change the value of the subobject `numbers[3]` of `numbers` this will also change `primes`. Variables do **not** point to a certain block of storage memory but they do point to an object that occupies storage memory. So the assignment `numbers:= primes` did **not** create a new list in a different place of memory but only created the new name `numbers` for the same old list of primes.

The same object can have several names.

If you want to change a list with the contents of `primes` independently from `primes` you will have to make a **copy** of `primes` by the function `Copy` which takes an object as its argument and returns a copy of the argument. (We will first restore the old value of `primes`.)

```
gap> primes[3]:= 5;
5
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> numbers:= Copy(primes);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> numbers = primes;
true
gap> numbers[3]:= 4;
4
gap> numbers = primes;
false
```

Now `numbers` is no longer equal to `primes` and `primes` still is a list of primes. Check this by printing the values of `numbers` and `primes`.

The only objects that can be changed this way are records and lists, because only GAP3 objects of these types have subobjects. To clarify this statement consider the following example.

```
gap> i:= 1;; j:= i;; i:= i+1;;
```

By adding 1 to `i` the value of `i` has changed. What happens to `j`? After the second statement `j` points to the same object as `i`, namely to the integer 1. The addition does **not** change the object `1` but creates a new object according to the instruction `i+1`. It is actually the assignment that changes the value of `i`. Therefore `j` still points to the object `1`. Integers (like permutations and booleans) have no subobjects. Objects of these types cannot be changed but can only be replaced by other objects. And a replacement does not change the values of other variables. In the above example an assignment of a new value to the variable `numbers` would also not change the value of `primes`.

Finally try the following examples and explain the results.

```
gap> l:= [];
[  ]
gap> l:= [l];
[ [  ] ]
gap> l[1]:= l;
[ ~ ]
```

Now return to the preceding section 1.9 and find out whether the functions `Add` and `Append` change their arguments.

In this section you have seen the difference between equal lists and identical lists. Lists are objects that have subobjects and therefore can be changed. Changing an object will change the values of all variables that point to that object. Be careful, since one object can have several names. The function `Copy` creates a copy of a list which is then a new object.

You will find more about lists in chapter 27, and more about identical lists in 27.9.

1.11 About Sets

GAP3 knows several special kinds of lists. A set in GAP3 is a special kind of list. A set contains no holes and its elements are sorted according to the GAP3 ordering of all its objects. Moreover a set contains no object twice.

The function `IsSet` tests whether an object is a set. It returns a boolean value. For any list there exists a corresponding set. This set is constructed by the function `Set` which takes the list as its argument and returns a set obtained from this list by ignoring holes and duplicates and by sorting the elements.

The elements of the sets used in the examples of this section are strings.

```
gap> fruits:= ["apple", "strawberry", "cherry", "plum"];
[ "apple", "strawberry", "cherry", "plum" ]
gap> IsSet(fruits);
false
gap> fruits:= Set(fruits);
[ "apple", "cherry", "plum", "strawberry" ]
```

Note that the original list `fruits` is not changed by the function `Set`. We have to make a new assignment to the variable `fruits` in order to make it a set.

The `in` operator is used to test whether an object is an element of a set. It returns a boolean value `true` or `false`.

```
gap> "apple" in fruits;
true
gap> "banana" in fruits;
false
```

The `in` operator may as well be applied to ordinary lists. It is however much faster to perform a membership test for sets since sets are always sorted and a binary search can be used instead of a linear search.

New elements may be added to a set by the function `AddSet` which takes the set `fruits` as its first argument and an element as its second argument and adds the element to the set if it wasn't already there. Note that the object `fruits` is changed.

```
gap> AddSet(fruits, "banana");
gap> fruits;         # The banana is inserted in the right place.
[ "apple", "banana", "cherry", "plum", "strawberry" ]
gap> AddSet(fruits, "apple");
gap> fruits;         # fruits has not changed.
[ "apple", "banana", "cherry", "plum", "strawberry" ]
```

Sets can be intersected by the function `Intersection` and united by the function `Union` which both take two sets as their arguments and return the intersection (union) of the two sets as a new object.

```
gap> breakfast:= ["tea", "apple", "egg"];
[ "tea", "apple", "egg" ]
gap> Intersection(breakfast, fruits);
[ "apple" ]
```

It is however not necessary for the objects collected in a set to be of the same type. You may as well have additional integers and boolean values for `breakfast`.

The arguments of the functions `Intersection` and `Union` may as well be ordinary lists, while their result is always a set. Note that in the preceding example at least one argument of `Intersection` was not a set.

The functions `IntersectSet` and `UniteSet` also form the intersection resp. union of two sets. They will however not return the result but change their first argument to be the result. Try them carefully.

In this section you have seen that sets are a special kind of list. There are functions to expand sets, intersect or unite sets, and there is the membership test with the `in` operator.

A more detailed description of strings is contained in chapter 30. Sets are described in more detail in chapter 28.

1.12 About Vectors and Matrices

A **vector** is a list of elements from a common field. A **matrix** is a list of vectors of equal length. Vectors and matrices are special kinds of lists without holes.

```
gap> v:= [3, 6, 2, 5/2];
[ 3, 6, 2, 5/2 ]
gap> IsVector(v);
true
```

Vectors may be multiplied by scalars from their field. Multiplication of vectors of equal length results in their scalar product.

```
gap> 2 * v;
[ 6, 12, 4, 5 ]
gap> v * 1/3;
[ 1, 2, 2/3, 5/6 ]
gap> v * v;
221/4  # the scalar product of v with itself
```

Note that the expression `v * 1/3` is actually evaluated by first multiplying `v` by 1 (which yields again `v`) and by then dividing by 3. This is also an allowed scalar operation. The expression `v/3` would result in the same value.

A matrix is a list of vectors of equal length.

```
gap> m:= [[1, -1, 1],
>         [2, 0, -1],
>         [1, 1, 1]];
[ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ] ]
gap> m[2][1];
2
```

Syntactically a matrix is a list of lists. So the number 2 in the second row and the first column of the matrix `m` is referred to as the first element of the second element of the list `m` via `m[2][1]`.

A matrix may be multiplied by scalars, vectors and other matrices. The vectors and matrices involved in such a multiplication must however have suitable dimensions.

```
gap> m:= [[1, 2, 3, 4],
>         [5, 6, 7, 8],
>         [9,10,11,12]];
[ [ 1, 2, 3, 4 ], [ 5, 6, 7, 8 ], [ 9, 10, 11, 12 ] ]
gap> PrintArray(m);
[ [   1,   2,   3,   4 ],
  [   5,   6,   7,   8 ],
  [   9,  10,  11,  12 ] ]
gap> [1, 0, 0, 0] * m;
Error, Vector *: vectors must have the same length
gap> [1, 0, 0] * m;
[ 1, 2, 3, 4 ]
gap> m * [1, 0, 0];
Error, Vector *: vectors must have the same length
gap> m * [1, 0, 0, 0];
[ 1, 5, 9 ]
gap> m * [0, 1, 0, 0];
[ 2, 6, 10 ]
```

Note that multiplication of a vector with a matrix will result in a linear combination of the rows of the matrix, while multiplication of a matrix with a vector results in a linear combination of the columns of the matrix. In the latter case the vector is considered as a column vector.

Submatrices can easily be extracted and assigned using the `{ }{ }` operator.

```
gap> sm := m{ [ 1, 2 ] }{ [ 3, 4 ] };
[ [ 3, 4 ], [ 7, 8 ] ]
gap> sm{ [ 1, 2 ] }{ [2] } := [[1],[-1]];
[ [ 1 ], [ -1 ] ]
gap> sm;
[ [ 3, 1 ], [ 7, -1 ] ]
```

The first curly brackets contain the selection of rows, the second that of columns.

In this section you have met vectors and matrices as special lists. You have seen how to refer to elements of a matrix and how to multiply scalars, vectors, and matrices.

Fields are described in chapter 6. The known fields in GAP3 are described in chapters 12, 13, 14, 15 and 18. Vectors and matrices are described in more detail in chapters 32 and 34. Vector spaces are described in chapter 9 and further matrix related structures are described in chapters 36 and 37.

1.13 About Records

A record provides another way to build new data structures. Like a list a record is a collection of other objects. In a record the elements are not indexed by numbers but by names (i.e., identifiers). An entry in a record is called a **record component** (or sometimes also record field).

```
gap> date:= rec(year:= 1992,
>               month:= "Jan",
>               day:= 13);
rec(
  year := 1992,
  month := "Jan",
  day := 13 )
```

Initially a record is defined as a comma separated list of assignments to its record components. Then the value of a record component is accessible by the record name and the record component name separated by one dot as the record component selector.

```
gap> date.year;
1992
gap> date.time:= rec(hour:= 19, minute:= 23, second:= 12);
rec(
  hour := 19,
  minute := 23,
  second := 12 )
gap> date;
rec(
  year := 1992,
  month := "Jan",
  day := 13,
  time := rec(
      hour := 19,
      minute := 23,
      second := 12 ) )
```

Assignments to new record components are possible in the same way. The record is automatically resized to hold the new component.

Most of the complex structures that are handled by GAP3 are represented as records, for instance groups and character tables.

Records are objects that may be changed. An assignment to a record component changes the original object. There are many functions in the library that will do such assignments to a record component of one of their arguments. The function `Size` for example, will compute the size of its argument which may be a group for instance, and then store the value in the record component `size`. The next call of `Size` for this object will use this stored value rather than compute it again.

Lists and records are the **only** types of GAP3 objects that can be changed.

Sometimes it is interesting to know which components of a certain record are bound. This information is available from the function `RecFields` (yes, this function should be called `RecComponentNames`), which takes a record as its argument and returns a list of all bound components of this record as a list of strings.

```
gap> RecFields(date);
[ "year", "month", "day", "time" ]
```

Finally try the following examples and explain the results.

```
gap> r:= rec();
rec(
   )
gap> r:= rec(r:= r);
rec(
  r := rec(
       ) )
gap> r.r:= r;
rec(
  r := ~ )
```

Now return to section 1.10 and find out what that section means for records.

In this section you have seen how to define and how to use records. Record objects are changed by assignments to record fields. Lists and records are the only types of objects that can be changed.

Records and functions for records are described in detail in chapter 46. More about identical records is found in 46.3.

1.14 About Ranges

A **range** is a finite sequence of integers. This is another special kind of list. A range is described by its minimum (the first entry), its second entry and its maximum, separated by a comma resp. two dots and enclosed in brackets. In the usual case of an ascending list of consecutive integers the second entry may be omitted.

```
gap> [1..999999];      # a range of almost a million numbers
[ 1 .. 999999 ]
gap> [1, 2..999999];   # this is equivalent
[ 1 .. 999999 ]
gap> [1, 3..999999];   # here the step is 2
[ 1, 3 .. 999999 ]
gap> Length( last );
500000
gap> [ 999999, 999997 .. 1 ];
[ 999999, 999997 .. 1 ]
```

This compact printed representation of a fairly long list corresponds to a compact internal representation. The function `IsRange` tests whether an object is a range. If this is true for a list but the list is not yet represented in the compact form of a range this will be done then.

```
gap> a:= [-2,-1,0,1,2,3,4,5];
[ -2, -1, 0, 1, 2, 3, 4, 5 ]
gap> IsRange(a);
true
gap> a;
[ -2 .. 5 ]
gap> a[5];
2
gap> Length(a);
8
```

Note that this change of representation does **not** change the value of the list `a`. The list `a` still behaves in any context in the same way as it would have in the long representation.

In this section you have seen that ascending lists of consecutive integers can be represented in a compact way as ranges.

Chapter 31 contains a detailed description of ranges. A fundamental application of ranges is introduced in the next section.

1.15 About Loops

Given a list `pp` of permutations we can form their product by means of a `for` loop instead of writing down the product explicitly.

```
gap> pp:= [ (1,3,2,6,8)(4,5,9), (1,6)(2,7,8)(4,9), (1,5,7)(2,3,8,6),
>           (1,8,9)(2,3,5,6,4), (1,9,8,6,3,4,7,2) ];;
gap> prod:= ();
()
gap> for p in pp do
>       prod:= prod * p;
>    od;
gap> prod;
(1,8,4,2,3,6,5)
```

First a new variable `prod` is initialized to the identity permutation `()`. Then the loop variable `p` takes as its value one permutation after the other from the list `pp` and is multiplied with the present value of `prod` resulting in a new value which is then assigned to `prod`.

The `for` loop has the following syntax.

`for` *var* `in` *list* `do` *statements* `od;`

The effect of the `for` loop is to execute the *statements* for every element of the *list*. A `for` loop is a statement and therefore terminated by a semicolon. The list of *statements* is enclosed by the keywords `do` and `od` (reverse `do`). A `for` loop returns no value. Therefore we had to ask explicitly for the value of `prod` in the preceding example.

The `for` loop can loop over any kind of list, even a list with holes. In many programming languages (and in former versions of GAP3, too) the `for` loop has the form

`for` *var* `from` *first* `to` *last* `do` *statements* `od;`

But this is merely a special case of the general `for` loop as defined above where the *list* in the loop body is a range.

`for` *var* `in [`*first*`..`*last*`] do` *statements* `od;`

You can for instance loop over a range to compute the factorial 15! of the number 15 in the following way.

```
gap> ff:= 1;
1
gap> for i in [1..15] do
>       ff:= ff * i;
>    od;
gap> ff;
1307674368000
```

The following example introduces the `while` loop which has the following syntax.

`while` *condition* `do` *statements* `od;`

The `while` loop loops over the *statements* as long as the *condition* evaluates to `true`. Like the `for` loop the `while` loop is terminated by the keyword `od` followed by a semicolon.

We can use our list `primes` to perform a very simple factorization. We begin by initializing a list `factors` to the empty list. In this list we want to collect the prime factors of the number 1333. Remember that a list has to exist before any values can be assigned to positions of the list. Then we will loop over the list `primes` and test for each prime whether it divides the number. If it does we will divide the number by that prime, add it to the list `factors` and continue.

```
gap> n:= 1333;
1333
gap> factors:= [];
[  ]
gap> for p in primes do
>       while n mod p = 0 do
>          n:= n/p;
>          Add(factors, p);
>       od;
>    od;
gap> factors;
[ 31, 43 ]
gap> n;
1
```

As `n` now has the value 1 all prime factors of 1333 have been found and `factors` contains a complete factorization of 1333. This can of course be verified by multiplying 31 and 43.

This loop may be applied to arbitrary numbers in order to find prime factors. But as `primes` is not a complete list of all primes this loop may fail to find all prime factors of a number greater than 2000, say. You can try to improve it in such a way that new primes are added to the list `primes` if needed.

You have already seen that list objects may be changed. This holds of course also for the list in a loop body. In most cases you have to be careful not to change this list, but there are situations where this is quite useful. The following example shows a quick way to determine the primes smaller than 1000 by a sieve method. Here we will make use of the function `Unbind` to delete entries from a list.

```
gap> primes:= [];
[  ]
gap> numbers:= [2..1000];
[ 2 .. 1000 ]
gap> for p in numbers do
>       Add(primes, p);
>       for n in numbers do
>          if n mod p = 0 then
>             Unbind(numbers[n-1]);
>          fi;
>       od;
>    od;
```

The inner loop removes all entries from `numbers` that are divisible by the last detected prime `p`. This is done by the function `Unbind` which deletes the binding of the list position `numbers[n-1]` to the value `n` so that afterwards `numbers[n-1]` no longer has an assigned value. The next element encountered in `numbers` by the outer loop necessarily is the next prime.

In a similar way it is possible to enlarge the list which is looped over. This yields a nice and short orbit algorithm for the action of a group, for example.

In this section you have learned how to loop over a list by the `for` loop and how to loop with respect to a logical condition with the `while` loop. You have seen that even the list in the loop body can be changed.

The `for` loop is described in 2.17. The `while` loop is described in 2.15.

1.16 About Further List Operations

There is however a more comfortable way to compute the product of a list of numbers or permutations.

```
gap> Product([1..15]);
1307674368000
gap> Product(pp);
(1,8,4,2,3,6,5)
```

The function `Product` takes a list as its argument and computes the product of the elements of the list. This is possible whenever a multiplication of the elements of the list is defined. So `Product` is just an implementation of the loop in the example above as a function.

There are other often used loops available as functions. Guess what the function `Sum` does. The function `List` may take a list and a function as its arguments. It will then apply the function to each element of the list and return the corresponding list of results. A list of cubes is produced as follows with the function `cubed` from 1.8.

```
gap> List([2..10], cubed);
[ 8, 27, 64, 125, 216, 343, 512, 729, 1000 ]
```

To add all these cubes we might apply the function `Sum` to the last list. But we may as well give the function `cubed` to `Sum` as an additional argument.

```
gap> Sum(last) = Sum([2..10], cubed);
true
```

The primes less than 30 can be retrieved out of the list `primes` from section 1.9 by the function `Filtered`. This function takes the list `primes` and a property as its arguments and will return the list of those elements of `primes` which have this property. Such a property will be represented by a function that returns a boolean value. In this example the property of being less than 30 can be reresented by the function `x-> x <30` since `x <30` will evaluate to `true` for values `x` less than 30 and to `false` otherwise.

```
gap> Filtered(primes, x-> x < 30);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

Another useful thing is the operator `{ }` that forms sublists. It takes a list of positions as its argument and will return the list of elements from the original list corresponding to these positions.

```
gap> primes{ [1 .. 10] };
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

In this section you have seen some functions which implement often used `for` loops. There are functions like `Product` to form the product of the elements of a list. The function `List` can apply a function to all elements of a list and the functions `Filtered` and `Sublist` create sublists of a given list.

You will find more predefined `for` loops in chapter 27.

1.17 About Writing Functions

You have already seen how to use the functions of the GAP3 library, i.e., how to apply them to arguments. This section will show you how to write your own functions.

Writing a function that prints `hello, world.` on the screen is a simple exercise in GAP3.

```
gap> sayhello:= function()
> Print("hello, world.\n");
> end;
function (  ) ... end
```

This function when called will only execute the `Print` statement in the second line. This will print the string `hello, world.` on the screen followed by a newline character `\n` that causes the GAP3 prompt to appear on the next line rather than immediately following the printed characters.

The function definition has the following syntax.

`function(`*arguments*`)` *statements* `end`

A function definition starts with the keyword `function` followed by the formal parameter list *arguments* enclosed in parenthesis. The formal parameter list may be empty as in the example. Several parameters are separated by commas. Note that there must be **no** semicolon behind the closing parenthesis. The function definition is terminated by the keyword `end`.

A GAP3 function is an expression like integers, sums and lists. It therefore may be assigned to a variable. The terminating semicolon in the example does not belong to the function definition but terminates the assignment of the function to the name `sayhello`. Unlike in the case of integers, sums, and lists the value of the function `sayhello` is echoed in the abbreviated fashion `function ( ) ... end`. This shows the most interesting part of a function: its formal parameter list (which is empty in this example). The complete value of `sayhello` is returned if you use the function `Print`.

```
gap> Print(sayhello, "\n");
function (  )
    Print( "hello, world.\n" );
end
```

Note the additional newline character `"\n"` in the `Print` statement. It is printed after the object `sayhello` to start a new line.

The newly defined function `sayhello` is executed by calling `sayhello()` with an empty argument list.

```
gap> sayhello();
hello, world.
```

This is however not a typical example as no value is returned but only a string is printed.

A more useful function is given in the following example. We define a function `sign` which shall determine the sign of a number.

```
gap> sign:= function(n)
>         if n < 0 then
>            return -1;
>         elif n = 0 then
>            return 0;
>         else
>            return 1;
>         fi;
>     end;
function ( n ) ... end
gap> sign(0); sign(-99); sign(11);
0
-1
1
gap> sign("abc");
1         # strings are defined to be greater than 0
```

This example also introduces the `if` statement which is used to execute statements depending on a condition. The `if` statement has the following syntax.

`if` *condition* `then` *statements* `elif` *condition* `then` *statements* `else` *statements* `fi;`

There may be several `elif` parts. The `elif` part as well as the `else` part of the `if` statement may be omitted. An `if` statement is no expression and can therefore not be assigned to a variable. Furthermore an `if` statement does not return a value.

Fibonacci numbers are defined recursively by $f(1) = f(2) = 1$ and $f(n) = f(n-1) + f(n-2)$. Since functions in GAP3 may call themselves, a function `fib` that computes Fibonacci numbers can be implemented basically by typing the above equations.

```
gap> fib:= function(n)
>       if n in [1, 2] then
>          return 1;
>       else
>          return fib(n-1) + fib(n-2);
>       fi;
>    end;
function ( n ) ... end
gap> fib(15);
610
```

There should be additional tests for the argument `n` being a positive integer. This function `fib` might lead to strange results if called with other arguments. Try to insert the tests in this example.

A function `gcd` that computes the greatest common divisor of two integers by Euclid's algorithm will need a variable in addition to the formal arguments.

```
gap> gcd:= function(a, b)
>       local c;
>       while b <> 0 do
>          c:= b;
>          b:= a mod b;
>          a:= c;
>       od;
>       return c;
>    end;
function ( a, b ) ... end
gap> gcd(30, 63);
3
```

The additional variable `c` is declared as a **local** variable in the `local` statement of the function definition. The `local` statement, if present, must be the first statement of a function definition. When several local variables are declared in only one `local` statement they are separated by commas.

The variable `c` is indeed a local variable, that is local to the function `gcd`. If you try to use the value of `c` in the main loop you will see that `c` has no assigned value unless you have already assigned a value to the variable `c` in the main loop. In this case the local nature of `c` in the function `gcd` prevents the value of the `c` in the main loop from being overwritten.

We say that in a given scope an identifier identifies a unique variable. A **scope** is a lexical part of a program text. There is the global scope that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A local scope introduces new variables, whose identifiers are given in the formal argument list and the local declaration of the function. The usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name.

We will now write a function to determine the number of partitions of a positive integer. A partition of a positive integer is a descending list of numbers whose sum is the given integer. For example $[4, 2, 1, 1]$ is a partition of 8. The complete set of all partitions of an integer n

may be divided into subsets with respect to the largest element. The number of partitions of n therefore equals the sum of the numbers of partitions of $n-i$ with elements less than i for all possible i. More generally the number of partitions of n with elements less than m is the sum of the numbers of partitions of $n-i$ with elements less than i for i less than m and n. This description yields the following function.

```
gap> nrparts:= function(n)
>    local np;
>    np:= function(n, m)
>       local i, res;
>       if n = 0 then
>          return 1;
>       fi;
>       res:= 0;
>       for i in [1..Minimum(n,m)] do
>          res:= res + np(n-i, i);
>       od;
>       return res;
>    end;
>    return np(n,n);
> end;
function ( n ) ... end
```

We wanted to write a function that takes one argument. We solved the problem of determining the number of partitions in terms of a recursive procedure with two arguments. So we had to write in fact two functions. The function `nrparts` that can be used to compute the number of partitions takes indeed only one argument. The function `np` takes two arguments and solves the problem in the indicated way. The only task of the function `nrparts` is to call `np` with two equal arguments.

We made `np` local to `nrparts`. This illustrates the possibility of having local functions in GAP3. It is however not necessary to put it there. `np` could as well be defined on the main level. But then the identifier `np` would be bound and could not be used for other purposes. And if it were used the essential function `np` would no longer be available for `nrparts`.

Now have a look at the function `np`. It has two local variables `res` and `i`. The variable `res` is used to collect the sum and `i` is a loop variable. In the loop the function `np` calls itself again with other arguments. It would be very disturbing if this call of `np` would use the same `i` and `res` as the calling `np`. Since the new call of `np` creates a new scope with new variables this is fortunately not the case.

The formal parameters n and m are treated like local variables.

It is however cheaper (in terms of computing time) to avoid such a recursive solution if this is possible (and it is possible in this case), because a function call is not very cheap.

In this section you have seen how to write functions in the GAP3 language. You have also seen how to use the `if` statement. Functions may have local variables which are declared in an initial `local` statement in the function definition. Functions may call themselves.

The function syntax is described in 2.18. The `if` statement is described in more detail in 2.14. More about Fibonacci numbers is found in 47.22 and more about partitions in 47.13.

1.18 About Groups

In this section we will show some easy computations with groups. The example uses permutation groups, but this is visible for the user only because the output contains permutations. The functions, like `Group`, `Size` or `SylowSubgroup` (for detailed information, see chapters 4, 7), are the same for all kinds of groups, although the algorithms which compute the information of course will be different in most cases.

It is not even necessary to know more about permutations than the two facts that they are elements of permutation groups and that they are written in disjoint cycle notation (see chapter 20). So let's construct a permutation group:

```
gap> s8:= Group( (1,2), (1,2,3,4,5,6,7,8) );
Group( (1,2), (1,2,3,4,5,6,7,8) )
```

We formed the group generated by the permutations `(1,2)` and `(1,2,3,4,5,6,7,8)`, which is well known as the symmetric group on eight points, and assigned it to the identifier `s8`. `s8` contains the alternating group on eight points which can be described in several ways, e.g., as group of all even permutations in `s8`, or as its commutator subgroup.

```
gap> a8:= CommutatorSubgroup( s8, s8 );
Subgroup( Group( (1,2), (1,2,3,4,5,6,7,8) ),
[ (1,3,2), (2,4,3), (2,3)(4,5), (2,4,6,5,3), (2,5,3)(4,7,6),
  (2,3)(5,6,8,7) ] )
```

The alternating group `a8` is printed as instruction to compute that subgroup of the group `s8` that is generated by the given six permutations. This representation is much shorter than the internal structure, and it is completely self–explanatory; one could, for example, print such a group to a file and read it into GAP3 later. But if one object occurs several times it is useful to refer to this object; this can be settled by assigning a name to the group.

```
gap> s8.name:= "s8";
"s8"
gap> a8;
Subgroup( s8, [ (1,3,2), (2,4,3), (2,3)(4,5), (2,4,6,5,3),
  (2,5,3)(4,7,6), (2,3)(5,6,8,7) ] )
gap> a8.name:= "a8";
"a8"
gap> a8;
a8
```

Whenever a group has a component `name`, GAP3 prints this name instead of the group itself. Note that there is no link between the name and the identifier, but it is of course useful to choose name and identifier compatible.

```
gap> copya8:= Copy( a8 );
a8
```

We examine the group `a8`. Like all complex GAP3 structures, it is represented as a record (see 7.118).

```
gap> RecFields( a8 );
[ "isDomain", "isGroup", "parent", "identity", "generators",
  "operations", "isPermGroup", "1", "2", "3", "4", "5", "6",
```

```
  "stabChainOptions", "stabChain", "orbit", "transversal",
  "stabilizer", "name" ]
```

Many functions store information about the group in this group record, this avoids duplicate computations. But we are not interested in the organisation of data but in the group, e.g., some of its properties (see chapter 7, especially 7.45):

```
gap> Size( a8 ); IsAbelian( a8 ); IsPerfect( a8 );
20160
false
true
```

Some interesting subgroups are the Sylow p subgroups for prime divisors p of the group order; a call of `SylowSubgroup` stores the required subgroup in the group record:

```
gap> Set( Factors( Size( a8 ) ) );
[ 2, 3, 5, 7 ]
gap> for p in last do
>      SylowSubgroup( a8, p );
>    od;
gap> a8.sylowSubgroups;
[ , Subgroup( s8, [ (1,5)(7,8), (1,5)(2,6), (3,4)(7,8), (2,3)(4,6),
      (1,7)(2,3)(4,6)(5,8), (1,2)(3,7)(4,8)(5,6) ] ),
  Subgroup( s8, [ (3,8,7), (2,6,4)(3,7,8) ] ),,
  Subgroup( s8, [ (3,7,8,6,4) ] ),,
  Subgroup( s8, [ (2,8,4,5,7,3,6) ] ) ]
```

The record component `sylowSubgroups` is a list which stores at the p–th position, if bound, the Sylow p subgroup; in this example this means that there are holes at positions 1, 4 and 6. Note that a call of `SylowSubgroup` for the cyclic group of order 65521 and for the prime 65521 would cause GAP3 to store the group at the end of a list of length 65521, so there are special situations where it is possible to bring GAP3 and yourselves into troubles.

We now can investigate the Sylow 2 subgroup.

```
gap> syl2:= last[2];;
gap> Size( syl2 );
64
gap> Normalizer( a8, syl2 );
Subgroup( s8, [ (3,4)(7,8), (2,3)(4,6), (1,2)(3,7)(4,8)(5,6) ] )
gap> last = syl2;
true
gap> Centre( syl2 );
Subgroup( s8, [ ( 1, 5)( 2, 6)( 3, 4)( 7, 8) ] )
gap> cent:= Centralizer( a8, last );
Subgroup( s8, [ ( 1, 5)( 2, 6)( 3, 4)( 7, 8), (3,4)(7,8), (3,7)(4,8),
  (2,3)(4,6), (1,2)(5,6) ] )
gap> Size( cent );
192
gap> DerivedSeries( cent );
[ Subgroup( s8, [ ( 1, 5)( 2, 6)( 3, 4)( 7, 8), (3,4)(7,8),
      (3,7)(4,8), (2,3)(4,6), (1,2)(5,6) ] ),
```

```
  Subgroup( s8, [ ( 1, 6, 3)( 2, 4, 5), ( 1, 8, 3)( 4, 5, 7),
      ( 1, 7)( 2, 3)( 4, 6)( 5, 8), ( 1, 5)( 2, 6) ] ),
  Subgroup( s8, [ ( 1, 3)( 2, 7)( 4, 5)( 6, 8),
      ( 1, 6)( 2, 5)( 3, 8)( 4, 7), ( 1, 5)( 3, 4), ( 1, 5)( 7, 8) ] )
    , Subgroup( s8, [ ( 1, 5)( 2, 6)( 3, 4)( 7, 8) ] ),
  Subgroup( s8, [  ] ) ]
gap> List( last, Size );
[ 192, 96, 32, 2, 1 ]
gap> low:= LowerCentralSeries( cent );
[ Subgroup( s8, [ ( 1, 5)( 2, 6)( 3, 4)( 7, 8), (3,4)(7,8),
      (3,7)(4,8), (2,3)(4,6), (1,2)(5,6) ] ),
  Subgroup( s8, [ ( 1, 6, 3)( 2, 4, 5), ( 1, 8, 3)( 4, 5, 7),
      ( 1, 7)( 2, 3)( 4, 6)( 5, 8), ( 1, 5)( 2, 6) ] ) ]
```

Another kind of subgroups is given by the point stabilizers.

```
gap> stab:= Stabilizer( a8, 1 );
Subgroup( s8, [ (2,5,6), (2,5)(3,6), (2,5,6,4,3), (2,5,3)(4,6,8),
  (2,5)(3,4,7,8) ] )
gap> Size( stab );
2520
gap> Index( a8, stab );
8
```

We can fetch an arbitrary group element and look at its centralizer in `a8`, and then get other subgroups by conjugation and intersection of already known subgroups. Note that we form the subgroups inside `a8`, but GAP3 regards these groups as subgroups of `s8` because this is the common "parent" group of all these groups and of `a8` (for the idea of parent groups, see 7.6).

```
gap> Random( a8 );
(1,6,3,2,7)(4,5,8)
gap> Random( a8 );
(1,3,2,4,7,5,6)
gap> cent:= Centralizer( a8, (1,2)(3,4)(5,8)(6,7) );
Subgroup( s8, [ (1,2)(3,4)(5,8)(6,7), (5,6)(7,8), (5,7)(6,8),
  (3,4)(6,7), (3,5)(4,8), (1,3)(2,4) ] )
gap> Size( cent );
192
gap> conj:= ConjugateSubgroup( cent, (2,3,4) );
Subgroup( s8, [ (1,3)(2,4)(5,8)(6,7), (5,6)(7,8), (5,7)(6,8),
  (2,4)(6,7), (2,8)(4,5), (1,4)(2,3) ] )
gap> inter:= Intersection( cent, conj );
Subgroup( s8, [ (5,6)(7,8), (5,7)(6,8), (1,2)(3,4), (1,3)(2,4) ] )
gap> Size( inter );
16
gap> IsElementaryAbelian( inter );
true
gap> norm:= Normalizer( a8, inter );
Subgroup( s8, [ (6,7,8), (5,6,8), (3,4)(6,8), (2,3)(6,8), (1,2)(6,8),
  (1,5)(2,6,3,7,4,8) ] )
```

```
gap> Size( norm );
576
```

Suppose we do not only look which funny things may appear in our group but want to construct a subgroup, e.g., a group of structure $2^3 : L_3(2)$ in `a8`. One idea is to look for an appropriate 2^3 which is specified by the fact that all its involutions are fixed point free, and then compute its normalizer in `a8`:

```
gap> elab:= Group( (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8),
>                  (1,5)(2,6)(3,7)(4,8) );;
gap> Size( elab );
8
gap> IsElementaryAbelian( elab );
true
gap> norm:= Normalizer( a8, AsSubgroup( s8, elab ) );
Subgroup( s8, [ (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6),
  (2,3)(6,7), (1,2)(7,8) ] )
gap> Size( norm );
1344
```

Note that `elab` was defined as separate group, thus we had to call `AsSubgroup` to achieve that it has the same parent group as `a8`. Let's look at some usual misuses:

```
Normalizer( a8, elab );
```

Intuitively, it is clear that here again we wanted to compute the normalizer of `elab` in `a8`, and in fact we would get it by this call. However, this would be a misuse in the sense that now GAP3 cannot use some clever method for the computation of the normalizer. So, for larger groups, the computation may be very time consuming. That is the reason why we used the the function `AsSubgroup` in the preceding example.

Let's have a closer look at that function.

```
gap> IsSubgroup( a8, AsSubgroup( a8, elab ) );
Error, <G> must be a parent group in
AsSubgroup( a8, elab ) called from
main loop
brk> quit;
gap> IsSubgroup( a8, AsSubgroup( s8, elab ) );
true
```

What we tried here was not correct. Since all our computations up to now are done inside `s8` which is the parent of `a8`, it is easy to understand that `IsSubgroup` works for two subgroups with this parent.

By the way, you should not try the operator `<` instead of the function `IsSubgroup`. Something like

```
gap> elab < a8;
false
```

or

```
gap> AsSubgroup( s8, elab ) < a8;
false
```

will not cause an error, but the result does not tell anything about the inclusion of one group in another; `<` looks at the element lists for the two domains which means that it computes them if they are not already stored –which is not desirable to do for large groups– and then simply compares the lists with respect to lexicographical order (see 4.7).

On the other hand, the equality operator `=` in fact does test the equality of groups. Thus

```
gap> elab = AsSubgroup( s8, elab );
true
```

means that the two groups are equal in the sense that they have the same elements. Note that they may behave differently since they have different parent groups. In our example, it is necessary to work with subgroups of `s8`:

```
gap> elab:= AsSubgroup( s8, elab );;
gap> elab.name:= "elab";;
```

If we are given the subgroup `norm` of order 1344 and its subgroup `elab`, the factor group can be considered.

```
gap> f:= norm / elab;
(Subgroup( s8, [ (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6),
  (2,3)(6,7), (1,2)(7,8) ] ) / elab)
gap> Size( f );
168
```

As the output shows, this is not a permutation group. The factor group and its elements can, however, be handled in the usual way.

```
gap> Random( f );
FactorGroupElement( elab, (2,8,7)(3,5,6) )
gap> Order( f, last );
3
```

The natural link between the group `norm` and its factor group `f` is the natural homomorphism onto `f`, mapping each element of `norm` to its coset modulo the kernel `elab`. In GAP3 you can construct the homomorphism, but note that the images lie in `f` since they are elements of the factor group, but the preimage of each such element is only a coset, not a group element (for cosets, see the relevant sections in chapter 7, for homomorphisms see chapters 8 and 43).

```
gap> f.name:= "f";;
gap> hom:= NaturalHomomorphism( norm, f );
NaturalHomomorphism( Subgroup( s8,
[ (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6), (2,3)(6,7),
  (1,2)(7,8) ] ), (Subgroup( s8,
[ (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6), (2,3)(6,7),
  (1,2)(7,8) ] ) / elab) )
gap> Kernel( hom ) = elab;
true
gap> x:= Random( norm );
(1,7,5,8,3,6,2)
gap> Image( hom, x );
FactorGroupElement( elab, (2,7,3,4,6,8,5) )
```

```
gap> coset:= PreImages( hom, last );
(elab*(2,7,3,4,6,8,5))
gap> IsCoset( coset );
true
gap> x in coset;
true
gap> coset in f;
false
```

The group `f` acts on its elements (**not** on the cosets) via right multiplication, yielding the regular permutation representation of `f` and thus a new permutation group, namely the linear group $L_3(2)$. A more elaborate discussion of operations of groups can be found in section 1.19 and chapter 8.

```
gap> op:= Operation( f, Elements( f ), OnRight );;
gap> IsPermGroup( op );
true
gap> Maximum( List( op.generators, LargestMovedPointPerm ) );
168
gap> IsSimple( op );
true
```

`norm` acts on the seven nontrivial elements of its normal subgroup `elab` by conjugation, yielding a representation of $L_3(2)$ on seven points. We embed this permutation group in `norm` and deduce that `norm` is a split extension of an elementary abelian group 2^3 with $L_3(2)$.

```
gap> op:= Operation( norm, Elements( elab ), OnPoints );
Group( (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6), (2,3)(6,7),
(3,4)(5,6) )
gap> IsSubgroup(   a8, AsSubgroup( s8, op ) );
true
gap> IsSubgroup( norm, AsSubgroup( s8, op ) );
true
gap> Intersection( elab, op );
Group( () )
```

Yet another kind of information about our `a8` concerns its conjugacy classes.

```
gap> ccl:= ConjugacyClasses( a8 );
[ ConjugacyClass( a8, () ), ConjugacyClass( a8, (1,3)(2,6)(4,7)(5,8) )
    , ConjugacyClass( a8, (1,3)(2,8,5)(6,7) ),
  ConjugacyClass( a8, (2,5,8) ), ConjugacyClass( a8, (1,3)(6,7) ),
  ConjugacyClass( a8, (1,3,2,5,4,7,8) ),
  ConjugacyClass( a8, (1,5,8,2,7,3,4) ),
  ConjugacyClass( a8, (1,5)(2,8,7,4,3,6) ),
  ConjugacyClass( a8, (2,7,3)(4,6,8) ),
  ConjugacyClass( a8, (1,6)(3,8,5,4) ),
  ConjugacyClass( a8, (1,3,5,2)(4,6,8,7) ),
  ConjugacyClass( a8, (1,8,6,2,5) ),
  ConjugacyClass( a8, (1,7,2,4,3)(5,8,6) ),
  ConjugacyClass( a8, (1,2,3,7,4)(5,8,6) ) ]
gap> Length( ccl );
```

```
14
gap> reps:= List( ccl, Representative );
[ (), (1,3)(2,6)(4,7)(5,8), (1,3)(2,8,5)(6,7), (2,5,8), (1,3)(6,7),
  (1,3,2,5,4,7,8), (1,5,8,2,7,3,4), (1,5)(2,8,7,4,3,6),
  (2,7,3)(4,6,8), (1,6)(3,8,5,4), (1,3,5,2)(4,6,8,7), (1,8,6,2,5),
  (1,7,2,4,3)(5,8,6), (1,2,3,7,4)(5,8,6) ]
gap> List( reps, r -> Order( a8, r ) );
[ 1, 2, 6, 3, 2, 7, 7, 6, 3, 4, 4, 5, 15, 15 ]
gap> List( ccl, Size );
[ 1, 105, 1680, 112, 210, 2880, 2880, 3360, 1120, 2520, 1260, 1344,
  1344, 1344 ]
```

Note the difference between `Order` (which means the element order), `Size` (which means the size of the conjugacy class) and `Length` (which means the length of a list).

Having the conjugacy classes, we can consider class functions, i.e., maps that are defined on the group elements, and that are constant on each conjugacy class. One nice example is the number of fixed points; here we use that permutations act on points via `^`.

```
gap> nrfixedpoints:= function( perm, support )
> return Number( [ 1 .. support ], x -> x^perm = x );
> end;
function ( perm, support ) ... end
```

Note that we must specify the support since a permutation does not know about the group it is an element of; e.g. the trivial permutation `()` has as many fixed points as the support denotes.

```
gap> permchar1:= List( reps, x -> nrfixedpoints( x, 8 ) );
[ 8, 0, 1, 5, 4, 1, 1, 0, 2, 2, 0, 3, 0, 0 ]
```

This is the character of the natural permutation representation of `a8` (More about characters can be found in chapters 49 ff.). In order to get another representation of `a8`, we consider another action, namely that on the elements of a conjugacy class by conjugation; note that this is denoted by `OnPoints`, too.

```
gap> class := First( ccl, c -> Size(c) = 112 );
ConjugacyClass( a8, (2,5,8) )
gap> op:= Operation( a8, Elements( class ), OnPoints );;
```

We get a permutation representation `op` on 112 points. It is more useful to look for properties than at the permutations.

```
gap> IsPrimitive( op, [ 1 .. 112 ] );
false
gap> blocks:= Blocks( op, [ 1 .. 112 ] );
[ [ 1, 2 ], [ 6, 8 ], [ 14, 19 ], [ 17, 20 ], [ 36, 40 ], [ 32, 39 ],
  [ 3, 5 ], [ 4, 7 ], [ 10, 15 ], [ 65, 70 ], [ 60, 69 ], [ 54, 63 ],
  [ 55, 68 ], [ 50, 67 ], [ 13, 16 ], [ 27, 34 ], [ 22, 29 ],
  [ 28, 38 ], [ 24, 37 ], [ 31, 35 ], [ 9, 12 ], [ 106, 112 ],
  [ 100, 111 ], [ 11, 18 ], [ 93, 104 ], [ 23, 33 ], [ 26, 30 ],
  [ 94, 110 ], [ 88, 109 ], [ 49, 62 ], [ 44, 61 ], [ 43, 56 ],
  [ 53, 58 ], [ 48, 57 ], [ 45, 66 ], [ 59, 64 ], [ 87, 103 ],
  [ 81, 102 ], [ 80, 96 ], [ 92, 98 ], [ 47, 52 ], [ 42, 51 ],
```

```
  [ 41, 46 ], [ 82, 108 ], [ 99, 105 ], [ 21, 25 ], [ 75, 101 ],
  [ 74, 95 ], [ 86, 97 ], [ 76, 107 ], [ 85, 91 ], [ 73, 89 ],
  [ 72, 83 ], [ 79, 90 ], [ 78, 84 ], [ 71, 77 ] ]
gap> op2:= Operation( op, blocks, OnSets );;
gap> IsPrimitive( op2, [ 1 .. 56 ] );
true
```

The action of `op` on the given block system gave us a new representation on 56 points which is primitive, i.e., the point stabilizer is a maximal subgroup. We compute its preimage in the representation on eight points using homomorphisms (which of course are monomorphisms).

```
gap> ophom := OperationHomomorphism( a8, op  );;
gap> Kernel(ophom);
Subgroup( s8, [  ] )
gap> ophom2:= OperationHomomorphism( op, op2 );;
gap> stab:= Stabilizer( op2, 1 );;
gap> Size( stab );
360
gap> composition:= ophom * ophom2;;
gap> preim:= PreImage( composition, stab );
Subgroup( s8, [ (1,3,2), (2,4,3), (1,3)(7,8), (2,3)(4,5), (6,8,7) ] )
```

And this is the permutation character (with respect to the succession of conjugacy classes in `ccl`):

```
gap> permchar2:= List( reps, x->nrfixedpoints(x^composition,56) );
[ 56, 0, 3, 11, 12, 0, 0, 0, 2, 2, 0, 1, 1, 1 ]
```

The normalizer of an element in the conjugacy class `class` is a group of order 360, too. In fact, it is essentially the same as the maximal subgroup we had found before

```
gap> sgp:= Normalizer( a8,
>                       Subgroup( s8, [ Representative(class) ] ) );
Subgroup( s8, [ (2,5)(3,4), (1,3,4), (2,5,8), (1,3,7)(2,5,8),
  (1,4,7,3,6)(2,5,8) ] )
gap> Size( sgp );
360
gap> IsConjugate( a8, sgp, preim );
true
```

The scalar product of permutation characters of two subgroups U, V, say, equals the number of (U, V)–double cosets (again, see chapters 49 ff. for the details). For example, the norm of the permutation character `permchar1` of degree eight is two since the action of `a8` on the cosets of a point stabilizer is at least doubly transitive:

```
gap> stab:= Stabilizer( a8, 1 );;
gap> double:= DoubleCosets( a8, stab, stab );
[ DoubleCoset( Subgroup( s8, [ (3,8,7), (3,4)(7,8), (3,5,4,8,7),
      (3,6,5)(4,8,7), (2,6,4,5)(7,8) ] ), (), Subgroup( s8,
    [ (3,8,7), (3,4)(7,8), (3,5,4,8,7), (3,6,5)(4,8,7),
      (2,6,4,5)(7,8) ] ) ),
  DoubleCoset( Subgroup( s8, [ (3,8,7), (3,4)(7,8), (3,5,4,8,7),
      (3,6,5)(4,8,7), (2,6,4,5)(7,8) ] ), (1,2)(7,8), Subgroup( s8,
```

```
    [ (3,8,7), (3,4)(7,8), (3,5,4,8,7), (3,6,5)(4,8,7),
      (2,6,4,5)(7,8) ] ) ) ]
gap> Length( double );
2
```

We compute the numbers of (sgp, sgp) and (sgp, stab) double cosets.

```
gap> Length( DoubleCosets( a8, sgp, sgp ) );
4
gap> Length( DoubleCosets( a8, sgp, stab ) );
2
```

Thus both irreducible constituents of `permchar1` are also constituents of `permchar2`, i.e., the difference of the two permutation characters is a proper character of `a8` of norm two.

```
gap> permchar2 - permchar1;
[ 48, 0, 2, 6, 8, -1, -1, 0, 0, 0, 0, -2, 1, 1 ]
```

1.19 About Operations of Groups

One of the most important tools in group theory is the **operation** or **action** of a group on a certain set.

We say that a group G operates on a set D if we have a function that takes each pair (d, g) with $d \in D$ and $g \in G$ to another element $d^g \in D$, which we call the image of d under g, such that $d^{identity} = d$ and $(d^g)^h = d^{gh}$ for each $d \in D$ and $g, h \in G$.

This is equivalent to saying that an operation is a homomorphism of the group G into the full symmetric group on D. We usually call D the **domain** of the operation and its elements **points**.

In this section we will demonstrate how you can compute with operations of groups. For an example we will use the alternating group on 8 points.

```
gap> a8 := Group( (1,2,3), (2,3,4,5,6,7,8) );;
gap> a8.name := "a8";;
```

It is important to note however, that the applicability of the functions from the operation package is not restricted to permutation groups. All the functions mentioned in this section can also be used to compute with the operation of a matrix group on the vectors, etc. We only use a permutation group here because this makes the examples more compact.

The standard operation in GAP3 is always denoted by the caret (^) operator. That means that when no other operation is specified (we will see below how this can be done) all the functions from the operations package will compute the image of a point p under an element g as p^g. Note that this can already denote different operations, depending on the type of points and the type of elements. For example if the group elements are permutations it can either denote the normal operation when the points are integers or the conjugation when the points are permutations themselves (see 20.2). For another example if the group elements are matrices it can either denote the multiplication from the right when the points are vectors or again the conjugation when the points are matrices (of the same dimension) themselves (see 34.1). Which operations are available through the caret operator for a particular type of group elements is described in the chapter for this type of group elements.

```
gap> 2 ^ (1,2,3);
```

```
3
gap> 1 ^ a8.2;
1
gap> (2,4) ^ (1,2,3);
(3,4)
```

The most basic function of the operations package is the function `Orbit`, which computes the orbit of a point under the operation of the group.

```
gap> Orbit( a8, 2 );
[ 2, 3, 1, 4, 5, 6, 7, 8 ]
```

Note that the orbit is not a set, because it is not sorted. See 8.16 for the definition in which order the points appear in an orbit.

We will try to find several subgroups in `a8` using the operations package. One subgroup is immediately available, namely the stabilizer of one point. The index of the stabilizer must of course be equal to the length of the orbit, i.e., 8.

```
gap> u8 := Stabilizer( a8, 1 );
Subgroup( a8, [ (2,3,4,5,6,7,8), (3,8,7) ] )
gap> Index( a8, u8 );
8
```

This gives us a hint how to find further subgroups. Each subgroup is the stabilizer of a point of an appropriate transitive operation (namely the operation on the cosets of that subgroup or another operation that is equivalent to this operation).

So the question is how to find other operations. The obvious thing is to operate on pairs of points. So using the function `Tuples` (see 47.9) we first generate a list of all pairs.

```
gap> pairs := Tuples( [1..8], 2 );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ],
  [ 1, 7 ], [ 1, 8 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 2, 4 ],
  [ 2, 5 ], [ 2, 6 ], [ 2, 7 ], [ 2, 8 ], [ 3, 1 ], [ 3, 2 ],
  [ 3, 3 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 3, 7 ], [ 3, 8 ],
  [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ], [ 4, 5 ], [ 4, 6 ],
  [ 4, 7 ], [ 4, 8 ], [ 5, 1 ], [ 5, 2 ], [ 5, 3 ], [ 5, 4 ],
  [ 5, 5 ], [ 5, 6 ], [ 5, 7 ], [ 5, 8 ], [ 6, 1 ], [ 6, 2 ],
  [ 6, 3 ], [ 6, 4 ], [ 6, 5 ], [ 6, 6 ], [ 6, 7 ], [ 6, 8 ],
  [ 7, 1 ], [ 7, 2 ], [ 7, 3 ], [ 7, 4 ], [ 7, 5 ], [ 7, 6 ],
  [ 7, 7 ], [ 7, 8 ], [ 8, 1 ], [ 8, 2 ], [ 8, 3 ], [ 8, 4 ],
  [ 8, 5 ], [ 8, 6 ], [ 8, 7 ], [ 8, 8 ] ]
```

Now we would like to have `a8` operate on this domain. But we cannot use the default operation (denoted by the caret) because *list* ^ *perm* is not defined. So we must tell the functions from the operations package how the group elements operate on the elements of the domain. In our example we can do this by simply passing `OnPairs` as optional last argument. All functions from the operations package accept such an optional argument that describes the operation. See 8.1 for a list of the available nonstandard operations.

Note that those operations are in fact simply functions that take an element of the domain and an element of the group and return the image of the element of the domain under the group element. So to compute the image of the pair `[1,2]` under the permutation `(1,4,5)` we can simply write

```
gap> OnPairs( [1,2], (1,4,5) );
[ 4, 2 ]
```

As was mentioned above we have to make sure that the operation is transitive. So we check this.

```
gap> IsTransitive( a8, pairs, OnPairs );
false
```

The operation is not transitive, so we want to find out what the orbits are. The function `Orbits` does that for you. It returns a list of all the orbits.

```
gap> orbs := Orbits( a8, pairs, OnPairs );
[ [ [ 1, 1 ], [ 2, 2 ], [ 3, 3 ], [ 4, 4 ], [ 5, 5 ], [ 6, 6 ],
      [ 7, 7 ], [ 8, 8 ] ],
  [ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 3, 1 ], [ 3, 4 ], [ 2, 1 ],
      [ 1, 4 ], [ 4, 1 ], [ 4, 5 ], [ 3, 2 ], [ 2, 4 ], [ 1, 5 ],
      [ 4, 2 ], [ 5, 1 ], [ 5, 6 ], [ 4, 3 ], [ 3, 5 ], [ 2, 5 ],
      [ 1, 6 ], [ 5, 3 ], [ 5, 2 ], [ 6, 1 ], [ 6, 7 ], [ 5, 4 ],
      [ 4, 6 ], [ 3, 6 ], [ 2, 6 ], [ 1, 7 ], [ 6, 4 ], [ 6, 3 ],
      [ 6, 2 ], [ 7, 1 ], [ 7, 8 ], [ 6, 5 ], [ 5, 7 ], [ 4, 7 ],
      [ 3, 7 ], [ 2, 7 ], [ 1, 8 ], [ 7, 5 ], [ 7, 4 ], [ 7, 3 ],
      [ 7, 2 ], [ 8, 1 ], [ 8, 2 ], [ 7, 6 ], [ 6, 8 ], [ 5, 8 ],
      [ 4, 8 ], [ 3, 8 ], [ 2, 8 ], [ 8, 6 ], [ 8, 5 ], [ 8, 4 ],
      [ 8, 3 ], [ 8, 7 ] ] ]
```

The operation of `a8` on the first orbit is of course equivalent to the original operation, so we ignore it and work with the second orbit.

```
gap> u56 := Stabilizer( a8, [1,2], OnPairs );
Subgroup( a8, [ (3,8,7), (3,6)(4,7,5,8), (6,7,8) ] )
gap> Index( a8, u56 );
56
```

So now we have found a second subgroup. To make the following computations a little bit easier and more efficient we would now like to work on the points `[1..56]` instead of the list of pairs. The function `Operation` does what we need. It creates a new group that operates on `[1..56]` in the same way that `a8` operates on the second orbit.

```
gap> a8_56 := Operation( a8, orbs[2], OnPairs );
Group( ( 1, 2, 4)( 3, 6,10)( 5, 7,11)( 8,13,16)(12,18,17)(14,21,20)
(19,27,26)(22,31,30)(28,38,37)(32,43,42)(39,51,50)(44,45,55),
( 1, 3, 7,12,19,28,39)( 2, 5, 9,15,23,33,45)( 4, 8,14,22,32,44, 6)
(10,16,24,34,46,56,51)(11,17,25,35,47,43,55)(13,20,29,40,52,38,50)
(18,26,36,48,31,42,54)(21,30,41,53,27,37,49) )
gap> a8_56.name := "a8_56";;
```

We would now like to know if the subgroup `u56` of index 56 that we found is maximal or not. Again we can make use of a function from the operations package. Namely a subgroup is maximal if the operation on the cosets of this subgroup is primitive, i.e., if there is no partition of the set of cosets into subsets such that the group operates setwise on those subsets.

```
gap> IsPrimitive( a8_56, [1..56] );
```

```
false
```

Note that we must specify the domain of the operation. You might think that in the last example `IsPrimitive` could use `[1..56]` as default domain if no domain was given. But this is not so simple, for example would the default domain of `Group( (2,3,4) )` be `[1..4]` or `[2..4]`? To avoid confusion, all operations package functions require that you specify the domain of operation.

We see that `a8_56` is not primitive. This means of course that the operation of `a8` on `orb[2]` is not primitive, because those two operations are equivalent. So the stabilizer `u56` is not maximal. Let us try to find its supergroups. We use the function `Blocks` to find a block system. The (optional) third argument in the following example tells `Blocks` that we want a block system where 1 and 10 lie in one block. There are several other block systems, which we could compute by specifying a different pair, it just turns out that `[1,10]` makes the following computation more interesting.

```
gap> blocks := Blocks( a8_56, [1..56], [1,10] );
[ [ 1, 10, 13, 21, 31, 43, 45 ], [ 2, 3, 16, 20, 30, 42, 55 ],
  [ 4, 6, 8, 14, 22, 32, 44 ], [ 5, 7, 11, 24, 29, 41, 54 ],
  [ 9, 12, 17, 18, 34, 40, 53 ], [ 15, 19, 25, 26, 27, 46, 52 ],
  [ 23, 28, 35, 36, 37, 38, 56 ], [ 33, 39, 47, 48, 49, 50, 51 ] ]
```

The result is a list of sets, i.e., sorted lists, such that `a8_56` operates on those sets. Now we would like the stabilizer of this operation on the sets. Because we wanted to operate on the sets we have to pass `OnSets` as third argument.

```
gap> u8_56 := Stabilizer( a8_56, blocks[1], OnSets );
Subgroup( a8_56,
[ (15,35,48)(19,28,39)(22,32,44)(23,33,52)(25,36,49)(26,37,50)
    (27,38,51)(29,41,54)(30,42,55)(31,43,45)(34,40,53)(46,56,47),
  ( 9,25)(12,19)(14,22)(15,34)(17,26)(18,27)(20,30)(21,31)(23,48)
    (24,29)(28,39)(32,44)(33,56)(35,47)(36,49)(37,50)(38,51)(40,52)
    (41,54)(42,55)(43,45)(46,53), ( 5,17)( 7,12)( 8,14)( 9,24)(11,18)
    (13,21)(15,25)(16,20)(23,47)(28,39)(29,34)(32,44)(33,56)(35,49)
    (36,48)(37,50)(38,51)(40,54)(41,53)(42,55)(43,45)(46,52),
  ( 2,11)( 3, 7)( 4, 8)( 5,16)( 9,17)(10,13)(20,24)(23,47)(25,26)
    (28,39)(29,30)(32,44)(33,56)(35,48)(36,50)(37,49)(38,51)(40,53)
    (41,55)(42,54)(43,45)(46,52), ( 1,10)( 2, 6)( 3, 4)( 5, 7)( 8,16)
    (12,17)(14,20)(19,26)(22,30)(23,47)(28,50)(32,55)(33,56)(35,48)
    (36,49)(37,39)(38,51)(40,53)(41,54)(42,44)(43,45)(46,52) ] )
gap> Index( a8_56, u8_56 );
8
```

Now we have a problem. We have found a new subgroup, but not as a subgroup of `a8`, instead it is a subgroup of `a8_56`. We know that `a8_56` is isomorphic to `a8` (in general the result of `Operation` is only isomorphic to a factor group of the original group, but in this case it must be isomorphic to `a8`, because `a8` is simple and has only the full group as nontrivial factor group). But we only know that an isomorphism exists, we do not know it.

Another function comes to our rescue. `OperationHomomorphism` returns the homomorphism of a group onto the group that was constructed by `Operation`. A later section in this chapter will introduce mappings and homomorphisms in general, but for the moment we can just regard the result of `OperationHomomorphism` as a black box that we can use to transfer information from `a8` to `a8_56` and back.

```
gap> h56 := OperationHomomorphism( a8, a8_56 );
OperationHomomorphism( a8, a8_56 )
gap> u8b := PreImages( h56, u8_56 );
Subgroup( a8, [ (6,7,8), (5,6)(7,8), (4,5)(7,8), (3,4)(7,8),
  (1,3)(7,8) ] )
gap> Index( a8, u8b );
8
gap> u8 = u8b;
false
```

So we have in fact found a new subgroup. However if we look closer we note that `u8b` is not totally new. It fixes the point 2, thus it lies in the stabilizer of 2, and because it has the same index as this stabilizer it must in fact be the stabilizer. Thus `u8b` is conjugated to `u8`. A nice way to check this is to check that the operation on the 8 blocks is equivalent to the original operation.

```
gap> IsEquivalentOperation( a8, [1..8], a8_56, blocks, OnSets );
true
```

Now the choice of the third argument `[1,10]` of `Blocks` becomes clear. Had we not given that argument we would have obtained the block system that has `[1,3,7,12,19,28,39]` as first block. The preimage of the stabilizer of this set would have been `u8` itself, and we would not have been able to introduce `IsEquivalentOperation`. Of course we could also use the general function `IsConjugate`, but we want to demonstrate `IsEquivalentOperation`.

Actually there is a third block system of `a8_56` that gives rise to a third subgroup.

```
gap> blocks := Blocks( a8_56, [1..56], [1,6] );
[ [ 1, 6 ], [ 2, 10 ], [ 3, 4 ], [ 5, 16 ], [ 7, 8 ], [ 9, 24 ],
  [ 11, 13 ], [ 12, 14 ], [ 15, 34 ], [ 17, 20 ], [ 18, 21 ],
  [ 19, 22 ], [ 23, 46 ], [ 25, 29 ], [ 26, 30 ], [ 27, 31 ],
  [ 28, 32 ], [ 33, 56 ], [ 35, 40 ], [ 36, 41 ], [ 37, 42 ],
  [ 38, 43 ], [ 39, 44 ], [ 45, 51 ], [ 47, 52 ], [ 48, 53 ],
  [ 49, 54 ], [ 50, 55 ] ]
gap> u28_56 := Stabilizer( a8_56, [1,6], OnSets );
Subgroup( a8_56,
[ ( 2,38,51)( 3,28,39)( 4,32,44)( 5,41,54)(10,43,45)(16,36,49)
    (17,40,53)(20,35,48)(23,47,30)(26,46,52)(33,55,37)(42,56,50),
  ( 5,17,26,37,50)( 7,12,19,28,39)( 8,14,22,32,44)( 9,15,23,33,54)
    (11,18,27,38,51)(13,21,31,43,45)(16,20,30,42,55)(24,34,46,56,49)
    (25,35,47,41,53)(29,40,52,36,48),
  ( 1, 6)( 2,39,38,19,18, 7)( 3,51,28,27,12,11)( 4,45,32,31,14,13)
    ( 5,55,33,23,15, 9)( 8,10,44,43,22,21)(16,50,56,46,34,24)
    (17,54,42,47,35,25)(20,49,37,52,40,29)(26,53,41,30,48,36) ] )
gap> u28 := PreImages( h56, u28_56 );
Subgroup( a8, [ (3,7,8), (4,5,6,7,8), (1,2)(3,8,7,6,5,4) ] )
gap> Index( a8, u28 );
28
```

We know that the subgroup u28 of index 28 is maximal, because we know that a8 has no subgroups of index 2, 4, or 7. However we can also quickly verify this by checking that a8_56 operates primitively on the 28 blocks.

```
gap> IsPrimitive( a8_56, blocks, OnSets );
true
```

There is a different way to obtain u28. Instead of operating on the 56 pairs [[1,2], [1,3], ..., [8,7]] we could operate on the 28 sets of two elements from [1..8]. But suppose we make a small mistake.

```
gap> OrbitLength( a8, [2,1], OnSets );
Error, OnSets: <tuple> must be a set
```

It is your responsibility to make sure that the points that you pass to functions from the operations package are in normal form. That means that they must be sets if you operate on sets with OnSets, they must be lists of length 2 if you operate on pairs with OnPairs, etc. This also applies to functions that accept a domain of operation, e.g., Operation and IsPrimitive. All points in such a domain must be in normal form. **It is not guaranteed that a violation of this rule will signal an error, you may obtain incorrect results.**

Note that Stabilizer is not only applicable to groups like a8 but also to their subgroups like u56. So another method to find a new subgroup is to compute the stabilizer of another point in u56. Note that u56 already leaves 1 and 2 fixed.

```
gap> u336 := Stabilizer( u56, 3 );
Subgroup( a8, [ (4,6,5), (5,6)(7,8), (6,7,8) ] )
gap> Index( a8, u336 );
336
```

Other functions are also applicable to subgroups. In the following we show that u336 operates regularly on the 60 triples of [4..8] which contain no element twice, which means that this operation is equivalent to the operations of u336 on its 60 elements from the right. Note that OnTuples is a generalization of OnPairs.

```
gap> IsRegular( u336, Orbit( u336, [4,5,6], OnTuples ), OnTuples );
true
```

Just as we did in the case of the operation on the pairs above, we now construct a new permutation group that operates on [1..336] in the same way that a8 operates on the cosets of u336. Note that the operation of a group on the cosets is by multiplication from the right, thus we have to specify OnRight.

```
gap> a8_336 := Operation( a8, Cosets( a8, u336 ), OnRight );;
gap> a8_336.name := "a8_336";;
```

To find subgroups above u336 we again check if the operation is primitive.

```
gap> blocks := Blocks( a8_336, [1..336], [1,43] );
[ [ 1, 43, 85 ], [ 2, 102, 205 ], [ 3, 95, 165 ], [ 4, 106, 251 ],
  [ 5, 117, 334 ], [ 6, 110, 294 ], [ 7, 122, 127 ], [ 8, 144, 247 ],
  [ 9, 137, 207 ], [ 10, 148, 293 ], [ 11, 45, 159 ],
  [ 12, 152, 336 ], [ 13, 164, 169 ], [ 14, 186, 289 ],
  [ 15, 179, 249 ], [ 16, 190, 335 ], [ 17, 124, 201 ],
  [ 18, 44, 194 ], [ 19, 206, 211 ], [ 20, 228, 331 ],
```

```
[ 21, 221, 291 ], [ 22, 46, 232 ], [ 23, 166, 243 ],
[ 24, 126, 236 ], [ 25, 248, 253 ], [ 26, 48, 270 ],
[ 27, 263, 333 ], [ 28, 125, 274 ], [ 29, 208, 285 ],
[ 30, 168, 278 ], [ 31, 290, 295 ], [ 32, 121, 312 ],
[ 33, 47, 305 ], [ 34, 167, 316 ], [ 35, 250, 327 ],
[ 36, 210, 320 ], [ 37, 74, 332 ], [ 38, 49, 163 ], [ 39, 81, 123 ],
[ 40, 59, 209 ], [ 41, 70, 292 ], [ 42, 66, 252 ], [ 50, 142, 230 ],
[ 51, 138, 196 ], [ 52, 146, 266 ], [ 53, 87, 131 ],
[ 54, 153, 302 ], [ 55, 160, 174 ], [ 56, 182, 268 ],
[ 57, 178, 234 ], [ 58, 189, 304 ], [ 60, 86, 199 ],
[ 61, 198, 214 ], [ 62, 225, 306 ], [ 63, 218, 269 ],
[ 64, 88, 235 ], [ 65, 162, 245 ], [ 67, 233, 254 ],
[ 68, 90, 271 ], [ 69, 261, 301 ], [ 71, 197, 288 ],
[ 72, 161, 281 ], [ 73, 265, 297 ], [ 75, 89, 307 ],
[ 76, 157, 317 ], [ 77, 229, 328 ], [ 78, 193, 324 ],
[ 79, 116, 303 ], [ 80, 91, 158 ], [ 82, 101, 195 ],
[ 83, 112, 267 ], [ 84, 108, 231 ], [ 92, 143, 237 ],
[ 93, 133, 200 ], [ 94, 150, 273 ], [ 96, 154, 309 ],
[ 97, 129, 173 ], [ 98, 184, 272 ], [ 99, 180, 238 ],
[ 100, 188, 308 ], [ 103, 202, 216 ], [ 104, 224, 310 ],
[ 105, 220, 276 ], [ 107, 128, 241 ], [ 109, 240, 256 ],
[ 111, 260, 311 ], [ 113, 204, 287 ], [ 114, 130, 277 ],
[ 115, 275, 296 ], [ 118, 132, 313 ], [ 119, 239, 330 ],
[ 120, 203, 323 ], [ 134, 185, 279 ], [ 135, 175, 242 ],
[ 136, 192, 315 ], [ 139, 171, 215 ], [ 140, 226, 314 ],
[ 141, 222, 280 ], [ 145, 244, 258 ], [ 147, 262, 318 ],
[ 149, 170, 283 ], [ 151, 282, 298 ], [ 155, 246, 329 ],
[ 156, 172, 319 ], [ 176, 227, 321 ], [ 177, 217, 284 ],
[ 181, 213, 257 ], [ 183, 264, 322 ], [ 187, 286, 300 ],
[ 191, 212, 325 ], [ 219, 259, 326 ], [ 223, 255, 299 ] ]
```

To find the subgroup of index 112 that belongs to this operation we could use the same methods as before, but we actually use a different trick. From the above we see that the subgroup is the union of `u336` with its 43rd and its 85th coset. Thus we simply add a representative of the 43rd coset to the generators of `u336`.

```
gap> u112 := Closure( u336, Representative( Cosets(a8,u336)[43] ) );
Subgroup( a8, [ (4,6,5), (5,6)(7,8), (6,7,8), (1,3,2) ] )
gap> Index( a8, u112 );
112
```

Above this subgroup of index 112 lies a subgroup of index 56, which is not conjugate to `u56`. In fact, unlike `u56` it is maximal. We obtain this subgroup in the same way that we obtained `u112`, this time forcing two points, namely 39 and 43 into the first block.

```
gap> blocks := Blocks( a8_336, [1..336], [1,39,43] );;
gap> Length( blocks );
56
gap> u56b := Closure( u112, Representative( Cosets(a8,u336)[39] ) );
Subgroup( a8, [ (4,6,5), (5,6)(7,8), (6,7,8), (1,3,2), (2,3)(7,8) ] )
gap> Index( a8, u56b );
```

```
56
gap> IsPrimitive( a8_336, blocks, OnSets );
true
```

We already mentioned in the beginning that there is another standard operation of permutations, namely the conjugation. E.g., because no other operation is specified in the following example `OrbitLength` simply operates using the caret operator and because *perm1*`^`*perm2* is defined as the conjugation of *perm2* on *perm1* we effectively compute the length of the conjugacy class of `(1,2)(3,4)(5,6)(7,8)`. (In fact *element1*`^`*element2* is always defined as the conjugation if *element1* and *element2* are group elements of the same type. So the length of a conjugacy class of any element *elm* in an arbitrary group G can be computed as `OrbitLength(` G`,` *elm* `)`. In general however this may not be a good idea, `Size( ConjugacyClass(` G`,` *elm* `) )` is probably more efficient.)

```
gap> OrbitLength( a8, (1,2)(3,4)(5,6)(7,8) );
105
gap> orb := Orbit( a8, (1,2)(3,4)(5,6)(7,8) );;
gap> u105 := Stabilizer( a8, (1,2)(3,4)(5,6)(7,8) );
Subgroup( a8, [ (5,6)(7,8), (1,2)(3,4)(5,6)(7,8), (5,7)(6,8),
  (3,4)(7,8), (3,5)(4,6), (1,3)(2,4) ] )
gap> Index( a8, u105 );
105
```

Of course the last stabilizer is in fact the centralizer of the element `(1,2)(3,4)(5,6)(7,8)`. `Stabilizer` notices that and computes the stabilizer using the centralizer algorithm for permutation groups.

In the usual way we now look for the subgroups that lie above `u105`.

```
gap> blocks := Blocks( a8, orb );;
gap> Length( blocks );
15
gap> blocks[1];
[ (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8), (1,4)(2,3)(5,8)(6,7),
  (1,5)(2,6)(3,7)(4,8), (1,6)(2,5)(3,8)(4,7), (1,7)(2,8)(3,5)(4,6),
  (1,8)(2,7)(3,6)(4,5) ]
```

To find the subgroup of index 15 we again use closure. Now we must be a little bit careful to avoid confusion. `u105` is the stabilizer of `(1,2)(3,4)(5,6)(7,8)`. We know that there is a correspondence between the points of the orbit and the cosets of `u105`. The point `(1,2)(3,4)(5,6)(7,8)` corresponds to `u105`. To get the subgroup of index 15 we must add to `u105` an element of the coset that corresponds to the point `(1,3)(2,4)(5,7)(6,8)` (or any other point in the first block). That means that we must use an element of `a8` that maps `(1,2)(3,4)(5,6)(7,8)` to `(1,3)(2,4)(5,7)(6,8)`. The important thing is that `(1,3)(2,4)(5,7)(6,8)` will not do, in fact `(1,3)(2,4)(5,7)(6,8)` lies in `u105`.

The function `RepresentativeOperation` does what we need. It takes a group and two points and returns an element of the group that maps the first point to the second. In fact it also allows you to specify the operation as optional fourth argument as usual, but we do not need this here. If no such element exists in the group, i.e., if the two points do not lie in one orbit under the group, `RepresentativeOperation` returns `false`.

```
gap> rep := RepresentativeOperation( a8, (1,2)(3,4)(5,6)(7,8),
```

```
>                                         (1,3)(2,4)(5,7)(6,8) );
(2,3)(6,7)
gap> u15 := Closure( u105, rep );
Subgroup( a8, [ (5,6)(7,8), (1,2)(3,4)(5,6)(7,8), (5,7)(6,8),
  (3,4)(7,8), (3,5)(4,6), (1,3)(2,4), (2,3)(6,7) ] )
gap> Index( a8, u15 );
15
```

`u15` is of course a maximal subgroup, because `a8` has no subgroups of index 3 or 5.

There is in fact another class of subgroups of index 15 above `u105` that we get by adding `(2,3)(6,8)` to `u105`.

```
gap> u15b := Closure( u105, (2,3)(6,8) );
Subgroup( a8, [ (5,6)(7,8), (1,2)(3,4)(5,6)(7,8), (5,7)(6,8),
  (3,4)(7,8), (3,5)(4,6), (1,3)(2,4), (2,3)(6,8) ] )
gap> Index( a8, u15b );
15
```

We now show that `u15` and `u15b` are not conjugate. We showed that `u8` and `u8b` are conjugate by showing that the operations on the cosets where equivalent. We could show that `u15` and `u15b` are not conjugate by showing that the operations on their cosets are not equivalent. Instead we simply call `RepresentativeOperation` again.

```
gap> RepresentativeOperation( a8, u15, u15b );
false
```

`RepresentativeOperation` tells us that there is no element g in `a8` such that `u15^`g `=` `u15b`. Because `^` also denotes the conjugation of subgroups this tells us that `u15` and `u15b` are not conjugate. Note that this operation should only be used rarely, because it is usually not very efficient. The test in this case is however reasonable efficient, and is in fact the one employed by `IsConjugate` (see 7.54).

This concludes our example. In this section we demonstrated some functions from the operations package. There is a whole class of functions that we did not mention, namely those that take a single element instead of a whole group as first argument, e.g., `Cycle` and `Permutation`. All functions are described in the chapter 8.

1.20 About Finitely Presented Groups and Presentations

In this section we will show you the investigation of a Coxeter group that is given by its presentation. You will see that finitely presented groups and presentations are different kinds of objects in GAP3. While finitely presented groups can never be changed after they have been created as factor groups of free groups, presentations allow manipulations of the generators and relators by Tietze transformations. The investigation of the example will involve methods and algorithms like Todd-Coxeter, Reidemeister-Schreier, Nilpotent Quotient, and Tietze transformations.

We start by defining a Coxeter group `c` on five generators as a factor group of the free group of rank 5, whose generators we already call `c.1`, ..., `c.5`.

```
gap> c := FreeGroup( 5, "c" );;
```

```
gap> r := List( c.generators, x -> x^2 );;
gap> Append( r, [ (c.1*c.2)^3, (c.1*c.3)^2, (c.1*c.4)^3,
>  (c.1*c.5)^3, (c.2*c.3)^3, (c.2*c.4)^2, (c.2*c.5)^3,
>  (c.3*c.4)^3, (c.3*c.5)^3, (c.4*c.5)^3,
>  (c.1*c.2*c.5*c.2)^2, (c.3*c.4*c.5*c.4)^2 ] );
gap> c := c / r;
Group( c.1, c.2, c.3, c.4, c.5 )
```

If we call the function `Size` for this group GAP3 will invoke the Todd-Coxeter method, which however will fail to get a result going up to the default limit of defining 64000 cosets:

```
gap> Size(c);
Error, the coset enumeration has defined more than 64000 cosets:
type 'return;' if you want to continue with a new limit of
128000 cosets,
type 'quit;' if you want to quit the coset enumeration,
type 'maxlimit := 0; return;' in order to continue without a limit,
 in
AugmentedCosetTableMtc( G, H, -1, "_x" ) called from
D.operations.Size( D ) called from
Size( c ) called from
main loop
brk> quit;
```

In fact, as we shall see later, our finitely presented group is infinite and hence we would get the same answer also with larger limits. So we next look for subgroups of small index, in our case limiting the index to four.

```
gap> lis := LowIndexSubgroupsFpGroup( c, TrivialSubgroup(c), 4 );;
gap> Length(lis);
10
```

The `LowIndexSubgroupsFpGroup` function in fact determines generators for the subgroups, written in terms of the generators of the given group. We can find the index of these subgroups by the function `Index`, and the permutation representation on the cosets of these subgroups by the function `OperationCosetsFpGroup`, which use a Todd-Coxeter method. The size of the image of this permutation representation is found using `Size` which in this case uses a Schreier-Sims method for permutation groups.

```
gap> List(lis, x -> [Index(c,x),Size(OperationCosetsFpGroup(c,x))]);
[ [ 1, 1 ], [ 4, 24 ], [ 4, 24 ], [ 4, 24 ], [ 4, 24 ], [ 4, 24 ],
  [ 4, 24 ], [ 4, 24 ], [ 3, 6 ], [ 2, 2 ] ]
```

We next determine the commutator factor groups of the kernels of these permutation representations. Note that here the difference of finitely presented groups and presentations has to be observed: We first determine the kernel of the permutation representation by the function `Core` as a subgroup of `c`, then a presentation of this subgroup using `PresentationSubgroup`, which has to be converted into a finitely presented group of its own right using `FpGroupPresentation`, before its commutator factor group and the abelian invariants can be found using integer matrix diagonalisation of the relators matrix by an elementary divisor algorithm. The conversion is necessary because `Core` computes a subgroup given by words in the generators of `c` but `CommutatorFactorGroup` needs a parent group given by generators and relators.

```
gap> List( lis, x -> AbelianInvariants( CommutatorFactorGroup(
>   FpGroupPresentation( PresentationSubgroup( c, Core(c,x) ) ) ) ) );
[ [ 2 ], [ 2, 2, 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ], [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 0, 0, 0, 0, 0, 0 ], [ 2, 2, 2, 2, 2, 2 ], [ 3 ] ]
```

More clearly arranged, this is

```
[ [ 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 2, 2, 2, 2, 2, 2, 2, 2 ],
  [ 0, 0, 0, 0, 0, 0 ],
  [ 2, 2, 2, 2, 2, 2 ],
  [ 3 ] ]
```

Note that there is another function `AbelianInvariantsSubgroupFpGroup` which we could have used to obtain this list which will do an abelianized Reduced Reidemeister-Schreier. This function is much faster because it does not compute a complete presentation for the core.

The output obtained shows that the third last of the kernels has a free abelian commutator factor group of rank 6. We turn our attention to this kernel which we call `n`, while we call the associated presentation `pr`.

```
gap> lis[8];
Subgroup( Group( c.1, c.2, c.3, c.4, c.5 ),
[ c.1, c.2, c.3*c.2*c.5^-1, c.3*c.4*c.3^-1, c.4*c.1*c.5^-1 ] )
gap> pr := PresentationSubgroup( c, Core( c, lis[8] ) );
<< presentation with 22 gens and 41 rels of total length 156 >>
gap> n := FpGroupPresentation(pr);;
```

We first determine p-factor groups for primes 2, 3, 5, and 7.

```
gap> InfoPQ1:= Ignore;;
gap> List( [2,3,5,7], p -> PrimeQuotient(n,p,5).dimensions );
[ [ 6, 10, 18, 30, 54 ], [ 6, 10, 18, 30, 54 ], [ 6, 10, 18, 30, 54 ],
  [ 6, 10, 18, 30, 54 ] ]
```

Observing that the ranks of the lower exponent-p central series are the same for these primes we suspect that the lower central series may have free abelian factors. To investigate this we have to call the package "nq".

```
gap> RequirePackage("nq");
gap> NilpotentQuotient( n, 5 );
[ [ 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0 ] ]
gap> List( last, Length );
```

```
[ 6, 4, 8, 12, 24 ]
```

The ranks of the factors except the first are divisible by four, and we compare them with the corresponding ranks of a free group on two generators.

```
gap> f2 := FreeGroup(2);
Group( f.1, f.2 )
gap> PrimeQuotient( f2, 2, 5 ).dimensions;
[ 2, 3, 5, 8, 14 ]
gap> NilpotentQuotient( f2, 5 );
[ [ 0, 0 ], [ 0 ], [ 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0 ] ]
gap> List( last, Length );
[ 2, 1, 2, 3, 6 ]
```

The result suggests a close relation of our group to the direct product of four free groups of rank two. In order to study this we want a simple presentation for our kernel `n` and obtain this by repeated use of Tietze transformations, using first the default simplification function `TzGoGo` and later specific introduction of new generators that are obtained as product of two of the existing ones using the function `TzSubstitute`. (Of course, this latter sequence of Tietze transformations that we display here has only been found after some trial and error.)

```
gap> pr := PresentationSubgroup( c, Core( c, lis[8] ) );
<< presentation with 22 gens and 41 rels of total length 156 >>
gap> TzGoGo(pr);
#I  there are 6 generators and 14 relators of total length 74
gap> TzGoGo(pr);
#I  there are 6 generators and 13 relators of total length 66
gap> TzGoGo(pr);
gap> TzPrintPairs(pr);
#I  1.  3  occurrences of  _x6 * _x11^-1
#I  2.  3  occurrences of  _x3 * _x15
#I  3.  2  occurrences of  _x11^-1 * _x15^-1
#I  4.  2  occurrences of  _x6 * _x15
#I  5.  2  occurrences of  _x6^-1 * _x15^-1
#I  6.  2  occurrences of  _x4 * _x15
#I  7.  2  occurrences of  _x4^-1 * _x15^-1
#I  8.  2  occurrences of  _x4^-1 * _x11
#I  9.  2  occurrences of  _x4 * _x6
#I  10.  2  occurrences of  _x3^-1 * _x11
gap> TzSubstitute(pr,10,2);
#I  substituting new generator _x26 defined by _x3^-1*_x11
#I  eliminating _x11 = _x3*_x26
#I  there are 6 generators and 13 relators of total length 70
gap> TzGoGo(pr);
#I  there are 6 generators and 12 relators of total length 62
#I  there are 6 generators and 12 relators of total length 60
gap> TzGoGo(pr);
gap> TzSubstitute(pr,9,2);
#I  substituting new generator _x27 defined by _x1^-1*_x15
#I  eliminating _x15 = _x27*_x1
#I  there are 6 generators and 12 relators of total length 64
```

```
gap> TzGoGo(pr);
#I  there are 6 generators and 11 relators of total length 56
gap> TzGoGo(pr);
gap> p2 := Copy(pr);
<< presentation with 6 gens and 11 rels of total length 56 >>
gap> TzPrint(p2);
#I  generators: [ _x1, _x3, _x4, _x6, _x26, _x27 ]
#I  relators:
#I  1.  4  [ -6, -1, 6, 1 ]
#I  2.  4  [ 4, 6, -4, -6 ]
#I  3.  4  [ 5, 4, -5, -4 ]
#I  4.  4  [ 4, -2, -4, 2 ]
#I  5.  4  [ -3, 2, 3, -2 ]
#I  6.  4  [ -3, -1, 3, 1 ]
#I  7.  6  [ -4, 3, 4, 6, -3, -6 ]
#I  8.  6  [ -1, -6, -2, 6, 1, 2 ]
#I  9.  6  [ -6, -2, -5, 6, 2, 5 ]
#I  10.  6  [ 2, 5, 1, -5, -2, -1 ]
#I  11.  8  [ -1, -6, -5, 3, 6, 1, 5, -3 ]
gap> TzPrintPairs(p2);
#I  1.  5  occurrences of  _x1^-1 * _x27^-1
#I  2.  3  occurrences of  _x6 * _x27
#I  3.  3  occurrences of  _x3 * _x26
#I  4.  2  occurrences of  _x3 * _x27
#I  5.  2  occurrences of  _x1 * _x4
#I  6.  2  occurrences of  _x1 * _x3
#I  7.  1  occurrence  of  _x26 * _x27
#I  8.  1  occurrence  of  _x26 * _x27^-1
#I  9.  1  occurrence  of  _x26^-1 * _x27
#I  10.  1  occurrence  of  _x6 * _x27^-1
gap> TzSubstitute(p2,1,2);
#I  substituting new generator _x28 defined by _x1^-1*_x27^-1
#I  eliminating _x27 = _x1^-1*_x28^-1
#I  there are 6 generators and 11 relators of total length 58
gap> TzGoGo(p2);
#I  there are 6 generators and 11 relators of total length 54
gap> TzGoGo(p2);
gap> p3 := Copy(p2);
<< presentation with 6 gens and 11 rels of total length 54 >>
gap> TzSubstitute(p3,3,2);
#I  substituting new generator _x29 defined by _x3*_x26
#I  eliminating _x26 = _x3^-1*_x29
gap> TzGoGo(p3);
#I  there are 6 generators and 11 relators of total length 52
gap> TzGoGo(p3);
gap> TzPrint(p3);
#I  generators: [ _x1, _x3, _x4, _x6, _x28, _x29 ]
#I  relators:
```

```
#I  1.  4  [ 6, 4, -6, -4 ]
#I  2.  4  [ 1, -6, -1, 6 ]
#I  3.  4  [ -5, -1, 5, 1 ]
#I  4.  4  [ -2, -5, 2, 5 ]
#I  5.  4  [ 4, -2, -4, 2 ]
#I  6.  4  [ -3, 2, 3, -2 ]
#I  7.  4  [ -3, -1, 3, 1 ]
#I  8.  6  [ -2, 5, -6, 2, -5, 6 ]
#I  9.  6  [ 4, -1, -5, -4, 5, 1 ]
#I  10.  6  [ -6, 3, -5, 6, -3, 5 ]
#I  11.  6  [ 3, -5, 4, -3, -4, 5 ]
```

The resulting presentation could further be simplified by Tietze transformations using `TzSubstitute` and `TzGoGo` until one reaches finally a presentation on 6 generators with 11 relators, 9 of which are commutators of the generators. Working by hand from these, the kernel can be identified as a particular subgroup of the direct product of four copies of the free group on two generators.

1.21 About Fields

In this section we will show you some basic computations with fields. GAP3 supports at present the following fields. The rationals, cyclotomic extensions of rationals and their subfields (which we will refer to as number fields in the following), and finite fields.

Let us first take a look at the infinite fields mentioned above. While the set of rational numbers is a predefined domain in GAP3 to which you may refer by its identifier `Rationals`, cyclotomic fields are constructed by using the function `CyclotomicField`, which may be abbreviated as `CF`.

```
gap> IsField( Rationals );
true
gap> Size( Rationals );
"infinity"
gap> f := CyclotomicField( 8 );
CF(8)
gap> IsSubset( f, Rationals );
true
```

The integer argument `n` of the function call to `CF` specifies that the cyclotomic field containing all n-th roots of unity should be returned.

Cyclotomic fields are constructed as extensions of the Rationals by primitive roots of unity. Thus a primitive n-th root of unity is always an element of CF(n), where n is a natural number. In GAP3, one may construct a primitive n-th root of unity by calling `E(n)`.

```
gap> (E(8) + E(8)^3)^2;
-2
gap> E(8) in f;
true
```

For every field extension you can compute the Galois group, i.e., the group of automorphisms that leave the subfield fixed. For an example, cyclotomic fields are an extension of the rationals, so you can compute their Galois group over the rationals.

```
gap> Galf := GaloisGroup( f );
Group( NFAutomorphism( CF(8) , 7 ), NFAutomorphism( CF(8) , 5 ) )
gap> Size( Galf );
4
```

The above cyclotomic field is a small example where the Galois group is not cyclic.

```
gap> IsCyclic( Galf );
false
gap> IsAbelian( Galf );
true
gap> AbelianInvariants( Galf );
[ 2, 2 ]
```

This shows us that the 8th cyclotomic field has a Galois group which is isomorphic to group V_4.

The elements of the Galois group are GAP3 automorphisms, so they may be applied to the elements of the field in the same way as all mappings are usually applied to objects in GAP3.

```
gap> g := Galf.generators[1];
NFAutomorphism( CF(8) , 7 )
gap> E(8) ^ g;
-E(8)^3
```

There are two functions, `Norm` and `Trace`, which compute the norm and the trace of elements of the field, respectively. The norm and the trace of an element a are defined to be the product and the sum of the images of a under the Galois group. You should usually specify the field as a first argument. This argument is however optional. If you omit a default field will be used. For a cyclotomic a this is the smallest cyclotomic field that contains a (note that this is not the smallest field that contains a, which may be a number field that is not a cyclotomic field).

```
gap> orb := List( Elements( Galf ), x -> E(8) ^ x );
[ E(8), E(8)^3, -E(8), -E(8)^3 ]
gap> Sum( orb ) = Trace( f, E(8) );
true
gap> Product( orb ) = Norm( f, E(8) );
true
gap> Trace( f, 1 );
4
```

The basic way to construct a finite field is to use the function `GaloisField` which may be abbreviated, as usual in algebra, as `GF`. Thus

```
gap> k := GF( 3, 4 );
GF(3^4)
```

or

```
gap> k := GaloisField( 81 );
GF(3^4)
```

will assign the finite field of order 3^4 to the variable `k`.

In fact, what `GF` does is to set up a record which contains all necessary information, telling that it represents a finite field of degree 4 over its prime field with 3 elements. Of course, all arguments to GF others than those which represent a prime power are rejected – for obvious reasons.

Some of the more important entries of the field record are zero, one and root, which hold the corresponding elements of the field. All elements of a finite field are represented as a certain power of an appropriate primitive root, which is written as Z(q). As can be seen below the smallest possible primitive root is used.

```
gap> k.one + k.root + k.root^10 - k.zero;
Z(3^4)^52
gap> k.root;
Z(3^4)
gap> k.root ^ 20;
Z(3^2)^2
gap> k.one;
Z(3)^0
```

Note that of course elements from fields of different characteristic cannot be combined in operations.

```
gap> Z(3^2) * k.root + k.zero + Z(3^8);
Z(3^8)^6534
gap> Z(2) * k.one;
Error, Finite field *: operands must have the same characteristic
```

In this example we tried to multiply a primitive root of the field with two elements with the identity element of the field k. As the characteristic of k equals 3, there is no way to perform the multiplication. The first statement of the example shows, that if all the elements of the expression belong to fields of the same characteristic, the result will be computed.

As soon as a primitive root is demanded, GAP3 internally sets up all relevant data structures that are necessary to compute in the corresponding finite field. Each finite field is constructed as a splitting field of a Conway polynomial. These polynomials, as a set, have special properties that make it easy to embed smaller fields in larger ones and to convert the representation of the elements when doing so. All Conway polynomials for fields up to an order of 65536 have been computed and installed in the GAP3 kernel.

But now look at the following example.

```
gap> Z(3^3) * Z(3^4);
Error, Finite field *: smallest common superfield to large
```

Although both factors are elements of fields of characteristic 3, the product can not be evaluated by GAP3. The reason for this is very easy to explain:In order to compute the product, GAP3 has to find a field in which both of the factors lie. Here in our example the smallest field containing $Z(3^3)$ and $Z(3^4)$ is $GF(3^{12})$, the field with 531441 elements. As we have mentioned above that the size of finite fields in GAP3 is limited at present by 65536 we now see that there is no chance to set up the internal data structures for the common field to perform the computation.

As before with cyclotomic fields, the Galois group of a finite field and the norm and trace of its elements may be computed. The calling conventions are the same as for cyclotomic fields.

```
gap> Galk := GaloisGroup( k );
```

```
Group( FrobeniusAutomorphism( GF(3^4) ) )
gap> Size( Galk );
4
gap> IsCyclic( Galk );
true
gap> Norm( k, k.root ^ 20 );
Z(3)^0
gap> Trace( k, k.root ^ 20 );
0*Z(3)
```

So far, in our examples, we were always interested in the Galois group of a field extension k over its prime field. In fact it often will occur that, given a subfield l of k the Galois group of k over l is desired. In GAP3 it is possible to change the structure of a field by using the / operator. So typing

```
gap> l := GF(3^2);
GF(3^2)
gap> IsSubset( k, l );
true
gap> k / l;
GF(3^4)/GF(3^2)
```

changes the representation of k from a field extension of degree 4 over $GF(3)$ to a field given as an extension of degree 2 over $GF(3^2)$. The actual elements of the fields are still the same, only the structure of the field has changed.

```
gap> k = k / l;
true
gap> Galkl := GaloisGroup( k / l );
Group( FrobeniusAutomorphism( GF(3^4)/GF(3^2) )^2 )
gap> Size( Galkl );
2
```

Of course, all the relevant functions behave in a different way when they refer to `k / l` instead of `k`

```
gap> Norm( k / l, k.root ^ 20 );
Z(3)
gap> Trace( k / l, k.root ^ 20 );
Z(3^2)^6
```

This feature, to change the structure of the field without changing the underlying set of elements, is also available for cyclotomic fields, which we have seen at the beginning of this chapter.

```
gap> g := CyclotomicField( 4 );
GaussianRationals
gap> IsSubset( f, g );
true
gap> f / g;
CF(8)/GaussianRationals
gap> Galfg := GaloisGroup( f / g );
Group( NFAutomorphism( CF(8)/GaussianRationals , 5 ) )
```

```
gap> Size( Galfg );
2
```

The examples should have shown that, although the structure of finite fields and cyclotomic fields is rather different, there is a similar interface to them in GAP3, which makes it easy to write programs that deal with both types of fields in the same way.

1.22 About Matrix Groups

This section intends to show you the things you could do with matrix groups in GAP3. In principle all the set theoretic functions mentioned in chapter 4 and all group functions mentioned in chapter 7 can be applied to matrix groups. However, you should note that at present only very few functions can work efficiently with matrix groups. Especially infinite matrix groups (over the rationals or cyclotomic fields) can not be dealt with at all.

Matrix groups are created in the same way as the other types of groups, by using the function `Group`. Of course, in this case the arguments have to be invertable matrices over a field.

```
gap> m1 := [ [ Z(3)^0, Z(3)^0, Z(3) ],
>            [ Z(3), 0*Z(3), Z(3) ],
>            [ 0*Z(3), Z(3), 0*Z(3) ] ];;
gap> m2 := [ [ Z(3), Z(3), Z(3)^0 ],
>          [ Z(3), 0*Z(3), Z(3) ],
>          [ Z(3)^0, 0*Z(3), Z(3) ] ];;
gap> m := Group( m1, m2 );
Group( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ] )
```

As usual for groups, the matrix group that we have constructed is represented by a record with several entries. For matrix groups, there is one additional entry which holds the field over which the matrix group is written.

```
gap> m.field;
GF(3)
```

Note that you do not specify the field when you construct the group. `Group` automatically takes the smallest field over which all its arguments can be written.

At this point there is the question what special functions are available for matrix groups. The size of our group, for example, may be computed using the function `Size`.

```
gap> Size( m );
864
```

If we now compute the size of the corresponding general linear group

```
gap> (3^3 - 3^0) * (3^3 - 3^1) * (3^3 - 3^2);
11232
```

we see that we have constructed a proper subgroup of index 13 of $GL(3,3)$.

Let us now set up a subgroup of `m`, which is generated by the matrix `m2`.

```
gap> n := Subgroup( m, [ m2 ] );
```

```
Subgroup( Group( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ] ),
[ [ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
      [ Z(3)^0, 0*Z(3), Z(3) ] ] ] )
gap> Size( n );
6
```

And to round up this example we now compute the centralizer of this subgroup in `m`.

```
gap> c := Centralizer( m, n );
Subgroup( Group( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ] ),
[ [ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
      [ Z(3)^0, 0*Z(3), Z(3) ] ],
  [ [ Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3) ] ] ] )
gap> Size( c );
12
```

In this section you have seen that matrix groups are constructed in the same way that all groups are constructed. You have also been warned that only very few functions can work efficiently with matrix groups. See chapter 37 to read more about matrix groups.

1.23 About Domains and Categories

Domain is GAP3's name for structured sets. We already saw examples of domains in the previous sections. For example, the groups `s8` and `a8` in sections 1.18 and 1.19 are domains. Likewise the fields in section 1.21 are domains. **Categories** are sets of domains. For example, the set of all groups forms a category, as does the set of all fields.

In those sections we treated the domains as black boxes. They were constructed by special functions such as `Group` and `GaloisField`, and they could be passed as arguments to other functions such as `Size` and `Orbits`.

In this section we will also treat domains as black boxes. We will describe how domains are created in general and what functions are applicable to all domains. Next we will show how domains with the same structure are grouped into categories and will give an overview of the categories that are available. Then we will discuss how the organization of the GAP3 library around the concept of domains and categories is reflected in this manual. In a later section we will open the black boxes and give an overview of the mechanism that makes all this work (see 1.27).

The first thing you must know is how you can obtain domains. You have basically three possibilities. You can use the domains that are predefined in the library, you can create new domains with domain constructors, and you can use the domains returned by many library functions. We will now discuss those three possibilities in turn.

The GAP3 library predefines some domains. That means that there is a global variable whose value is this domain. The following example shows some of the more important predefined domains.

```
gap> Integers;
Integers    # the ring of all integers
gap> Size( Integers );
"infinity"
gap> GaussianRationals;
GaussianRationals    # the field of all Gaussian
gap> (1/2+E(4)) in GaussianRationals;
true    # E(4) is GAP3's name for the complex root of -1
gap> Permutations;
Permutations    # the domain of all permutations
```

Note that GAP3 prints those domains using the name of the global variable.

You can create new domains using **domain constructors** such as `Group`, `Field`, etc. A domain constructor is a function that takes a certain number of arguments and returns the domain described by those arguments. For example, `Group` takes an arbitrary number of group elements (of the same type) and returns the group generated by those elements.

```
gap> gf16 := GaloisField( 16 );
GF(2^4)    # the finite field with 16 elements
gap> Intersection( gf16, GaloisField( 64 ) );
GF(2^2)
gap> a5 := Group( (1,2,3), (3,4,5) );
Group( (1,2,3), (3,4,5) )    # the alternating group on 5 points
gap> Size( a5 );
60
```

Again GAP3 prints those domains using more or less the expression that you entered to obtain the domain.

As with groups (see 1.18) a name can be assigned to an arbitrary domain D with the assignment D`.name :=` *string*`;`, and GAP3 will use this name from then on in the output.

Many functions in the GAP3 library return domains. In the last example you already saw that `Intersection` returned a finite field domain. Below are more examples.

```
gap> GaloisGroup( gf16 );
Group( FrobeniusAutomorphism( GF(2^4) ) )
gap> SylowSubgroup( a5, 2 );
Subgroup( Group( (1,2,3), (3,4,5) ), [ (2,4)(3,5), (2,3)(4,5) ] )
```

The distinction between domain constructors and functions that return domains is a little bit arbitrary. It is also not important for the understanding of what follows. If you are nevertheless interested, here are the principal differences. A constructor performs no computation, while a function performs a more or less complicated computation. A constructor creates the representation of the domain, while a function relies on a constructor to create the domain. A constructor knows the dirty details of the domain's representation, while a function may be independent of the domain's representation. A constructor may appear as printed representation of a domain, while a function usually does not.

After showing how domains are created, we will now discuss what you can do with domains. You can assign a domain to a variable, put a domain into a list or into a record, pass a domain as argument to a function, and return a domain as result of a function. In this regard there is no difference between an integer value such as 17 and a domain such as

`Group( (1,2,3), (3,4,5) )`. Of course many functions will signal an error when you call them with domains as arguments. For example, `Gcd` does not accept two groups as arguments, because they lie in no Euclidean ring.

There are some functions that accept domains of any type as their arguments. Those functions are called the **set theoretic functions**. The full list of set theoretic functions is given in chapter 4.

Above we already used one of those functions, namely `Size`. If you look back you will see that we applied `Size` to the domain `Integers`, which is a ring, and the domain `a5`, which is a group. Remember that a domain was a structured set. The size of the domain is the number of elements in the set. `Size` returns this number or the string `"infinity"` if the domain is infinite. Below are more examples.

```
gap> Size( GaussianRationals );
"infinity"     # this string is returned for infinite domains
gap> Size( SylowSubgroup( a5, 2 ) );
4
```

`IsFinite(` *D* `)` returns `true` if the domain *D* is finite and `false` otherwise. You could also test if a domain is finite using `Size(` *D* `) < "infinity"` (GAP3 evaluates *n* `< "infinity"` to `true` for any number *n*). `IsFinite` is more efficient. For example, if *D* is a permutation group, `IsFinite(` *D* `)` can immediately return `true`, while `Size(` *D* `)` may take quite a while to compute the size of *D*.

The other function that you already saw is `Intersection`. Above we computed the intersection of the field with 16 elements and the field with 64 elements. The following example is similar.

```
gap> Intersection( a5, Group( (1,2), (1,2,3,4) ) );
Group( (2,3,4), (1,2)(3,4) )     # alternating group on 4 points
```

In general `Intersection` tries to return a domain. In general this is not possible however. Remember that a domain is a structured set. If the two domain arguments have different structure the intersection may not have any structure at all. In this case `Intersection` returns the result as a proper set, i.e., as a sorted list without holes and duplicates. The following example shows such a case. `ConjugacyClass` returns the conjugacy class of `(1,2,3,4,5)` in the alternating group on 6 points as a domain. If we intersect this class with the symmetric group on 5 points we obtain a proper set of 12 permutations, which is only one half of the conjugacy class of 5 cycles in `s5`.

```
gap> a6 := Group( (1,2,3), (2,3,4,5,6) );
Group( (1,2,3), (2,3,4,5,6) )
gap> class := ConjugacyClass( a6, (1,2,3,4,5) );
ConjugacyClass( Group( (1,2,3), (2,3,4,5,6) ), (1,2,3,4,5) )
gap> Size( class );
72
gap> s5 := Group( (1,2), (2,3,4,5) );
Group( (1,2), (2,3,4,5) )
gap> Intersection( class, s5 );
[ (1,2,3,4,5), (1,2,4,5,3), (1,2,5,3,4), (1,3,5,4,2), (1,3,2,5,4),
  (1,3,4,2,5), (1,4,3,5,2), (1,4,5,2,3), (1,4,2,3,5), (1,5,4,3,2),
  (1,5,2,4,3), (1,5,3,2,4) ]
```

You can intersect arbitrary domains as the following example shows.

```
gap> Intersection( Integers, a5 );
[  ]     # the empty set
```

Note that we optimized `Intersection` for typical cases, e.g., computing the intersection of two permutation groups, etc. The above computation is done with a very simple–minded method, all elements of `a5` are listed (with `Elements`, described below), and for each element `Intersection` tests whether it lies in `Integers` (with `in`, described below). So the same computation with the alternating group on 10 points instead of `a5` will probably exhaust your patience.

Just as `Intersection` returns a proper set occasionally, it also accepts proper sets as arguments. `Intersection` also takes an arbitrary number of arguments. And finally it also accepts a list of domains or sets to intersect as single argument.

```
gap> Intersection( a5, [ (1,2), (1,2,3), (1,2,3,4), (1,2,3,4,5) ] );
[ (1,2,3), (1,2,3,4,5) ]
gap> Intersection( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] );
[  ]
gap> Intersection( [ [1,2,4], [2,3,4], [1,3,4] ] );
[ 4 ]
```

The function `Union` is the obvious counterpart of `Intersection`. Note that `Union` usually does **not** return a domain. This is because the union of two domains, even of the same type, is usually not again a domain of that type. For example, the union of two subgroups is a subgroup if and only if one of the subgroups is a subset of the other. Of course this is exactly the reason why `Union` is less important than `Intersection` in algebra.

Because domains are structured sets there ought to be a membership test that tests whether an object lies in this domain or not. This is not implemented by a function, instead the operator `in` is used. *elm* `in` *D* returns `true` if the element *elm* lies in the domain *D* and `false` otherwise. We already used the `in` operator above when we tested whether `1/2 + E(4)` lies in the domain of Gaussian integers.

```
gap> (1,2,3) in a5;
true
gap> (1,2) in a5;
false
gap> (1,2,3,4,5,6,7) in a5;
false
gap> 17 in a5;
false    # of course an integer does not lie in a permutation group
gap> a5 in a5;
false
```

As you can see in the last example, `in` only implements the membership test. It does not allow you to test whether a domain is a subset of another domain. For such tests the function `IsSubset` is available.

```
gap> IsSubset( a5, a5 );
true
gap> IsSubset( a5, Group( (1,2,3) ) );
true
```

```
gap> IsSubset( Group( (1,2,3) ), a5 );
false
```

In the above example you can see that `IsSubset` tests whether the second argument is a subset of the first argument. As a general rule GAP3 library functions take as **first** arguments those arguments that are in some sense **larger** or more structured.

Suppose that you want to loop over all elements of a domain. For example, suppose that you want to compute the set of element orders of elements in the group `a5`. To use the `for` loop you need a list of elements in the domain D, because `for` *var* `in` D `do` *statements* `od` will not work. The function `Elements` does exactly that. It takes a domain D and returns the proper set of elements of D.

```
gap> Elements( Group( (1,2,3), (2,3,4) ) );
[ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
  (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ]
gap> ords := [];;
gap> for elm  in Elements( a5 )  do
>        Add( ords, Order( a5, elm ) );
>    od;
gap> Set( ords );
[ 1, 2, 3, 5 ]
gap> Set( List( Elements( a5 ), elm -> Order( a5, elm ) ) );
[ 1, 2, 3, 5 ]      # an easier way to compute the set of orders
```

Of course, if you apply `Elements` to an infinite domain, `Elements` will signal an error. It is also not a good idea to apply `Elements` to very large domains because the list of elements will take much space and computing this large list will probably exhaust your patience.

```
gap> Elements( GaussianIntegers );
Error, the ring <R> must be finite to compute its elements in
D.operations.Elements( D ) called from
Elements( GaussianIntegers ) called from
main loop
brk> quit;
```

There are a few more set theoretic functions. See chapter 4 for a complete list.

All the set theoretic functions treat the domains as if they had no structure. Now a domain is a structured set (excuse us for repeating this again and again, but it is really important to get this across). If the functions ignore the structure than they are effectively viewing a domain only as the set of elements.

In fact all set theoretic functions also accept proper sets, i.e., sorted lists without holes and duplicates as arguments (we already mentioned this for `Intersection`). Also set theoretic functions may occasionally return proper sets instead of domains as result.

This equivalence of a domain and its set of elements is particularly important for the definition of equality of domains. Two domains D and E are equal (in the sense that D `=` E evaluates to `true`) if and only if the set of elements of D is equal to the set of elements of E (as returned by `Elements(` D `)` and `Elements(` E `)`). As a special case either of the operands of `=` may also be a proper set, and the value is `true` if this set is equal to the set of elements of the domain.

```
gap> a4 := Group( (1,2,3), (2,3,4) );
```

```
Group( (1,2,3), (2,3,4) )
gap> elms := Elements( a4 );
[ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
  (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ]
gap> elms = a4;
true
```

However the following example shows that this does not imply that all functions return the same answer for two domains (or a domain and a proper set) that are equal. This is because those function may take the structure into account.

```
gap> IsGroup( a4 );
true
gap> IsGroup( elms );
false
gap> Intersection( a4, Group( (1,2), (1,2,3) ) );
Group( (1,2,3) )
gap> Intersection( elms, Group( (1,2), (1,2,3) ) );
[ (), (1,2,3), (1,3,2) ]      # this is not a group
gap> last = last2;
true                          # but it is equal to the above result
gap> Centre( a4 );
Subgroup( Group( (1,2,3), (2,3,4) ), [  ] )
gap> Centre( elms );
Error, <struct> must be a record in
Centre( elms ) called from
main loop
brk> quit;
```

Generally three things may happen if you have two domains D and E that are equal but have different structure (or a domain D and a set E that are equal). First a function that tests whether a domain has a certain structure may return `true` for D and `false` for E. Second a function may return a domain for D and a proper set for E. Third a function may work for D and fail for E, because it requires the structure.

A slightly more complex example for the second case is the following.

```
gap> v4 := Subgroup( a4, [ (1,2)(3,4), (1,3)(2,4) ] );
Subgroup( Group( (1,2,3), (2,3,4) ), [ (1,2)(3,4), (1,3)(2,4) ] )
gap> v4.name := "v4";;
gap> rc := v4 * (1,2,3);
(v4*(2,4,3))
gap> lc := (1,2,3) * v4;
((1,2,3)*v4)
gap> rc = lc;
true
gap> rc * (1,3,2);
(v4*())
gap> lc * (1,3,2);
[ (1,3)(2,4), (), (1,2)(3,4), (1,4)(2,3) ]
gap> last = last2;
```

```
false
```

The two domains `rc` and `lc` (yes, cosets are domains too) are equal, because they have the same set of elements. However if we multiply both with `(1,3,2)` we obtain the trivial right coset for `rc` and a list for `lc`. The result for `lc` is **not** a proper set, because it is not sorted, therefore `=` evaluates to `false`. (For the curious. The multiplication of a left coset with an element from the right will generally not yield another coset, i.e., nothing that can easily be represented as a domain. Thus to multiply `lc` with `(1,3,2)` GAP3 first converts `lc` to the set of its elements with `Elements`. But the definition of multiplication requires that a list l multiplied by an element e yields a new list n such that each element n`[`i`]` in the new list is the product of the element l`[`i`]` at the **same position** of the operand list l with e.)

Note that the above definition only defines **what** the result of the equality comparison of two domains D and E should be. It does not prescribe that this comparison is actually performed by listing all elements of D and E. For example, if D and E are groups, it is sufficient to check that all generators of D lie in E and that all generators of E lie in D. If GAP3 would really compute the whole set of elements, the following test could not be performed on any computer.

```
gap> Group( (1,2), (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18) )
> = Group( (17,18), (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18) );
true
```

If we could only apply the set theoretic functions to domains, domains would be of little use. Luckily this is not so. We already saw that we could apply `GaloisGroup` to the finite field with 16 elements, and `SylowSubgroup` to the group `a5`. But those functions are not applicable to all domains. The argument of `GaloisGroup` must be a field, and the argument of `SylowSubgroup` must be a group.

A **category** is a set of domains. So we say that the argument of `GaloisGroup` must be an element of the category of fields, and the argument of `SylowSubgroup` must be an element of the category of groups. The most important categories are **rings**, **fields**, **groups**, and **vector spaces**. Which category a domain belongs to determines which functions are applicable to this domain and its elements. We want to emphasize the each domain belongs to **one and only one** category. This is necessary because domains in different categories have, sometimes incompatible, representations.

Note that the categories only exist conceptually. That means that there is no GAP3 object for the categories, e.g., there is no object `Groups`. For each category there exists a function that tests whether a domain is an element of this category.

```
gap> IsRing( gf16 );
false
gap> IsField( gf16 );
true
gap> IsGroup( gf16 );
false
gap> IsVectorSpace( gf16 );
false
```

Note that of course mathematically the field `gf16` is also a ring and a vector space. However in GAP3 a domain can only belong to one category. So a domain is conceptually a set of elements with **one** structure, e.g., a field structure. That the same set of elements may also support a different structure, e.g., a ring or vector space structure, can not be represented by this domain. So you need a different domain to represent this different structure. (We are planning to add functions that changes the structure of a domain, e.g. `AsRing(` *field* `)` should return a new domain with the same elements as *field* but with a ring structure.)

Domains may have certain properties. For example a ring may be commutative and a group may be nilpotent. Whether a domain has a certain property *Property* can be tested with the function `Is`*Property*.

```
gap> IsCommutativeRing( GaussianIntegers );
true
gap> IsNilpotent( a5 );
false
```

There are also similar functions that test whether a domain (especially a group) is represented in a certain way. For example `IsPermGroup` tests whether a group is represented as a permutation group.

```
gap> IsPermGroup( a5 );
true
gap> IsPermGroup( a4 / v4 );
false    # a4 / v4 is represented as a generic factor group
```

There is a slight difference between a function such as `IsNilpotent` and a function such as `IsPermGroup`. The former tests properties of an **abstract** group and its outcome is independent of the representation of that group. The latter tests whether a group is given in a certain representation.

This (rather philosophical) issue is further complicated by the fact that sometimes representations and properties are not independent. This is especially subtle with `IsSolvable` (see 7.61) and `IsAgGroup` (see 25.26). `IsSolvable` tests whether a group G is solvable. `IsAgGroup` tests whether a group G is represented as a finite polycyclic group, i.e., by a finite presentation that allows to efficiently compute canonical normal forms of elements (see 25). Of course every finite polycyclic group is solvable, so `IsAgGroup(` G `)` implies `IsSolvable(` G `)`. On the other hand `IsSolvable(` G `)` does not imply `IsAgGroup(` G `)`, because, even though each solvable group **can** be represented as a finite polycyclic group, it **need** not, e.g., it could also be represented as a permutation group.

The organization of the manual follows the structure of domains and categories.

After the description of the programming language and the environment chapter 4 describes the domains and the functions applicable to all domains.

Next come the chapters that describe the categories rings, fields, groups, and vector spaces.

The remaining chapters describe GAP3's data–types and the domains one can make with those elements of those data-types. The order of those chapters roughly follows the order of the categories. The data–types whose elements form rings and fields come first (e.g., integers and finite fields), followed by those whose elements form groups (e.g., permutations), and so on. The data–types whose elements support little or no algebraic structure come last (e.g., booleans). In some cases there may be two chapters for one data–type, one describing the elements and the other describing the domains made with those elements (e.g., permutations and permutation groups).

The GAP3 manual not only describes what you can do, it also gives some hints how GAP3 performs its computations. However, it can be tricky to find those hints. The index of this manual can help you.

Suppose that you want to intersect two permutation groups. If you read the section that describes the function `Intersection` (see 4.12) you will see that the last paragraph describes the default method used by `Intersection`. Such a last paragraph that describes the default method is rather typical. In this case it says that `Intersection` computes the proper set of elements of both domains and intersect them. It also says that this method is often overlaid with a more efficient one. You wonder whether this is the case for permutation groups. How can you find out? Well you look in the index under **Intersection**. There you will find a reference **Intersection, for permutation groups** to section **Set Functions for Permutation Groups** (see 21.20). This section tells you that `Intersection` uses a backtrack for permutation groups (and cites a book where you can find a description of the backtrack).

Let us now suppose that you intersect two factor groups. There is no reference in the index for **Intersection, for factor groups**. But there is a reference for **Intersection, for groups** to the section **Set Functions for Groups** (see 7.114). Since this is the next best thing, look there. This section further directs you to the section **Intersection for Groups** (see 7.116). This section finally tells you that `Intersection` computes the intersection of two groups G and H as the stabilizer in G of the trivial coset of H under the operation of G on the right cosets of H.

In this section we introduced domains and categories. You have learned that a domain is a structured set, and that domains are either predefined, created by domain constructors, or returned by library functions. You have seen most functions that are applicable to all domains. Those functions generally ignore the structure and treat a domain as the set of its elements. You have learned that categories are sets of domains, and that the category a domain belongs to determines which functions are applicable to this domain.

More information about domains can be found in chapter 4. Chapters 5, 6, 7, and 9 define the categories known to GAP3. The section 1.27 opens that black boxes and shows how all this works.

1.24 About Mappings and Homomorphisms

A mapping is an object which maps each element of its source to a value in its range. Source and range can be arbitrary sets of elements. But in most applications the source and range are structured sets and the mapping, in such applications called homomorphism, is compatible with this structure.

In the last sections you have already encountered examples of homomorphisms, namely natural homomorphisms of groups onto their factor groups and operation homomorphisms of groups into symmetric groups.

Finite fields also bear a structure and homomorphisms between fields are always bijections. The Galois group of a finite field is generated by the Frobenius automorphism. It is very easy to construct.

```
gap> f := FrobeniusAutomorphism( GF(81) );
FrobeniusAutomorphism( GF(3^4) )
```

```
gap> Image( f, Z(3^4) );
Z(3^4)^3
gap> A := Group( f );
Group( FrobeniusAutomorphism( GF(3^4) ) )
gap> Size( A );
4
gap> IsCyclic( A );
true
gap> Order( Mappings, f );
4
gap> Kernel( f );
[ 0*Z(3) ]
```

For finite fields and cyclotomic fields the function `GaloisGroup` is an easy way to construct the Galois group.

```
gap> GaloisGroup( GF(81) );
Group( FrobeniusAutomorphism( GF(3^4) ) )
gap> Size( last );
4
gap> GaloisGroup( CyclotomicField( 18 ) );
Group( NFAutomorphism( CF(9) , 2 ) )
gap> Size( last );
6
```

Not all group homomorphisms are bijections of course, natural homomorphisms do have a kernel in most cases and operation homomorphisms need neither be surjective nor injective.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s4.name := "s4";;
gap> v4 := Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] );
Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] )
gap> v4.name := "v4";;
gap> s3 := s4 / v4;
(s4 / v4)
gap> f := NaturalHomomorphism( s4, s3 );
NaturalHomomorphism( s4, (s4 / v4) )
gap> IsHomomorphism( f );
true
gap> IsEpimorphism( f );
true
gap> Image( f );
(s4 / v4)
gap> IsMonomorphism( f );
false
gap> Kernel( f );
v4
```

The image of a group homomorphism is always one element of the range but the preimage can be a coset. In order to get one representative of this coset you can use the function `PreImagesRepresentative`.

```
gap> Image( f, (1,2,3,4) );
FactorGroupElement( v4, (2,4) )
gap> PreImages( f, s3.generators[1] );
(v4*(2,4))
gap> PreImagesRepresentative( f, s3.generators[1] );
(2,4)
```

But even if the homomorphism is a monomorphism but not surjective you can use the function `PreImagesRepresentative` in order to get the preimage of an element of the range.

```
gap> A := Z(3) * [ [ 0, 1 ], [ 1, 0 ] ];;
gap> B := Z(3) * [ [ 0, 1 ], [ -1, 0 ] ];;
gap> G := Group( A, B );
Group( [ [ 0*Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ],
[ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ] )
gap> Size( G );
8
gap> G.name := "G";;
gap> d8 := Operation( G, Orbit( G, Z(3)*[1,0] ) );
Group( (1,2)(3,4), (1,2,3,4) )
gap> e := OperationHomomorphism( Subgroup( G, [B] ), d8 );
OperationHomomorphism( Subgroup( G,
[ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ] ] ), Group( (1,2)(3,4),
(1,2,3,4) ) )
gap> Kernel( e );
Subgroup( G, [  ] )
gap> IsSurjective( e );
false
gap> PreImages( e, (1,3)(2,4) );
(Subgroup( G, [  ] )*[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ])
gap> PreImage( e, (1,3)(2,4) );
Error, <bij> must be a bijection, not an arbitrary mapping in
bij.operations.PreImageElm( bij, img ) called from
PreImage( e, (1,3)(2,4) ) called from
main loop
brk> quit;
gap> PreImagesRepresentative( e, (1,3)(2,4) );
[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ]
```

Only bijections allow `PreImage` in order to get the preimage of an element of the range.

```
gap> Operation( G, Orbit( G, Z(3)*[1,0] ) );
Group( (1,2)(3,4), (1,2,3,4) )
gap> d := OperationHomomorphism( G, last );
OperationHomomorphism( G, Group( (1,2)(3,4), (1,2,3,4) ) )
gap> PreImage( d, (1,3)(2,4) );
[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ]
```

Both `PreImage` and `PreImages` can also be applied to sets. They return the complete preimage.

```
gap> PreImages( d, Group( (1,2)(3,4), (1,3)(2,4) ) );
```

```
Subgroup( G, [ [ [ 0*Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ],
  [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ] )
gap> Size( last );
4
gap> f := NaturalHomomorphism( s4, s3 );
NaturalHomomorphism( s4, (s4 / v4) )
gap> PreImages( f, s3 );
Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4), (2,4), (3,4) ] )
gap> Size( last );
24
```

Another way to construct a group automorphism is to use elements in the normalizer of a subgroup and construct the induced automorphism. A special case is the inner automorphism induced by an element of a group, a more general case is a surjective homomorphism induced by arbitrary elements of the parent group.

```
gap> d12 := Group((1,2,3,4,5,6),(2,6)(3,5));;  d12.name := "d12";;
gap> i1 := InnerAutomorphism( d12, (1,2,3,4,5,6) );
InnerAutomorphism( d12, (1,2,3,4,5,6) )
gap> Image( i1, (2,6)(3,5) );
(1,3)(4,6)
gap> IsAutomorphism( i1 );
true
```

Mappings can also be multiplied, provided that the range of the first mapping is a subgroup of the source of the second mapping. The multiplication is of course defined as the composition. Note that, in line with the fact that mappings operate **from the right**, `Image(` *map1* `*` *map2*`,` *elm* `)` is defined as `Image(` *map2*`, Image(` *map1*`,` *elm* `) )`.

```
gap> i2 := InnerAutomorphism( d12, (2,6)(3,5) );
InnerAutomorphism( d12, (2,6)(3,5) )
gap> i1 * i2;
InnerAutomorphism( d12, (1,6)(2,5)(3,4) )
gap> Image( last, (2,6)(3,5) );
(1,5)(2,4)
```

Mappings can also be inverted, provided that they are bijections.

```
gap> i1 ^ -1;
InnerAutomorphism( d12, (1,6,5,4,3,2) )
gap> Image( last, (2,6)(3,5) );
(1,5)(2,4)
```

Whenever you have a set of bijective mappings on a finite set (or domain) you can construct the group generated by those mappings. So in the following example we create the group of inner automorphisms of `d12`.

```
gap> autd12 := Group( i1, i2 );
Group( InnerAutomorphism( d12,
(1,2,3,4,5,6) ), InnerAutomorphism( d12, (2,6)(3,5) ) )
gap> Size( autd12 );
6
gap> Index( d12, Centre( d12 ) );
```

```
6
```

Note that the computation with such automorphism groups in their present implementation is not very efficient. For example to compute the size of such an automorphism group all elements are computed. Thus work with such automorphism groups should be restricted to very small examples.

The function `ConjugationGroupHomomorphism` is a generalization of `InnerAutomorphism`. It accepts a source and a range and an element that conjugates the source into the range. Source and range must lie in a common parent group, and the conjugating element must also lie in this parent group.

```
gap> c2 := Subgroup( d12, [ (2,6)(3,5) ] );
Subgroup( d12, [ (2,6)(3,5) ] )
gap> v4 := Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6) ] );
Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6) ] )
gap> x := ConjugationGroupHomomorphism( c2, v4, (1,3,5)(2,4,6) );
ConjugationGroupHomomorphism( Subgroup( d12,
[ (2,6)(3,5) ] ), Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6)
 ] ), (1,3,5)(2,4,6) )
gap> IsSurjective( x );
false
gap> Image( x );
Subgroup( d12, [ (1,5)(2,4) ] )
```

But how can we construct homomorphisms which are not induced by elements of the parent group? The most general way to construct a group homomorphism is to define the source, range and the images of the generators under the homomorphism in mind.

```
gap> c := GroupHomomorphismByImages( G, s4, [A,B], [(1,2),(3,4)] );
GroupHomomorphismByImages( G, s4,
[ [ [ 0*Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ],
  [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ] ], [ (1,2), (3,4) ] )
gap> Kernel( c );
Subgroup( G, [ [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ] )
gap> Image( c );
Subgroup( s4, [ (1,2), (3,4) ] )
gap> IsHomomorphism( c );
true
gap> Image( c, A );
(1,2)
gap> PreImages( c, (1,2) );
(Subgroup( G, [ [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ] )*
[ [ 0*Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ])
```

Note that it is possible to construct a general mapping this way that is not a homomorphism, because `GroupHomomorphismByImages` does not check if the given images fulfill the relations of the generators.

```
gap> b := GroupHomomorphismByImages( G, s4, [A,B], [(1,2,3),(3,4)] );
GroupHomomorphismByImages( G, s4,
[ [ [ 0*Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ],
```

```
  [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ] ], [ (1,2,3), (3,4) ] )
gap> IsHomomorphism( b );
false
gap> Images( b, A );
(Subgroup( s4, [ (1,3,2), (2,3,4), (1,3,4), (1,4)(2,3), (1,4,2)
 ] )*())
```

The result is a **multi valued mapping**, i.e., one that maps each element of its source to a set of elements in its range. The set of images of `A` under `b` is defined as follows. Take all the words of two letters $w(x,y)$ such that $w(A,B) = A$, e.g., x and $xyxyx$. Then the set of images is the set of elements that you get by inserting the images of `A` and `B` in those words, i.e., $w((1,2,3),(3,4))$, e.g., $(1,2,3)$ and $(1,4,2)$. One can show that the set of images of the identity under a multi valued mapping such as `b` is a subgroup and that the set of images of other elements are cosets of this subgroup.

1.25 About Character Tables

This section contains some examples of the use of GAP3 in character theory. First a few very simple commands for handling character tables are introduced, and afterwards we will construct the character tables of $(A_5 \times 3){:}2$ and of $A_6.2^2$.

GAP3 has a large library of character tables, so let us look at one of these tables, e.g., the table of the Mathieu group M_{11}:

```
gap> m11:= CharTable( "M11" );
CharTable( "M11" )
```

Character tables contain a lot of information. This is not printed in full length since the internal structure is not easy to read. The next statement shows a more comfortable output format.

```
gap> DisplayCharTable( m11 );
M11

      2  4  4  1  3  .  1  3  3   .   .
      3  2  1  2  .  .  1  .  .   .   .
      5  1  .  .  .  1  .  .  .   .   .
     11  1  .  .  .  .  .  .  .   1   1

        1a 2a 3a 4a 5a 6a 8a 8b 11a 11b
     2P 1a 1a 3a 2a 5a 3a 4a 4a 11b 11a
     3P 1a 2a 1a 4a 5a 2a 8a 8b 11a 11b
     5P 1a 2a 3a 4a 1a 6a 8b 8a 11a 11b
    11P 1a 2a 3a 4a 5a 6a 8a 8b  1a  1a

X.1      1  1  1  1  1  1  1  1   1   1
X.2     10  2  1  2  . -1  .  .  -1  -1
X.3     10 -2  1  .  .  1  A -A  -1  -1
X.4     10 -2  1  .  .  1 -A  A  -1  -1
X.5     11  3  2 -1  1  . -1 -1   .   .
X.6     16  . -2  .  1  .  .  .   B  /B
```

```
X.7      16  . -2  .  1  .  .  .  /B   B
X.8      44  4 -1  . -1  1  .  .   .   .
X.9      45 -3  .  1  .  . -1 -1   1   1
X.10     55 -1  1 -1  . -1  1  1   .   .

A = E(8)+E(8)^3
  = ER(-2) = i2
B = E(11)+E(11)^3+E(11)^4+E(11)^5+E(11)^9
  = (-1+ER(-11))/2 = b11
```

We are not too much interested in the internal structure of this character table (see 49.2); but of course we can access all information about the centralizer orders (first four lines), element orders (next line), power maps for the prime divisors of the group order (next four lines), irreducible characters (lines parametrized by `X.1` ... `X.10`) and irrational character values (last four lines), see 49.37 for a detailed description of the format of the displayed table. E.g., the irreducible characters are a list with name `m11.irreducibles`, and each character is a list of cyclotomic integers (see chapter 13). There are various ways to describe the irrationalities; e.g., the square root of -2 can be entered as `E(8) + E(8)^3` or `ER(-2)`, the famous ATLAS of Finite Groups [CCN+85] denotes it as `i2`.

```
gap> m11.irreducibles[3];
[ 10, -2, 1, 0, 0, 1, E(8)+E(8)^3, -E(8)-E(8)^3, -1, -1 ]
```

We can for instance form tensor products of this character with all irreducibles, and compute the decomposition into irreducibles.

```
gap> tens:= Tensored( [ last ], m11.irreducibles );;
gap> MatScalarProducts( m11, m11.irreducibles, tens );
[ [ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 ],
  [ 0, 0, 0, 0, 1, 0, 0, 1, 1, 0 ], [ 1, 0, 0, 0, 0, 0, 0, 1, 0, 1 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 1, 1 ], [ 0, 0, 0, 0, 0, 0, 1, 1, 1, 1 ],
  [ 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 ], [ 0, 0, 1, 1, 0, 1, 1, 2, 3, 3 ],
  [ 0, 1, 0, 1, 1, 1, 1, 3, 2, 3 ], [ 0, 1, 1, 0, 1, 1, 1, 3, 3, 4 ] ]
```

The decomposition means for example that the third character in the list `tens` is the sum of the irreducible characters at positions 5, 8 and 9.

```
gap> tens[3];
[ 100, 4, 1, 0, 0, 1, -2, -2, 1, 1 ]
gap> tens[3] = Sum( Sublist( m11.irreducibles, [ 5, 8, 9 ] ) );
true
```

Or we can compute symmetrizations, e.g., the characters χ^{2+}, defined by $\chi^{2+}(g) = \frac{1}{2}(\chi^2(g) + \chi(g^2))$, for all irreducibles.

```
gap> sym:= SymmetricParts( m11, m11.irreducibles, 2 );;
gap> MatScalarProducts( m11, m11.irreducibles, sym );
[ [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 1, 1, 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 ],
  [ 1, 1, 0, 0, 1, 0, 0, 1, 0, 0 ], [ 0, 1, 0, 0, 1, 0, 1, 1, 0, 1 ],
  [ 0, 1, 0, 0, 1, 1, 0, 1, 0, 1 ], [ 1, 3, 0, 0, 3, 2, 2, 8, 4, 6 ],
  [ 1, 2, 0, 0, 3, 2, 2, 8, 4, 7 ],
  [ 1, 3, 1, 1, 4, 3, 3, 11, 7, 10 ] ]
```

```
gap> sym[2];
[ 55, 7, 1, 3, 0, 1, 1, 1, 0, 0 ]
gap> sym[2] = Sum( Sublist( m11.irreducibles, [ 1, 2, 8 ] ) );
true
```

If the subgroup fusion into a supergroup is known, characters can be induced to this group, e.g., to obtain the permutation character of the action of M_{12} on the cosets of M_{11}.

```
gap> m12:= CharTable( "M12" );;
gap> permchar:= Induced( m11, m12, [ m11.irreducibles[1] ] );
[ [ 12, 0, 4, 3, 0, 0, 4, 2, 0, 1, 0, 2, 0, 1, 1 ] ]
gap> MatScalarProducts( m12, m12.irreducibles, last );
[ [ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
gap> DisplayCharTable( m12, rec( chars:= permchar ) );
M12

      2  6  4  6  1  2  5  5  1  2  1  3  3   1   .   .
      3  3  1  1  3  2  .  .  .  1  1  .  .   .   .   .
      5  1  1  .  .  .  .  .  1  .  .  .  .   1   .   .
     11  1  .  .  .  .  .  .  .  .  .  .  .   .   1   1

        1a 2a 2b 3a 3b 4a 4b 5a 6a 6b 8a 8b 10a 11a 11b
     2P 1a 1a 1a 3a 3b 2b 2b 5a 3b 3a 4a 4b  5a 11b 11a
     3P 1a 2a 2b 1a 1a 4a 4b 5a 2a 2b 8a 8b 10a 11a 11b
     5P 1a 2a 2b 3a 3b 4a 4b 1a 6a 6b 8a 8b  2a 11a 11b
    11P 1a 2a 2b 3a 3b 4a 4b 5a 6a 6b 8a 8b 10a  1a  1a

Y.1     12  .  4  3  .  .  4  2  .  1  .  2   .   1   1
```

It should be emphasized that the heart of character theory is dealing with lists. Characters are lists, and also the maps which occur are represented as lists. Note that the multiplication of group elements is not available, so we neither have homomorphisms. All we can talk of are class functions, and the lists are regarded as such functions, being the lists of images with respect to a fixed order of conjugacy classes. Therefore we do not write `chi( cl )` or `cl^chi` for the value of the character `chi` on the class `cl`, but `chi[i]` where `i` is the position of the class `cl`.

Since the data structures are so basic, most calculations involve compositions of maps; for example, the embedding of a subgroup in a group is described by the so–called subgroup fusion which is a class function that maps each class c of the subgroup to that class of the group that contains c. Consider the symmetric group $S_5 \cong A_5.2$ as subgroup of M_{11}. (Do not worry about the names that are used to get library tables, see 49.12 for an overview.)

```
gap> s5:= CharTable( "A5.2" );;
gap> map:= GetFusionMap( s5, m11 );
[ 1, 2, 3, 5, 2, 4, 6 ]
```

The subgroup fusion is already stored on the table. We see that class 1 of `s5` is mapped to class 1 of `m11` (which means that the identity of S_5 maps to the identity of M_{11}), classes 2 and 5 of `s5` both map to class 2 of `m11` (which means that all involutions of S_5 are conjugate in M_{11}), and so on.

The restriction of a character of `m11` to `s5` is just the composition of this character with the subgroup fusion map. Viewing this map as list one would call this composition an indirection.

```
gap> chi:= m11.irreducibles[3];
[ 10, -2, 1, 0, 0, 1, E(8)+E(8)^3, -E(8)-E(8)^3, -1, -1 ]
gap> rest:= List( map, x -> chi[x] );
[ 10, -2, 1, 0, -2, 0, 1 ]
```

This looks very easy, and many GAP3 functions in character theory do such simple calculations. But note that it is not always obvious that a list is regarded as a map, where preimages and/or images refer to positions of certain conjugacy classes.

```
gap> alt:= s5.irreducibles[2];
[ 1, 1, 1, 1, -1, -1, -1 ]
gap> kernel:= KernelChar( last );
[ 1, 2, 3, 4 ]
```

The kernel of a character is represented as the list of (positions of) classes lying in the kernel. We know that the kernel of the alternating character `alt` of `s5` is the alternating group A_5. The order of the kernel can be computed as sum of the lengths of the contained classes from the character table, using that the classlengths are stored in the `classes` component of the table.

```
gap> s5.classes;
[ 1, 15, 20, 24, 10, 30, 20 ]
gap> last{ kernel };
[ 1, 15, 20, 24 ]
gap> Sum( last );
60
```

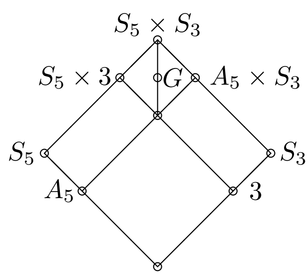
We chose those classlengths of `s5` that belong to the S_5–classes contained in the alternating group. The same thing is done in the following command, reflecting the view of the kernel as map.

```
gap> List( kernel, x -> s5.classes[x] );
[ 1, 15, 20, 24 ]
gap> Sum( kernel, x -> s5.classes[x] );
60
```

This small example shows how the functions `List` and `Sum` can be used. These functions as well as `Filtered` were introduced in 1.16, and we will make heavy use of them; in many cases such a command might look very strange, but it is just the translation of a (hardly less complicated) mathematical formula to character theory.

And now let us construct some small character tables!

The group $G = (A_5 \times 3):2$ is a maximal subgroup of the alternating group A_8; G extends to $S_5 \times S_3$ in S_8. We want to construct the character table of G.



First the tables of the subgroup $A_5 \times 3$ and the supergroup $S_5 \times S_3$ are constructed; the tables of the factors of each direct product are again got from the table library using admissible names, see 49.12 for this.

```
gap> a5:= CharTable( "A5" );;
gap> c3:= CharTable( "Cyclic", 3 );;
gap> a5xc3:= CharTableDirectProduct( a5, c3 );;
gap> s5:= CharTable( "A5.2" );;
gap> s3:= CharTable( "Symmetric", 3 );;
gap> s3.irreducibles;
[ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ] ]
# The trivial character shall be the first one.
gap> SortCharactersCharTable( s3 );  # returns the applied permutation
(1,2,3)
gap> s5xs3:= CharTableDirectProduct( s5, s3 );;
```

G is the normal subgroup of index 2 in $S_5 \times S_3$ which contains neither S_5 nor the normal S_3. We want to find the classes of `s5xs3` whose union is G. For that, we compute the set of kernels of irreducibles –remember that they are given simply by lists of numbers of contained classes– and then choose those kernels belonging to normal subgroups of index 2.

```
gap> kernels:= Set( List( s5xs3.irreducibles, KernelChar ) );
[ [ 1 ], [ 1, 2, 3 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
      19, 20, 21 ], [ 1, 3 ],
  [ 1, 3, 4, 6, 7, 9, 10, 12, 13, 15, 16, 18, 19, 21 ],
  [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ], [ 1, 4, 7, 10 ],
  [ 1, 4, 7, 10, 13, 16, 19 ] ]
gap> sizes:= List( kernels, x -> Sum( Sublist( s5xs3.classes, x ) ) );
[ 1, 6, 360, 720, 3, 360, 360, 60, 120 ]
gap> s5xs3.size;
720
gap> index2:= Sublist( kernels, [ 3, 6, 7 ] );
[ [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
  [ 1, 3, 4, 6, 7, 9, 10, 12, 13, 15, 16, 18, 19, 21 ],
  [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ] ]
```

In order to decide which kernel describes G, we consider the embeddings of `s5` and `s3` in `s5xs3`, given by the subgroup fusions.

```
gap> s5ins5xs3:= GetFusionMap( s5, s5xs3 );
[ 1, 4, 7, 10, 13, 16, 19 ]
gap> s3ins5xs3:= GetFusionMap( s3, s5xs3 );
[ 1, 2, 3 ]
gap> Filtered( index2, x->Intersection(x,s5ins5xs3)<>s5ins5xs3 and
>                         Intersection(x,s3ins5xs3)<>s3ins5xs3     );
[ [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ] ]
gap> nsg:= last[1];
[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ]
```

We now construct a first approximation of the character table of this normal subgroup, namely the restriction of `s5xs3` to the classes given by `nsg`.

```
gap> sub:= CharTableNormalSubgroup( s5xs3, nsg );;
#I CharTableNormalSubgroup: classes in [ 8 ] necessarily split
gap> PrintCharTable( sub );
```

```
rec( identifier := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 2\
0 ])", size :=
360, name := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ])",\
 order := 360, centralizers := [ 360, 180, 24, 12, 18, 9, 15, 15/2,
  12, 4, 6 ], orders := [ 1, 3, 2, 6, 3, 3, 5, 15, 2, 4, 6
 ], powermap := [ , [ 1, 2, 1, 2, 5, 6, 7, 8, 1, 3, 5 ],
  [ 1, 1, 3, 3, 1, 1, 7, 7, 9, 10, 9 ],,
  [ 1, 2, 3, 4, 5, 6, 1, 2, 9, 10, 11 ] ], classes :=
[ 1, 2, 15, 30, 20, 40, 24, 48, 30, 90, 60
 ], operations := CharTableOps, irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 2, -1, 2, -1, 2, -1, 2, -1, 0, 0, 0 ],
  [ 6, 6, -2, -2, 0, 0, 1, 1, 0, 0, 0 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, 2, 0, -1 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, -2, 0, 1 ],
  [ 8, -4, 0, 0, 2, -1, -2, 1, 0, 0, 0 ],
  [ 5, 5, 1, 1, -1, -1, 0, 0, 1, -1, 1 ],
  [ 5, 5, 1, 1, -1, -1, 0, 0, -1, 1, -1 ],
  [ 10, -5, 2, -1, -2, 1, 0, 0, 0, 0, 0 ] ], fusions := [ rec(
      name := [ 'A', '5', '.', '2', 'x', 'S', '3' ],
      map := [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ] ) ] )
```

Not all restrictions of irreducible characters of `s5xs3` to `sub` remain irreducible. We compute those restrictions with norm larger than 1.

```
gap> red:= Filtered( Restricted( s5xs3, sub, s5xs3.irreducibles ),
>                    x -> ScalarProduct( sub, x, x ) > 1 );
[ [ 12, -6, -4, 2, 0, 0, 2, -1, 0, 0, 0 ] ]
gap> Filtered( [ 1 .. Length( nsg ) ],
>              x -> not IsInt( sub.centralizers[x] ) );
[ 8 ]
```

Note that `sub` is not actually a character table in the sense of mathematics but only a record with components like a character table. GAP3 does not know about this subtleties and treats it as a character table.

As the list `centralizers` of centralizer orders shows, at least class 8 splits into two conjugacy classes in G, since this is the only possibility to achieve integral centralizer orders.

Since 10 restrictions of irreducible characters remain irreducible for G (`sub` contains 10 irreducibles), only one of the 11 irreducibles of $S_5 \times S_3$ splits into two irreducibles of G, in other words, class 8 is the only splitting class.

Thus we create a new approximation of the desired character table (which we call `split`) where this class is split; 8th and 9th column of the known irreducibles are of course equal, and due to the splitting the second powermap for these columns is ambiguous.

```
gap> splitting:= [ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ];;
gap> split:= CharTableSplitClasses( sub, splitting );;
gap> PrintCharTable( split );
rec( identifier := "Split(Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14,\
```

```
 17, 20 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ])", size :=
360, order :=
360, name := "Split(Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 2\
0 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ])", centralizers :=
[ 360, 180, 24, 12, 18, 9, 15, 15, 15, 12, 4, 6 ], classes :=
[ 1, 2, 15, 30, 20, 40, 24, 24, 24, 30, 90, 60 ], orders :=
[ 1, 3, 2, 6, 3, 3, 5, 15, 15, 2, 4, 6 ], powermap :=
[ , [ 1, 2, 1, 2, 5, 6, 7, [ 8, 9 ], [ 8, 9 ], 1, 3, 5 ],
  [ 1, 1, 3, 3, 1, 1, 7, 7, 7, 10, 11, 10 ],,
  [ 1, 2, 3, 4, 5, 6, 1, 2, 2, 10, 11, 12 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 2, -1, 2, -1, 2, -1, 2, -1, -1, 0, 0, 0 ],
  [ 6, 6, -2, -2, 0, 0, 1, 1, 1, 0, 0, 0 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, -1, 2, 0, -1 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, -1, -2, 0, 1 ],
  [ 8, -4, 0, 0, 2, -1, -2, 1, 1, 0, 0, 0 ],
  [ 5, 5, 1, 1, -1, -1, 0, 0, 0, 1, -1, 1 ],
  [ 5, 5, 1, 1, -1, -1, 0, 0, 0, -1, 1, -1 ],
  [ 10, -5, 2, -1, -2, 1, 0, 0, 0, 0, 0, 0 ] ], fusions := [ rec(
      name := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ])"
       ,
      map := [ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ] )
 ], operations := CharTableOps )
gap> Restricted( sub, split, red );
[ [ 12, -6, -4, 2, 0, 0, 2, -1, -1, 0, 0, 0 ] ]
```

To complete the table means to find the missing two irreducibles and to complete the powermaps. For this, there are different possibilities. First, one can try to embed G in A_8.

```
gap> a8:= CharTable( "A8" );;
gap> fus:= SubgroupFusions( split, a8 );
[ [ 1, 4, 3, 9, 4, 5, 8, 13, 14, 3, 7, 9 ],
  [ 1, 4, 3, 9, 4, 5, 8, 14, 13, 3, 7, 9 ] ]
gap> fus:= RepresentativesFusions( split, fus, a8 );
#I RepresentativesFusions: no subtable automorphisms stored
[ [ 1, 4, 3, 9, 4, 5, 8, 13, 14, 3, 7, 9 ] ]
gap> StoreFusion( split, a8, fus[1] );
```

The subgroup fusion is unique up to table automorphisms. Now we restrict the irreducibles of A_8 to G and reduce.

```
gap> rest:= Restricted( a8, split, a8.irreducibles );;
gap> red:= Reduced( split, split.irreducibles, rest );
rec(
  remainders := [  ],
  irreducibles :=
   [ [ 6, -3, -2, 1, 0, 0, 1, -E(15)-E(15)^2-E(15)^4-E(15)^8,
          -E(15)^7-E(15)^11-E(15)^13-E(15)^14, 0, 0, 0 ],
      [ 6, -3, -2, 1, 0, 0, 1, -E(15)^7-E(15)^11-E(15)^13-E(15)^14,
          -E(15)-E(15)^2-E(15)^4-E(15)^8, 0, 0, 0 ] ] )
```

```
gap> Append( split.irreducibles, red.irreducibles );
```

The list of irreducibles is now complete, but the powermaps are not yet adjusted. To complete the 2nd powermap, we transfer that of A_8 to G using the subgroup fusion.

```
gap> split.powermap;
[ , [ 1, 2, 1, 2, 5, 6, 7, [ 8, 9 ], [ 8, 9 ], 1, 3, 5 ],
  [ 1, 1, 3, 3, 1, 1, 7, 7, 7, 10, 11, 10 ],,
  [ 1, 2, 3, 4, 5, 6, 1, 2, 2, 10, 11, 12 ] ]
gap> TransferDiagram( split.powermap[2], fus[1], a8.powermap[2] );;
```

And this is the complete table.

```
gap> split.identifier:= "(A5x3):2";;
gap> DisplayCharTable( split );
Split(Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ]),[ 1, 2, 3\
, 4, 5, 6, 7, 8, 8, 9, 10, 11 ])

      2  3  2  3  2  1  .  .   .   .  2  2  1
      3  2  2  1  1  2  2  1   1   1  1  .  1
      5  1  1  .  .  .  .  1   1   1  .  .  .

        1a 3a 2a 6a 3b 3c 5a 15a 15b 2b 4a 6b
     2P 1a 3a 1a 3a 3b 3c 5a 15a 15b 1a 2a 3b
     3P 1a 1a 2a 2a 1a 1a 5a  5a  5a 2b 4a 2b
     5P 1a 3a 2a 6a 3b 3c 1a  3a  3a 2b 4a 6b

X.1      1  1  1  1  1  1  1   1   1  1  1  1
X.2      1  1  1  1  1  1  1   1   1 -1 -1 -1
X.3      2 -1  2 -1  2 -1  2  -1  -1  .  .  .
X.4      6  6 -2 -2  .  .  1   1   1  .  .  .
X.5      4  4  .  .  1  1 -1  -1  -1  2  . -1
X.6      4  4  .  .  1  1 -1  -1  -1 -2  .  1
X.7      8 -4  .  .  2 -1 -2   1   1  .  .  .
X.8      5  5  1  1 -1 -1  .   .   .  1 -1  1
X.9      5  5  1  1 -1 -1  .   .   . -1  1 -1
X.10    10 -5  2 -1 -2  1  .   .   .  .  .  .
X.11     6 -3 -2  1  .  .  1   A  /A  .  .  .
X.12     6 -3 -2  1  .  .  1  /A   A  .  .  .

A = -E(15)-E(15)^2-E(15)^4-E(15)^8
  = (-1-ER(-15))/2 = -1-b15
```

There are many ways around the block, so two further methods to complete the table `split` shall be demonstrated; but we will not go into details.

Without use of GAP3 one could work as follows:

The irrationalities –and there must be irrational entries in the character table of G, since the outer 2 can conjugate at most two of the four Galois conjugate classes of elements of order 15– could also have been found from the structure of G and the restriction of the irreducible $S_5 \times S_3$ character of degree 12.

On the classes that did not split the values of this character must just be divided by 2. Let x be one of the irrationalities. The second orthogonality relation tells us that $x \cdot \overline{x} = 4$ (at class 15a) and $x + x* = -1$ (at classes 1a and 15a); here $x*$ denotes the nontrivial Galois conjugate of x. This has no solution for $x = \overline{x}$, otherwise it leads to the quadratic equation $x^2 + x + 4 = 0$ with solutions $b15 = \frac{1}{2}(-1 + \sqrt{-15})$ and $-1 - b15$.

The third possibility to complete the table is to embed $A_5 \times 3$:

```
gap> split.irreducibles := split.irreducibles{ [ 1 .. 10 ] };;
gap> SubgroupFusions( a5xc3, split );
[ [ 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, [ 8, 9 ], [ 8, 9 ], 7, [ 8, 9 ],
      [ 8, 9 ] ] ]
```

The images of the four classes of element order 15 are not determined, the returned list parametrizes the 2^4 possibilities.

```
gap> fus:= ContainedMaps( last[1] );;
gap> Length( fus );
16
gap> fus[1];
[ 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 8, 7, 8, 8 ]
```

Most of these 16 possibilities are excluded using scalar products of induced characters. We take a suitable character `chi` of `a5xc3` and compute the norm of the induced character with respect to each possible map.

```
gap> chi:= a5xc3.irreducibles[5];
[ 3, 3*E(3), 3*E(3)^2, -1, -E(3), -E(3)^2, 0, 0, 0, -E(5)-E(5)^4,
  -E(15)^2-E(15)^8, -E(15)^7-E(15)^13, -E(5)^2-E(5)^3,
  -E(15)^11-E(15)^14, -E(15)-E(15)^4 ]
gap> List( fus, x -> List( Induced( a5xc3, split, [ chi ], x ),
>                          y -> ScalarProduct( split, y, y ) )[1] );
[ 8/15, -2/3*E(5)-11/15*E(5)^2-11/15*E(5)^3-2/3*E(5)^4,
  -2/3*E(5)-11/15*E(5)^2-11/15*E(5)^3-2/3*E(5)^4, 2/3,
  -11/15*E(5)-2/3*E(5)^2-2/3*E(5)^3-11/15*E(5)^4, 3/5, 1,
  -11/15*E(5)-2/3*E(5)^2-2/3*E(5)^3-11/15*E(5)^4,
  -11/15*E(5)-2/3*E(5)^2-2/3*E(5)^3-11/15*E(5)^4, 1, 3/5,
  -11/15*E(5)-2/3*E(5)^2-2/3*E(5)^3-11/15*E(5)^4, 2/3,
  -2/3*E(5)-11/15*E(5)^2-11/15*E(5)^3-2/3*E(5)^4,
  -2/3*E(5)-11/15*E(5)^2-11/15*E(5)^3-2/3*E(5)^4, 8/15 ]
gap> Filtered( [ 1 .. Length( fus ) ], x -> IsInt( last[x] ) );
[ 7, 10 ]
```

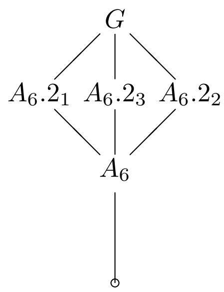
So only fusions 7 and 10 may be possible. They are equivalent (with respect to table automorphisms), and the list of induced characters contains the missing irreducibles of G:

```
gap> Sublist( fus, last );
[ [ 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 9, 7, 9, 8 ],
  [ 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 9, 8, 7, 8, 9 ] ]
gap> ind:= Induced( a5xc3, split, a5xc3.irreducibles, last[1] );;
gap> Reduced( split, split.irreducibles, ind );
rec(
  remainders := [  ],
```

```
irreducibles :=
 [ [ 6, -3, -2, 1, 0, 0, 1, -E(15)-E(15)^2-E(15)^4-E(15)^8,
        -E(15)^7-E(15)^11-E(15)^13-E(15)^14, 0, 0, 0 ],
    [ 6, -3, -2, 1, 0, 0, 1, -E(15)^7-E(15)^11-E(15)^13-E(15)^14,
        -E(15)-E(15)^2-E(15)^4-E(15)^8, 0, 0, 0 ] ] )
```

The following example is thought mainly for experts. It shall demonstrate how one can work together with GAP3 and the ATLAS [CCN+85], so better leave out the rest of this section if you are not familiar with the ATLAS.

We shall construct the character table of the group $G = A_6.2^2 \cong Aut(A_6)$ from the tables of the normal subgroups $A_6.2_1 \cong S_6$, $A_6.2_2 \cong PGL(2,9)$ and $A_6.2_3 \cong M_{10}$.
We regard G as a downward extension of the Klein four-group 2^2 with A_6. The set of classes of all preimages of cyclic subgroups of 2^2 covers the classes of G, but it may happen that some representatives are conjugate in G, i.e., the classes fuse.
The ATLAS denotes the character tables of G, $G.2_1$, $G.2_2$ and $G.2_3$ as follows:



	;	@	@	@	@	@	@	@	;	;	@	@	@	@	@
		360	8	9	9	4	5	5			24	24	4	3	3
	p power		A	A	A	A	A	A			A	A	A	AB	BC
	p' part		A	A	A	A	A	A			A	A	A	AB	BC
	ind	1A	2A	3A	3B	4A	5A	B*	fus	ind	2B	2C	4B	6A	6B
χ_1	+	1	1	1	1	1	1	1	:	++	1	1	1	1	1
χ_2	+	5	1	2	-1	-1	0	0	:	++	3	-1	1	0	-1
χ_3	+	5	1	-1	2	-1	0	0	:	++	-1	3	1	-1	0
χ_4	+	8	0	-1	-1	0	-b5	*		+	0	0	0	0	0
χ_5	+	8	0	-1	-1	0	*	-b5							
χ_6	+	9	1	0	0	1	-1	-1	:	++	3	3	-1	0	0
χ_7	+	10	-2	1	1	0	0	0	:	++	2	-2	0	-1	1

```
   ;   ;  @    @    @    @    @    ;   ;  @    @    @

         10    4    4    5    5          2    4    4
          A    A    A   BD   AD          A    A    A
          A    A    A   AD   BD          A    A    A
 fus ind 2D   8A   B*  10A   B* fus ind 4C   8C  D**

   :  ++  1    1    1    1    1    :  ++  1    1    1   χ1

       +  0    0    0    0    0        +  0    0    0   χ2

                                                         χ3

   :  ++  2    0    0   b5    *        +  0    0    0   χ4

   :  ++  2    0    0    *   b5                          χ5

   :  ++ -1    1    1   -1   -1    :  ++  1   -1   -1   χ6

   :  ++  0   r2  -r2    0    0    :  oo  0   i2  -i2   χ7
```

First we construct a table whose classes are those of the three subgroups. Note that the exponent of A_6 is 60, so the representative orders could become at most 60 times the value in 2^2.

```
gap> s1:= CharTable( "A6.2_1" );;
gap> s2:= CharTable( "A6.2_2" );;
gap> s3:= CharTable( "A6.2_3" );;
gap> c2:= CharTable( "Cyclic", 2 );;
gap> v4:= CharTableDirectProduct( c2, c2 );;
#I CharTableDirectProduct: existing subgroup fusion on <tbl2> replaced
#I    by actual one
gap> for tbl in [ s1, s2, s3 ] do
>      Print( tbl.irreducibles[2], "\n" );
>    od;
[ 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1 ]
[ 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1 ]
[ 1, 1, 1, 1, 1, -1, -1, -1 ]
gap> split:= CharTableSplitClasses( v4,
>               [1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4], 60 );;
gap> PrintCharTable( split );
rec( identifier := "Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, \
3, 3, 3, 4, 4, 4 ])", size := 4, order :=
4, name := "Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3,\
 4, 4, 4 ])", centralizers := [ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 4, 4, 4 ], classes := [ 1/5, 1/5, 1/5, 1/5, 1/5, 1/5, 1/5,
  1/5, 1/5, 1/5, 1/5, 1/5, 1/5, 1/5, 1/5, 1/3, 1/3, 1/3 ], orders :=
[ 1, [ 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60 ],
  [ 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60 ],
```

```
    [ 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60 ],
    [ 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ],
    [ 2, 4, 6, 8, 10, 12, 20, 24, 30, 40, 60, 120 ] ], powermap :=
  [ , [ 1, [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ],
        [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ],
        [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ],
        [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ],
        [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ],
        [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 5 ] ] ], irreducibles :=
  [ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, -1 ],
    [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1 ],
    [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1 ]
   ], fusions := [ rec(
        name := [ 'C', '2', 'x', 'C', '2' ],
        map := [ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4 ]
       ) ], operations := CharTableOps )
```

Now we embed the subgroups and adjust the classlengths, order, centralizers, powermaps and thus the representative orders.

```
gap> StoreFusion( s1, split, [1,2,3,3,4,5,6,7,8,9,10]);
gap> StoreFusion( s2, split, [1,2,3,4,5,5,11,12,13,14,15]);
gap> StoreFusion( s3, split, [1,2,3,4,5,16,17,18]);
gap> for tbl in [ s1, s2, s3 ] do
>      fus:= GetFusionMap( tbl, split );
>      for class in Difference( [ 1 .. Length( tbl.classes ) ],
>                               KernelChar(tbl.irreducibles[2]) ) do
>        split.classes[ fus[ class ] ]:= tbl.classes[ class ];
>      od;
>    od;
gap> for class in [ 1 .. 5 ] do
>      split.classes[ class ]:= s3.classes[ class ];
>    od;
gap> split.classes;
[ 1, 45, 80, 90, 144, 15, 15, 90, 120, 120, 36, 90, 90, 72, 72, 180,
  90, 90 ]
```

```
gap> split.size:= Sum( last );
1440
gap> split.order:= last;
gap> split.centralizers:= List( split.classes, x -> split.order / x );
[ 1440, 32, 18, 16, 10, 96, 96, 16, 12, 12, 40, 16, 16, 20, 20, 8,
  16, 16 ]
gap> split.powermap[3]:= InitPowermap( split, 3 );;
gap> split.powermap[5]:= InitPowermap( split, 5 );;
gap> for tbl in [ s1, s2, s3 ] do
>      fus:= GetFusionMap( tbl, split );
>      for p in [ 2, 3, 5 ] do
>        TransferDiagram( tbl.powermap[p], fus, split.powermap[p] );
>      od;
>    od;
gap> split.powermap;
[ , [ 1, 1, 3, 2, 5, 1, 1, 2, 3, 3, 1, 4, 4, 5, 5, 2, 4, 4 ],
  [ 1, 2, 1, 4, 5, 6, 7, 8, 6, 7, 11, 13, 12, 15, 14, 16, 17, 18 ],,
  [ 1, 2, 3, 4, 1, 6, 7, 8, 9, 10, 11, 13, 12, 11, 11, 16, 18, 17 ] ]
gap> split.orders:= ElementOrdersPowermap( split.powermap );
[ 1, 2, 3, 4, 5, 2, 2, 4, 6, 6, 2, 8, 8, 10, 10, 4, 8, 8 ]
```

In order to decide which classes fuse in G, we look at the norms of suitable induced characters, first the + extension of χ_2 to $A_6.2_1$.

```
gap> ind:= Induced( s1, split, [ s1.irreducibles[3] ] )[1];
[ 10, 2, 1, -2, 0, 6, -2, 2, 0, -2, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> ScalarProduct( split, ind, ind );
3/2
```

The inertia group of this character is $A_6.2_1$, thus the norm of the induced character must be 1. If the classes 2B and 2C fuse, the contribution of these classes is changed from $15 \cdot 6^2 + 15 \cdot (-2)^2$ to $30 \cdot 2^2$, the difference is 480. But we have to subtract 720 which is half the group order, so also 6A and 6B fuse. This is not surprising, since it reflects the action of the famous outer automorphism of S_6. Next we examine the + extension of χ_4 to $A_6.2_2$.

```
gap> ind:= Induced( s2, split, [ s2.irreducibles[4] ] )[1];
[ 16, 0, -2, 0, 1, 0, 0, 0, 0, 0, 4, 0, 0, 2*E(5)+2*E(5)^4,
  2*E(5)^2+2*E(5)^3, 0, 0, 0 ]
gap> ScalarProduct( split, ind, ind );
3/2
```

Again, the norm must be 1, 10A and 10B fuse.

```
gap> collaps:= CharTableCollapsedClasses( split,
>                    [1,2,3,4,5,6,6,7,8,8,9,10,11,12,12,13,14,15] );;
gap> PrintCharTable( collaps );
rec( identifier := "Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2,\
 2, 3, 3, 3, 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 1\
1, 12, 12, 13, 14, 15 ])", size := 1440, order :=
1440, name := "Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3\
, 3, 3, 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 11, 12\
, 12, 13, 14, 15 ])", centralizers := [ 1440, 32, 18, 16, 10, 48, 16,
```

```
  6, 40, 16, 16, 10, 8, 16, 16 ], orders :=
[ 1, 2, 3, 4, 5, 2, 4, 6, 2, 8, 8, 10, 4, 8, 8 ], powermap :=
[ , [ 1, 1, 3, 2, 5, 1, 2, 3, 1, 4, 4, 5, 2, 4, 4 ],
  [ 1, 2, 1, 4, 5, 6, 7, 6, 9, 11, 10, 12, 13, 14, 15 ],,
  [ 1, 2, 3, 4, 1, 6, 7, 8, 9, 11, 10, 9, 13, 15, 14 ]
 ], fusionsource :=
[ "Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4 \
])" ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, 1, 1, 1 ]
 ], classes := [ 1, 45, 80, 90, 144, 30, 90, 240, 36, 90, 90, 144,
  180, 90, 90 ], operations := CharTableOps )
gap> split.fusions;
[ rec(
      name := [ 'C', '2', 'x', 'C', '2' ],
      map := [ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4 ]
     ), rec(
      name :=
       "Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3,\
 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 11, 12, 12, 1\
3, 14, 15 ])",
      map := [ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 11, 12, 12, 13,
          14, 15 ] ) ]
gap> for tbl in [ s1, s2, s3 ] do
>      StoreFusion( tbl, collaps,
>                   CompositionMaps( GetFusionMap( split, collaps ),
>                                    GetFusionMap( tbl, split ) ) );
>    od;
gap> ind:= Induced( s1, collaps, [ s1.irreducibles[10] ] )[1];
[ 20, -4, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> ScalarProduct( collaps, ind, ind );
1
```

This character must be equal to any induced character of an irreducible character of degree 10 of $A_6.2_2$ and $A_6.2_3$. That means, `8A` fuses with `8B`, and `8C` with `8D`.

```
gap> a6v4:= CharTableCollapsedClasses( collaps,
>     [1,2,3,4,5,6,7,8,9,10,10,11,12,13,13] );;
gap> PrintCharTable( a6v4 );
rec( identifier := "Collapsed(Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2\
, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8\
, 9, 10, 11, 12, 12, 13, 14, 15 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10\
, 11, 12, 13, 13 ])", size := 1440, order :=
1440, name := "Collapsed(Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, \
2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, \
10, 11, 12, 12, 13, 14, 15 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11,\
 12, 13, 13 ])", centralizers := [ 1440, 32, 18, 16, 10, 48, 16, 6,
```

```
  40, 8, 10, 8, 8 ], orders := [ 1, 2, 3, 4, 5, 2, 4, 6, 2, 8, 10, 4,
  8 ], powermap := [ , [ 1, 1, 3, 2, 5, 1, 2, 3, 1, 4, 5, 2, 4 ],
  [ 1, 2, 1, 4, 5, 6, 7, 6, 9, 10, 11, 12, 13 ],,
  [ 1, 2, 3, 4, 1, 6, 7, 8, 9, 10, 9, 12, 13 ] ], fusionsource :=
[ "Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3\
, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 11, 12, 12, 13, 14\
, 15 ])" ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1 ] ], classes :=
[ 1, 45, 80, 90, 144, 30, 90, 240, 36, 180, 144, 180, 180
 ], operations := CharTableOps )
gap> for tbl in [ s1, s2, s3 ] do
>      StoreFusion( tbl, a6v4,
>                   CompositionMaps( GetFusionMap( collaps, a6v4 ),
>                                    GetFusionMap( tbl, collaps ) ) );
>    od;
```

Now the classes of G are known, the only remaining work is to compute the irreducibles.

```
gap> a6v4.irreducibles;
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1 ] ]
gap> for tbl in [ s1, s2, s3 ] do
>      ind:= Set( Induced( tbl, a6v4, tbl.irreducibles ) );
>      Append( a6v4.irreducibles,
>              Filtered( ind, x -> ScalarProduct( a6v4,x,x ) = 1 ) );
>    od;
gap> a6v4.irreducibles:= Set( a6v4.irreducibles );
[ [ 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1 ],
  [ 1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 10, 2, 1, -2, 0, -2, -2, 1, 0, 0, 0, 0, 0 ],
  [ 10, 2, 1, -2, 0, 2, 2, -1, 0, 0, 0, 0, 0 ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, -4, 0, 1, 0, 0 ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, 4, 0, -1, 0, 0 ],
  [ 20, -4, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
gap> sym:= Symmetrizations( a6v4, [ a6v4.irreducibles[5] ], 2 );
[ [ 45, -3, 0, 1, 0, -3, 1, 0, -5, 1, 0, -1, 1 ],
  [ 55, 7, 1, 3, 0, 7, 3, 1, 5, -1, 0, 1, -1 ] ]
gap> Reduced( a6v4, a6v4.irreducibles, sym );
rec(
  remainders := [ [ 27, 3, 0, 3, -3, 3, -1, 0, 1, -1, 1, 1, -1 ] ],
  irreducibles := [ [ 9, 1, 0, 1, -1, -3, 1, 0, -1, 1, -1, -1, 1 ] ] )
gap> Append( a6v4.irreducibles,
```

```
>           Tensored( last.irreducibles,
>                     Sublist( a6v4.irreducibles, [ 1 .. 4 ] ) ) );
gap> SortCharactersCharTable( a6v4,
>                          (1,4)(2,3)(5,6)(7,8)(9,13,10,11,12) );;
gap> a6v4.identifier:= "A6.2^2";;
gap> DisplayCharTable( a6v4 );
Collapsed(Collapsed(Split(C2xC2,[ 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, \
3, 3, 3, 4, 4, 4 ]),[ 1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 9, 10, 11, 12, 12,\
 13, 14, 15 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 11, 12, 13, 13 ])

     2  5  5  1  4  1  4  4  1  3  3   1  3  3
     3  2  .  2  .  .  1  .  1  .  .   .  .  .
     5  1  .  .  .  1  .  .  .  1  .   1  .  .

       1a 2a 3a 4a 5a 2b 4b 6a 2c 8a 10a 4c 8b
    2P 1a 1a 3a 2a 5a 1a 2a 3a 1a 4a  5a 2a 4a
    3P 1a 2a 1a 4a 5a 2b 4b 2b 2c 8a 10a 4c 8b
    5P 1a 2a 3a 4a 1a 2b 4b 6a 2c 8a  2c 4c 8b

X.1      1  1  1  1  1  1  1  1  1  1   1  1  1
X.2      1  1  1  1  1  1  1  1 -1 -1  -1 -1 -1
X.3      1  1  1  1  1 -1 -1 -1  1  1   1 -1 -1
X.4      1  1  1  1  1 -1 -1 -1 -1 -1  -1  1  1
X.5     10  2  1 -2  .  2  2 -1  .  .   .  .  .
X.6     10  2  1 -2  . -2 -2  1  .  .   .  .  .
X.7     16  . -2  .  1  .  .  .  4  .  -1  .  .
X.8     16  . -2  .  1  .  .  . -4  .   1  .  .
X.9      9  1  .  1 -1 -3  1  .  1 -1   1  1 -1
X.10     9  1  .  1 -1 -3  1  . -1  1  -1 -1  1
X.11     9  1  .  1 -1  3 -1  .  1 -1   1 -1  1
X.12     9  1  .  1 -1  3 -1  . -1  1  -1  1 -1
X.13    20 -4  2  .  .  .  .  .  .  .   .  .  .
```

1.26 About Group Libraries

When you start GAP3 it already knows several groups. For example, some basic groups such as cyclic groups or symmetric groups, all primitive permutation groups of degree at most 50, and all 2-groups of size at most 256.

Each of the sets above is called a **group library**. The set of all groups that GAP3 knows initially is called the **collection of group libraries**.

In this section we show you how you can access the groups in those libraries and how you can extract groups with certain properties from those libraries.

Let us start with the basic groups, because they are not accessed in the same way as the groups in the other libraries.

To access such a basic group you just call a function with an appropriate name, such as `CyclicGroup` or `SymmetricGroup`.

```
gap> c13 := CyclicGroup( 13 );
Group( ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13) )
gap> Size( c13 );
13
gap> s8 := SymmetricGroup( 8 );
Group( (1,8), (2,8), (3,8), (4,8), (5,8), (6,8), (7,8) )
gap> Size( s8 );
40320
```

The functions above also accept an optional first argument that describes the type of group. For example you can pass `AgWords` to `CyclicGroup` to get a cyclic group as a finite polycyclic group (see 25).

```
gap> c13 := CyclicGroup( AgWords, 13 );
Group( c13 )
```

Of course you cannot pass `AgWords` to `SymmetricGroup`, because symmetric groups are in general not polycyclic.

The default is to construct the groups as permutation groups, but you can also explicitly pass `Permutations`. Other possible arguments are `AgWords` for finite polycyclic groups, `Words` for finitely presented groups, and `Matrices` for matrix groups (however only `Permutations` and `AgWords` currently work).

Let us now turn to the other libraries. They are all accessed in a uniform way. For a first example we will use the group library of primitive permutation groups.

To extract a group from a group library you generally use the **extraction function**. In our example this function is called `PrimitiveGroup`. It takes two arguments. The first is the degree of the primitive permutation group that you want and the second is an integer that specifies which of the primitive permutation groups of that degree you want.

```
gap> g := PrimitiveGroup( 12, 3 );
M(11)
gap> g.generators;
[ ( 2, 6)( 3, 5)( 4, 7)( 9,10), ( 1, 5, 7)( 2, 9, 4)( 3, 8,10),
  ( 1,11)( 2, 7)( 3, 5)( 4, 6), ( 2, 5)( 3, 6)( 4, 7)(11,12) ]
gap> Size( g );
7920
gap> IsSimple( g );
true
gap> h := PrimitiveGroup( 16, 19 );
2^4.A(7)
gap> Size( h );
40320
```

The reason for the extraction function is as follows. A group library is usually not stored as a list of groups. Instead a more compact representation for the groups is used. For example the groups in the library of 2-groups are represented by 4 integers. The extraction function hides this representation from you, and allows you to access the group library as if it was a table of groups (two dimensional in the above example).

What arguments the extraction function accepts, and how they are interpreted is described in the sections that describe the individual group libraries in chapter 38. Those functions will of course signal an error when you pass illegal arguments.

Suppose that you want to get a list of all primitive permutation groups that have a degree 10 and are simple but not cyclic. It would be very difficult to use the extraction function to extract all groups in the group library, and test each of those. It is much simpler to use the **selection function**. The name of the selection function always begins with `All` and ends with `Groups`, in our example it is thus called `AllPrimitiveGroups`.

```
gap> AllPrimitiveGroups( DegreeOperation,   10,
>                        IsSimple,          true,
>                        IsCyclic,          false );
[ A(5), PSL(2,9), A(10) ]
```

`AllPrimitiveGroups` takes a variable number of argument pairs consisting of a function (e.g. `DegreeOperation`) and a value (e.g. 10). To understand what `AllPrimitiveGroups` does, imagine that the group library was stored as a long list of permutation groups. `AllPrimitiveGroups` takes all those groups in turn. To each group it applies each function argument and compares the result with the corresponding value argument. It selects a group if and only if all the function results are equal to the corresponding value. So in our example `AllPrimitiveGroups` selects those groups g for which `DegreeOperation(`g`) = 10` **and** `IsSimple(`g`) = true` **and** `IsCyclic(`g`) = false`. Finally `AllPrimitiveGroups` returns the list of the selected groups.

Next suppose that you want all the primitive permutation groups that have degree **at most** 10, are simple but are not cyclic. You could obtain such a list with 10 calls to `AllPrimitiveGroups` (i.e., one call for the degree 1 groups, another for the degree 2 groups and so on), but there is a simple way. Instead of specifying a single value that a function must return you can simply specify a list of such values.

```
gap> AllPrimitiveGroups( DegreeOperation,   [1..10],
>                        IsSimple,          true,
>                        IsCyclic,          false );
[ A(5), PSL(2,5), A(6), PSL(3,2), A(7), PSL(2,7), A(8), PSL(2,8),
  A(9), A(5), PSL(2,9), A(10) ]
```

Note that the list that you get contains `A(5)` twice, first in its primitive presentation on 5 points and second in its primitive presentation on 10 points.

Thus giving several argument pairs to the selection function allows you to express the logical **and** of properties that a group must have to be selected, and giving a list of values allows you to express a (restricted) logical **or** of properties that a group must have to be selected.

There is no restriction on the functions that you can use. It is even possible to use functions that you have written yourself. Of course, the functions must be unary, i.e., accept only one argument, and must be able to deal with the groups.

```
gap> NumberConjugacyClasses := function ( g )
>        return Length( ConjugacyClasses( g ) );
> end;
function ( g ) ... end
gap> AllPrimitiveGroups( DegreeOperation,          [1..10],
>                        IsSimple,                 true,
>                        IsCyclic,                 false,
>                        NumberConjugacyClasses,   9 );
[ A(7), PSL(2,8) ]
```

Note that in some cases a selection function will issue a warning. For example if you call `AllPrimitiveGroups` without specifying the degree, it will issue such a warning.

```
gap> AllPrimitiveGroups( Size,      [100..400],
>                        IsSimple,  true,
>                        IsCyclic,  false );
#W  AllPrimitiveGroups: degree automatically restricted to [1..50]
[ A(6), PSL(3,2), PSL(2,7), PSL(2,9), A(6) ]
```

If selection functions would really run over the list of all groups in a group library and apply the function arguments to each of those, they would be very inefficient. For example the 2-groups library contains 58760 groups. Simply creating all those groups would take a very long time.

Instead selection functions recognize certain functions and handle them more efficiently. For example `AllPrimitiveGroups` recognizes `DegreeOperation`. If you pass `DegreeOperation` to `AllPrimitiveGroups` it does not create a group to apply `DegreeOperation` to it. Instead it simply consults an index and quickly eliminates all groups that have a different degree. Other functions recognized by `AllPrimitiveGroups` are `IsSimple`, `Size`, and `Transitivity`.

So in our examples `AllPrimitiveGroups`, recognizing `DegreeOperation` and `IsSimple`, eliminates all but 16 groups. Then it creates those 16 groups and applies `IsCyclic` to them. This eliminates 4 more groups (`C(2)`, `C(3)`, `C(5)`, and `C(7)`). Then in our last example it applies `NumberConjugacyClasses` to the remaining 12 groups and eliminates all but `A(7)` and `PSL(2,8)`.

The catch is that the selection functions will take a large amount of time if they cannot recognize any special functions. For example the following selection will take a large amount of time, because only `IsSimple` is recognized, and there are 116 simple groups in the primitive groups library.

```
AllPrimitiveGroups( IsSimple, true, NumberConjugacyClasses, 9 );
```

So you should specify a sufficiently large set of recognizable functions when you call a selection function. It is also advisable to put those functions first (though in some group libraries the selection function will automatically rearrange the argument pairs so that the recognized functions come first). The sections describing the individual group libraries in chapter 38 tell you which functions are recognized by the selection function of that group library.

There is another function, called the **example function** that behaves similar to the selection function. Instead of returning a list of all groups with a certain set of properties it only returns one such group. The name of the example function is obtained by replacing `All` by `One` and stripping the `s` at the end of the name of the selection function.

```
gap> OnePrimitiveGroup( DegreeOperation,          [1..10],
>                       IsSimple,                 true,
>                       IsCyclic,                 false,
>                       NumberConjugacyClasses,   9 );
A(7)
```

The example function works just like the selection function. That means that all the above comments about the special functions that are recognized also apply to the example function.

Let us now look at the 2-groups library. It is accessed in the same way as the primitive groups library. There is an extraction function `TwoGroup`, a selection function `AllTwoGroups`, and an example function `OneTwoGroup`.

```
gap> g := TwoGroup( 128, 5 );
Group( a1, a2, a3, a4, a5, a6, a7 )
gap> Size( g );
128
gap> NumberConjugacyClasses( g );
80
```

The groups are all displayed as `Group( a1, a2, ..., a`n` )`, where 2^n is the size of the group.

```
gap> AllTwoGroups( Size,   256,
>                  Rank,   3,
>                  pClass, 2 );
[ Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ) ]
gap> l := AllTwoGroups( Size,                             256,
>                       Rank,                             3,
>                       pClass,                           5,
>                       g -> Length( DerivedSeries( g ) ), 4 );;
gap> Length( l );
28
```

The selection and example function of the 2-groups library recognize `Size`, `Rank`, and `pClass`. Note that `Rank` and `pClass` are functions that can in fact only be used in this context, i.e., they can not be applied to arbitrary groups.

The following discussion is a bit technical and you can ignore it safely.

For very big group libraries, such as the 2-groups library, the groups (or their compact representations) are not stored on a single file. This is because this file would be very large and loading it would take a long time and a lot of main memory.

Instead the groups are stored on a small number of files (27 in the case of the 2-groups). The selection and example functions are careful to load only those files that may actually contain groups with the specified properties. For example in the above example the files containing the groups of size less than 256 are never loaded. In fact in the above example only one very small file is loaded.

When a file is loaded the selection and example functions also unload the previously loaded file. That means that they forget all the groups in this file again (except those selected of course). Thus even if the selection or example functions have to search through the whole group library, only a small part of the library is held in main memory at any time. In principle it should be possible to search the whole 2-groups library with as little as 2 MByte of main memory.

If you have sufficient main memory available you can explicitly load files from the 2-groups library with `ReadTwo(` *filename* `)`, e.g., `Read( "twogp64")` to load the file with the groups of size 64. Those files will then not be unloaded again. This will take up more main memory, but the selection and example function will work faster, because they do not have to load those files again each time they are needed.

In this section you have seen the basic groups library and the group libraries of primitive groups and 2-groups. You have seen how you can extract a single group from such a library with the extraction function. You have seen how you can select groups with certain properties with the selection and example function. Chapter 38 tells you which other group libraries are available.

1.27 About the Implementation of Domains

In this section we will open the black boxes and describe how all this works. This is complex and you do not need to understand it if you are content to use domains only as black boxes. So you may want to skip this section (and the remainder of this chapter).

Domains are represented by records, which we will call **domain records** in the following. Which components have to be present, which may, and what those components hold, differs from category to category, and, to a smaller extent, from domain to domain. It is possible, though, to generally distinguish four types of components.

The first type of components are called the **category components**. They determine to which category a domain belongs. A domain D in a category *Cat* has a component `is`*Cat* with the value `true`. For example, each group has the component `isGroup`. Also each domain has the component `isDomain` (again with the value `true`). Finally a domain may also have components that describe the representation of this domain. For example, each permutation group has a component `isPermGroup` (again with the value `true`). Functions such as `IsPermGroup` test whether such a component is present, and whether it has the value `true`.

The second type of components are called the **identification components**. They distinguish the domain from other domains in the same category. The identification components uniquely identify the domain. For example, for groups the identification components are `generators`, which holds a list of generators of the group, and `identity`, which holds the identity of the group (needed for the trivial group, for which the list of generators is empty).

The third type of components are called **knowledge components**. They hold all the knowledge GAP3 has about the domain. For example the size of the domain D is stored in the knowledge component D`.size`, the commutator subgroup of a group is stored in the knowledge component D`.commutatorSubgroup`, etc. Of course, the knowledge about a certain domain will usually increase as you work with a domain. For example, a group record may initially hold only the knowledge that the group is finite, but may later hold all kinds of knowledge, for example the derived series, the Sylow subgroups, etc.

Finally each domain record contains an **operations record**. The operations record is discussed below.

We want to emphasize that really all information that GAP3 has about a domain is stored in the knowledge components. That means that you can access all this information, at least if you know where to look and how to interpret what you see. The chapters describing categories and domains will tell you what knowledge components a domain may have, and how the knowledge is represented in those components.

For an example let us return to the permutation group `a5` from section 1.23. If we print the record using the function `PrintRec` we see all the information. GAP3 stores the stabilizer chain of `a5` in the components `orbit`, `transversal`, and `stabilizer`. It is not important that you understand what a stabilizer chain is (this is discussed in chapter 21), the important point here is that it is the vital information that GAP3 needs to work efficiently with `a5` and that you can access it.

```
gap> a5 := Group( (1,2,3), (3,4,5) );
Group( (1,2,3), (3,4,5) )
gap> Size( a5 );
```

```
60
gap> PrintRec( a5 );
rec(
  isDomain         := true,
  isGroup          := true,
  identity         := (),
  generators       := [ (1,2,3), (3,4,5) ],
  operations       := ...,
  isPermGroup      := true,
  isFinite         := true,
  1                := (1,2,3),
  2                := (3,4,5),
  orbit            := [ 1, 3, 2, 5, 4 ],
  transversal      := [ (), (1,2,3), (1,2,3), (3,4,5), (3,4,5) ],
  stabilizer       := rec(
    identity    := (),
    generators  := [ (3,4,5), (2,5,3) ],
    orbit       := [ 2, 3, 5, 4 ],
    transversal := [ , (), (2,5,3), (3,4,5), (3,4,5) ],
    stabilizer  := rec(
      identity    := (),
      generators  := [ (3,4,5) ],
      orbit       := [ 3, 5, 4 ],
      transversal := [ ,, (), (3,4,5), (3,4,5) ],
      stabilizer  := rec(
        identity   := (),
        generators := [  ],
        operations := ... ),
      operations  := ... ),
    operations  := ... ),
  isParent         := true,
  stabChainOptions := rec(
    random     := 1000,
    operations := ... ),
  stabChain        := rec(
    generators  := [ (1,2,3), (3,4,5) ],
    identity    := (),
    orbit       := [ 1, 3, 2, 5, 4 ],
    transversal := [ (), (1,2,3), (1,2,3), (3,4,5), (3,4,5) ],
    stabilizer  := rec(
      identity    := (),
      generators  := [ (3,4,5), (2,5,3) ],
      orbit       := [ 2, 3, 5, 4 ],
      transversal := [ , (), (2,5,3), (3,4,5), (3,4,5) ],
      stabilizer  := rec(
        identity    := (),
        generators  := [ (3,4,5) ],
        orbit       := [ 3, 5, 4 ],
```

```
          transversal := [ ,, (), (3,4,5), (3,4,5) ],
          stabilizer  := rec(
            identity   := (),
            generators := [  ],
            operations := ... ),
          operations  := ... ),
        operations  := ... ),
      operations  := ... ),
    size             := 60 )
```

Note that you can not only read this information, you can also modify it. However, unless you truly understand what you are doing, we discourage you from playing around. All GAP3 functions assume that the information in the domain record is in a consistent state, and everything will go wrong if it is not.

```
gap> a5.size := 120;
120
gap> Size( ConjugacyClass( a5, (1,2,3,4,5) ) );
24     #  this is of course wrong
```

As was mentioned above, each domain record has an operations record. We have already seen that functions such as `Size` can be applied to various types of domains. It is clear that there is no general method that will compute the size of all domains efficiently. So `Size` must somehow decide which method to apply to a given domain. The operations record makes this possible.

The operations record of a domain *D* is the component with the name *D*`.operations`, its value is a record. For each function that you can apply to *D* this record contains a function that will compute the required information (hopefully in an efficient way).

To understand this let us take a look at what happens when we compute `Size( a5 )`. Not much happens. `Size` simply calls `a5.operations.Size( a5 )`. `a5.operations.Size` is a function written especially for permutation groups. It computes the size of `a5` and returns it. Then `Size` returns this value.

Actually `Size` does a little bit more than that. It first tests whether `a5` has the knowledge component `a5.size`. If this is the case, `Size` simply returns that value. Otherwise it calls `a5.operations.Size( a5 )` to compute the size. `Size` remembers the result in the knowledge component `a5.size` so that it is readily available the next time `Size( a5 )` is called. The complete definition of `Size` is as follows.

```
gap> Size := function ( D )
>     local  size;
>     if IsSet( D )  then
>         size := Length( D );
>     elif IsRec( D ) and IsBound( D.size )  then
>         size := D.size;
>     elif IsDomain( D )  then
>         D.size := D.operations.Size( D );
>         size := D.size;
>     else
>         Error( "<D> must be a domain or a set" );
```

```
>     fi;
>     return size;
> end;;
```

Because functions such as `Size` only dispatch to the functions in the operations record, they are called **dispatcher functions**. Almost all functions that you call directly are dispatcher functions, and almost all functions that do the hard work are components in an operations record.

Which function is called by a dispatcher obviously depends on the domain and its operations record (that is the whole point of having an operations record). In principle each domain could have its own `Size` function. In practice however, this would require too many functions. So different domains share the functions in their operations records, usually all domains with the same representation share all their operations record functions. For example all permutation groups share the same `Size` function. Because this shared `Size` function must be able to access the information in the domain record to compute the correct result, the `Size` dispatcher function (and all other dispatchers as well) pass the domain as first argument

In fact the domains not only have the same functions in their operations record, they share the operations record. So for example all permutation groups share a common operations record, which is called `PermGroupOps`. This means that changing a function in the operations record for a domain D in the following way D`.operations.`*function* `:=` *new-function*`;` will also change this function for all domains of the same type, even those that do not yet exist at the moment of the assignment and will only be constructed later. This is usually not desirable, since supposedly *new-function* uses some special properties of the domain D to work more efficiently. We suggest therefore that you first make a copy of the operations record with D`.operations := Copy(` D`.operations );` and only afterwards do D`.operations.`*function* `:=` *new-function*`;`.

If a programmer that implements a new domain D, a new type of groups say, would have to write all functions applicable to D, this would require a lot of effort. For example, there are about 120 functions applicable to groups. Luckily many of those functions are independent of the particular type of groups. For example the following function will compute the commutator subgroup of any group, assuming that `TrivialSubgroup`, `Closure`, and `NormalClosure` work. We say that this function is **generic**.

```
gap> GroupOps.CommutatorSubgroup := function ( U, V )
>     local   C, u, v, c;
>     C := TrivialSubgroup( U );
>     for u  in U.generators  do
>         for v  in V.generators  do
>             c := Comm( u, v );
>             if not c in C  then
>                 C := Closure( C, c );
>             fi;
>         od;
>     od;
>     return NormalClosure( Closure( U, V ), C );
> end;;
```

So it should be possible to use this function for the new type of groups. The mechanism to do this is called **inheritance**. How it works is described in 1.28, but basically the programmer just copies the generic functions from the generic group operations record into the operations record for his new type of groups.

The generic functions are also called **default functions**, because they are used by default, unless the programmer **overlaid** them for the new type of groups.

There is another mechanism through which work can be simplified. It is called **delegation**. Suppose that a generic function works for the new type of groups, but that some special cases can be handled more efficiently for the new type of groups. Then it is possible to handle only those cases and delegate the general cases back to the generic function. An example of this is the function that computes the orbit of a point under a permutation group. If the point is an integer then the generic algorithm can be improved by keeping a second list that remembers which points have already been seen. The other cases (remember that `Orbit` can also be used for other operations, e.g., the operation of a permutation group on pairs of points or the operations on subgroups by conjugation) are delegated back to the generic function. How this is done can be seen in the following definition.

```
gap> PermGroupOps.Orbit := function ( G, d, opr )
>     local   orb,          # orbit of d under G, result
>             max,          # largest point moved by the group G
>             new,          # boolean list indicating if a point is new
>             gen,          # one generator of the group G
>             pnt,          # one point in the orbit orb
>             img;          # image of pnt under gen
>
>     # standard operation
>     if   opr = OnPoints  and IsInt(d)  then
>
>         # get the largest point max moved by the group G
>         max := 0;
>         for gen  in G.generators  do
>             if max < LargestMovedPointPerm(gen)  then
>                 max := LargestMovedPointPerm(gen);
>             fi;
>         od;
>
>         # handle fixpoints
>         if not d in [1..max]  then
>             return [ d ];
>         fi;
>
>         # start with the singleton orbit
>         orb := [ d ];
>         new := BlistList( [1..max], [1..max] );
>         new[d] := false;
>
>         # loop over all points found
>         for pnt  in orb  do
>             for gen  in G.generators  do
```

```
>                 img := pnt ^ gen;
>                 if new[img]  then
>                     Add( orb, img );
>                     new[img] := false;
>                 fi;
>             od;
>         od;
>
>     #  other operation, delegate back on default function
>     else
>         orb := GroupOps.Orbit( G, d, opr );
>     fi;
>
>     #  return the orbit orb
>     return orb;
> end;;
```

Inheritance and delegation allow the programmer to implement a new type of groups by merely specifying how those groups differ from generic groups. This is far less work than having to implement all possible functions (apart from the problem that in this case it is very likely that the programmer would forget some of the more exotic functions).

To make all this clearer let us look at an extended example to show you how a computation in a domain may use default and special functions to achieve its goal. Suppose you defined `g`, `x`, and `y` as follows.

```
gap> g := SymmetricGroup( 8 );;
gap> x := [ (2,7,4)(3,5), (1,2,6)(4,8) ];;
gap> y := [ (2,5,7)(4,6), (1,5)(3,8,7) ];;
```

Now you ask for an element of `g` that conjugates `x` to `y`, i.e., a permutation on 8 points that takes `(2,7,4)(3,5)` to `(2,5,7)(4,6)` and `(1,2,6)(4,8)` to `(1,5)(3,8,7)`. This is done as follows (see 8.25 and 8.1).

```
gap> RepresentativeOperation( g, x, y, OnTuples );
(1,8)(2,7)(3,4,5,6)
```

Now lets look at what happens step for step. First `RepresentativeOperation` is called. After checking the arguments it calls the function `g.operations.RepresentativeOperation`, which is the function `SymmetricGroupOps.RepresentativeOperation`, passing the arguments `g`, `x`, `y`, and `OnTuples`.

`SymmetricGroupOps.RepresentativeOperation` handles a lot of cases special, but the operation on tuples of permutations is not among them. Therefore it delegates this problem to the function that it overlays, which is `PermGroupOps.RepresentativeOperation`.

`PermGroupOps.RepresentativeOperation` also does not handle this special case, and delegates the problem to the function that it overlays, which is the default function called `GroupOps.RepresentativeOperation`.

`GroupOps.RepresentativeOperation` views this problem as a general tuples problem, i.e., it does not care whether the points in the tuples are integers or permutations, and decides to solve it one step at a time. So first it looks for an element taking `(2,7,4)(3,5)` to

(2,5,7)(4,6) by calling RepresentativeOperation(g, (2,7,4)(3,5), (2,5,7)(4,6)).

RepresentativeOperation calls g.operations.RepresentativeOperation next, which is the function SymmetricGroupOps.RepresentativeOperation, passing the arguments g, (2,7,4)(3,5), and (2,5,7)(4,6).

SymmetricGroupOps.RepresentativeOperation can handle this case. It **knows** that g contains every permutation on 8 points, so it contains (3,4,7,5,6), which obviously does what we want, namely it takes x[1] to y[1]. We will call this element t.

Now GroupOps.RepresentativeOperation (see above) looks for an s in the stabilizer of x[1] taking x[2] to y[2]^(t^-1), since then for r=s*t we have x[1]^r = (x[1]^s)^t = x[1]^t = y[1] and also x[2]^r = (x[2]^s)^t = (y[2]^(t^-1))^t = y[2]. So the next step is to compute the stabilizer of x[1] in g. To do this it calls Stabilizer(g, (2,7,4)(3,5)).

Stabilizer calls g.operations.Stabilizer, which is SymmetricGroupOps.Stabilizer, passing the arguments g and (2,7,4)(3,5). SymmetricGroupOps.Stabilizer detects that the second argument is a permutation, i.e., an element of the group, and calls Centralizer(g, (2,7,4)(3,5)). Centralizer calls the function g.operations.Centralizer, which is SymmetricGroupOps.Centralizer, again passing the arguments g, (2,7,4)(3,5).

SymmetricGroupOps.Centralizer again **knows** how centralizer in symmetric groups look, and after looking at the permutation (2,7,4)(3,5) sharply for a short while returns the centralizer as Subgroup(g, [(1,6), (6,8), (2,7,4), (3,5)]), which we will call c. Note that c is of course not a symmetric group, therefore SymmetricGroupOps.Subgroup gives it PermGroupOps as operations record and not SymmetricGroupOps.

As explained above GroupOps.RepresentativeOperation needs an element of c taking x[2] ((1,2,6)(4,8)) to y[2]^(t^-1) ((1,7)(4,6,8)). So RepresentativeOperation(c, (1,2,6)(4,8), (1,7)(4,6,8)) is called. RepresentativeOperation in turn calls the function c.operations.RepresentativeOperation, which is, since c is a permutation group, the function PermGroupOps.RepresentativeOperation, passing the arguments c, (1,2,6)(4,8), and (1,7)(4,6,8).

PermGroupOps.RepresentativeOperation detects that the points are permutations and and performs a backtrack search through c. It finds and returns (1,8)(2,4,7)(3,5), which we call s.

Then GroupOps.RepresentativeOperation returns r = s*t = (1,8)(2,7)(3,6)(4,5), and we are done.

In this example you have seen how functions use the structure of their domain to solve a problem most efficiently, for example SymmetricGroupOps.RepresentativeOperation but also the backtrack search in PermGroupOps.RepresentativeOperation, how they use other functions, for example SymmetricGroupOps.Stabilizer called Centralizer, and how they delegate cases which they can not handle more efficiently back to the function they overlaid, for example SymmetricGroupOps.RepresentativeOperation delegated to PermGroupOps.RepresentativeOperation, which in turn delegated to to the function GroupOps.RepresentativeOperation.

If you think this whole mechanism using dispatcher functions and the operations record is overly complex let us look at some of the alternatives. This is even more technical than the previous part of this section so you may want to skip the remainder of this section.

One alternative would be to let the dispatcher know about the various types of domains, test which category a domain lies in, and dispatch to an appropriate function. Then we would not need an operations record. The dispatcher function `CommutatorSubgroup` would then look as follows. Note this is **not** how `CommutatorSubgroup` is implemented in GAP3.

```
CommutatorSubgroup := function ( G )
    local   C;
    if IsAgGroup( G )  then
        C := CommutatorSubgroupAgGroup( G );
    elif IsMatGroup( G )  then
        C := CommutatorSubgroupMatGroup( G );
    elif IsPermGroup( G )  then
        C := CommutatorSubgroupPermGroup( G );
    elif IsFpGroup( G )  then
        C := CommutatorSubgroupFpGroup( G );
    elif IsFactorGroup( G )  then
        C := CommutatorSubgroupFactorGroup( G );
    elif IsDirectProduct( G )  then
        C := CommutatorSubgroupDirectProduct( G );
    elif IsDirectProductAgGroup( G )  then
        C := CommutatorSubgroupDirectProductAgGroup( G );
    elif IsSubdirectProduct( G )  then
        C := CommutatorSubgroupSubdirectProduct( G );
    elif IsSemidirectProduct( G )  then
        C := CommutatorSubgroupSemidirectProduct( G );
    elif IsWreathProduct( G )  then
        C := CommutatorSubgroupWreathProduct( G );
    elif IsGroup( G )  then
        C := CommutatorSubgroupGroup( G );
    else
        Error("<G> must be a group");
    fi;
    return C;
end;
```

You already see one problem with this approach. The number of cases that the dispatcher functions would have to test is simply to large. It is even worse for set theoretic functions, because they would have to handle all different types of domains (currently about 30).

The other problem arises when a programmer implements a new domain. Then he would have to rewrite all dispatchers and add a new case to each. Also the probability that the programmer forgets one dispatcher is very high.

Another problem is that inheritance becomes more difficult. Instead of just copying one operations record the programmer would have to copy each function that should be inherited. Again the probability that he forgets one is very high.

Another alternative would be to do completely without dispatchers. In this case there would be the functions `CommutatorSugroupAgGroup`, `CommutatorSubgroupPermGroup`, etc., and it would be your responsibility to call the right function. For example to compute the size of a permutation group you would call `SizePermGroup` and to compute the size of a coset you would call `SizeCoset` (or maybe even `SizeCosetPermGroup`).

The most obvious problem with this approach is that it is much more cumbersome. You would always have to know what kind of domain you are working with and which function you would have to call.

Another problem is that writing generic functions would be impossible. For example the above generic implementation of `CommutatorSubgroup` could not work, because for a concrete group it would have to call `ClosurePermGroup` or `ClosureAgGroup` etc.

If generic functions are impossible, inheritance and delegation can not be used. Thus for each type of domain all functions must be implemented. This is clearly a lot of work, more work than we are willing to do.

So we argue that our mechanism is the easiest possible that serves the following two goals. It is reasonably convenient for you to use. It allows us to implement a large (and ever increasing) number of different types of domains.

This may all sound a lot like object oriented programming to you. This is not surprising because we want to solve the same problems that object oriented programming tries to solve. Let us briefly discuss the similarities and differences to object oriented programming, taking C++ as an example (because it is probably the widest known object oriented programming language nowadays). This discussion is very technical and again you may want to skip the remainder of this section.

Let us first recall the problems that the GAP3 mechanism wants to handle.

1 How can we represent domains in such a way that we can handle domains of different type in a common way?

2 How can we make it possible to allow functions that take domains of different type and perform the same operation for those domains (but using different methods)?

3 How can we make it possible that the implementation of a new type of domains only requires that one implements what distinguishes this new type of domains from domains of an old type (without the need to change any old code)?

For object oriented programming the problems are the same, though the names used are different. We talk about domains, object oriented programming talks about objects, and we talk about categories, object oriented programming talks about classes.

1 How can we represent objects in such a way that we can handle objects of different classes in a common way (e.g., declare variables that can hold objects of different classes)?

2 How can we make it possible to allow functions that take objects of different classes (with a common base class) and perform the same operation for those objects (but using different methods)?

3 How can we make it possible that the implementation of a new class of objects only requires that one implements what distinguishes the objects of this new class from the objects of an old (base) class (without the need to change any old code)?

In GAP3 the first problem is solved by representing all domains using records. Actually because GAP3 does not perform strong static type checking each variable can hold objects of arbitrary type, so it would even be possible to represent some domains using lists or something else. But then, where would we put the operations record?

C++ does something similar. Objects are represented by `struct`-s or pointers to structures. C++ then allows that a pointer to an object of a base class actually holds a pointer to an object of a derived class.

In GAP3 the second problem is solved by the dispatchers and the operations record. The operations record of a given domain holds the methods that should be applied to that domain, and the dispatcher does nothing but call this method.

In C++ it is again very similar. The difference is that the dispatcher only exists conceptually. If the compiler can already decide which method will be executed by a given call to the dispatcher it directly calls this function. Otherwise (for virtual functions that may be overlaid in derived classes) it basically inlines the dispatcher. This inlined code then dispatches through the so–called **virtual method table** (`vmt`). Note that this virtual method table is the same as the operations record, except that it is a table and not a record.

In GAP3 the third problem is solved by inheritance and delegation. To inherit functions you simply copy them from the operations record of domains of the old category to the operations record of domains of the new category. Delegation to a method of a larger category is done by calling *super-category-operations-record*`.`*function*

C++ also supports inheritance and delegation. If you derive a class from a base class, you copy the methods from the base class to the derived class. Again this copying is only done conceptually in C++. Delegation is done by calling a qualified function *base-class*`::`*function*.

Now that we have seen the similarities, let us discuss the differences.

The first differences is that GAP3 is not an object oriented programming language. We only programmed the library in an object oriented way using very few features of the language (basically all we need is that GAP3 has no strong static type checking, that records can hold functions, and that records can grow dynamically). Following Stroustrup's convention we say that the GAP3 language only **enables** object oriented programming, but does not **support** it.

The second difference is that C++ adds a mechanism to support data hiding. That means that fields of a `struct` can be private. Those fields can only be accessed by the functions belonging to this class (and `friend` functions). This is not possible in GAP3. Every field of every domain is accessible. This means that you can also modify those fields, with probably catastrophic results.

The final difference has to do with the relation between categories and their domains and classes and their objects. In GAP3 a category is a set of domains, thus we say that a domain is an element of a category. In C++ (and most other object oriented programming languages) a class is a prototype for its objects, thus we say that an object is an instance of the class. We believe that GAP3's relation better resembles the mathematical model.

In this section you have seen that domains are represented by domain records, and that you can therefore access all information that GAP3 has about a certain domain. The following sections in this chapter discuss how new domains can be created (see 1.28, and 1.29) and how you can even define a new type of elements (see 1.30).

1.28 About Defining New Domains

In this section we will show how one can add a new domain to GAP3. All domains are implemented in the library in this way. We will use the ring of Gaussian integers as our example.

Note that everything defined here is already in the library file `LIBNAME/"gaussian.g"`, so there is no need for you to type it in. You may however like to make a copy of this file and modify it.

The elements of this domain are already available, because Gaussian integers are just a special case of cyclotomic numbers. As is described in chapter 13 `E(4)` is GAP3's name for the complex root of -1. So all Gaussian integers can be represented as *a* `+` *b*`*E(4)`, where *a* and *b* are ordinary integers.

As was already mentioned each domain is represented by a record. So we create a record to represent the Gaussian integers, which we call `GaussianIntegers`.

```
gap> GaussianIntegers := rec();;
```

The first components that this record must have are those that identify this record as a record denoting a ring domain. Those components are called the **category components**.

```
gap> GaussianIntegers.isDomain := true;;
gap> GaussianIntegers.isRing := true;;
```

The next components are those that uniquely identify this ring. For rings this must be `generators`, `zero`, and `one`. Those components are called the **identification components** of the domain record. We also assign a **name component**. This name will be printed when the domain is printed.

```
gap> GaussianIntegers.generators := [ 1, E(4) ];;
gap> GaussianIntegers.zero := 0;;
gap> GaussianIntegers.one := 1;;
gap> GaussianIntegers.name := "GaussianIntegers";;
```

Next we enter some components that represent knowledge that we have about this domain. Those components are called the **knowledge components**. In our example we know that the Gaussian integers form a infinite, commutative, integral, Euclidean ring, which has an unique factorization property, with the four units 1, -1, `E(4)`, and `-E(4)`.

```
gap> GaussianIntegers.size                       := "infinity";;
gap> GaussianIntegers.isFinite                   := false;;
gap> GaussianIntegers.isCommutativeRing          := true;;
gap> GaussianIntegers.isIntegralRing             := true;;
gap> GaussianIntegers.isUniqueFactorizationRing  := true;;
gap> GaussianIntegers.isEuclideanRing            := true;;
gap> GaussianIntegers.units                      := [1,-1,E(4),-E(4)];;
```

This was the easy part of this example. Now we have to add an **operations record** to the domain record. This operations record (`GaussianIntegers.operations`) shall contain functions that implement all the functions mentioned in chapter 5, e.g., `DefaultRing`, `IsCommutativeRing`, `Gcd`, or `QuotientRemainder`.

Luckily we do not have to implement all this functions. The first class of functions that we need not implement are those that can simply get the result from the knowledge components. E.g., `IsCommutativeRing` looks for the knowledge component `isCommutativeRing`, finds it and returns this value. So `GaussianIntegers.operations.IsCommutativeRing` is never called.

```
gap> IsCommutativeRing( GaussianIntegers );
true
gap> Units( GaussianIntegers );
[ 1, -1, E(4), -E(4) ]
```

The second class of functions that we need not implement are those for which there is a general algorithm that can be applied for all rings. For example once we can do a division with remainder (which we will have to implement) we can use the general Euclidean algorithm to compute the greatest common divisor of elements.

So the question is, how do we get those general functions into our operations record. This is very simple, we just initialize the operations record as a copy of the record `RingOps`, which contains all those general functions. We say that `GaussianIntegers.operations` **inherits** the general functions from `RingOps`.

```
gap> GaussianIntegersOps := OperationsRecord(
>        "GaussianIntegersOps", RingOps );;
gap> GaussianIntegers.operations := GaussianIntegersOps;;
```

So now we have to add those functions whose result can not (easily) be derived from the knowledge components and that we can not inherit from `RingOps`.

The first such function is the membership test. This function must test whether an object is an element of the domain `GaussianIntegers`. `IsCycInt(`x`)` tests whether x is a cyclotomic integer and `NofCyc(`x`)` returns the smallest n such that the cyclotomic x can be written as a linear combination of powers of the primitive n-th root of unity `E(`n`)`. If `NofCyc(`x`)` returns 1, x is an ordinary rational number.

```
gap> GaussianIntegersOps.\in := function ( x, GaussInt )
>    return IsCycInt( x ) and (NofCyc( x ) = 1 or NofCyc( x ) = 4);
> end;;
```

Note that the second argument `GaussInt` is not used in the function. Whenever this function is called, the second argument must be `GaussianIntegers`, because `GaussianIntegers` is the only domain that has this particular function in its operations record. This also happens for most other functions that we will write. This argument can not be dropped though, because there are other domains that share a common `in` function, for example all permutation groups have the same `in` function. If the operator `in` would not pass the second argument, this function could not know for which permutation group it should perform the membership test.

So now we can test whether a certain object is a Gaussian integer or not.

```
gap> E(4) in GaussianIntegers;
true
gap> 1/2 in GaussianIntegers;
false
gap> GaussianIntegers in GaussianIntegers;
false
```

Another function that is just as easy is the function `Random` that should return a random Gaussian integer.

```
gap> GaussianIntegersOps.Random := function ( GaussInt )
>     return Random( Integers ) + Random( Integers ) * E( 4 );
```

```
> end;;
```

Note that actually a `Random` function was inherited from `RingOps`. But this function can not be used. It tries to construct the sorted list of all elements of the domain and then picks a random element from that list. Therefor this function is only applicable for finite domains, and can not be used for `GaussianIntegers`. So we **overlay** this default function by simply putting another function in the operations record.

Now we can already test whether a Gaussian integer is a unit or not. This is because the default function inherited from `RingOps` tests whether the knowledge component `units` is present, and it returns `true` if the element is in that list and `false` otherwise.

```
gap> IsUnit( GaussianIntegers, E(4) );
true
gap> IsUnit( GaussianIntegers, 1 + E(4) );
false
```

Now we finally come to more interesting stuff. The function `Quotient` should return the quotient of its two arguments x and y. If the quotient does not exist in the ring (i.e., if it is a proper Gaussian rational), it must return `false`. (Without this last requirement we could do without the `Quotient` function and always simply use the `/` operator.)

```
gap> GaussianIntegersOps.Quotient := function ( GaussInt, x, y )
>     local   q;
>     q := x / y;
>     if not IsCycInt( q )  then
>         q := false;
>     fi;
>     return q;
> end;;
```

The next function is used to test if two elements are associate in the ring of Gaussian integers. In fact we need not implement this because the function that we inherit from `RingOps` will do fine. The following function is a little bit faster though that the inherited one.

```
gap> GaussianIntegersOps.IsAssociated := function ( GaussInt, x, y )
>     return x = y  or x = -y  or x = E(4)*y  or x = -E(4)*y;
> end;;
```

We must however implement the function `StandardAssociate`. It should return an associate that is in some way standard. That means, whenever we apply `StandardAssociate` to two associated elements we must obtain the same value. For Gaussian integers we return that associate that lies in the first quadrant of the complex plane. That is, the result is that associated element that has positive real part and nonnegative imaginary part. 0 is its own standard associate of course. Note that this is a generalization of the absolute value function, which is `StandardAssociate` for the integers. The reason that we must implement `StandardAssociate` is of course that there is no general way to compute a standard associate for an arbitrary ring, there is not even a standard way to define this!

```
gap> GaussianIntegersOps.StandardAssociate := function ( GaussInt, x )
>     if   IsRat(x)  and 0 <= x  then
>         return x;
>     elif IsRat(x)  then
```

```
>         return -x;
>     elif 0 <  COEFFSCYC(x)[1]      and 0 <= COEFFSCYC(x)[2]      then
>         return x;
>     elif      COEFFSCYC(x)[1] <= 0 and 0 <  COEFFSCYC(x)[2]      then
>         return - E(4) * x;
>     elif      COEFFSCYC(x)[1] <  0 and      COEFFSCYC(x)[2] <= 0 then
>         return - x;
>     else
>         return E(4) * x;
>     fi;
> end;;
```

Note that `COEFFSCYC` is an internal function that returns the coefficients of a Gaussian integer (actually of an arbitrary cyclotomic) as a list.

Now we have implemented all functions that are necessary to view the Gaussian integers plainly as a ring. Of course there is not much we can do with such a plain ring, we can compute with its elements and can do a few things that are related to the group of units.

```
gap> Quotient( GaussianIntegers, 2, 1+E(4) );
1-E(4)
gap> Quotient( GaussianIntegers, 3, 1+E(4) );
false
gap> IsAssociated( GaussianIntegers, 1+E(4), 1-E(4) );
true
gap> StandardAssociate( GaussianIntegers, 3 - E(4) );
1+3*E(4)
```

The remaining functions are related to the fact that the Gaussian integers are an Euclidean ring (and thus also a unique factorization ring).

The first such function is `EuclideanDegree`. In our example the Euclidean degree of a Gaussian integer is of course simply its norm. Just as with `StandardAssociate` we must implement this function because there is no general way to compute the Euclidean degree for an arbitrary Euclidean ring. The function itself is again very simple. The Euclidean degree of a Gaussian integer x is the product of x with its complex conjugate, which is denoted in GAP3 by `GaloisCyc(` x`, -1 )`.

```
gap> GaussianIntegersOps.EuclideanDegree := function ( GaussInt, x )
>     return x * GaloisCyc( x, -1 );
> end;;
```

Once we have defined the Euclidean degree we want to implement the `QuotientRemainder` function that gives us the Euclidean quotient and remainder of a division.

```
gap> GaussianIntegersOps.QuotientRemainder := function ( GaussInt, x, y )
>     return [ RoundCyc( x/y ),  x - RoundCyc( x/y ) * y ];
> end;;
```

Note that in the definition of `QuotientRemainder` we must use the function `RoundCyc`, which views the Gaussian rational x/y as a point in the complex plane and returns the point of the lattice spanned by 1 and `E(4)` **closest** to the point x/y. If we would truncate towards the origin instead (this is done by the function `IntCyc`) we could not guarantee that the result of EuclideanRemainder always has Euclidean degree less than the Euclidean degree of y as the following example shows.

```
gap> x := 2 - E(4);;  EuclideanDegree( GaussianIntegers, x );
5
gap> y := 2 + E(4);;  EuclideanDegree( GaussianIntegers, y );
5
gap> q := x / y;  q := IntCyc( q );
3/5-4/5*E(4)
0
gap> EuclideanDegree( GaussianIntegers, x - q * y );
5
```

Now that we have implemented the QuotientRemainder function we can compute greatest common divisors in the ring of Gaussian integers. This is because we have inherited from RingOps the general function Gcd that computes the greatest common divisor using Euclid's algorithm, which only uses QuotientRemainder (and StandardAssociate to return the result in a normal form). Of course we can now also compute least common multiples, because that only uses Gcd.

```
gap> Gcd( GaussianIntegers, 2, 5 - E(4) );
1+E(4)
gap> Lcm( GaussianIntegers, 2, 5 - E(4) );
6+4*E(4)
```

Since the Gaussian integers are a Euclidean ring they are also a unique factorization ring. The next two functions implement the necessary operations. The first is the test for primality. A rational integer is a prime in the ring of Gaussian integers if and only if it is congruent to 3 modulo 4 (the other rational integer primes split into two irreducibles), and a Gaussian integer that is not a rational integer is a prime if its norm is a rational integer prime.

```
gap> GaussianIntegersOps.IsPrime := function ( GaussInt, x )
>     if IsInt( x )  then
>         return x mod 4 = 3  and IsPrimeInt( x );
>     else
>         return IsPrimeInt( x * GaloisCyc( x, -1 ) );
>     fi;
> end;;
```

The factorization is based on the same observation. We compute the Euclidean degree of the number that we want to factor, and factor this rational integer. Then for every rational integer prime that is congruent to 3 modulo 4 we get one factor, and we split the other rational integer primes using the function TwoSquares and test which irreducible divides.

```
gap> GaussianIntegersOps.Factors := function ( GaussInt, x )
>     local   facs,       # factors (result)
>             prm,        # prime factors of the norm
>             tsq;        # representation of prm as x^2 + y^2
>
>     # handle trivial cases
>     if x in [ 0, 1, -1, E(4), -E(4) ]  then
```

```
>         return [ x ];
>     fi;
>
>     # loop over all factors of the norm of x
>     facs := [];
>     for prm  in Set( FactorsInt( EuclideanDegree( x ) ) )  do
>
>         # p = 2 and primes p = 1 mod 4 split according to p = x^2+y^2
>         if prm = 2  or prm mod 4 = 1  then
>             tsq := TwoSquares( prm );
>             while IsCycInt( x / (tsq[1]+tsq[2]*E(4)) )  do
>                 Add( facs, (tsq[1]+tsq[2]*E(4)) );
>                 x := x / (tsq[1]+tsq[2]*E(4));
>             od;
>             while IsCycInt( x / (tsq[2]+tsq[1]*E(4)) )  do
>                 Add( facs, (tsq[2]+tsq[1]*E(4)) );
>                 x := x / (tsq[2]+tsq[1]*E(4));
>             od;
>
>         # primes p = 3 mod 4 stay prime
>         else
>             while IsCycInt( x / prm )  do
>                 Add( facs, prm );
>                 x := x / prm;
>             od;
>         fi;
>
>     od;
>
>     # the first factor takes the unit
>     facs[1] := x * facs[1];
>
>     # return the result
>     return facs;
> end;;
```

So now we can factorize numbers in the ring of Gaussian integers.

```
gap> Factors( GaussianIntegers, 10 );
[ -1-E(4), 1+E(4), 1+2*E(4), 2+E(4) ]
gap> Factors( GaussianIntegers, 103 );
[ 103 ]
```

Now we have written all the functions for the operations record that implement the operations. We would like one more thing however. Namely that we can simply write `Gcd( 2, 5 - E(4) )` without having to specify `GaussianIntegers` as first argument. `Gcd` and the other functions should be clever enough to find out that the arguments are Gaussian integers and call `GaussianIntegers.operations.Gcd` automatically.

To do this we must first understand what happens when `Gcd` is called without a ring as first argument. For an example suppose that we have called `Gcd( 66, 123 )` (and want to compute the gcd over the integers).

First `Gcd` calls `DefaultRing( [ 66, 123 ] )`, to obtain a ring that contains 66 and 123. `DefaultRing` then calls `Domain( [ 66, 123 ] )` to obtain a domain, which need not be a ring, that contains 66 and 123. `Domain` is the **only** function in the whole GAP3 library that knows about the various types of elements. So it looks at its argument and decides to return the domain `Integers` (which is in fact already a ring, but it could in principle also return `Rationals`). `DefaultRing` now calls `Integers.operations.DefaultRing( [ 66, 123 ] )` and expects a ring in which the requested gcd computation can be performed. `Integers.operations.DefaultRing( [ 66, 123 ] )` also returns `Integers`. So `DefaultRing` returns `Integers` to `Gcd` and `Gcd` finally calls `Integers.operations.Gcd( Integers, 66, 123 )`.

So the first thing we must do is to tell `Domain` about Gaussian integers. We do this by extending `Domain` with the two lines

```
elif ForAll( elms, IsGaussInt )  then
    return GaussianIntegers;
```

so that it now looks as follows.

```
gap> Domain := function ( elms )
>     local   elm;
>     if   ForAll( elms, IsInt )  then
>         return Integers;
>     elif ForAll( elms, IsRat )  then
>         return Rationals;
>     elif ForAll( elms, IsFFE )  then
>         return FiniteFieldElements;
>     elif ForAll( elms, IsPerm )  then
>         return Permutations;
>     elif ForAll( elms, IsMat )  then
>         return Matrices;
>     elif ForAll( elms, IsWord )  then
>         return Words;
>     elif ForAll( elms, IsAgWord )  then
>         return AgWords;
>     elif ForAll( elms, IsGaussInt )  then
>         return GaussianIntegers;
>     elif ForAll( elms, IsCyc )  then
>         return Cyclotomics;
>     else
>         for elm  in elms do
>             if    IsRec(elm)  and IsBound(elm.domain)
>               and ForAll( elms, l -> l in elm.domain )
>             then
>                 return elm.domain;
>             fi;
>         od;
>         Error("sorry, the elements lie in no common domain");
>     fi;
```

```
> end;;
```

Of course we must define a function `IsGaussInt`, otherwise this could not possibly work. This function is similar to the membership test we already defined above.

```
gap> IsGaussInt := function ( x )
>     return IsCycInt( x ) and (NofCyc( x ) = 1 or NofCyc( x ) = 4);
> end;;
```

Then we must define a function `DefaultRing` for the Gaussian integers that does nothing but return `GaussianIntegers`.

```
gap> GaussianIntegersOps.DefaultRing := function ( elms )
>     return GaussianIntegers;
> end;;
```

Now we can call `Gcd` with two Gaussian integers without having to pass `GaussianIntegers` as first argument.

```
gap> Gcd( 2, 5 - E(4) );
1+E(4)
```

Of course GAP3 can not read your mind. In the following example it assumes that you want to factor 10 over the ring of integers, not over the ring of Gaussian integers (because `Integers` is the default ring containing 10). So if you want to factor a rational integer over the ring of Gaussian integers you must pass `GaussianIntegers` as first argument.

```
gap> Factors( 10 );
[ 2, 5 ]
gap> Factors( GaussianIntegers, 10 );
[ -1-E(4), 1+E(4), 1+2*E(4), 2+E(4) ]
```

This concludes our example. In the file `LIBNAME/"gaussian.g"` you will also find the definition of the field of Gaussian rationals. It is so similar to the above definition that there is no point in discussing it here. The next section shows you what further considerations are necessary when implementing a type of parametrized domains (demonstrated by implementing full symmetric permutation groups). For further details see chapter 14 for a description of the Gaussian integers and rationals and chapter 5 for a list of all functions applicable to rings.

1.29 About Defining New Parametrized Domains

In this section we will show you an example that is slightly more complex than the example in the previous section. Namely we will demonstrate how one can implement parametrized domains. As an example we will implement symmetric permutation groups. This works similar to the implementation of a single domain. Therefore we can be very brief. Of course you should have read the previous section.

Note that everything defined here is already in the file `GRPNAME/"permgrp.grp"`, so there is no need for you to type it in. You may however like to make a copy of this file and modify it.

In the example of the previous section we simply had a variable (`GaussianIntegers`), whose value was the domain. This can not work in this example, because there is not **one** symmetric permutation group. The solution is obvious. We simply define a function that takes the degree and returns the symmetric permutation group of this degree (as a domain).

```
gap> SymmetricPermGroup := function ( n )
>     local   G;          # symmetric group on <n> points, result
>
>     # make the group generated by (1,n), (2,n), .., (n-1,n)
>     G := Group( List( [1..n-1], i -> (i,n) ), () );
>     G.degree := n;
>
>     # give it the correct operations record
>     G.operations := SymmetricPermGroupOps;
>
>     # return the symmetric group
>     return G;
> end;;
```

The key is of course to give the domains returned by `SymmetricPermGroup` a new operations record. This operations record will hold functions that are written especially for symmetric permutation groups. Note that all symmetric groups created by `SymmetricPermGroup` share one operations record.

Just as we inherited in the example in the previous section from the operations record `RingOps`, here we can inherit from the operations record `PermGroupOps` (after all, each symmetric permutation group is also a permutation group).

```
gap> SymmetricPermGroupOps := Copy( PermGroupOps );
```

We will now overlay some of the functions in this operations record with new functions that make use of the fact that the domain is a full symmetric permutation group. The first function that does this is the membership test function.

```
gap> SymmetricPermGroupOps.\in := function ( g, G )
>     return     IsPerm( g )
>            and (   g = ()
>                 or LargestMovedPointPerm( g ) <= G.degree);
> end;;
```

The most important knowledge for a permutation group is a base and a strong generating set with respect to that base. It is not important that you understand at this point what this is mathematically. The important point here is that such a strong generating set with respect to an appropriate base is used by many generic permutation group functions, most of which we inherit for symmetric permutation groups. Therefore it is important that we are able to compute a strong generating set as fast as possible. Luckily it is possible to simply write down such a strong generating set for a full symmetric group. This is done by the following function.

```
gap> SymmetricPermGroupOps.MakeStabChain := function ( G, base )
>     local   sgs,        # strong generating system of G wrt. base
>             last;       # last point of the base
>
>     # remove all unwanted points from the base
>     base := Filtered( base, i -> i <= G.degree );
>
>     # extend the base with those points not already in the base
```

```
>     base := Concatenation( base, Difference( [1..G.degree], base ) );
>
>     # take the last point
>     last := base[ Length(base) ];
>
>     # make the strong generating set
>     sgs := List( [1..Length(base)-1], i -> ( base[i], last ) );
>
>     # make the stabilizer chain
>     MakeStabChainStrongGenerators( G, base, sgs );
> end;;
```

One of the things that are very easy for symmetric groups is the computation of centralizers of elements. The next function does this. Again it is not important that you understand this mathematically. The centralizer of an element g in the symmetric group is generated by the cycles c of g and an element x for each pair of cycles of g of the same length that maps one cycle to the other.

```
gap> SymmetricPermGroupOps.Centralizer := function ( G, g )
>     local   C,      # centralizer of g in G, result
>             sgs,    # strong generating set of C
>             gen,    # one generator in sgs
>             cycles, # cycles of g
>             cycle,  # one cycle from cycles
>             lasts,  # lasts[l] is the last cycle of length l
>             last,   # one cycle from lasts
>             i;      # loop variable
>
>     # handle special case
>     if IsPerm( g )  and g in G  then
>
>         # start with the empty strong generating system
>         sgs := [];
>
>         # compute the cycles and find for each length the last one
>         cycles := Cycles( g, [1..G.degree] );
>         lasts := [];
>         for cycle  in cycles  do
>             lasts[Length(cycle)] := cycle;
>         od;
>
>         # loop over the cycles
>         for cycle  in cycles  do
>
>             # add that cycle itself to the strong generators
>             if Length( cycle ) <> 1  then
>                 gen := [1..G.degree];
>                 for i  in [1..Length(cycle)-1]  do
>                     gen[cycle[i]] := cycle[i+1];
```

```
>                 od;
>                 gen[cycle[Length(cycle)]] := cycle[1];
>                 gen := PermList( gen );
>                 Add( sgs, gen );
>             fi;
>
>             # and it can be mapped to the last cycle of this length
>             if cycle <> lasts[ Length(cycle) ]  then
>                 last := lasts[ Length(cycle) ];
>                 gen := [1..G.degree];
>                 for i  in [1..Length(cycle)]  do
>                     gen[cycle[i]] := last[i];
>                     gen[last[i]] := cycle[i];
>                 od;
>                 gen := PermList( gen );
>                 Add( sgs, gen );
>             fi;
>
>         od;
>
>         # make the centralizer
>         C := Subgroup( G, sgs );
>
>         # make the stabilizer chain
>         MakeStabChainStrongGenerators( C, [1..G.degree], sgs );
>
>     # delegate general case
>     else
>         C := PermGroupOps.Centralizer( G, g );
>     fi;
>
>     # return the centralizer
>     return C;
> end;;
```

Note that the definition `C := Subgroup( G, sgs );` defines a subgroup of a symmetric permutation group. But this subgroup is usually not a full symmetric permutation group itself. Thus `C` must not have the operations record `SymmetricPermGroupOps`, instead it should have the operations record `PermGroupOps`. And indeed `C` will have this operations record. This is because `Subgroup` calls G`.operations.Subgroup`, and we inherited this function from `PermGroupOps`.

Note also that we only handle one special case in the function above. Namely the computation of a centralizer of a single element. This function can also be called to compute the centralizer of a whole subgroup. In this case `SymmetricPermGroupOps.Centralizer` simply **delegates** the problem by calling `PermGroupOps.Centralizer`.

The next function computes the conjugacy classes of elements in a symmetric group. This is very easy, because two elements are conjugated in a symmetric group when they have the same cycle structure. Thus we can simply compute the partitions of the degree, and for each degree we get one conjugacy class.

```
gap> SymmetricPermGroupOps.ConjugacyClasses := function ( G )
>     local   classes,    # conjugacy classes of G, result
>             prt,        # partition of G
>             sum,        # partial sum of the entries in prt
>             rep,        # representative of a conjugacy class of G
>             i;          # loop variable
>
>     # loop over the partitions
>     classes := [];
>     for prt  in Partitions( G.degree )  do
>
>         # compute the representative of the conjugacy class
>         rep := [2..G.degree];
>         sum := 1;
>         for i  in prt  do
>             rep[sum+i-1] := sum;
>             sum := sum + i;
>         od;
>         rep := PermList( rep );
>
>         # add the new class to the list of classes
>         Add( classes, ConjugacyClass( G, rep ) );
>
>     od;
>
>     # return the classes
>     return classes;
> end;;
```

This concludes this example. You have seen that the implementation of a parametrized domain is not much more difficult than the implementation of a single domain. You have also seen how functions that overlay generic functions may delegate problems back to the generic function. The library file for symmetric permutation groups contain some more functions for symmetric permutation groups.

1.30 About Defining New Group Elements

In this section we will show how one can add a new type of group elements to GAP3. A lot of group elements in GAP3 are implemented this way, for example elements of generic factor groups, or elements of generic direct products.

We will use prime residue classes modulo an integer as our example. They have the advantage that the arithmetic is very simple, so that we can concentrate on the implementation without being carried away by mathematical details.

Note that everything we define is already in the library in the file `LIBNAME/"numtheor.g"`, so there is no need for you to type it in. You may however like to make a copy of this file and modify it.

We will represent residue classes by records. This is absolutely typical, all group elements not built into the GAP3 kernel are realized by records.

To distinguish records representing residue classes from other records we require that residue class records have a component with the name `isResidueClass` and the value `true`. We also require that they have a component with the name `isGroupElement` and again the value `true`. Those two components are called the tag components.

Next each residue class record must of course have components that tell us which residue class this record represents. The component with the name `representative` contains the smallest nonnegative element of the residue class. The component with the name `modulus` contains the modulus. Those two components are called the identifying components.

Finally each residue class record must have a component with the name `operations` that contains an appropriate operations record (see below). In this way we can make use of the possibility to define operations for records (see 46.4 and 46.5).

Below is an example of a residue class record.

```
r13mod43 := rec(
    isGroupElement := true,
    isResidueClass := true,
    representative := 13,
    modulus        := 43,
    domain         := GroupElements,
    operations     := ResidueClassOps );
```

The first function that we have to write is very simple. Its only task is to test whether an object is a residue class. It does this by testing for the tag component `isResidueClass`.

```
gap> IsResidueClass := function ( obj )
>     return IsRec( obj )
>            and IsBound( obj.isResidueClass )
>            and obj.isResidueClass;
> end;;
```

Our next function takes a representative and a modulus and constructs a new residue class. Again this is not very difficult.

```
gap> ResidueClass := function ( representative, modulus )
>     local res;
>     res := rec();
>     res.isGroupElement  := true;
>     res.isResidueClass  := true;
>     res.representative  := representative mod modulus;
>     res.modulus         := modulus;
>     res.domain          := GroupElements;
>     res.operations      := ResidueClassOps;
>     return res;
> end;;
```

Now we have to define the operations record for residue classes. Remember that this record contains a function for each binary operation, which is called to evaluate such a binary operation (see 46.4 and 46.5). The operations `=`, `<`, `*`, `/`, `mod`, `^`, `Comm`, and `Order` are the ones that are applicable to all group elements. The meaning of those operations for group elements is described in 7.2 and 7.3.

Luckily we do not have to define everything. Instead we can inherit a lot of those functions from generic group elements. For example, for all group elements g/h should be equivalent to $g*h$`^-1`. So the function for `/` could simply be `function(g,h) return g*h^-1; end`. Note that this function can be applied to all group elements, independently of their type, because all the dependencies are in `*` and `^`.

The operations record `GroupElementOps` contains such functions that can be used by all types of group elements. Note that there is no element that has `GroupElementsOps` as its operations record. This is impossible, because there is for example no generic method to multiply or invert group elements. Thus `GroupElementsOps` is only used to inherit general methods as is done below.

```
gap> ResidueClassOps := Copy( GroupElementOps );;
```

Note that the copy is necessary, otherwise the following assignments would not only change `ResidueClassOps` but also `GroupElementOps`.

The first function we are implementing is the equality comparison. The required operation is described simply enough. `=` should evaluate to `true` if the operands are equal and `false` otherwise. Two residue classes are of course equal if they have the same representative and the same modulus. One complication is that when this function is called either operand may not be a residue class. Of course at least one must be a residue class otherwise this function would not have been called at all.

```
gap> ResidueClassOps.\= := function ( l, r )
>     local   isEql;
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             isEql :=    l.representative = r.representative
>                     and l.modulus        = r.modulus;
>         else
>             isEql := false;
>         fi;
>     else
>         if IsResidueClass( r )  then
>             isEql := false;
>         else
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return isEql;
> end;;
```

Note that the quotes around the equal sign `=` are necessary, otherwise it would not be taken as a record component name, as required, but as the symbol for equality, which must not appear at this place.

Note that we do not have to implement a function for the inequality operator `<>`, because it is in the GAP3 kernel implemented by the equivalence l `<>` r is `not` l `=` r.

The next operation is the comparison. We define that one residue class is smaller than another residue class if either it has a smaller modulus or, if the moduli are equal, it has a smaller representative. We must also implement comparisons with other objects.

```
gap> ResidueClassOps.\< := function ( l, r )
>     local   isLess;
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             isLess :=   l.representative < r.representative
>                     or (l.representative = r.representative
>                         and l.modulus    < r.modulus);
>         else
>             isLess := not IsInt( r ) and not IsRat( r )
>                   and not IsCyc( r ) and not IsPerm( r )
>                   and not IsWord( r ) and not IsAgWord( r );
>         fi;
>     else
>         if IsResidueClass( r )  then
>             isLess :=  IsInt( l ) or IsRat( l )
>                     or IsCyc( l ) or IsPerm( l )
>                     or IsWord( l ) or IsAgWord( l );
>         else
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return isLess;
> end;;
```

The next operation that we must implement is the multiplication *. This function is quite complex because it must handle several different tasks. To make its implementation easier to understand we will start with a very simple–minded one, which only multiplies residue classes, and extend it in the following paragraphs.

```
gap> ResidueClassOps.\* := function ( l, r )
>     local   prd;        # product of l and r, result
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             if l.modulus <> r.modulus  then
>                 Error("<l> and <r> must have the same modulus");
>             fi;
>             prd := ResidueClass(
>                         l.representative * r.representative,
>                         l.modulus );
>         else
>             Error("product of <l> and <r> must be defined");
>         fi;
>     else
>         if IsResidueClass( r )  then
>             Error("product of <l> and <r> must be defined");
>         else
```

```
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return prd;
> end;;
```

This function correctly multiplies residue classes, but there are other products that must be implemented. First every group element can be multiplied with a list of group elements, and the result shall be the list of products (see 7.3 and 27.13). In such a case the above function would only signal an error, which is not acceptable. Therefore we must extend this definition.

```
gap> ResidueClassOps.\* := function ( l, r )
>     local   prd;        # product of l and r, result
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             if l.modulus <> r.modulus  then
>                 Error( "<l> and <r> must have the same modulus" );
>             fi;
>             prd := ResidueClass(
>                         l.representative * r.representative,
>                         l.modulus );
>         elif IsList( r )  then
>             prd := List( r, x -> l * x );
>         else
>             Error("product of <l> and <r> must be defined");
>         fi;
>     elif IsList( l )  then
>         if IsResidueClass( r )  then
>             prd := List( l, x -> x * r );
>         else
>             Error("panic: neither <l> nor <r> is a residue class");
>         fi;
>     else
>         if IsResidueClass( r )  then
>             Error( "product of <l> and <r> must be defined" );
>         else
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return prd;
> end;;
```

This function is almost complete. However it is also allowed to multiply a group element with a subgroup and the result shall be a coset (see 7.86). The operations record of subgroups, which are of course also represented by records (see 7.118), contains a function that constructs such a coset. The problem is that in an expression like *subgroup* `*` *residue-class*, this function is not called. This is because the multiplication function in the operations record of the **right** operand is called if both operands have such a function (see 46.5). Now in the above case both operands have such a function. The left operand *subgroup* has the operations record `GroupOps` (or some refinement thereof), the right operand *residue-class* has the operations record `ResidueClassOps`. Thus `ResidueClassOps.*` is called. But it does not and also should not know how to construct a coset. The solution is simple. The multiplication function for residue classes detects this special case and simply calls the multiplication function of the left operand.

```
gap> ResidueClassOps.\* := function ( l, r )
>     local   prd;        # product of l and r, result
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             if l.modulus <> r.modulus  then
>                 Error( "<l> and <r> must have the same modulus" );
>             fi;
>             prd := ResidueClass(
>                         l.representative * r.representative,
>                         l.modulus );
>         elif IsList( r )  then
>             prd := List( r, x -> l * x );
>         else
>             Error("product of <l> and <r> must be defined");
>         fi;
>     elif IsList( l )  then
>         if IsResidueClass( r )  then
>             prd := List( l, x -> x * r );
>         else
>             Error("panic: neither <l> nor <r> is a residue class");
>         fi;
>     else
>         if IsResidueClass( r )  then
>             if IsRec( l )  and IsBound( l.operations )
>                 and IsBound( l.operations.\* )
>                 and l.operations.\* <> ResidueClassOps.\*
>             then
>                 prd := l.operations.\*( l, r );
>             else
>                 Error("product of <l> and <r> must be defined");
>             fi;
>         else
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return prd;
> end;;
```

Now we are done with the multiplication.

Next is the powering operation `^`. It is not very complicated. The `PowerMod` function (see 5.25) does most of what we need, especially the inversion of elements with the Euclidean algorithm when the exponent is negative. Note however, that the definition of operations (see 7.3) requires that the conjugation is available as power of a residue class by another residue class. This is of course very easy since residue classes form an abelian group.

```
gap> ResidueClassOps.\^ := function ( l, r )
>     local    pow;
>     if IsResidueClass( l )  then
>         if IsResidueClass( r )  then
>             if l.modulus <> r.modulus  then
>                 Error("<l> and <r> must have the same modulus");
>             fi;
>             if GcdInt( r.representative, r.modulus ) <> 1  then
>                 Error("<r> must be invertable");
>             fi;
>             pow := l;
>         elif IsInt( r )  then
>             pow := ResidueClass(
>                         PowerMod( l.representative, r, l.modulus ),
>                         l.modulus );
>         else
>             Error("power of <l> and <r> must be defined");
>         fi;
>     else
>         if IsResidueClass( r )  then
>             Error("power of <l> and <r> must be defined");
>         else
>             Error("panic, neither <l> nor <r> is a residue class");
>         fi;
>     fi;
>     return pow;
> end;;
```

The last function that we have to write is the printing function. This is called to print a residue class. It prints the residue class in the form `ResidueClass(` *representative*`,` *modulus* `)`. It is fairly typical to print objects in such a form. This form has the advantage that it can be read back, resulting in exactly the same element, yet it is very concise.

```
gap> ResidueClassOps.Print := function ( r )
>     Print("ResidueClass( ",r.representative,", ",r.modulus," )");
> end;;
```

Now we are done with the definition of residue classes as group elements. Try them. We can at this point actually create groups of such elements, and compute in them.

However, we are not yet satisfied. There are two problems with the code we have implemented so far. Different people have different opinions about which of those problems is the graver one, but hopefully all agree that we should try to attack those problems.

The first problem is that it is still possible to define objects via `Group` (see 7.9) that are not actually groups.

```
gap> G := Group( ResidueClass(13,43), ResidueClass(13,41) );
Group( ResidueClass( 13, 43 ), ResidueClass( 13, 41 ) )
```

The other problem is that groups of residue classes constructed with the code we have implemented so far are not handled very efficiently. This is because the generic group algorithms are used, since we have not implemented anything else. For example to test whether a residue class lies in a residue class group, all elements of the residue class group are computed by a Dimino algorithm, and then it is tested whether the residue class is an element of this proper set.

To solve the first problem we must first understand what happens with the above code if we create a group with `Group(` *res1* `,` *res2* `...` `)`. `Group` tries to find a domain that contains all the elements *res1*, *res2*, etc. It first calls `Domain( [` *res1* `,` *res2* `... ] )` (see 4.5). `Domain` looks at the residue classes and sees that they all are records and that they all have a component with the name `domain`. This is understood to be a domain in which the elements lie. And in fact *res1* `in GroupElements` is `true`, because `GroupElements` accepts all records with tag `isGroupElement`. So `Domain` returns `GroupElements`. `Group` then calls `GroupElements.operations.Group(GroupElements,[`*res1*`,`*res2*`...],`*id*`)`, where *id* is the identity residue class, obtained by *res1* `^ 0`, and returns the result.

`GroupElementsOps.Group` is the function that actually creates the group. It does this by simply creating a record with its second argument as generators list, its third argument as identity, and the generic `GroupOps` as operations record. It ignores the first argument, which is passed only because convention dictates that a dispatcher passes the domain as first argument.

So to solve the first problem we must achieve that another function instead of the generic function `GroupElementsOps.Group` is called. This can be done by persuading `Domain` to return a different domain. And this will happen if the residue classes hold this other domain in their `domain` component.

The obvious choice for such a domain is the (yet to be written) domain `ResidueClasses`. So `ResidueClass` must be slightly changed.

```
gap> ResidueClass := function ( representative, modulus )
>     local res;
>     res := rec();
>     res.isGroupElement  := true;
>     res.isResidueClass  := true;
>     res.representative  := representative mod modulus;
>     res.modulus         := modulus;
>     res.domain          := ResidueClasses;
>     res.operations      := ResidueClassOps;
>     return res;
> end;;
```

The main purpose of the domain `ResidueClasses` is to construct groups, so there is very little we have to do. And in fact most of that can be inherited from `GroupElements`.

```
gap> ResidueClasses := Copy( GroupElements );;
gap> ResidueClasses.name := "ResidueClasses";;
gap> ResidueClassesOps := Copy( GroupElementsOps );;
gap> ResidueClasses.operations := ResidueClassesOps;;
```

So now we must implement `ResidueClassesOps.Group`, which should check whether the passed elements do in fact form a group. After checking it simply delegates to the generic function `GroupElementsOps.Group` to create the group as before.

```
gap> ResidueClassesOps.Group := function ( ResidueClasses, gens, id )
```

```
>     local   g;          # one generator from gens
>     for g  in gens  do
>         if g.modulus <> id.modulus  then
>             Error("the generators must all have the same modulus");
>         fi;
>         if GcdInt( g.representative, g.modulus ) <> 1  then
>           Error("the generators must all be prime residue classes");
>         fi;
>     od;
>     return GroupElementOps.Group( ResidueClasses, gens, id );
> end;;
```

This solves the first problem. To solve the second problem, i.e., to make operations with residue class groups more efficient, we must extend the function `ResidueClassesOps.Group`. It now enters a new operations record into the group. It also puts the modulus into the group record, so that it is easier to access.

```
gap> ResidueClassesOps.Group := function ( ResidueClasses, gens, id )
>     local   G,          # group G, result
>             gen;        # one generator from gens
>     for gen  in gens  do
>         if gen.modulus <> id.modulus  then
>             Error("the generators must all have the same modulus");
>         fi;
>         if GcdInt( gen.representative, gen.modulus ) <> 1  then
>           Error("the generators must all be prime residue classes");
>         fi;
>     od;
>     G := GroupElementsOps.Group( ResidueClasses, gens, id );
>     G.modulus    := id.modulus;
>     G.operations := ResidueClassGroupOps;
>     return G;
> end;;
```

Of course now we must build such an operations record. Luckily we do not have to implement all functions, because we can inherit a lot of functions from `GroupOps`. This is done by copying `GroupOps` as we have done before for `ResidueClassOps` and `ResidueClassesOps`.

```
gap> ResidueClassGroupOps := Copy( GroupOps );;
```

Now the first function that we must write is the `Subgroup` function to ensure that not only groups constructed by `Group` have the correct operations record, but also subgroups of those groups created by `Subgroup`. As in `Group` we only check the arguments and then leave the work to `GroupOps.Subgroup`.

```
gap> ResidueClassGroupOps.Subgroup := function ( G, gens )
>     local   S,          # subgroup of G, result
>             gen;        # one generator from gens
>     for gen  in gens  do
>         if gen.modulus <> G.modulus  then
>             Error("the generators must all have the same modulus");
>         fi;
```

```
>         if GcdInt( gen.representative, gen.modulus ) <> 1  then
>           Error("the generators must all be prime residue classes");
>         fi;
>     od;
>     S := GroupOps.Subgroup( G, gens );
>     S.modulus    := G.modulus;
>     S.operations := ResidueClassGroupOps;
>     return S;
> end;;
```

The first function that we write especially for residue class groups is `SylowSubgroup`. Since residue class groups are abelian we can compute a Sylow subgroup of such a group by simply taking appropriate powers of the generators.

```
gap> ResidueClassGroupOps.SylowSubgroup := function ( G, p )
>     local   S,          # Sylow subgroup of G, result
>             gen,        # one generator of G
>             ord,        # order of gen
>             gens;       # generators of S
>     gens := [];
>     for gen  in G.generators  do
>         ord := OrderMod( gen.representative, G.modulus );
>         while ord mod p = 0  do ord := ord / p;  od;
>         Add( gens, gen ^ ord );
>     od;
>     S := Subgroup( Parent( G ), gens );
>     return S;
> end;;
```

To allow the other functions that are applicable to residue class groups to work efficiently we now want to make use of the fact that residue class groups are direct products of cyclic groups and that we know what those factors are and how we can project onto those factors.

To do this we write `ResidueClassGroupOps.MakeFactors` that adds the components `facts`, `roots`, `sizes`, and `sgs` to the group record G. This information, detailed below, will enable other functions to work efficiently with such groups. Creating such information is a fairly typical thing, for example for permutation groups the corresponding information is the stabilizer chain computed by `MakeStabChain`.

G`.facts` will be the list of prime power factors of G`.modulus`. Actually this is a little bit more complicated, because the residue class group modulo the largest power q of 2 that divides G`.modulus` need not be cyclic. So if q is a multiple of 4, G`.facts[1]` will be 4, corresponding to the projection of G into $(Z/4Z)^*$ (of size 2), furthermore if q is a multiple of 8, G`.facts[2]` will be q, corresponding to the projection of G into the subgroup generated by 5 in $(Z/qZ)^*$ (of size $q/4$).

G`.roots` will be a list of primitive roots, i.e., of generators of the corresponding factors in G`.facts`. G`.sizes` will be a list of the sizes of the corresponding factors in G`.facts`, i.e., G`.sizes[`i`] = Phi(` G`.facts[`i`] )`. (If G`.modulus` is a multiple of 8, G`.roots[2]` will be 5, and G`.sizes[2]` will be $q/4$.)

Now we can represent each element g of the group G by a list e, called the exponent vector, of the length of G`.facts`, where e`[`i`]` is the logarithm of g`.representative mod`

G.`facts[`*i*`]` with respect to *G*.`roots[`*i*`]`. The multiplication of elements of G corresponds to the componentwise addition of their exponent vectors, where we add modulo *G*.`sizes[`*i*`]` in the i-th component. (Again special consideration are necessary if *G*.`modulus` is divisible by 8.)

Next we compute the exponent vectors of all generators of G, and represent this information as a matrix. Then we bring this matrix into upper triangular form, with an algorithm that is very much like the ordinary Gaussian elimination, modified to account for the different sizes of the components. This upper triangular matrix of exponent vectors is the component *G*.`sgs`. This new matrix obviously still contains the exponent vectors of a generating system of G, but a much nicer one, which allows us to tackle problems one component at a time. (It is not necessary that you fully check this, the important thing here is not the mathematical side.)

```
gap> ResidueClassGroupOps.MakeFactors := function ( G )
>     local   p, q,       # prime factor of modulus and largest power
>             r, s,       # two rows of the standard generating system
>             g,          # extended gcd of leading entries in r, s
>             x, y,       # two entries in r and s
>             i, k, l;    # loop variables
>
>     # find the factors of the direct product
>     G.facts := [];
>     G.roots := [];
>     G.sizes := [];
>     for p  in Set( Factors( G.modulus ) )  do
>         q := p;
>         while G.modulus mod (p*q) = 0  do q := p*q;  od;
>         if q mod 4 = 0  then
>             Add( G.facts, 4 );
>             Add( G.roots, 3 );
>             Add( G.sizes, 2 );
>         fi;
>         if q mod 8 = 0  then
>             Add( G.facts, q );
>             Add( G.roots, 5 );
>             Add( G.sizes, q/4 );
>         fi;
>         if p <> 2  then
>             Add( G.facts, q );
>             Add( G.roots, PrimitiveRootMod( q ) );
>             Add( G.sizes, (p-1)*q/p );
>         fi;
>     od;
>
>     # represent each generator in this factorization
>     G.sgs := [];
>     for k  in [ 1 .. Length( G.generators ) ]  do
>         G.sgs[k] := [];
```

```
>         for i  in [ 1 .. Length( G.facts ) ]  do
>             if G.facts[i] mod 8 = 0  then
>                 if G.generators[k].representative mod 4 = 1  then
>                     G.sgs[k][i] := LogMod(
>                         G.generators[k].representative,
>                         G.roots[i], G.facts[i] );
>                 else
>                     G.sgs[k][i] := LogMod(
>                         -G.generators[k].representative,
>                         G.roots[i], G.facts[i] );
>                 fi;
>             else
>                 G.sgs[k][i] := LogMod(
>                         G.generators[k].representative,
>                         G.roots[i], G.facts[i] );
>             fi;
>         od;
>     od;
>     for i  in [ Length( G.sgs ) + 1 .. Length( G.facts ) ]  do
>         G.sgs[i] := 0 * G.facts;
>     od;
>
>     # bring this matrix to diagonal form
>     for i  in [ 1 .. Length( G.facts ) ]  do
>         r := G.sgs[i];
>         for k  in [ i+1 .. Length( G.sgs ) ]  do
>             s := G.sgs[k];
>             g := Gcdex( r[i], s[i] );
>             for l  in [ i .. Length( r ) ]  do
>                 x := r[l];  y := s[l];
>                 r[l] := (g.coeff1 * x + g.coeff2 * y) mod G.sizes[l];
>                 s[l] := (g.coeff3 * x + g.coeff4 * y) mod G.sizes[l];
>             od;
>         od;
>         s := [];
>         x := G.sizes[i] / GcdInt( G.sizes[i], r[i] );
>         for l  in [ 1 .. Length( r ) ]  do
>             s[l] := (x * r[l]) mod G.sizes[l];
>         od;
>         Add( G.sgs, s );
>     od;
>
> end;;
```

With the information computed by `MakeFactors` it is now of course very easy to compute the size of a residue class group. We just look at the G`.sgs`, and multiply the orders of the leading exponents of the nonzero exponent vectors.

```
gap> ResidueClassGroupOps.Size := function ( G )
```

```
>     local   s,          # size of G, result
>             i;          # loop variable
>     if not IsBound( G.facts )  then
>         G.operations.MakeFactors( G );
>     fi;
>     s := 1;
>     for i  in [ 1 .. Length( G.facts ) ]  do
>         s := s * G.sizes[i] / GcdInt( G.sizes[i], G.sgs[i][i] );
>     od;
>     return s;
> end;;
```

The membership test is a little bit more complicated. First we test that the first argument is really a residue class with the correct modulus. Then we compute the exponent vector of this residue class and reduce this exponent vector using the upper triangular matrix G.sgs.

```
gap> ResidueClassGroupOps.\in := function ( res, G )
>     local   s,          # exponent vector of res
>             g,          # extended gcd
>             x, y,       # two entries in s and G.sgs[i]
>             i, l;       # loop variables
>     if not IsResidueClass( res )
>         or res.modulus <> G.modulus
>         or GcdInt( res.representative, res.modulus ) <> 1
>     then
>         return false;
>     fi;
>     if not IsBound( G.facts )  then
>         G.operations.MakeFactors( G );
>     fi;
>     s := [];
>     for i  in [ 1 .. Length( G.facts ) ]  do
>         if G.facts[i] mod 8 = 0  then
>             if res.representative mod 4 = 1  then
>                 s[i] := LogMod( res.representative,
>                                 G.roots[i], G.facts[i] );
>             else
>                 s[i] := LogMod( -res.representative,
>                                 G.roots[i], G.facts[i] );
>             fi;
>         else
>             s[i] := LogMod( res.representative,
>                             G.roots[i], G.facts[i] );
>         fi;
>     od;
>     for i  in [ 1 .. Length( G.facts ) ]  do
>         if s[i] mod GcdInt( G.sizes[i], G.sgs[i][i] ) <> 0  then
>             return false;
>         fi;
```

```
>         g := Gcdex( s[i], G.sgs[i][i] );
>         for l  in [ i .. Length( G.facts ) ]  do
>             x := s[l];  y := G.sgs[i][l];
>             s[l] := (g.coeff3 * x + g.coeff4 * y) mod G.sizes[l];
>         od;
>     od;
>     return true;
> end;;
```

We also add a function `Random` that works by creating a random exponent as a random linear combination of the exponent vectors in G`.sgs`, and converts this exponent vector to a residue class. (The main purpose of this function is to allow you to create random test examples for the other functions.)

```
gap> ResidueClassGroupOps.Random := function ( G )
>     local   s,          # exponent vector of random element
>             r,          # vector of remainders in each factor
>             i, k, l;    # loop variables
>     if not IsBound( G.facts )  then
>         G.operations.MakeFactors( G );
>     fi;
>     s := 0 * G.facts;
>     for i  in [ 1 .. Length( G.facts ) ]  do
>         l := G.sizes[i] / GcdInt( G.sizes[i], G.sgs[i][i] );
>         k := Random( [ 0 .. l-1 ] );
>         for l  in [ i .. Length( s ) ]  do
>             s[l] := (s[l] + k * G.sgs[i][l]) mod G.sizes[l];
>         od;
>     od;
>     r := [];
>     for l  in [ 1 .. Length( s ) ]  do
>         r[l] := PowerModInt( G.roots[l], s[l], G.facts[l] );
>         if G.facts[l] mod 8 = 0  and r[1] = 3  then
>             r[l] := G.facts[l] - r[l];
>         fi;
>     od;
>     return ResidueClass( ChineseRem( G.facts, r ), G.modulus );
> end;;
```

There are a lot more functions that would benefit from being implemented especially for residue class groups. We do not show them here, because the above functions already displayed how such functions can be written.

To round things up, we finally add a function that constructs the full residue class group given a modulus m. This function is totally independent of the implementation of residue classes and residue class groups. It only has to find a (minimal) system of generators of the full prime residue classes group, and to call `Group` to construct this group. It also adds the information entry `size` to the group record, of course with the value $\phi(n)$.

```
gap> PrimeResidueClassGroup := function ( m )
>     local   G,          # group Z/mZ, result
```

```
>           gens,       # generators of G
>           p, q,       # prime and prime power dividing m
>           r,          # primitive root modulo q
>           g;          # is = r mod q and = 1 mod m/q
>
>     # add generators for each prime power factor q of m
>     gens := [];
>     for p  in Set( Factors( m ) )  do
>         q := p;
>         while m mod (q * p) = 0  do q := q * p;  od;
>
>         # (Z/4Z)^* = < 3 >
>         if   q = 4  then
>             r := 3;
>             g := r + q * (((1/q mod (m/q)) * (1 - r)) mod (m/q));
>             Add( gens, ResidueClass( g, m ) );
>
>         # (Z/8nZ)^* = < 5, -1 > is not cyclic
>         elif q mod 8 = 0  then
>             r := q-1;
>             g := r + q * (((1/q mod (m/q)) * (1 - r)) mod (m/q));
>             Add( gens, ResidueClass( g, m ) );
>             r := 5;
>             g := r + q * (((1/q mod (m/q)) * (1 - r)) mod (m/q));
>             Add( gens, ResidueClass( g, m ) );
>
>         # for odd q, (Z/qZ)^* is cyclic
>         elif q <> 2  then
>             r :=  PrimitiveRootMod( q );
>             g := r + q * (((1/q mod (m/q)) * (1 - r)) mod (m/q));
>             Add( gens, ResidueClass( g, m ) );
>         fi;
>
>     od;
>
>     # return the group generated by gens
>     G := Group( gens, ResidueClass( 1, m ) );
>     G.size := Phi( n );
>     return G;
> end;;
```

There is one more thing that we can learn from this example. Mathematically a residue class is not only a group element, but a set as well. We can reflect this in GAP3 by turning residue classes into domains (see 4). Section 1.28 gives an example of how to implement a new domain, so we will here only show the code with few comments.

First we must change the function that constructs a residue class, so that it enters the necessary fields to tag this record as a domain. It also adds the information that residue classes are infinite.

```
gap> ResidueClass := function ( representative, modulus )
>     local res;
>     res := rec();
>     res.isGroupElement  := true;
>     res.isDomain        := true;
>     res.isResidueClass  := true;
>     res.representative  := representative mod modulus;
>     res.modulus         := modulus;
>     res.isFinite        := false;
>     res.size            := "infinity";
>     res.domain          := ResidueClasses;
>     res.operations      := ResidueClassOps;
>     return res;
> end;;
```

The initialization of the ResidueClassOps record must be changed too, because now we want to inherit both from GroupElementsOps and DomainOps. This is done by the function MergedRecord, which takes two records and returns a new record that contains all components from either record.

Note that the record returned by MergedRecord does not have those components that appear in both arguments. This forces us to explicitly write down from which record we want to inherit those functions, or to define them anew. In our example the components common to GroupElementOps and DomainOps are only the equality and ordering functions, which we have to define anyhow. (This solution for the problem of which definition to choose in the case of multiple inheritance is also taken by C++.)

With this function definition we can now initialize ResidueClassOps.

```
gap> ResidueClassOps := MergedRecord( GroupElementOps, DomainOps );;
```

Now we add all functions to this record as described above.

Next we add a function to the operations record that tests whether a certain object is in a residue class.

```
gap> ResidueClassOps.\in := function ( element, class )
>     if IsInt( element )  then
>         return (element mod class.modulus = class.representative);
>     else
>         return false;
>     fi;
> end;;
```

Finally we add a function to compute the intersection of two residue classes.

```
gap> ResidueClassOps.Intersection := function ( R, S )
>     local   I,          # intersection of R and S, result
>             gcd;        # gcd of the moduli
>     if IsResidueClass( R )  then
>         if IsResidueClass( S )  then
>             gcd := GcdInt( R.modulus, S.modulus );
>             if     R.representative mod gcd
>                 <> S.representative mod gcd
```

```
>             then
>                 I := [];
>             else
>                 I := ResidueClass(
>                         ChineseRem(
>                             [ R.modulus,        S.modulus ] ,
>                             [ R.representative, S.representative ]),
>                         Lcm(  R.modulus,        S.modulus ) );
>             fi;
>         else
>             I := DomainOps.Intersection( R, S );
>         fi;
>     else
>         I := DomainOps.Intersection( R, S );
>     fi;
>     return I;
> end;;
```

There is one further thing that we have to do. When `Group` is called with a single argument that is a domain, it assumes that you want to create a new group such that there is a bijection between the original domain and the new group. This is not what we want here. We want that in this case we get the cyclic group that is generated by the single residue class. (This overloading of `Group` is probably a mistake, but so is the overloading of residue classes, which are both group elements and domains.) The following definition solves this problem.

```
gap> ResidueClassOps.Group := function ( R )
>     return ResidueClassesOps.Group( ResidueClasses, [R], R^0 );
> end;;
```

This concludes our example. There are however several further things that you could do. One is to add functions for the quotient, the modulus, etc. Another is to fix the functions so that they do not hang if asked for the residue class group mod 1. Also you might try to implement residue class rings analogous to residue class groups. Finally it might be worthwhile to improve the speed of the multiplication of prime residue classes. This can be done by doing some precomputation in `ResidueClass` and adding some information to the residue class record for prime residue classes ([Mon85]).

Chapter 2

The Programming Language

This chapter describes the GAP3 programming language. It should allow you in principle to predict the result of each and every input. In order to know what we are talking about, we first have to look more closely at the process of interpretation and the various representations of data involved.

First we have the input to GAP3, given as a string of characters. How those characters enter GAP3 is operating system dependent, e.g., they might be entered at a terminal, pasted with a mouse into a window, or read from a file. The mechanism does not matter. This representation of expressions by characters is called the **external representation** of the expression. Every expression has at least one external representation that can be entered to get exactly this expression.

The input, i.e., the external representation, is transformed in a process called **reading** to an internal representation. At this point the input is analyzed and inputs that are not legal external representations, according to the rules given below, are rejected as errors. Those rules are usually called the **syntax** of a programming language.

The internal representation created by reading is called either an **expression** or a **statement**. Later we will distinguish between those two terms, however now we will use them interchangeably. The exact form of the internal representation does not matter. It could be a string of characters equal to the external representation, in which case the reading would only need to check for errors. It could be a series of machine instructions for the processor on which GAP3 is running, in which case the reading would more appropriately be called compilation. It is in fact a tree–like structure.

After the input has been read it is again transformed in a process called **evaluation** or **execution**. Later we will distinguish between those two terms too, but for the moment we will use them interchangeably. The name hints at the nature of this process, it replaces an expression with the value of the expression. This works recursively, i.e., to evaluate an expression first the subexpressions are evaluated and then the value of the expression is computed according to rules given below from those values. Those rules are usually called the **semantics** of a programming language.

The result of the evaluation is, not surprisingly, called a **value**. The set of values is of course a much smaller set than the set of expressions; for every value there are several expressions that will evaluate to this value. Again the form in which such a value is represented internally does not matter. It is in fact a tree–like structure again.

The last process is called **printing**. It takes the value produced by the evaluation and creates an external representation, i.e., a string of characters again. What you do with this external representation is up to you. You can look at it, paste it with the mouse into another window, or write it to a file.

Lets look at an example to make this more clear. Suppose you type in the following string of 8 characters

```
1 + 2 * 3;
```

GAP3 takes this external representation and creates a tree like internal representation, which we can picture as follows

```
    +
   / \
  1   *
     / \
    2   3
```

This expression is then evaluated. To do this GAP3 first evaluates the right subexpression `2*3`. Again to do this GAP3 first evaluates its subexpressions 2 and 3. However they are so simple that they are their own value, we say that they are self–evaluating. After this has been done, the rule for `*` tells us that the value is the product of the values of the two subexpressions, which in this case is clearly 6. Combining this with the value of the left operand of the `+`, which is self–evaluating too gives us the value of the whole expression 7. This is then printed, i.e., converted into the external representation consisting of the single character `7`.

In this fashion we can predict the result of every input when we know the syntactic rules that govern the process of reading and the semantic rules that tell us for every expression how its value is computed in terms of the values of the subexpressions. The syntactic rules are given in sections 2.1, 2.2, 2.3, 2.4, 2.5, and 2.20, the semantic rules are given in sections 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15, 2.16, 2.17, 2.18, and the chapters describing the individual data types.

2.1 Lexical Structure

The input of GAP3 consists of sequences of the following characters.

Digits, uppercase and lowercase letters, *space*, *tab*, *newline*, and the special characters

```
"    '    (    )    *    +    ,    _
.    /    :    ;    <    =    >    ~
[    \    ]    ^    _    {    }    #
```

Other characters will be signalled as illegal. Inside strings and comments the full character set supported by the computer is allowed.

2.2 Language Symbols

The process of reading, i.e., of assembling the input into expressions, has a subprocess, called **scanning**, that assembles the characters into symbols. A **symbol** is a sequence of characters that form a lexical unit. The set of symbols consists of keywords, identifiers, strings, integers, and operator and delimiter symbols.

A keyword is a reserved word consisting entirely of lowercase letters (see 2.4). An identifier is a sequence of letters and digits that contains at least one letter and is not a keyword (see 2.5). An integer is a sequence of digits (see 10). A string is a sequence of arbitrary characters enclosed in double quotes (see 30).

Operator and delimiter symbols are

```
+       -       *       /       ^       ~
=       <>      <       <=      >       >=
:=      .       ..      ->      ,       ;
[       ]       {       }       (       )
```

Note that during the process of scanning also all whitespace is removed (see 2.3).

2.3 Whitespaces

The characters *space*, *tab*, *newline*, and *return* are called **whitespace characters**. Whitespace is used as necessary to separate lexical symbols, such as integers, identifiers, or keywords. For example `Thorondor` is a single identifier, while `Th or ondor` is the keyword `or` between the two identifiers `Th` and `ondor`. Whitespace may occur between any two symbols, but not within a symbol. Two or more adjacent whitespaces are equivalent to a single whitespace. Apart from the role as separator of symbols, whitespaces are otherwise insignificant. Whitespaces may also occur inside a string, where they are significant. Whitespaces should also be used freely for improved readability.

A **comment** starts with the character `#`, which is sometimes called sharp or hatch, and continues to the end of the line on which the comment character appears. The whole comment, including `#` and the *newline* character is treated as a single whitespace. Inside a string, the comment character `#` looses its role and is just an ordinary character.

For example, the following statement

```
if i<0 then a:=-i;else a:=i;fi;
```

is equivalent to

```
if i < 0  then      # if i is negative
    a := -i;        #     take its inverse
else                # otherwise
    a := i;         #     take itself
fi;
```

(which by the way shows that it is possible to write superfluous comments). However the first statement is not equivalent to

```
ifi<0thena:=-i;elsea:=i;fi;
```

since the keyword `if` must be separated from the identifier `i` by a whitespace, and similarly `then` and `a`, and `else` and `a` must be separated.

2.4 Keywords

Keywords are reserved words that are used to denote special operations or are part of statements. They must not be used as identifiers. The keywords are

```
and      do       elif     else     end      fi
for      function if       in       local    mod
not      od       or       repeat   return   then
until    while    quit
```

Note that all keywords are written in lowercase. For example only `else` is a keyword; `Else`, `eLsE`, `ELSE` and so forth are ordinary identifiers. Keywords must not contain whitespace, for example `el if` is not the same as `elif`.

2.5 Identifiers

An identifier is used to refer to a variable (see 2.7). An identifier consists of letters, digits, and underscores _, and must contain at least one letter or underscore. An identifier is terminated by the first character not in this class. Examples of valid identifiers are

```
a                   foo                 aLongIdentifier
hello               Hello               HELLO
x100                100x                _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
```

Note that case is significant, so the three identifiers in the second line are distinguished.

The backslash `\` can be used to include other characters in identifiers; a backslash followed by a character is equivalent to the character, except that this escape sequence is considered to be an ordinary letter. For example `G\(2\,5\)` is an identifier, not a call to a function `G`.

An identifier that starts with a backslash is never a keyword, so for example `\*` and `\mod` are identifier.

The length of identifiers is not limited, however only the first 1023 characters are significant. The escape sequence *newline* is ignored, making it possible to split long identifiers over multiple lines.

2.6 Expressions

An **expression** is a construct that evaluates to a value. Syntactic constructs that are executed to produce a side effect and return no value are called **statements** (see 2.11). Expressions appear as right hand sides of assignments (see 2.12), as actual arguments in function calls (see 2.8), and in statements.

Note that an expression is not the same as a value. For example `1 + 11` is an expression, whose value is the integer 12. The external representation of this integer is the character sequence `12`, i.e., this sequence is output if the integer is printed. This sequence is another expression whose value is the integer 12. The process of finding the value of an expression is done by the interpreter and is called the **evaluation** of the expression.

Variables, function calls, and integer, permutation, string, function, list, and record literals (see 2.7, 2.8, 10, 20, 30, 2.18, 27, 46), are the simplest cases of expressions.

Expressions, for example the simple expressions mentioned above, can be combined with the operators to form more complex expressions. Of course those expressions can then be combined further with the operators to form even more complex expressions. The **operators** fall into three classes. The **comparisons** are `=`, `<>`, `<=`, `>`, `>=`, and `in` (see 2.9 and 27.14). The **arithmetic operators** are `+`, `-`, `*`, `/`, `mod`, and `^` (see 2.10). The **logical operators** are `not`, `and`, and `or` (see 45.2).

```
gap> 2 * 2;      # a very simple expression with value
4
gap> 2 * 2 + 9 = Fibonacci(7) and  Fibonacci(13) in Primes;
true              # a more complex expression
```

2.7 Variables

A **variable** is a location in a GAP3 program that points to a value. We say the variable is **bound** to this value. If a variable is evaluated it evaluates to this value.

Initially an ordinary variable is not bound to any value. The variable can be bound to a value by **assigning** this value to the variable (see 2.12). Because of this we sometimes say that a variable that is not bound to any value has no assigned value. Assignment is in fact the only way by which a variable, which is not an argument of a function, can be bound to a value. After a variable has been bound to a value an assignment can also be used to bind the variable to another value.

A special class of variables are **arguments** of functions. They behave similarly to other variables, except they are bound to the value of the actual arguments upon a function call (see 2.8).

Each variable has a name that is also called its **identifier**. This is because in a given scope an identifier identifies a unique variable (see 2.5). A **scope** is a lexical part of a program text. There is the global scope that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A local scope introduces new variables, whose identifiers are given in the formal argument list and the `local` declaration of the function (see 2.18). Usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name. Because this mapping from identifiers to variables is done when the program is read, not when it is executed, GAP3 is said to have lexical scoping. The following example shows how one identifier refers to different variables at different points in the program text.

```
g := 0;           # global variable g
x := function ( a, b, c )
   local   y;
   g := c;        # c refers to argument c of function x
   y := function ( y )
       local  d, e, f;
       d := y;    # y refers to argument y of function y
       e := b;    # b refers to argument b of function x
       f := g;    # g refers to global variable g
       return d + e + f;
   end;
```

```
        return y( a );  # y refers to local y of function x
    end;
```

It is important to note that the concept of a variable in GAP3 is quite different from the concept of a variable in programming languages like PASCAL. In those languages a variable denotes a block of memory. The value of the variable is stored in this block. So in those languages two variables can have the same value, but they can never have identical values, because they denote different blocks of memory. (Note that PASCAL has the concept of a reference argument. It seems as if such an argument and the variable used in the actual function call have the same value, since changing the argument's value also changes the value of the variable used in the actual function call. But this is not so; the reference argument is actually a pointer to the variable used in the actual function call, and it is the compiler that inserts enough magic to make the pointer invisible.) In order for this to work the compiler needs enough information to compute the amount of memory needed for each variable in a program, which is readily available in the declarations PASCAL requires for every variable.

In GAP3 on the other hand each variable justs points to a value.

2.8 Function Calls

function-var`()`
function-var`(` *arg-expr* {`,` *arg-expr*} `)`

The function call has the effect of calling the function *function-var*. The precise semantics are as follows.

First GAP3 evaluates the *function-var*. Usually *function-var* is a variable, and GAP3 does nothing more than taking the value of this variable. It is allowed though that *function-var* is a more complex expression, namely it can for example be a selection of a list element *list-var*`[`*int-expr*`]`, or a selection of a record component *record-var*`.`*ident*. In any case GAP3 tests whether the value is a function. If it is not, GAP3 signals an error.

Next GAP3 checks that the number of actual arguments *arg-expr*s agrees with the number of formal arguments as given in the function definition. If they do not agree GAP3 signals an error. An exception is the case when there is exactly one formal argument with the name `arg`, in which case any number of actual arguments is allowed.

Now GAP3 allocates for each formal argument and for each formal local a new variable. Remember that a variable is a location in a GAP3 program that points to a value. Thus for each formal argument and for each formal local such a location is allocated.

Next the arguments *arg-expr*s are evaluated, and the values are assigned to the newly created variables corresponding to the formal arguments. Of course the first value is assigned to the new variable corresponding to the first formal argument, the second value is assigned to the new variable corresponding to the second formal argument, and so on. However, GAP3 does not make any guarantee about the order in which the arguments are evaluated. They might be evaluated left to right, right to left, or in any other order, but each argument is evaluated once. An exception again occurs if the function has only one formal argument with the name `arg`. In this case the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument `arg`.

The new variables corresponding to the formal locals are initially not bound to any value. So trying to evaluate those variables before something has been assigned to them will signal an error.

Now the body of the function, which is a statement, is executed. If the identifier of one of the formal arguments or formal locals appears in the body of the function it refers to the new variable that was allocated for this formal argument or formal local, and evaluates to the value of this variable.

If during the execution of the body of the function a `return` statement with an expression (see 2.19) is executed, execution of the body is terminated and the value of the function call is the value of the expression of the `return`. If during the execution of the body a `return` statement without an expression is executed, execution of the body is terminated and the function call does not produce a value, in which case we call this call a procedure call (see 2.13). If the execution of the body completes without execution of a `return` statement, the function call again produces no value, and again we talk about a procedure call.

```
gap> Fibonacci( 11 );
    # a call to the function Fibonacci with actual argument 11
89

gap> G.operations.RightCosets( G, Intersection( U, V ) );;
    # a call to the function in G.operations.RightCosets
    # where the second actual argument is another function call
```

2.9 Comparisons

left-expr `=` *right-expr*
left-expr `<>` *right-expr*

The operator `=` tests for equality of its two operands and evaluates to `true` if they are equal and to `false` otherwise. Likewise `<>` tests for inequality of its two operands. Note that any two objects can be compared, i.e., `=` and `<>` will never signal an error. For each type of objects the definition of equality is given in the respective chapter. Objects of different types are never equal, i.e., `=` evaluates in this case to `false`, and `<>` evaluates to `true`.

left-expr `<` *right-expr*
left-expr `>` *right-expr*
left-expr `<=` *right-expr*
left-expr `>=` *right-expr*

`<` denotes less than, `<=` less than or equal, `>` greater than, and `>=` greater than or equal of its two operands. For each type of objects the definition of the ordering is given in the respective chapter. The ordering of objects of different types is as follows. Rationals are smallest, next are cyclotomics, followed by finite field elements, permutations, words, words in solvable groups, boolean values, functions, lists, and records are largest.

Comparison operators, which includes the operator `in` (see 27.14) are not associative, i.e., it is not allowed to write *a* `=` *b* `<>` *c* `=` *d*, you must use `(`*a* `=` *b*`) <> (`*c* `=` *d*`)` instead. The comparison operators have higher precedence than the logical operators (see 45.2), but lower precedence than the arithmetic operators (see 2.10). Thus, for example, *a* `*` *b* `=` *c* `and` *d* is interpreted, `((`*a* `*` *b*`) =` *c*`) and` *d*`)`.

```
gap> 2 * 2 + 9 = Fibonacci(7);     # a comparison where the left
true                                # operand is an expression
```

2.10 Operations

`+` *right-expr*
`-` *right-expr*
left-expr `+` *right-expr*
left-expr `-` *right-expr*
left-expr `*` *right-expr*
left-expr `/` *right-expr*
left-expr `mod` *right-expr*
left-expr `^` *right-expr*

The arithmetic operators are `+`, `-`, `*`, `/`, `mod`, and `^`. The meanings (semantic) of those operators generally depend on the types of the operands involved, and they are defined in the various chapters describing the types. However basically the meanings are as follows.

`+` denotes the addition, and `-` the subtraction of ring and field elements. `*` is the multiplication of group elements, `/` is the multiplication of the left operand with the inverse of the right operand. `mod` is only defined for integers and rationals and denotes the modulo operation. `+` and `-` can also be used as unary operations. The unary `+` is ignored and unary `-` is equivalent to multiplication by -1. `^` denotes powering of a group element if the right operand is an integer, and is also used to denote operation if the right operand is a group element.

The **precedence** of those operators is as follows. The powering operator `^` has the highest precedence, followed by the unary operators `+` and `-`, which are followed by the multiplicative operators `*`, `/`, and `mod`, and the additive binary operators `+` and `-` have the lowest precedence. That means that the expression `-2 ^ -2 * 3 + 1` is interpreted as `(-(2 ^ (-2)) * 3) + 1`. If in doubt use parentheses to clarify your intention.

The **associativity** of the arithmetic operators is as follows.`^` is not associative, i.e., it is illegal to write `2^3^4`, use parentheses to clarify whether you mean `(2^3) ^ 4` or `2 ^ (3^4)`. The unary operators `+` and `-` are right associative, because they are written to the left of their operands. `*`, `/`, `mod`, `+`, and `-` are all left associative, i.e., `1-2-3` is interpreted as `(1-2)-3` not as `1-(2-3)`. Again, if in doubt use parentheses to clarify your intentions.

The arithmetic operators have higher precedence than the comparison operators (see 2.9 and 27.14) and the logical operators (see 45.2). Thus, for example, *a* `*` *b* `=` *c* `and` *d* is interpreted, `((`*a* `*` *b*`)` `=` *c*`)` `and` *d*.

```
gap> 2 * 2 + 9;    # a very simple arithmetic expression
13
```

2.11 Statements

Assignments (see 2.12), Procedure calls (see 2.13), `if` statements (see 2.14), `while` (see 2.15), `repeat` (see 2.16) and `for` loops (see 2.17), and the `return` statement (see 2.19) are called statements. They can be entered interactively or be part of a function definition. Every statement must be terminated by a semicolon.

Statements, unlike expressions, have no value. They are executed only to produce an effect. For example an assignment has the effect of assigning a value to a variable, a `for` loop has the effect of executing a statement sequence for all elements in a list and so on. We will talk about **evaluation** of expressions but about **execution** of statements to emphasize this difference.

It is possible to use expressions as statements. However this does cause a warning.

```
gap> if i <> 0  then  k = 16/i;  fi;
Syntax error: warning, this statement has no effect
if i <> 0  then  k = 16/i;  fi;
                          ^
```

As you can see from the example this is useful for those users who are used to languages where `=` instead of `:=` denotes assignment.

A sequence of one or more statements is a statement sequence, and may occur everywhere instead of a single statement. There is nothing like PASCAL's BEGIN-END, instead each construct is terminated by a keyword. The most simple statement sequence is a single semicolon, which can be used as an empty statement sequence.

2.12 Assignments

var `:=` *expr*`;`

The **assignment** has the effect of assigning the value of the expressions *expr* to the variable *var*.

The variable *var* may be an ordinary variable (see 2.7), a list element selection *list-var*`[`*int-expr*`]` (see 27.6) or a record component selection *record-var*`.`*ident* (see 46.2). Since a list element or a record component may itself be a list or a record the left hand side of an assignment may be arbitrarily complex.

Note that variables do not have a type. Thus any value may be assigned to any variable. For example a variable with an integer value may be assigned a permutation or a list or anything else.

If the expression *expr* is a function call then this function must return a value. If the function does not return a value an error is signalled and you enter a break loop (see 3.2). As usual you can leave the break loop with `quit;`. If you enter `return` *return-expr*`;` the value of the expression *return-expr* is assigned to the variable, and execution continues after the assignment.

```
gap> S6 := rec( size := 720 );; S6;
rec(
  size := 720 )
gap> S6.generators := [ (1,2), (1,2,3,4,5) ];; S6;
rec(
  size := 720,
  generators := [ (1,2), (1,2,3,4,5) ] )
gap> S6.generators[2] := (1,2,3,4,5,6);; S6;
rec(
  size := 720,
  generators := [ (1,2), (1,2,3,4,5,6) ] )
```

2.13 Procedure Calls

procedure-var`();`
procedure-var`(` *arg-expr* {`,` *arg-expr*} `);`

The procedure call has the effect of calling the procedure *procedure-var*. A procedure call is done exactly like a function call (see 2.8). The distinction between functions and procedures is only for the sake of the discussion, GAP3 does not distinguish between them.

A **function** does return a value but does not produce a side effect. As a convention the name of a function is a noun, denoting what the function returns, e.g., `Length`, `Concatenation` and `Order`.

A **procedure** is a function that does not return a value but produces some effect. Procedures are called only for this effect. As a convention the name of a procedure is a verb, denoting what the procedure does, e.g., `Print`, `Append` and `Sort`.

```
gap> Read( "myfile.g" );     # a call to the procedure Read
gap> l := [ 1, 2 ];;
gap> Append( l, [3,4,5] );    # a call to the procedure Append
```

2.14 If

`if` *bool-expr1* `then` *statements1*
{ `elif` *bool-expr2* `then` *statements2* }
`[ else` *statements3* `]`
`fi;`

The `if` statement allows one to execute statements depending on the value of some boolean expression. The execution is done as follows.

First the expression *bool-expr1* following the `if` is evaluated. If it evaluates to `true` the statement sequence *statements1* after the first `then` is executed, and the execution of the `if` statement is complete.

Otherwise the expressions *bool-expr2* following the `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. As soon as an expression evaluates to `true` the corresponding statement sequence *statements2* is executed and execution of the `if` statement is complete.

If the `if` expression and all, if any, `elif` expressions evaluate to `false` and there is an `else` part, which is optional, its statement sequence *statements3* is executed and the execution of the `if` statement is complete. If there is no `else` part the `if` statement is complete without executing any statement sequence.

Since the `if` statement is terminated by the `fi` keyword there is no question where an `else` part belongs, i.e., GAP3 has no dangling else.
In `if` *expr1* `then if` *expr2* `then` *stats1* `else` *stats2* `fi; fi;`
the `else` part belongs to the second `if` statement, whereas in
`if` *expr1* `then if` *expr2* `then` *stats1* `fi; else` *stats2* `fi;`
the `else` part belongs to the first `if` statement.

Since an if statement is not an expression it is not possible to write

```
abs := if x  > 0  then  x;  else  -x;  fi;
```

which would, even if legal syntax, be meaningless, since the `if` statement does not produce a value that could be assigned to `abs`.

If one expression evaluates neither to `true` nor to `false` an error is signalled and a break loop (see 3.2) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `true`. Likewise, if you enter `return false;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `false`.

```
gap> i := 10;;
gap> if 0 < i  then
>        s := 1;
>    elif i < 0  then
>        s := -1;
>    else
>        s := 0;
>    fi;
gap> s;
1         #  the sign of i
```

2.15 While

`while` *bool-expr* `do` *statements* `od;`

The `while` loop executes the statement sequence *statements* while the condition *bool-expr* evaluates to `true`.

First *bool-expr* is evaluated. If it evaluates to `false` execution of the `while` loop terminates and the statement immediately following the `while` loop is executed next. Otherwise if it evaluates to `true` the *statements* are executed and the whole process begins again.

The difference between the `while` loop and the `repeat until` loop (see 2.16) is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` an error is signalled and a break loop (see 3.2) is entered. As usual you can leave the break loop with `quit;`. If you enter `return false;`, execution continues with the next statement immediately following the `while` loop. If you enter `return true;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

```
gap> i := 0;;  s := 0;;
gap> while s <= 200  do
>        i := i + 1;  s := s + i^2;
>    od;
gap> s;
204        #  first sum of the first i squares larger than 200
```

2.16 Repeat

`repeat` *statements* `until` *bool-expr*`;`

The `repeat` loop executes the statement sequence *statements* until the condition *bool-expr* evaluates to `true`.

First *statements* are executed. Then *bool-expr* is evaluated. If it evaluates to `true` the `repeat` loop terminates and the statement immediately following the `repeat` loop is executed next. Otherwise if it evaluates to `false` the whole process begins again with the execution of the *statements*.

The difference between the `while` loop (see 2.15) and the `repeat until` loop is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` a error is signalled and a break loop (see 3.2) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution continues with the next statement immediately following the `repeat` loop. If you enter `return false;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

```
gap> i := 0;;  s := 0;;
gap> repeat
>        i := i + 1;  s := s + i^2;
>    until s > 200;
gap> s;
204         # first sum of the first i squares larger than 200
```

2.17 For

`for` *simple-var* `in` *list-expr* `do` *statements* `od;`

The `for` loop executes the statement sequence *statements* for every element of the list *list-expr*.

The statement sequence *statements* is first executed with *simple-var* bound to the first element of the list *list*, then with *simple-var* bound to the second element of *list* and so on. *simple-var* must be a simple variable, it must not be a list element selection *list-var*`[`*int-expr*`]` or a record component selection *record-var*`.`*ident*.

The execution of the `for` loop is exactly equivalent to the `while` loop

```
loop-list := list;
loop-index := 1;
while loop-index <= Length(loop-list) do
    variable := loop-list[loop-index];
    statements
    loop-index := loop-index + 1;
od;
```

with the exception that *loop-list* and *loop-index* are different variables for each `for` loop that do not interfere with each other.

The list *list* is very often a range.
`for` *variable* `in [`*from*`..`*to*`] do` *statements* `od;`
corresponds to the more common
`for` *variable* `from` *from* `to` *to* `do` *statements* `od;`
in other programming languages.

```
gap> s := 0;;
```

```
gap> for i  in [1..100]  do
>        s := s + i;
> od;
gap> s;
5050
```

Note in the following example how the modification of the **list** in the loop body causes the loop body also to be executed for the new values

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i  in l  do
>        Print( i, " " );
>        if i mod 2 = 0  then Add( l, 3 * i / 2 );  fi;
> od;  Print( "\n" );
1 2 3 4 5 6 3 6 9 9
gap> l;
[ 1, 2, 3, 4, 5, 6, 3, 6, 9, 9 ]
```

Note in the following example that the modification of the **variable** that holds the list has no influence on the loop

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i  in l  do
>        Print( i, " " );
>        l := [];
> od;  Print( "\n" );
1 2 3 4 5 6
gap> l;
[  ]
```

2.18 Functions

`function` ([*arg-ident* {, *arg-ident*}])
    [`local` *loc-ident* {, *loc-ident*} ;]
    statements
`end`

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component. Later this function can be called as described in 2.8.

The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence (see 47.22)

```
gap> fib := function ( n )
>         local  f1,  f2,  f3,  i;
>         f1 := 1;  f2 := 1;
>         for i  in [3..n]  do
>             f3 := f1 + f2;
>             f1 := f2;
>             f2 := f3;
>         od;
```

```
>         return f2;
>     end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Because for each of the formal arguments *arg-ident* and for each of the formal locals *loc-ident* a new variable is allocated when the function is called (see 2.8), it is possible that a function calls itself. This is usually called **recursion**. The following is a recursive function that computes values of the Fibonacci sequence

```
gap> fib := function ( n )
>         if n < 3  then
>             return 1;
>         else
>             return fib(n-1) + fib(n-2);
>         fi;
>     end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Note that the recursive version needs `2 * fib(`n`)-1` steps to compute `fib(`n`)`, while the iterative version of `fib` needs only n`-2` steps. Both are not optimal however, the library function `Fibonacci` only needs on the order of `Log(`n`)` steps.

arg-ident `->` *expr*

This is a shorthand for
`function (` *arg-ident* `) return` *expr*`; end`.
arg-ident must be a single identifier, i.e., it is not possible to write functions of several arguments this way. Also `arg` is not treated specially, so it is also impossible to write functions that take a variable number of arguments this way.

The following is an example of a typical use of such a function

```
gap> Sum( List( [1..100], x -> x^2 ) );
338350
```

When a function *fun1* definition is evaluated inside another function *fun2*, GAP3 binds all the identifiers inside the function *fun1* that are identifiers of an argument or a local of *fun2* to the corresponding variable. This set of bindings is called the environment of the function *fun1*. When *fun1* is called, its body is executed in this environment. The following implementation of a simple stack uses this. Values can be pushed onto the stack and then later be popped off again. The interesting thing here is that the functions `push` and `pop` in the record returned by `Stack` access the local variable `stack` of `Stack`. When `Stack` is called a new variable for the identifier `stack` is created. When the function definitions of `push` and `pop` are then evaluated (as part of the `return` statement) each reference to `stack` is bound to this new variable. Note also that the two stacks `A` and `B` do not interfere, because each call of `Stack` creates a new variable for `stack`.

```
gap> Stack := function ()
>         local   stack;
>         stack := [];
>         return rec(
>             push := function ( value )
```

```
>                 Add( stack, value );
>             end,
>             pop := function ()
>                 local   value;
>                 value := stack[Length(stack)];
>                 Unbind( stack[Length(stack)] );
>                 return value;
>             end
>         );
>     end;;
gap> A := Stack();;
gap> B := Stack();;
gap> A.push( 1 );  A.push( 2 );  A.push( 3 );
gap> B.push( 4 );  B.push( 5 );  B.push( 6 );
gap> A.pop();  A.pop();  A.pop();
3
2
1
gap> B.pop();  B.pop();  B.pop();
6
5
4
```

This feature should be used rarely, since its implementation in GAP3 is not very efficient.

2.19 Return

`return;`

In this form `return` terminates the call of the innermost function that is currently executing, and control returns to the calling function. An error is signalled if no function is currently executing. No value is returned by the function.

`return` *expr*`;`

In this form `return` terminates the call of the innermost function that is currently executing, and returns the value of the expression *expr*. Control returns to the calling function. An error is signalled if no function is currently executing.

Both statements can also be used in break loops (see 3.2). `return;` has the effect that the computation continues where it was interrupted by an error or the user hitting *ctr*`C`. `return` *expr*`;` can be used to continue execution after an error. What happens with the value *expr* depends on the particular error.

2.20 The Syntax in BNF

This section contains the definition of the GAP3 syntax in Backus-Naur form.

A BNF is a set of rules, whose left side is the name of a syntactical construct. Those names are enclosed in angle brackets and written in *italics*. The right side of each rule contains a possible form for that syntactic construct. Each right side may contain names of other syntactic constructs, again enclosed in angle brackets and written in *italics*, or character sequences that must occur literally; they are written in `typewriter style`.

Furthermore each righthand side can contain the following metasymbols written in **boldface**. If the right hand side contains forms separated by a pipe symbol (|) this means that one of the possible forms can occur. If a part of a form is enclosed in square brackets ([ ]) this means that this part is optional, i.e. might be present or missing. If part of the form is enclosed in curly braces ({ }) this means that the part may occur arbitrarily often, or possibly be missing.

```
Ident       := a|...|z|A|...|Z|_ {a|...|z|A|...|Z|0|...|9|_}
Var         := Ident
            |  Var . Ident
            |  Var . ( Expr )
            |  Var [ Expr ]
            |  Var { Expr }
            |  Var ( [ Expr { , Expr } ] )
List        := [ [ Expr ] {, [ Expr ] } ]
            |  [ Expr [, Expr ] .. Expr ]
Record      := rec( [ Ident := Expr {, Ident := Expr } ] )
Permutation := ( Expr {, Expr } ) { ( Expr {, Expr } ) }
Function    := function ( [ Ident {, Ident } ] )
               [ local Ident {, Ident } ; ]
               Statements
               end
Char        := ' any character '
String      := " { any character } "
Int         := 0|1|...|9 { 0|1|...|9 }
Atom        := Int
            |  Var
            |  ( Expr )
            |  Permutation
            |  Char
            |  String
            |  Function
            |  List
            |  Record
Factor      := {+|-} Atom [ ^ {+|-} Atom ]
Term        := Factor { *|/|mod Factor }
Arith       := Term { +|- Term }
Rel         := { not } Arith { =|<>|<|>|<=|>=|in Arith }
And         := Rel { and Rel }
Log         := And { or And }
Expr        := Log
            |  Var [ -> Log ]
Statement   := Expr
            |  Var := Expr
            |  if Expr then Statements
               { elif Expr then Statements }
               [ else            Statements ] fi
            |  for Var in Expr do Statements od
            |  while Expr do Statements od
            |  repeat Statements until Expr
            |  return [ Expr ]
            |  quit
Statements  := { Statement ; }
            |  ;
```

Chapter 3

Environment

This chapter describes the interactive environment in which you use GAP3.

The first sections describe the main read eval print loop and the break loop (see 3.1, 3.2, and 3.3).

The next section describes the commands you can use to edit the current input line (see 3.4).

The next sections describe the GAP3 help system (see 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11).

The next sections describe the input and output functions (see 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.18, and 3.19).

The next sections describe the functions that allow you to collect statistics about a computation (see 3.20, 3.21).

The last sections describe the functions that allow you to execute other programs as subprocesses from within GAP3 (see 3.22 and 3.23).

3.1 Main Loop

The normal interaction with GAP3 happens in the so–called **read eval print** loop. This means that you type an input, GAP3 first reads it, evaluates it, and prints the result. The exact sequence is as follows.

To show you that it is ready to accept your input, GAP3 displays the **prompt** `gap>` . When you see this, you know that GAP3 is waiting for your input.

Note that every statement must be terminated by a semicolon. You must also enter *return* before GAP3 starts to read and evaluate your input. Because GAP3 does not do anything until you enter *return*, you can edit your input to fix typos and only when everything is correct enter *return* and have GAP3 take a look at it (see 3.4). It is also possible to enter several statements as input on a single line. Of course each statement must be terminated by a semicolon.

It is absolutely acceptable to enter a single statement on several lines. When you have entered the beginning of a statement, but the statement is not yet complete, and you enter *return*, GAP3 will display the **partial prompt** `>` . When you see this, you know that GAP3 is waiting for the rest of the statement. This happens also when you forget the semicolon `;` that terminates every GAP3 statement.

When you enter *return*, GAP3 first checks your input to see if it is syntactically correct (see chapter 2 for the definition of syntactically correct). If it is not, GAP3 prints an error message of the following form

```
gap> 1 * ;
Syntax error: expression expected
1 * ;
    ^
```

The first line tells you what is wrong about the input, in this case the `*` operator takes two expressions as operands, so obviously the right one is missing. If the input came from a file (see 3.12), this line will also contain the filename and the line number. The second line is a copy of the input. And the third line contains a caret pointing to the place in the previous line where GAP3 realized that something is wrong. This need not be the exact place where the error is, but it is usually quite close.

Sometimes, you will also see a partial prompt after you have entered an input that is syntactically incorrect. This is because GAP3 is so confused by your input, that it thinks that there is still something to follow. In this case you should enter `;`*return* repeatedly, ignoring further error messages, until you see the full prompt again. When you see the full prompt, you know that GAP3 forgave you and is now ready to accept your next – hopefully correct – input.

If your input is syntactically correct, GAP3 evaluates or executes it, i.e., performs the required computations (see chapter 2 for the definition of the evaluation).

If you do not see a prompt, you know that GAP3 is still working on your last input. Of course, you can **type ahead**, i.e., already start entering new input, but it will not be accepted by GAP3 until GAP3 has completed the ongoing computation.

When GAP3 is ready it will usually print the result of the computation, i.e., the value computed. Note that not all statements produce a value, for example, if you enter a `for` loop, nothing will be printed, because the `for` loop does not produce a value that could be printed.

Also sometimes you do not want to see the result. For example if you have computed a value and now want to assign the result to a variable, you probably do not want to see the value again. You can terminate statements by **two** semicolons to suppress the printing of the result.

If you have entered several statements on a single line GAP3 will first read, evaluate, and print the first one, then read evaluate, and print the second one, and so on. This means that the second statement will not even be checked for syntactical correctness until GAP3 has completed the first computation.

After the result has been printed GAP3 will display another prompt, and wait for your next input. And the whole process starts all over again. Note that a new prompt will only be printed after GAP3 has read, evaluated, and printed the last statement if you have entered several statements on a single line.

In each statement that you enter the result of the previous statement that produced a value is available in the variable `last`. The next to previous result is available in `last2` and the result produced before that is available in `last3`.

```
gap> 1; 2; 3;
1
2
3
gap> last3 + last2 * last;
7
```

Also in each statement the time spent by the last statement, whether it produced a value or not, is available in the variable `time`. This is an integer that holds the number of milliseconds.

3.2 Break Loops

When an error has occurred or when you interrupt GAP3, usually by hitting *ctr*-C, GAP3 enters a break loop, that is in most respects like the main read eval print loop (see 3.1). That is, you can enter statements, GAP3 reads them, evaluates them, and prints the result if any. However those evaluations happen within the context in which the error occurred. So you can look at the arguments and local variables of the functions that were active when the error happened and even change them. The prompt is changed from `gap>` to `brk>` to indicate that you are in a break loop.

There are two ways to leave a break loop.

The first is to quit the break loop and continue in the main loop. To do this you enter `quit;` or hit the *eof* (**e**nd **o**f **f**ile) character, which is usually *ctr*-D. In this case control returns to the main loop, and you can enter new statements.

The other way is to return from a break loop. To do this you enter `return;` or `return` *expr*`;`. If the break loop was entered because you interrupted GAP3, then you can continue by entering `return;`. If the break loop was entered due to an error, you usually have to return a value to continue the computation. For example, if the break loop was entered because a variable had no assigned value, you must return the value that this variable should have to continue the computation.

3.3 Error

`Error(` *messages*... `)`

`Error` signals an error. First the messages *messages* are printed, this is done exactly as if `Print` (see 3.14) were called with these arguments. Then a break loop (see 3.2) is entered, unless the standard error output is not connected to a terminal. You can leave this break loop with `return;` to continue execution with the statement following the call to `Error`.

3.4 Line Editing

GAP3 allows you to edit the current input line with a number of editing commands. Those commands are accessible either as **control keys** or as **escape keys**. You enter a control key by pressing the *ctr* key, and, while still holding the *ctr* key down, hitting another key *key*. You enter an escape key by hitting *esc* and then hitting another key *key*. Below we denote control keys by *ctr*-*key* and escape keys by *esc*-*key*. The case of *key* does not matter, i.e., *ctr*-A and *ctr*-a are equivalent.

Characters not mentioned below always insert themselves at the current cursor position.

The first few commands allow you to move the cursor on the current line.

ctr-`A` move the cursor to the beginning of the line.
esc-`B` move the cursor to the beginning of the previous word.
ctr-`B` move the cursor backward one character.
ctr-`F` move the cursor forward one character.
esc-`F` move the cursor to the end of the next word.
ctr-`E` move the cursor to the end of the line.

The next commands delete or kill text. The last killed text can be reinserted, possibly at a different position with the yank command.

ctr-`H` or *del* delete the character left of the cursor.
ctr-`D` delete the character under the cursor.
ctr-`K` kill up to the end of the line.
esc-`D` kill forward to the end of the next word.
esc-*del* kill backward to the beginning of the last word.
ctr-`X` kill entire input line, and discard all pending input.
ctr-`Y` insert (yank) a just killed text.

The next commands allow you to change the input.

ctr-`T` exchange (twiddle) current and previous character.
esc-`U` uppercase next word.
esc-`L` lowercase next word.
esc-`C` capitalize next word.

The *tab* character, which is in fact the control key *ctr*-`I`, looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier. If there is more than one possible completion, it completes to the longest common prefix of all those completions. If the characters to the left of the cursor are already the longest common prefix of all completions hitting *tab* a second time will display all possible completions.

tab complete the identifier before the cursor.

The next commands allow you to fetch previous lines, e.g., to correct typos, etc. This history is limited to about 8000 characters.

ctr-`L` insert last input line before current character.
ctr-`P` redisplay the last input line, another *ctr*-`P` will redisplay the line before that, etc. If the cursor is not in the first column only the lines starting with the string to the left of the cursor are taken.
ctr-`N` Like *ctr*-`P` but goes the other way round through the history.
esc-`<` goes to the beginning of the history.
esc-`>` goes to the end of the history.
ctr-`O` accepts this line and perform a *ctr*-`N`.

Finally there are a few miscellaneous commands.

ctr-`V` enter next character literally, i.e., enter it even if it is one of the control keys.
ctr-`U` execute the next command 4 times.
esc-*num* execute the next command *num* times.
esc-*ctr*-`L` repaint input line.

3.5 Help

This section describes together with the following sections the GAP3 help system. The help system lets you read the manual interactively.

?*section*

The help command `?` displays the section with the name *section* on the screen. For example `?Help` will display this section on the screen. You should not type in the single quotes, they are only used in help sections to delimit text that you should enter into GAP3 or that GAP3 prints in response. When the whole section has been displayed the normal GAP3 prompt `gap>` is shown and normal GAP3 interaction resumes.

The section 3.6 tells you what actions you can perform while you are reading a section. You command GAP3 to display this section by entering `?Reading Sections`, without quotes. The section 3.7 describes the format of sections and the conventions used, 3.8 lists the commands you use to flip through sections, 3.9 describes how to read a section again, 3.10 tells you how to avoid typing the long section names, and 3.11 describes the index command.

3.6 Reading Sections

If the section is longer than 24 lines GAP3 stops after 24 lines and displays

```
-- <space> for more --
```

If you press *space* GAP3 displays the next 24 lines of the section and then stops again. This goes on until the whole section has been displayed, at which point GAP3 will return immediately to the main GAP3 loop. Pressing `f` has the same effect as *space*.

You can also press `b` or the key labeled *del* which will scroll back to the **previous** 24 lines of the section. If you press `b` or *del* when GAP3 is displaying the top of a section GAP3 will ring the bell.

You can also press `q` to quit and return immediately back to the main GAP3 loop without reading the rest of the section.

Actually the 24 is only a default, if you have a larger screen that can display more lines of text you may want to tell this to GAP3 with the `-y` *rows* option when you start GAP3.

3.7 Format of Sections

This section describes the format of sections when they are displayed on the screen and the special conventions used.

As you can see GAP3 indents sections 4 spaces and prints a header line containing the name of the section on the left and the name of the chapter on the right.

`<text>`

Text enclosed in angle brackets is used for arguments in the descriptions of functions and for other placeholders. It means that you should not actually enter this text into GAP3 but replace it by an appropriate text depending on what you want to do. For example when we write that you should enter ?*section* to see the section with the name *section*, *section* servers as a placeholder, indicating that you can enter the name of the section that you want to see at this place. In the printed manual such text is printed in italics.

`'text'`

Text enclosed in single quotes is used for names of variables and functions and other text that you may actually enter into your computer and see on your screen. The text enclosed in single quotes may contain placeholders enclosed in angle brackets as described above. For example when the help text for `IsPrime` says that the form of the call is `'IsPrime( <n> )'` this means that you should actually enter the IsPrime(and), without the quotes, but replace the n with the number (or expression) that you want to test. In the printed manual this text is printed in a monospaced (all characters have the same width) typewriter font.

`"text"`

Text enclosed in double quotes is used for cross references to other parts of the manual. So the text inside the double quotes is the name of another section of the manual. This is used to direct you to other sections that describe a topic or a function used in this section. So for example 3.10 is a cross reference to the next section. In the printed manual the text is replaced by the number of the section.

`_` and `^`

In mathematical formulas the underscore and the caret are used to denote subscription and superscription. Ordinarily they apply only to the very next character following, unless a whole expression enclosed in parentheses follows. So for example `x_1^(i+1)` denotes the variable `x` with subscript 1 raised to the `i+1` power. In the printed manual mathematical formulas are typeset in italics (actually mathitalics) and subscripts and superscripts are actually lowered and raised.

Longer examples are usually paragraphs of their own that are indented 8 spaces from the left margin, i.e. 4 spaces further than the surrounding text. Everything on the lines with the prompts `gap>` and `>`, except the prompts themselves of course, is the input you have to type, everything else is GAP3's response. In the printed manual examples are also indented 4 spaces and are printed in a monospaced typewriter font.

```
gap> ?Format of Sections
Format of Sections ______________________________________ Environment

This section describes the format of sections when they are displayed
on the screen and the special conventions used.

...
```

3.8 Browsing through the Sections

The help sections are organized like a book into chapters. This should not surprise you, since the same source is used both for the printed manual and the online help. Just as you can flip through the pages of a book there are special commands to browse through the help sections.

`?>`
`?<`

The two help commands `?<` and `?>` correspond to the flipping of pages. `?<` takes you to the section preceding the current section and displays it, and `?>` takes you to the section following the current section.

`?<<`
`?>>`

`?<<` is like `?<`, only more so. It takes you back to the first section of the current chapter, which gives an overview of the sections described in this chapter. If you are already in this section `?<<` takes you to the first section of the previous chapter. `?>>` takes you to the first section of the next chapter.

`?-`
`?+`

GAP3 remembers the sections that you have read. `?-` takes you to the one that you have read before the current one, and displays it again. Further `?-` takes you further back in this history. `?+` reverses this process, i.e., it takes you back to the section that you have read **after** the current one. It is important to note, that `?-` and `?+` do **not** alter the history like the other help commands.

3.9 Redisplaying a Section

`?`

The help command `?` followed by no section name redisplays the last help section again. So if you reach the bottom of a long help section and already forgot what was mentioned at the beginning, or, for example, the examples do not seem to agree with your interpretation of the explanations, use `?` to read the whole section again from the beginning.

When `?` is used before any section has been read GAP3 displays the section `Welcome to GAP`.

3.10 Abbreviating Section Names

Upper and lower case in *section* are not distinguished, so typing either `?Abbreviating Section Names` or `?abbreviating section names` will show this very section.

Each word in *section* may be abbreviated. So instead of typing `?abbreviating section names` you may also type `?abb sec nam`, or even `?a s n`. You must not omit the spaces separating the words. For each word in the section name you must give at least the first character. As another example you may type `?oper for int` instead of `?operations for integers`, which is especially handy when you can not remember whether it was `operations` or `operators`.

If an abbreviation matches multiple section names a list of all these section names is displayed.

3.11 Help Index

`??`*topic*

`??` looks up *topic* in GAP3's index and prints all the index entries that contain the substring *topic*. Then you can decide which section is the one you are actually interested in and request this one.

```
gap> ??help
    help ______________________________________________________ Index
```

```
    Help
    Reading Sections (help!scrolling)
    Format of the Sections (help!format)
    Browsing through the Sections (help!browsing)
    Redisplaying a Section (help!redisplaying)
    Abbreviating Section Names (help!abbreviating)
    Help Index
gap>
```

The first thing on each line is the name of the section. If the name of the section matches *topic* nothing more is printed. Otherwise the index entry that matched *topic* is printed in parentheses following the section name. For each section only the first matching index entry is printed. The order of the sections corresponds to their order in the GAP3 manual, so that related sections should be adjacent.

3.12 Read

`Read(` *filename* `)`

`Read` reads the input from the file with the filename *filename*, which must be a string.

`Read` first opens the file *filename*. If the file does not exist, or if GAP3 can not open it, e.g., because of access restrictions, an error is signalled.

Then the contents of the file are read and evaluated, but the results are not printed. The reading and printing happens exactly as described for the main loop (see 3.1).

If an input in the file contains a syntactical error, a message is printed, and the rest of this statement is ignored, but the rest of the file is read.

If a statement in the file causes an error a break loop is entered (see 3.2). The input for this break loop is not taken from the file, but from the input connected to the `stderr` output of GAP3. If `stderr` is not connected to a terminal, no break loop is entered. If this break loop is left with `quit` (or *ctr*-`D`) the file is closed and GAP3 does not continue to read from it.

Note that a statement may not begin in one file and end in another, i.e., *eof* (**e**nd **o**f **f**ile) is not treated as whitespace, but as a special symbol that must not appear inside any statement.

Note that one file may very well contain a read statement causing another file to be read, before input is again taken from the first file. There is an operating system dependent maximum on the number of files that may be open at once, usually it is 15.

The special file name `"*stdin*"` denotes the standard input, i.e., the stream through which the user enters commands to GAP3. The exact behaviour of `Read( "*stdin*")` is operating system dependent, but usually the following happens. If GAP3 was started with no input redirection, statements are read from the terminal stream until the user enters the end of file character, which is usually *ctr*-`D`. Note that terminal streams are special, in that they may yield ordinary input **after** an end of file. Thus when control returns to the main read eval print loop the user can continue with GAP3. If GAP3 was started with an input redirection, statements are read from the current position in the input file up to the end of the file. When control returns to the main read eval print loop the input stream will still return end of file, and GAP3 will terminate. The special file name `"*errin*"` denotes the stream connected with the `stderr` output. This stream is usually connected to the terminal, even if the standard input was redirected, unless the standard error stream was also redirected, in which case opening of `"*errin*"` fails, and `Read` will signal an error.

`Read` is implemented in terms of the function `READ`, which behaves exactly like `Read`, except that `READ` does not signal an error when it can not open the file. Instead it returns `true` or `false` to indicate whether opening the file was successful or not.

3.13 ReadLib

`ReadLib(` *name* `)`

`ReadLib` reads input from the library file with the name *name*. `ReadLib` prefixes *name* with the value of the variable `LIBNAME` and appends the string `".g"` and calls `Read` (see 3.12) with this file name.

3.14 Print

`Print(` *obj1* `,` *obj2* `...` `)`

`Print` prints the objects *obj1*, *obj2*... etc. to the standard output. The output looks exactly like the printed representation of the objects printed by the main loop. The exception are strings, which are printed without the enclosing quotes and a few other transformations (see 30). Note that no space or newline is printed between the objects. `PrintTo` can be used to print to a file (see 3.15).

```
gap> for i  in [1..5]  do
>     Print( i, " ", i^2, " ", i^3, "\n" );
> od;
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

3.15 PrintTo

`PrintTo(` *filename* `,` *obj1* `,` *obj2* `...` `)`

`PrintTo` works like `Print`, except that the output is printed to the file with the name *filename* instead of the standard output. This file must of course be writable by GAP3, otherwise an error is signalled. Note that `PrintTo` will overwrite the previous contents of this file if it already existed. `AppendTo` can be used to append to a file (see 3.16).

The special file name `"*stdout*"` can be used to print to the standard output. This is equivalent to a plain `Print`, except that a plain `Print` that is executed while evaluating an argument to a `PrintTo` call will also print to the output file opened by the last `PrintTo` call, while `PrintTo( "*stdout*",` *obj1* `,` *obj2* `...` `)` always prints to the standard output. The special file name `"*errout*"` can be used to print to the standard error output file, which is usually connected with the terminal, even if the standard output was redirected.

There is an operating system dependent maximum to the number of output files that may be open at once, usually this is 14.

3.16 AppendTo

`AppendTo(` *filename*`,` *obj1*`,` *obj2*`...` `)`

`AppendTo` works like `PrintTo` (see 3.15), except that the output does not overwrite the previous contents of the file, but is appended to the file.

3.17 LogTo

`LogTo(` *filename* `)`

`LogTo` causes the subsequent interaction to be logged to the file with the name *filename*, i.e., everything you see on your terminal will also appear in this file. This file must of course be writable by GAP3, otherwise an error is signalled. Note that `LogTo` will overwrite the previous contents of this file if it already existed.

`LogTo()`

In this form `LogTo` stops logging again.

3.18 LogInputTo

`LogInputTo(` *filename* `)`

`LogInputTo` causes the subsequent input lines to be logged to the file with the name *filename*, i.e., every line you type will also appear in this file. This file must of course be writable by GAP3, otherwise an error is signalled. Note that `LogInputTo` will overwrite the previous contents of this file if it already existed.

`LogInputTo()`

In this form `LogInputTo` stops logging again.

3.19 SizeScreen

`SizeScreen()`

In this form `SizeScreen` returns the size of the screen as a list with two entries. The first is the length of each line, the second is the number of lines.

`SizeScreen( [` x`,` y `] )`

In this form `SizeScreen` sets the size of the screen. x is the length of each line, y is the number of lines. Either value may be missing, to leave this value unaffected. Note that those parameters can also be set with the command line options `-x` x and `-y` y (see 56).

3.20 Runtime

`Runtime()`

`Runtime` returns the time spent by GAP3 in milliseconds as an integer. This is usually the cpu time, i.e., not the wall clock time. Also time spent by subprocesses of GAP3 (see 3.22) is not counted.

3.21 Profile

`Profile( true )`

In this form `Profile` turns the profiling on. Subsequent computations will record the time spent by each function and the number of times each function was called. Old profiling information is cleared.

`Profile( false )`

In this form `Profile` turns the profiling off again. Recorded information is still kept, so you can display it even after turning the profiling off.

`Profile()`

In this form `Profile` displays the collected information in the following format.

```
gap> Factors( 10^21+1 );;      # make sure that the library is loaded
gap> Profile( true );
gap> Factors( 10^42+1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
gap> Profile( false );
gap> Profile();
 count    time percent time/call child function
     4    1811      76      452    2324  FactorsRho
    18     171       7        9     237  PowerModInt
   127      94       3        0      94  GcdInt
    41      83       3        2     415  IsPrimeInt
    91      59       2        0      59  TraceModQF
   511      47       1        0      39  QuoInt
    22      23       0        1      23  Jacobi
   116      20       0        0      31  log
     3      20       0        6      70  SmallestRootInt
     1      19       0       19    2370  FactorsInt
    26      15       0        0      39  LogInt
     4       4       0        1       4  Concatenation
     5       4       0        0      20  RootInt
     7       0       0        0       0  Add
    26       0       0        0       0  Length
    13       0       0        0       0  NextPrimeInt
     4       0       0        0       0  AddSet
     4       0       0        0       0  IsList
     4       0       0        0       0  Sort
     8       0       0        0       0  Append
          2369     100                   TOTAL
```

The last column contains the name of the function. The first column contains the number of times each function was called. The second column contains the time spent in this function. The third column contains the percentage of the total time spent in this function. The fourth column contains the time per call, i.e., the quotient of the second by the first number. The fifth column contains the time spent in this function and all other functions called, directly or indirectly, by this function.

3.22 Exec

`Exec( ` *command* ` )`

`Exec` executes the command given by the string *command* in the operating system. How this happens is operating system dependent. Under UNIX, for example, a new shell is started and *command* is passed as a command to this shell.

```
gap> Exec( "date" );
Fri Dec 13 17:00:29 MET 1991
```

`Edit` (see 3.23) should be used to call an editor from within GAP3.

3.23 Edit

`Edit( ` *filename* ` )`

`Edit` starts an editor with the file whose filename is given by the string *filename*, and reads the file back into GAP3 when you exit the editor again. You should set the GAP3 variable `EDITOR` to the name of the editor that you usually use, e.g., `/usr/ucb/vi`. This can for example be done in your `.gaprc` file (see the sections on operating system dependent features in chapter 56).

Chapter 4

Domains

Domain is GAP3's name for structured sets. The ring of Gaussian integers $Z[I]$ is an example of a domain, the group D_{12} of symmetries of a regular hexahedron is another.

The GAP3 library predefines some domains. For example the ring of Gaussian integers is predefined as `GaussianIntegers` (see 14) and the field of rationals is predefined as `Rationals` (see 12). Most domains are constructed by functions, which are called **domain constructors**. For example the group D_{12} is constructed by the construction `Group( (1,2,3,4,5,6), (2,6)(3,5) )` (see 7.9) and the finite field with 16 elements is constructed by `GaloisField( 16 )` (see 18.10).

The first place where you need domains in GAP3 is the obvious one. Sometimes you simply want to talk about a domain. For example if you want to compute the size of the group D_{12}, you had better be able to represent this group in a way that the `Size` function can understand.

The second place where you need domains in GAP3 is when you want to be able to specify that an operation or computation takes place in a certain domain. For example suppose you want to factor 10 in the ring of Gaussian integers. Saying `Factors( 10 )` will not do, because this will return the factorization in the ring of integers `[ 2, 5 ]`. To allow operations and computations to happen in a specific domain, `Factors`, and many other functions as well, accept this domain as optional first argument. Thus `Factors( GaussianIntegers, 10 )` yields the desired result `[ 1+E(4), 1-E(4), 2+E(4), 2-E(4) ]`.

Each domain in GAP3 belongs to one or more **categories**, which are simply sets of domains. The categories in which a domain lies determine the functions that are applicable to this domain and its elements. Examples of domains are **rings** (the functions applicable to a domain that is a ring are described in 5), **fields** (see 6), **groups** (see 7), **vector spaces** (see 9), and of course the category **domains** that contains all domains (the functions applicable to any domain are described in this chapter).

This chapter describes how domains are represented in GAP3 (see 4.1), how functions that can be applied to different types of domains know how to solve a problem for each of those types (see 4.2, 4.3, and 4.4), how domains are compared (see 4.7), and the set theoretic functions that can be applied to any domain (see 4.6, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.16).

The functions described in this chapter are implemented in the file `LIBNAME/"domain.g"`.

4.1 Domain Records

Domains are represented by records (see 46), which are called **domain records** in the following. Which components need to be present, which may, and what those components hold, differs from category to category, and, to a smaller extent, from domain to domain. It is generally possible though to distinguish four types of components.

Each domain record has the component `isDomain`, which has the value `true`. Furthermore, most domains also have a component that specifies which category this domain belongs to. For example, each group has the component `isGroup`, holding the value `true`. Those components are called the **category components** of the domain record. A domain that only has the component `isDomain` is a member only of the category **Domains** and only the functions described in this chapter are applicable to such a domain.

Every domain record also contains enough information to identify uniquely the domain in the so called **identification components**. For example, for a group the domain record, called group record in this case, has a component called `generators` containing a system of generators (and also a component `identity` holding the identity element of the group, needed if the generator list is empty, as is the case for the trivial group).

Next the domain record holds all the knowledge GAP3 has about the domain, for example the size of the domain, in the so called **knowledge components**. Of course, the knowledge about a certain domain will usually increase as time goes by. For example, a group record may initially hold only the knowledge that the group is finite, but may end holding all kinds of knowledge, for example the derived series, the Sylow subgroups, etc.

Finally each domain record has a component, which is called its **operations record** (because it is the component with the name `operations` and it holds a record), that tells functions like `Size` how to compute this information for this domain. The exact mechanism is described later (see 4.2).

4.2 Dispatchers

In the previous section it was mentioned that domains are represented by domain records, and that each domain record has an operations record. This operations record is used by functions like `Size` to find out how to compute this information for the domain. Let us discuss this mechanism using the example of `Size`. Suppose you call `Size` with a domain D.

First `Size` tests whether D has a component called `size`, i.e., if D`.size` is bound. If it is, `Size` assumes that it holds the size of the domain and returns this value.

Let us suppose that this component has no assigned value. Then `Size` looks at the component D`.operations`, which must be a record. `Size` takes component D`.operations.Size` of this record, which must be a function. `Size` calls this function passing D as argument. If a domain record has no `Size` function in its operations record, an error is signalled.

Finally `Size` stores the value returned by D`.operations.Size(` D `)` in the component D`.size`, where it is available for the next call of `Size(` D `)`.

Because functions like `Size` do little except dispatch to the function in the operations record they are called **dispatcher functions**.

Which function is called through this mechanism obviously depends on the domain and its operations record. In principle each domain could have its own `Size` function. In practice however this is not the case. For example all permutation groups share the operations record `PermGroupOps` so they all use the same `Size` function `PermGroupOps.Size`.

Note that in fact domains of the same type not only share the functions, in fact they share the operations record. So for example all permutation groups have the same operations record. This means that changing such a function for a domain D in the following way D`.operations.`*function* `:=` *new-function*`;` will also change this function for all domains of the same type, even those that do not yet exist at the moment of the assignment and will only be constructed later. This is usually not desirable, since supposedly *new-function* uses some special properties of the domain D to work efficiently. We suggest therefore, that you use the following assignments instead:
D`.operations := Copy(` D`.operations );`
D`.operations.`*function* `:=` *new-function*`;`.

Some domains do not provide a special `Size` function, either because no efficient method is known or because the author that implemented the domain simply was too lazy to write one. In those cases the domain inherits the **default function**, which is `DomainOps.Size`. Such inheritance is uncommon for the `Size` function, but rather common for the `Union` function.

4.3 More about Dispatchers

Usually you need not care about the mechanism described in the previous section. You just call the dispatcher functions like `Size`. They will call the function in the operations record, which is hopefully implementing an algorithm that is well suited for their domain, by using the structure of this domain.

There are three reasons why you might want to avoid calling the dispatcher function and call the dispatched to function directly.

The first reason is efficiency. The dispatcher functions don't do very much. They only check the types of their arguments, check if the requested information is already present, and dispatch to the appropriate function in the operations record. But sometimes, for example in the innermost loop of your algorithm, even this little is too much. In those cases you can avoid the overhead introduced by the dispatcher function by calling the function in the operations record directly. For example, you would use G`.operations.Size(`G`)` instead of `Size(`G`)`.

The second reason is flexibility. Sometimes you do not want to call the function in the operations record, but another function that performs the same task, using a different algorithm. In that case you will call this different function. For example, if G is a permutation group, and the orbit of p under G is very short, `GroupOps.Orbit(`G`,`p`)`, which is the default function to compute an orbit, may be slightly more efficient than `Orbit(`G`,`p`)`, which calls G`.operations.Orbit(`G`,`p`)`, which is the same as `PermGroupOps.Orbit(`G`,`p`)`.

The third has to do with the fact that the dispatcher functions check for knowledge components like D`.size` or D`.elements` and also store their result in such components. For example, suppose you know that the result of a computation takes up quite some space, as is the case with `Elements(`D`)`, and that you will never need the value again. In this case you would not want the dispatcher function to enter the value in the domain record, and therefore would call D`.operations.Elements(`D`)` directly. On the other hand you may not want to use the value in the domain record, because you mistrust it. In this case you should call the function in the operations record directly, e.g., you would use G`.operations.Size(`G`)` instead of `Size(`G`)` (and then compare the result with G`.size`).

4.4 An Example of a Computation in a Domain

This section contains an extended example to show you how a computation in a domain may use default and special functions to achieve its goal. Suppose you defined `G`, `x`, and `y` as follows.

```
gap> G := SymmetricGroup( 8 );;
gap> x := [ (2,7,4)(3,5), (1,2,6)(4,8) ];;
gap> y := [ (2,5,7)(4,6), (1,5)(3,8,7) ];;
```

Now you ask for an element of `G` that conjugates `x` to `y`, i.e., a permutation on 8 points that takes `(2,7,4)(3,5)` to `(2,5,7)(4,6)` and `(1,2,6)(4,8)` to `(1,5)(3,8,7)`. This is done as follows (see 8.25 and 8.1).

```
gap> RepresentativeOperation( G, x, y, OnTuples );
(1,8)(2,7)(3,4,5,6)
```

Let us look at what happens step by step. First `RepresentativeOperation` is called. After checking the arguments it calls the function `G.operations.RepresentativeOperation`, which is the function `SymmetricGroupOps.RepresentativeOperation`, passing the arguments `G`, `x`, `y`, and `OnTuples`.

`SymmetricGroupOps.RepresentativeOperation` handles a lot of cases specially, but the operation on tuples of permutations is not among them. Therefore it delegates this problem to the function that it overlays, which is `PermGroupOps.RepresentativeOperation`.

`PermGroupOps.RepresentativeOperation` also does not handle this special case, and delegates the problem to the function that it overlays, which is the default function called `GroupOps.RepresentativeOperation`.

`GroupOps.RepresentativeOperation` views this problem as a general tuples problem, i.e., it does not care whether the points in the tuples are integers or permutations, and decides to solve it one step at a time. So first it looks for an element taking `(2,7,4)(3,5)` to `(2,5,7)(4,6)` by calling `RepresentativeOperation( G, (2,7,4)(3,5), (2,5,7)(4,6) )`.

`RepresentativeOperation` calls `G.operations.RepresentativeOperation` next, which is the function `SymmetricGroupOps.RepresentativeOperation`, passing the arguments `G`, `(2,7,4)(3,5)`, and `(2,5,7)(4,6)`.

`SymmetricGroupOps.RepresentativeOperation` can handle this case. It **knows** that `G` contains every permutation on 8 points, so it contains `(3,4,7,5,6)`, which obviously does what we want, namely it takes `x[1]` to `y[1]`. We will call this element `t`.

Now `GroupOps.RepresentativeOperation` (see above) looks for an `s` in the stabilizer of `x[1]` taking `x[2]` to `y[2]^(t^-1)`, since then for `r=s*t` we have `x[1]^r = (x[1]^s)^t = x[1]^t = y[1]` and also `x[2]^r = (x[2]^s)^t = (y[2]^(t^-1))^t = y[2]`. So the next step is to compute the stabilizer of `x[1]` in `G`. To do this it calls `Stabilizer( G, (2,7,4)(3,5) )`.

Stabilizer calls G.operations.Stabilizer, which is SymmetricGroupOps.Stabilizer, passing the arguments G and (2,7,4)(3,5). SymmetricGroupOps.Stabilizer detects that the second argument is a permutation, i.e., an element of the group, and calls Centralizer(G, (2,7,4)(3,5)). Centralizer calls the function G.operations.Centralizer, which is SymmetricGroupOps.Centralizer, again passing the arguments G, (2,7,4)(3,5).

SymmetricGroupOps.Centralizer again **knows** how centralizers in symmetric groups look, and after looking at the permutation (2,7,4)(3,5) sharply for a short while returns the centralizer as Subgroup(G, [(1,6), (1,6,8), (2,7,4), (3,5)]), which we will call S. Note that S is of course not a symmetric group, therefore SymmetricGroupOps.Subgroup gives it PermGroupOps as operations record and not SymmetricGroupOps.

As explained above GroupOps.RepresentativeOperation needs an element of S taking x[2] ((1,2,6)(4,8)) to y[2]^(t^-1) ((1,7)(4,6,8)). So RepresentativeOperation(S, (1,2,6)(4,8), (1,7)(4,6,8)) is called. RepresentativeOperation in turn calls the function S.operations.RepresentativeOperation, which is, since S is a permutation group, the function PermGroupOps.RepresentativeOperation, passing the arguments S, (1,2,6)(4,8), and (1,7)(4,6,8).

PermGroupOps.RepresentativeOperation detects that the points are permutations and and performs a backtrack search through S. It finds and returns (1,8)(2,4,7)(3,5), which we call s.

Then GroupOps.RepresentativeOperation returns r = s*t = (1,8)(2,7)(3,6)(4,5), and we are done.

In this example you have seen how functions use the structure of their domain to solve a problem most efficiently, for example SymmetricGroupOps.RepresentativeOperation but also the backtrack search in PermGroupOps.RepresentativeOperation, how they use other functions, for example SymmetricGroupOps.Stabilizer called Centralizer, and how they delegate cases which they can not handle more efficiently back to the function they overlaid, for example SymmetricGroupOps.RepresentativeOperation delegated to PermGroupOps.RepresentativeOperation, which in turn delegated to to the function GroupOps.RepresentativeOperation.

4.5 Domain

Domain(*list*)

Domain returns a domain that contains all the elements in *list* and that knows how to make the ring, field, group, or vector space that contains those elements.

Note that the domain returned by Domain need in general not be a ring, field, group, or vector space itself. For example if passed a list of elements of finite fields Domain will return the domain FiniteFieldElements. This domain contains all finite field elements, no matter of which characteristic. This domain has a function FiniteFieldElementsOps.Field that knows how to make a finite field that contains the elements in *list*. This function knows that all elements must have the same characteristic for them to lie in a common field.

```
gap> D := Domain( [ Z(4), Z(8) ] );
FiniteFieldElements
gap> IsField( D );
false
```

```
gap> D.operations.Field( [ Z(4), Z(8) ] );
GF(2^6)
```

`Domain` is the only function in the whole GAP3 library that knows about the various types of elements. For example, when `Norm` is confronted by a field element z, it does not know what to do with it. So it calls `F := DefaultField( [` z `] )` to get a field in which z lies, because this field (more precisely `F.operations.Norm`) will know better. However, `DefaultField` also does not know what to do with z. So it calls `D := Domain( [` z `] )` to get a domain in which z lies, because it (more precisely `D.operations.DefaultField`) will know how to make a default field in which z lies.

4.6 Elements

`Elements(` D `)`

`Elements` returns the set of elements of the domain D. The set is returned as a new proper set, i.e., as a new sorted list without holes and duplicates (see 28). D may also be a list, in which case the set of elements of this list is returned. An error is signalled if D is an infinite domain.

```
gap> Elements( GaussianIntegers );
Error, the ring <R> must be finite to compute its elements
gap> D12 := Group( (2,6)(3,5), (1,2)(3,6)(4,5) );;
gap> Elements( D12 );
[ (), (2,6)(3,5), (1,2)(3,6)(4,5), (1,2,3,4,5,6), (1,3)(4,6),
  (1,3,5)(2,4,6), (1,4)(2,3)(5,6), (1,4)(2,5)(3,6), (1,5)(2,4),
  (1,5,3)(2,6,4), (1,6,5,4,3,2), (1,6)(2,5)(3,4) ]
```

`Elements` remembers the set of elements in the component D`.elements` and will return a shallow copy (see 46.12) next time it is called to compute the elements of D. If you want to avoid this, for example for a large domain, for which you know that you will not need the list of elements in the future, either unbind (see 46.10) D`.elements` or call D`.operation.Elements(`D`)` directly.

Since there is no general method to compute the elements of a domain the default function `DomainOps.Elements` just signals an error. This default function is overlaid for each special finite domain. In fact, implementors of domains, **must** implement this function for new domains, since it is, together with `IsFinite` (see 4.9) the most basic function for domains, used by most of the default functions in the domain package.

In general functions that return a set of elements are free, in fact encouraged, to return a domain instead of the proper set of elements. For one thing this allows to keep the structure, for another the representation by a domain record is usually more space efficient. `Elements` must not do this, its only purpose is to create the proper set of elements.

4.7 Comparisons of Domains

D `=` E
D `<>` E

`=` evaluates to `true` if the two domains D and E are equal, to `false` otherwise. `<>` evaluates to `true` if the two domains D and E are different and to `false` if they are equal.

Two domains are considered equal if and only if the sets of their elements as computed by `Elements` (see 4.6) are equal. Thus, in general `=` behaves as if each domain operand were replaced by its set of elements. Except that `=` will also sometimes, but not always, work for infinite domains, for which it is of course difficult to compute the set of elements. Note that this implies that domains belonging to different categories may well be equal. As a special case of this, either operand may also be a proper set, i.e., a sorted list without holes or duplicates (see 28.2), and the result will be `true` if and only if the set of elements of the domain is, as a set, equal to the set. It is also possible to compare a domain with something else that is not a domain or a set, but the result will of course always be `false` in this case.

```
gap> GaussianIntegers = D12;
false    # GAP3 knows that those domains cannot be equal because
         # GaussianIntegers is infinite and D12 is finite
gap> GaussianIntegers = Integers;
false    # GAP3 knows how to compare those two rings
gap> GaussianIntegers = Rationals;
Error, sorry, cannot compare the infinite domains <D> and <E>
gap> D12 = Group( (2,6)(3,5), (1,2)(3,6)(4,5) );
true
gap> D12 = [(),(2,6)(3,5),(1,2)(3,6)(4,5),(1,2,3,4,5,6),(1,3)(4,6),
>           (1,3,5)(2,4,6),(1,4)(2,3)(5,6),(1,4)(2,5)(3,6),
>           (1,5)(2,4),(1,5,3)(2,6,4),(1,6,5,4,3,2),(1,6)(2,5)(3,4)];
true
gap> D12 = [(1,6,5,4,3,2),(1,6)(2,5)(3,4),(1,5,3)(2,6,4),(1,5)(2,4),
>           (1,4)(2,5)(3,6),(1,4)(2,3)(5,6),(1,3,5)(2,4,6),(1,3)(4,6),
>           (1,2,3,4,5,6),(1,2)(3,6)(4,5),(2,6)(3,5),()];
false    # since the left operand behaves as a set
         # while the right operand is not a set
```

The default function `DomainOps.'='` checks whether both domains are infinite. If they are, an error is signalled. Otherwise, if one domain is infinite, `false` is returned. Otherwise the sizes (see 4.10) of the domains are compared. If they are different, `false` is returned. Finally the sets of elements of both domains are computed (see 4.6) and compared. This default function is overlaid by more special functions for other domains.

D `<` *E*
D `<=` *E*
D `>` *E*
D `>=` *E*

`<`, `<=`, `>`, and `>=` evaluate to `true` if the domain D is less than, less than or equal to, greater than, and greater than or equal to the domain E and to `false` otherwise.

A domain D is considered less than a domain E if and only if the set of elements of D is less than the set of elements of the domain E. Generally you may just imagine that each domain operand is replaced by the set of its elements, and that the comparison is performed on those sets (see 27.12). This implies that, if you compare a domain with an object that is not a list or a domain, this other object will be less than the domain, except if it is a record, in which case it is larger than the domain (see 2.9).

Note that `<` does **not** test whether the left domain is a subset of the right operand, even though it resembles the mathematical subset notation.

```
gap> GaussianIntegers < Rationals;
Error, sorry, cannot compare <E> with the infinite domain <D>
gap> Group( (1,2), (1,2,3,4,5,6) ) < D12;
true      # since (5,6), the second element of the left operand,
          # is less than (2,6)(3,5), the second element of D12.
gap> D12 < [(1,6,5,4,3,2),(1,6)(2,5)(3,4),(1,5,3)(2,6,4),(1,5)(2,4),
>           (1,4)(2,5)(3,6),(1,4)(2,3)(5,6),(1,3,5)(2,4,6),(1,3)(4,6),
>           (1,2,3,4,5,6),(1,2)(3,6)(4,5),(2,6)(3,5),()];
true      # since (), the first element of D12, is less than
          # (1,6,5,4,3,2), the first element of the right operand.
gap> 17 < D12;
true      # objects that are not lists or records are smaller
          # than domains, which behave as if they were a set
```

The default function `DomainOps.'<'` checks whether either domain is infinite. If one is, an error is signalled. Otherwise the sets of elements of both domains are computed (see 4.6) and compared. This default function is only very seldom overlaid by more special functions for other domains. Thus the operators `<`, `<=`, `>`, and `>=` are quite expensive and their use should be avoided if possible.

4.8 Membership Test for Domains

elm `in` *D*

`in` returns `true` if the element *elm*, which may be an object of any type, lies in the domain *D*, and `false` otherwise.

```
gap> 13 in GaussianIntegers;
true
gap> GaussianIntegers in GaussianIntegers;
false
gap> (1,2) in D12;
false
gap> (1,2)(3,6)(4,5) in D12;
true
```

The default function for domain membership tests is `DomainOps.'in'`, which computes the set of elements of the domain with the function `Elements` (see 4.6) and tests whether *elm* lies in this set. Special domains usually overlay this function with more efficient membership tests.

4.9 IsFinite

`IsFinite(` *D* `)`

`IsFinite` returns `true` if the domain *D* is finite and `false` otherwise. *D* may also be a proper set (see 28.2), in which case the result is of course always `true`.

```
gap> IsFinite( GaussianIntegers );
false
gap> IsFinite( D12 );
true
```

The default function `DomainOps.IsFinite` just signals an error, since there is no general method to determine whether a domain is finite or not. This default function is overlaid for each special domain. In fact, implementors of domains **must** implement this function for new domains, since it is, together with `Elements` (see 4.6), the most basic function for domains, used by most of the default functions in the domain package.

4.10 Size

`Size(` *D* `)`

`Size` returns the size of the domain *D*. If *D* is infinite, `Size` returns the string `"infinity"`. *D* may also be a proper set (see 28.2), in which case the result is the length of this list. `Size` will, however, signal an error if *D* is a list that is not a proper set, i.e., that is not sorted, or has holes, or contains duplicates.

```
gap> Size( GaussianIntegers );
"infinity"
gap> Size( D12 );
12
```

The default function to compute the size of a domain is `DomainOps.Size`, which computes the set of elements of the domain with the function `Elements` (see 4.6) and returns the length of this set. This default function is overlaid in practically every domain.

4.11 IsSubset

`IsSubset(` *D*`,` *E* `)`

`IsSubset` returns `true` if the domain *E* is a subset of the domain *D* and `false` otherwise.

E is considered a subset of *D* if and only if the set of elements of *E* is as a set a subset of the set of elements of *D* (see 4.6 and 28.9). That is `IsSubset` behaves as if implemented as `IsSubsetSet( Elements(`*D*`), Elements(`*E*`) )`, except that it will also sometimes, but not always, work for infinite domains, and that it will usually work much faster than the above definition. Either argument may also be a proper set.

```
gap> IsSubset( GaussianIntegers, [1,E(4)] );
true
gap> IsSubset( GaussianIntegers, Rationals );
Error, sorry, cannot compare the infinite domains <D> and <E>
gap> IsSubset( Group( (1,2), (1,2,3,4,5,6) ), D12 );
true
gap> IsSubset( D12, [ (), (1,2)(3,4)(5,6) ] );
false
```

The default function `DomainOps.IsSubset` checks whether both domains are infinite. If they are it signals an error. Otherwise if the *E* is infinite it returns `false`. Otherwise if *D* is infinite it tests if each element of *E* is **in** *D* (see 4.8). Otherwise it tests whether the proper set of elements of *E* is a subset of the proper set of elements of *D* (see 4.6 and 28.9).

4.12 Intersection

`Intersection(` *D1*`,` *D2*`...` `)`
`Intersection(` *list* `)`

In the first form `Intersection` returns the intersection of the domains *D1*, *D2*, etc. In the second form *list* must be a list of domains and `Intersection` returns the intersection of those domains. Each argument *D* or element of *list* respectively may also be an arbitrary list, in which case `Intersection` silently applies `Set` (see 28.2) to it first.

The result of `Intersection` is the set of elements that lie in every of the domains *D1*, *D2*, etc. Functions called by the dispatcher function `Intersection` however, are encouraged to keep as much structure as possible. So if *D1* and *D2* are elements of a common category and if this category is closed under taking intersections, then the result should be a domain lying in this category too. So for example the intersection of permutation groups will again be a permutation group.

```
gap> Intersection( CyclotomicField(9), CyclotomicField(12) );
CF(3)     # CF is a shorthand for CyclotomicField
          # this is one of the rare cases where the intersection
          # of two infinite domains works
gap> Intersection( GaussianIntegers, Rationals );
Error, sorry, cannot intersect infinite domains <D> and <E>
gap> Intersection( D12, Group( (1,2), (1,2,3,4,5) ) );
Group( (1,5)(2,4) )
gap> Intersection( D12, [ (1,3)(4,6), (1,2)(3,4) ] );
[ (1,3)(4,6) ]     # note that the second argument is not a set
gap> Intersection( D12, [ (), (1,2)(3,4), (1,3)(4,6), (1,4)(5,6) ] );
[ (), (1,3)(4,6) ]     # although the result is mathematically a
                       # group it is returned as a proper set
                       # because the second argument was not a group
gap> Intersection( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] );
[  ]     # two or more domains or sets as arguments are legal
gap> Intersection( [ [1,2,4], [2,3,4], [1,3,4] ] );
[ 4 ]     # or a list of domains or sets
gap> Intersection( [ ] );
Error, List Element: <list>[1] must have a value
```

The dispatcher function (see 4.2) `Intersection` is slightly different from other dispatcher functions. It does not simply call the function in the operations record passings its arguments. Instead it loops over its arguments (or the list of domains or sets) and calls the function in the operations record repeatedly, and passes each time only two domains. This obviously makes writing the function for the operations record simpler.

The default function `DomainOps.Intersection` checks whether both domains are infinite. If they are it signals an error. Otherwise, if one of the domains is infinite it loops over the elements of the other domain, and tests for each element whether it lies in the infinite domain. If both domains are finite it computes the proper sets of elements of both and intersects them (see 4.6 and 28.9). This default method is overlaid by more special functions for most other domains. Those functions usually are faster and keep the structure of the domains if possible.

4.13 Union

`Union(` *D1* `,` *D2* `...` `)`
`Union(` *list* `)`

In the first form `Union` returns the union of the domains *D1*, *D2*, etc. In the second form *list* must be a list of domains and `Union` returns the union of those domains. Each argument *D* or element of *list* respectively may also be an arbitrary list, in which case `Union` silently applies `Set` (see 28.2) to it first.

The result of `Union` is the set of elements that lie in any the domains *D1*, *D2*, etc. Functions called by the dispatcher function `Union` however, are encouraged to keep as much structure as possible. However, currently GAP3 does not support any category that is closed under taking unions except the category of all domains. So the only case that structure will be kept is when one argument *D* or element of *list* respectively is a superset of all the other arguments or elements of *list*.

```
gap> Union( GaussianIntegers, Rationals );
Error, sorry, cannot unite <E> with the infinite domain <D>
gap> Union( D12, Group( (1,2), (1,2,3) ) );
[ (), (2,3), (2,6)(3,5), (1,2), (1,2)(3,6)(4,5), (1,2,3),
  (1,2,3,4,5,6), (1,3,2), (1,3), (1,3)(4,6), (1,3,5)(2,4,6),
  (1,4)(2,3)(5,6), (1,4)(2,5)(3,6), (1,5)(2,4), (1,5,3)(2,6,4),
  (1,6,5,4,3,2), (1,6)(2,5)(3,4) ]
gap> Union( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] );
[ 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 20, 25 ]
    # two or more domains or sets as arguments are legal
gap> Union( [ [1,2,4], [2,3,4], [1,3,4] ] );
[ 1, 2, 3, 4 ]     # or a list of domains or sets
gap> Union( [ ] );
[  ]
```

The dispatcher function (see 4.2) `Union` is slightly different from other dispatcher functions. It does not simply call the function in the operations record passings its arguments. Instead it loops over its arguments (or the list of domains or sets) and calls the function in the operations record repeatedly, and passes each time only two domains. This obviously makes writing the function for the operations record simpler.

The default function `DomainOps.Union` checks whether either domain is infinite. If one is it signals an error. If both domains are finite it computes the proper sets of elements of both and unites them (see 4.6 and 28.9). This default method is overlaid by more special functions for some other domains. Those functions usually are faster.

4.14 Difference

`Difference(` *D*`,` *E* `)`

`Difference` returns the set difference of the domains *D* and *E*. Either argument may also be an arbitrary list, in which case `Difference` silently applies `Set` (see 28.2) to it first.

The result of `Difference` is the set of elements that lie in *D* but not in *E*. Note that *E* need not be a subset of *D*. The elements of *E*, however, that are not element of *D* play no role for the result.

```
gap> Difference( D12, [(),(1,2,3,4,5,6),(1,3,5)(2,4,6),
>                     (1,4)(2,5)(3,6),(1,6,5,4,3,2),(1,5,3)(2,6,4)] );
[ (2,6)(3,5), (1,2)(3,6)(4,5), (1,3)(4,6), (1,4)(2,3)(5,6),
```

```
(1,5)(2,4), (1,6)(2,5)(3,4) ]
```

The default function `DomainOps.Difference` checks whether D is infinite. If it is it signals an error. Otherwise `Difference` computes the proper sets of elements of D and E and returns the difference of those sets (see 4.6 and 28.8). This default function is currently not overlaid for any domain.

4.15 Representative

`Representative(` D `)`

`Representative` returns a representative of the domain D.

The existence of a representative, and the exact definition of what a representative is, depends on the category of D. The representative should be an element that, within a given context, identifies the domain D. For example if D is a cyclic group, its representative would be a generator of D, or if D is a coset, its representative would be an arbitrary element of the coset.

Note that `Representative` is pretty free in choosing a representative if there are several. It is not even guaranteed that `Representative` returns the same representative if it is called several times for one domain. Thus the main difference between `Representative` and `Random` (see 4.16) is that `Representative` is free to choose a value that is cheap to compute, while `Random` must make an effort to randomly distribute its answers.

```
gap> C := Coset( Subgroup( G, [(1,4)(2,5)(3,6)] ), (1,6,5,4,3,2) );;
gap> Representative( C );
(1,3,5)(2,4,6)
```

`Representative` first tests whether the component D`.representative` is bound. If the field is bound it returns its value. Otherwise it calls D`.operations.Representative(` D `)`, remembers the returned value in D`.representative`, and returns it.

The default function called this way is `DomainOps.Representative`, which simply signals an error, since there is no default way to find a representative.

4.16 Random

`Random(` D `)`

`Random` returns a random element of the domain D. The distribution of elements returned by `Random` depends on the domain D. For finite domains all elements are usually equally likely. For infinite domains some reasonable distribution is used. See the chapters of the various domains to find out which distribution is being used.

```
gap> Random( GaussianIntegers );
1-4*E(4)
gap> Random( GaussianIntegers );
1+2*E(4)
gap> Random( D12 );
()
gap> Random( D12 );
(1,4)(2,5)(3,6)
```

The default function for random selection is `DomainOps.Random`, which computes the set of elements using `Elements` and selects a random element of this list using `RandomList` (see 27.48 for a description of the pseudo random number generator used). This default function can of course only be applied to finite domains. It is overlaid by other functions for most other domains.

All random functions called this way rely on the low level random number generator provided by `RandomList` (see 27.48).

Chapter 5

Rings

Rings are important algebraic domains. Mathematically a **ring** is a set R with two operations `+` and `*` called addition and multiplication. $(R, +)$ must be an abelian group. The identity of this group is called 0_R. $(R - \{0_R\}, *)$ must be a monoid. If this monoid has an identity element it is called 1_R.

Important examples of rings are the integers (see 10), the Gaussian integers (see 14), the integers of a cyclotomic field (see 15), and matrices (see 34).

This chapter contains sections that describe how to test whether a domain is a ring (see 5.1), and how to find the smallest and the default ring in which a list of elements lies (see 5.2 and 5.3).

The next sections describe the operations applicable to ring elements (see 5.4, 5.5, 5.6).

The next sections describe the functions that test whether a ring has certain properties (5.7, 5.8, 5.9, and 5.10).

The next sections describe functions that are related to the units of a ring (see 5.11, 5.12, 5.13, 5.14, and 5.15).

Then come the sections that describe the functions that deal with the irreducible and prime elements of a ring (see 5.16, 5.17, and 5.18).

Then come the sections that describe the functions that are applicable to elements of rings (see 5.19, 5.20, 5.21, 5.22, 5.24, 5.25, 5.26, 5.27, 5.28).

The last section describes how ring records are represented internally (see 5.29).

Because rings are a category of domains all functions applicable to domains are also applicable to rings (see chapter 4) .

All functions described in this chapter are in `LIBNAME/"ring.g"`.

5.1 IsRing

`IsRing(` *domain* `)`

`IsRing` returns `true` if the object *domain* is a ring record, representing a ring (see 5.29), and `false` otherwise.

More precisely `IsRing` tests whether *domain* is a ring record (see 5.29). So for example a matrix group may in fact be a ring, yet `IsRing` would return `false`.

```
gap> IsRing( Integers );
true
gap> IsRing( Rationals );
false    # Rationals is a field record not a ring record
gap> IsRing( rec( isDomain := true, isRing := true ) );
true    # it is possible to fool IsRing
```

5.2 Ring

`Ring(` r, s... `)`
`Ring(` *list* `)`

In the first form `Ring` returns the smallest ring that contains all the elements r, s... etc. In the second form `Ring` returns the smallest ring that contains all the elements in the list *list*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

```
gap> Ring( 1, -1 );
Integers
gap> Ring( [10..20] );
Integers
```

`Ring` differs from `DefaultRing` (see 5.3) in that it returns the smallest ring in which the elements lie, while `DefaultRing` may return a larger ring if that makes sense.

5.3 DefaultRing

`DefaultRing(` r, s... `)`
`DefaultRing(` *list* `)`

In the first form `DefaultRing` returns the default ring that contains all the elements r, s... etc. In the second form `DefaultRing` returns the default ring that contains all the elements in the list *list*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

The ring returned by `DefaultRing` need not be the smallest ring in which the elements lie. For example for elements from cyclotomic fields `DefaultRing` may return the ring of integers of the smallest cyclotomic field in which the elements lie, which need not be the smallest ring overall, because the elements may in fact lie in a smaller number field which is not a cyclotomic field.

For the exact definition of the default ring of a certain type of elements read the chapter describing this type.

`DefaultRing` is used by the ring functions like `Quotient`, `IsPrime`, `Factors`, or `Gcd` if no explicit ring is given.

```
gap> DefaultRing( 1, -1 );
Integers
gap> DefaultRing( [10..20] );
Integers
```

`Ring` (see 5.2) differs from `DefaultRing` in that it returns the smallest ring in which the elements lie, while `DefaultRing` may return a larger ring if that makes sense.

5.4 Comparisons of Ring Elements

r `=` *s*
r `<>` *s*

The equality operator `=` evaluates to `true` if the two ring elements r and s are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the two ring elements r and s are not equal, and to `false` otherwise. Note that any two ring elements can be compared, even if they do not lie in compatible rings. In this case they can, of course, never be equal. For each type of rings the equality of those ring elements is given in the respective chapter.

Ring elements can also be compared with objects of other types. Of course they are never equal.

r `<` *s*
r `<=` *s*
r `>` *s*
r `>=` *s*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the ring element r is less than, less than or equal to, greater than, or greater than or equal to the ring element s, and to `false` otherwise. For each type of rings the definition of the ordering of those ring elements is given in the respective chapter. The ordering of ring elements is as follows. Rationals are smallest, next are cyclotomics, followed by finite ring elements.

Ring elements can also be compared with objects of other types. They are smaller than everything else.

5.5 Operations for Ring Elements

The following operations are always available for ring elements. Of course the operands must lie in compatible rings, i.e., the rings must be equal, or at least have a common superring.

r `+` *s*

The operator `+` evaluates to the sum of the two ring elements r and s, which must lie in compatible rings.

r `-` *s*

The operator `-` evaluates to the difference of the two ring elements r and s, which must lie in compatible rings.

r `*` *s*

The operator `*` evaluates to the product of the two ring elements r and s, which must lie in compatible rings.

r `^` *n*

The operator `^` evaluates to the n-th power of the ring element r. If n is a positive integer then *r*`^`*n* is *r*`*`*r*`*..*`*r* (n factors). If n is a negative integer *r*`^`*n* is defined as $1/r^{-n}$. If 0 is raised to a negative power an error is signalled. Any ring element, even 0, raised to the 0-th power yields 1.

For the precedence of the operators see 2.10.

Note that the quotient operator / usually performs the division in the quotient field of the ring. To compute a quotient in a ring use the function `Quotient` (see 5.6).

5.6 Quotient

`Quotient( r, s )`
`Quotient( R, r, s )`

In the first form `Quotient` returns the quotient of the two ring elements r and s in their default ring (see 5.3). In the second form `Quotient` returns the quotient of the two ring elements r and s in the ring R. It returns `false` if the quotient does not exist.

```
gap> Quotient( 4, 2 );
2
gap> Quotient( Integers, 3, 2 );
false
```

`Quotient` calls `R.operations.Quotient( R, r, s )` and returns the value.

The default function called this way is `RingOps.Quotient`, which just signals an error, because there is no generic method to compute the quotient of two ring elements. Thus special categories of rings must overlay this default function with other functions.

5.7 IsCommutativeRing

`IsCommutativeRing( R )`

`IsCommutativeRing` returns `true` if the ring R is commutative and `false` otherwise.

A ring R is called **commutative** if for all elements r and s of R we have $rs = sr$.

```
gap> IsCommutativeRing( Integers );
true
```

`IsCommutativeRing` first tests whether the flag `R.isCommutativeRing` is bound. If the flag is bound, it returns this value. Otherwise it calls `R.operations.IsCommutativeRing( R )`, remembers the returned value in `R.isCommutativeRing`, and returns it.

The default function called this way is `RingOps.IsCommutativeRing`, which tests whether all the generators commute if the component `R.generators` is bound, and tests whether all elements commute otherwise, unless R is infinite. This function is seldom overlaid, because most rings already have the flag bound.

5.8 IsIntegralRing

`IsIntegralRing( R )`

`IsIntegeralRing` returns `true` if the ring R is integral and `false` otherwise.

A ring R is called **integral** if it is commutative and if for all elements r and s of R we have $rs = 0_R$ implies that either r or s is 0_R.

```
gap> IsIntegralRing( Integers );
true
```

`IsIntegralRing` first tests whether the flag R`.isIntegralRing` is bound. If the flag is bound, it returns this value. Otherwise it calls R`.operations.IsIntegralRing(` R `)`, remembers the returned value in R`.isIntegralRing`, and returns it.

The default function called this way is `RingOps.IsIntegralRing`, which tests whether the product of each pair of nonzero elements is unequal to zero, unless R is infinite. This function is seldom overlaid, because most rings already have the flag bound.

5.9 IsUniqueFactorizationRing

`IsUniqueFactorizationRing(` R `)`

`IsUniqueFactorizationRing` returns `true` if R is a unique factorization ring and `false` otherwise.

A ring R is called a **unique factorization ring** if it is an integral ring, and every element has a unique factorization into irreducible elements, i.e., a unique representation as product of irreducibles (see 5.16). Unique in this context means unique up to permutations of the factors and up to multiplication of the factors by units (see 5.12).

```
gap> IsUniqueFactorizationRing( Integers );
true
```

`IsUniqueFactorizationRing` tests whether R`.isUniqueFactorizationRing` is bound. If the flag is bound, it returns this value. If this flag has no assigned value it calls the function R`.operations.IsUniqueFactorizationRing(` R `)`, remembers the returned value in R`.isUniqueFactorizationRing`, and returns it.

The default function called this way is `RingOps.IsUniqueFactorizationRing`, which just signals an error, since there is no generic method to test whether a ring is a unique factorization ring. Special categories of rings thus must either have the flag bound or overlay this default function.

5.10 IsEuclideanRing

`IsEuclideanRing(` R `)`

`IsEuclideanRing` returns `true` if the ring R is a Euclidean ring and `false` otherwise.

A ring R is called a Euclidean ring if it is an integral ring and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisor of two elements, which in turn implies that R is a unique factorization ring.

```
gap> IsEuclideanRing( Integers );
true
```

`IsEuclideanRing` first tests whether the flag R`.isEuclideanRing` is bound. If the flag is bound, it returns this value. Otherwise it calls R`.operations.IsEuclideanRing(` R `)`, remembers the returned value in R`.isEuclideanRing`, and returns it.

The default function called this way is `RingOps.IsEuclideanRing`, which just signals an error, because there is no generic way to test whether a ring is a Euclidean ring. This function is seldom overlaid because most rings already have the flag bound.

5.11 IsUnit

`IsUnit(` *r* `)`
`IsUnit(` *R*`,` *r* `)`

In the first form `IsUnit` returns `true` if the ring element r is a unit in its default ring (see 5.3). In the second form `IsUnit` returns `true` if r is a unit in the ring R.

An element r is called a **unit** in a ring R, if r has an inverse in R.

```
gap> IsUnit( Integers, 2 );
false
gap> IsUnit( Integers, -1 );
true
```

`IsUnit` calls *R*`.operations.IsUnit(` *R*`,` *r* `)` and returns the value.

The default function called this way is `RingOps.IsUnit`, which tries to compute the inverse of r with *R*`.operations.Quotient(` *R*`,` *R*`.one,` *r* `)` and returns `true` if the result is not `false`, and `false` otherwise. Special categories of rings overlay this default function with more efficient functions.

5.12 Units

`Units(` *R* `)`

`Units` returns the group of units of the ring R. This may either be returned as a list or as a group described by a group record (see 7).

An element r is called a **unit** of a ring R, if r has an inverse in R. It is easy to see that the set of units forms a multiplicative group.

```
gap> Units( Integers );
[ -1, 1 ]
```

`Units` first tests whether the component *R*`.units` is bound. If the component is bound, it returns this value. Otherwise it calls *R*`.operations.Units(` *R* `)`, remembers the returned value in *R*`.units`, and returns it.

The default function called this way is `RingOps.Units`, which runs over all elements of R and tests for each whether it is a unit, provided that R is finite. Special categories of rings overlay this default function with more efficient functions.

5.13 IsAssociated

`IsAssociated(` *r*`,` *s* `)`
`IsAssociated(` *R*`,` *r*`,` *s* `)`

In the first form `IsAssociated` returns `true` if the two ring elements r and s are associated in their default ring (see 5.3) and `false` otherwise. In the second form `IsAssociated` returns `true` if the two ring elements r and s are associated in the ring R and `false` otherwise.

Two elements r and s of a ring R are called **associates** if there is a unit u of R such that $ru = s$.

```
gap> IsAssociated( Integers, 2, 3 );
false
gap> IsAssociated( Integers, 17, -17 );
true
```

`IsAssociated` calls R`.operations.IsAssociated(` R`,` r`,` s `)` and returns the value.

The default function called this way is `RingOps.IsAssociated`, which tries to compute the quotient of r and s and returns `true` if the quotient exists and is a unit. Special categories of rings overlay this default function with more efficient functions.

5.14 StandardAssociate

`StandardAssociate(` r `)`
`StandardAssociate(` R`,` r `)`

In the first form `StandardAssociate` returns the standard associate of the ring element r in its default ring (see 5.3). In the second form `StandardAssociate` returns the standard associate of the ring element r in the ring R.

The **standard associate** of an ring element r of R is an associated element of r which is, in a ring dependent way, distinguished among the set of associates of r. For example, in the ring of integers the standard associate is the absolute value.

```
gap> StandardAssociate( Integers, -17 );
17
```

`StandardAssociate` calls R`.operations.StandardAssociate(` R`,` r `)` and returns the value.

The default function called this way is `RingOps.StandardAssociate`, which just signals an error, because there is no generic way even to define the standard associate. Thus special categories of rings must overlay this default function with other functions.

5.15 Associates

`Associates(` r `)`
`Associates(` R`,` r `)`

In the first form `Associates` returns the set of associates of the ring element r in its default ring (see 5.3). In the second form `Associates` returns the set of associates of r in the ring R.

Two elements r and s of a ring R are called **associate** if there is a unit u of R such that $ru = s$.

```
gap> Associates( Integers, 17 );
[ -17, 17 ]
```

`Associates` calls R`.operations.Associates(` R`,` r `)` and returns the value.

The default function called this way is `RingOps.Associates`, which multiplies the set of units of R with the element r, and returns the set of those elements. Special categories of rings overlay this default function with more efficient functions.

5.16 IsIrreducible

`IsIrreducible(` *r* `)`
`IsIrreducible(` *R*, *r* `)`

In the first form `IsIrreducible` returns `true` if the ring element r is irreducible in its default ring (see 5.3) and `false` otherwise. In the second form `IsIrreducible` returns `true` if the ring element r is irreducible in the ring R and `false` otherwise.

An element r of a ring R is called **irreducible** if there is no nontrivial factorization of r in R, i.e., if there is no representation of r as product st such that neither s nor t is a unit (see 5.11). Each prime element (see 5.17) is irreducible.

```
gap> IsIrreducible( Integers, 4 );
false
gap> IsIrreducible( Integers, 3 );
true
```

`IsIrreducible` calls *R*`.operations.IsIrreducible(` *R*, *r* `)` and returns the value.

The default function called this way is `RingOps.IsIrreducible`, which justs signals an error, because there is no generic way to test whether an element is irreducible. Thus special categories of rings must overlay this default function with other functions.

5.17 IsPrime

`IsPrime(` *r* `)`
`IsPrime(` *R*, *r* `)`

In the first form `IsPrime` returns `true` if the ring element r is a prime in its default ring (see 5.3) and `false` otherwise. In the second form `IsPrime` returns `true` if the ring element r is a prime in the ring R and `false` otherwise.

An element r of a ring R is called **prime** if for each pair s and t such that r divides st the element r divides either s or t. Note that there are rings where not every irreducible element (see 5.16) is a prime.

```
gap> IsPrime( Integers, 4 );
false
gap> IsPrime( Integers, 3 );
true
```

`IsPrime` calls *R*`.operations.IsPrime(` *R*, *r* `)` and returns the value.

The default function called this way is `RingOps.IsPrime`, which just signals an error, because there is no generic way to test whether an element is prime. Thus special categories of rings must overlay this default function with other functions.

5.18 Factors

`Factors(` *r* `)`
`Factors(` *R*, *r* `)`

In the first form `Factors` returns the factorization of the ring element r in its default ring (see 5.3). In the second form `Factors` returns the factorization of the ring element r in the ring R. The factorization is returned as a list of primes (see 5.17). Each element in the list is a standard associate (see 5.14) except the first one, which is multiplied by a unit as necessary to have `Product( Factors(` R`,` r `) ) =` r. This list is usually also sorted, thus smallest prime factors come first. If r is a unit or zero, `Factors(` R`,` r `) = [` r `]`.

```
gap> Factors( -Factorial(6) );
[ -2, 2, 2, 2, 3, 3, 5 ]
gap> Set( Factors( Factorial(13)/11 ) );
[ 2, 3, 5, 7, 13 ]
gap> Factors( 2^63 - 1 );
[ 7, 7, 73, 127, 337, 92737, 649657 ]
gap> Factors( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
```

`Factors` calls R`.operations.Factors(` R`,` r `)` and returns the value.

The default function called this way is `RingOps.Factors`, which just signals an error, because there is no generic way to compute the factorization of ring elements. Thus special categories of ring elements must overlay this default function with other functions.

5.19 EuclideanDegree

`EuclideanDegree(` r `)`
`EuclideanDegree(` R`,` r `)`

In the first form `EuclideanDegree` returns the Euclidean degree of the ring element r in its default ring. In the second form `EuclideanDegree` returns the Euclidean degree of the ring element in the ring R. R must of course be an Euclidean ring (see 5.10).

A ring R is called a Euclidean ring, if it is an integral ring, and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisors of two elements, which in turn implies that R is a unique factorization ring.

```
gap> EuclideanDegree( Integers, 17 );
17
gap> EuclideanDegree( Integers, -17 );
17
```

`EuclideanDegree` calls R`.operations.EuclideanDegree(` R`,` r `)` and returns the value.

The default function called this way is `RingOps.EuclideanDegree`, which justs signals an error, because there is no default way to compute the Euclidean degree of an element. Thus Euclidean rings must overlay this default function with other functions.

5.20 EuclideanRemainder

`EuclideanRemainder(` r`,` m `)`
`EuclideanRemainder(` R`,` r`,` m `)`

In the first form `EuclideanRemainder` returns the remainder of the ring element r modulo the ring element m in their default ring. In the second form `EuclideanRemainder` returns the remainder of the ring element r modulo the ring element m in the ring R. The ring R must be a Euclidean ring (see 5.10) otherwise an error is signalled.

A ring R is called a Euclidean ring, if it is an integral ring, and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisors of two elements, which in turn implies that R is a unique factorization ring. `EuclideanRemainder` returns this remainder $r - qs$.

```
gap> EuclideanRemainder( 16, 3 );
1
gap> EuclideanRemainder( Integers, 201, 11 );
3
```

`EuclideanRemainder` calls R`.operations.EuclideanRemainder(` R, r, m `)` in order to compute the remainder and returns the value.

The default function called this way uses `QuotientRemainder` in order to compute the remainder.

5.21 EuclideanQuotient

`EuclideanQuotient(` r, m `)`
`EuclideanQuotient(` R, r, m `)`

In the first form `EuclideanQuotient` returns the Euclidean quotient of the ring elements r and m in their default ring. In the second form `EuclideanQuotient` returns the Euclidean quotient of the ring elements $r$and m in the ring R. The ring R must be a Euclidean ring (see 5.10) otherwise an error is signalled.

A ring R is called a Euclidean ring, if it is an integral ring, and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisors of two elements, which in turn implies that R is a unique factorization ring. `EuclideanQuotient` returns the quotient q.

```
gap> EuclideanQuotient( 16, 3 );
5
gap> EuclideanQuotient( Integers, 201, 11 );
18
```

`EuclideanQuotient` calls R`.operations.EuclideanQuotient(` R, r, m `)` and returns the value.

The default function called this way uses `QuotientRemainder` in order to compute the quotient.

5.22 QuotientRemainder

`QuotientRemainder(` r, m `)`
`QuotientRemainder(` R, r, m `)`

In the first form `QuotientRemainder` returns the Euclidean quotient and the Euclidean remainder of the ring elements r and m in their default ring as pair of ring elements. In the second form `QuotientRemainder` returns the Euclidean quotient and the Euclidean remainder of the ring elements r and m in the ring R. The ring R must be a Euclidean ring (see 5.10) otherwise an error is signalled.

A ring R is called a Euclidean ring, if it is an integral ring, and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisors of two elements, which in turn implies that R is a unique factorization ring. `QuotientRemainder` returns this quotient q and the remainder $r - qs$.

```
gap> qr := QuotientRemainder( 16, 3 );
[ 5, 1 ]
gap> 3 * qr[1] + qr[2];
16
gap> QuotientRemainder( Integers, 201, 11 );
[ 18, 3 ]
```

`QuotientRemainder` calls R`.operations.QuotientRemainder(` R`,` r`,` m `)` and returns the value.

The default function called this way is `RingOps.QuotientRemainder`, which just signals an error, because there is no default function to compute the Euclidean quotient or remainder of one ring element modulo another. Thus Euclidean rings must overlay this default function with other functions.

5.23 Mod

`Mod(` r`,` m `)`
`Mod(` R`,` r`,` m `)`

`Mod` is a synonym for `EuclideanRemainder` and is obsolete, see 5.20.

5.24 QuotientMod

`QuotientMod(` r`,` s`,` m `)`
`QuotientMod(` R`,` r`,` s`,` m `)`

In the first form `QuotientMod` returns the quotient of the ring elements r and s modulo the ring element m in their default ring (see 5.3). In the second form `QuotientMod` returns the quotient of the ring elements r and s modulo the ring element m in the ring R. R must be a Euclidean ring (see 5.10) so that `EuclideanRemainder` (see 5.20) can be applied. If the modular quotient does not exist, `false` is returned.

The quotient q of r and s modulo m is an element of R such that $qs = r$ modulo m, i.e., such that $qs - r$ is divisable by m in R and that q is either 0 (if r is divisable by m) or the Euclidean degree of q is strictly smaller than the Euclidean degree of m.

```
gap> QuotientMod( Integers, 13, 7, 11 );
5
```

```
gap> QuotientMod( Integers, 13, 7, 21 );
false
```

`QuotientMod` calls R`.operations.QuotientMod(` R, r, s, m `)` and returns the value.

The default function called this way is `RingOps.QuotientMod`, which applies the Euclidean gcd algorithm to compute the gcd g of s and m, together with the representation of this gcd as linear combination in s and m, g `=` a `*` s `+` b `*` m. The modular quotient exists if and only if r is divisible by g, in which case the quotient is a `* Quotient(` R, r, g `)`. This default function is seldom overlaid, because there is seldom a better way to compute the quotient.

5.25 PowerMod

`PowerMod(` r, e, m `)`
`PowerMod(` R, r, e, m `)`

In the first form `PowerMod` returns the e-th power of the ring element r modulo the ring element m in their default ring (see 5.3). In the second form `PowerMod` returns the e-th power of the ring element r modulo the ring element m in the ring R. e must be an integer. R must be a Euclidean ring (see 5.10) so that `EuclideanRemainder` (see 5.20) can be applied to its elements.

If e is positive the result is r^e modulo m. If e is negative then `PowerMod` first tries to find the inverse of r modulo m, i.e., i such that $ir = 1$ modulo m. If the inverse does not exist an error is signalled. If the inverse does exist `PowerMod` returns `PowerMod(` R, i, `-`e, m `)`.

`PowerMod` reduces the intermediate values modulo m, improving performance drastically when e is large and m small.

```
gap> PowerMod( Integers, 2, 20, 100 );
76          # 2^20 = 1048576
gap> PowerMod( Integers, 3, 2^32, 2^32+1 );
3029026160          # which proves that 2^32 + 1 is not a prime
gap> PowerMod( Integers, 3, -1, 22 );
15          # 3*15 = 45 = 1 modulo 22
```

`PowerMod` calls R`.operations.PowerMod(` R, r, e, m `)` and returns the value.

The default function called this way is `RingOps.PowerMod`, which uses `QuotientMod` (see 5.24) if necessary to invert r, and then uses a right-to-left repeated squaring, reducing the intermediate results modulo m in each step. This function is seldom overlaid, because there is seldom a better way of computing the power.

5.26 Gcd

`Gcd(` *r1*, *r2*... `)`
`Gcd(` R, *r1*, *r2*... `)`

In the first form `Gcd` returns the greatest common divisor of the ring elements *r1*, *r2*... etc. in their default ring (see 5.3). In the second form `Gcd` returns the greatest common divisor of the ring elements *r1*, *r2*... etc. in the ring R. R must be a Euclidean ring (see 5.10) so that `QuotientRemainder` (see 5.22) can be applied to its elements. `Gcd` returns the standard associate (see 5.14) of the greatest common divisors.

A greatest common divisor of the elements r_1, r_2... etc. of the ring R is an element of largest Euclidean degree (see 5.19) that is a divisor of r_1, r_2... etc. We define $gcd(r, 0_R) = gcd(0_R, r) = StandardAssociate(r)$ and $gcd(0_R, 0_R) = 0_R$.

```
gap> Gcd( Integers, 123, 66 );
3
```

`Gcd` calls R`.operations.Gcd` repeatedly, each time passing the result of the previous call and the next argument, and returns the value of the last call.

The default function called this way is `RingOps.Gcd`, which applies the Euclidean algorithm to compute the greatest common divisor. Special categories of rings overlay this default function with more efficient functions.

5.27 GcdRepresentation

`GcdRepresentation(` *r1* `,` *r2* `... )`
`GcdRepresentation(` *R*`,` *r1* `,` *r2* `... )`

In the first form `GcdRepresentation` returns the representation of the greatest common divisor of the ring elements *r1*, *r2*... etc. in their default ring (see 5.3). In the second form `GcdRepresentation` returns the representation of the greatest common divisor of the ring elements *r1*, *r2*... etc. in the ring *R*. *R* must be a Euclidean ring (see 5.10) so that `Gcd` (see 5.26) can be applied to its elements. The representation is returned as a list of ring elements.

The representation of the gcd g of the elements r_1, r_2... etc. of a ring R is a list of ring elements s_1, s_2... etc. of R, such that $g = s_1r_1 + s_2r_2...$. That this representation exists can be shown using the Euclidean algorithm, which in fact can compute those coefficients.

```
gap> GcdRepresentation( 123, 66 );
[ 7, -13 ]     # 3 = 7*123 - 13*66
gap> Gcd( 123, 66 ) = last * [ 123, 66 ];
true
```

`GcdRepresentation` calls R`.operations.GcdRepresentation` repeatedly, each time passing the gcd result of the previous call and the next argument, and returns the value of the last call.

The default function called this way is `RingOps.GcdRepresentation`, which applies the Euclidean algorithm to compute the greatest common divisor and its representation. Special categories of rings overlay this default function with more efficient functions.

5.28 Lcm

`Lcm(` *r1* `,` *r2* `... )`
`Lcm(` *R*`,` *r1* `,` *r2* `... )`

In the first form `Lcm` returns the least common multiple of the ring elements *r1*, *r2*... etc. in their default ring (see 5.3). In the second form `Lcm` returns the least common multiple of the ring elements *r1*, *r2*,... etc. in the ring *R*. *R* must be a Euclidean ring (see 5.10) so that Gcd (see 5.26) can be applied to its elements. Lcm returns the standard associate (see 5.14) of the least common multiples.

A least common multiple of the elements r_1, r_2... etc. of the ring R is an element of smallest Euclidean degree (see 5.19) that is a multiple of r_1, r_2... etc. We define $lcm(r, 0_R) = lcm(0_R, r) = StandardAssociate(r)$ and $Lcm(0_R, 0_R) = 0_R$.

Lcm uses the equality $lcm(m, n) = m * n/gcd(m, n)$ (see 5.26).

```
gap> Lcm( Integers, 123, 66 );
2706
```

Lcm calls *R*.operations.Lcm repeatedly, each time passing the result of the previous call and the next argument, and returns the value of the last call.

The default function called this way is RingOps.Lcm, which simply returns the product of r with the quotient of s and the greatest common divisor of r and s. Special categories of rings overlay this default function with more efficient functions.

5.29 Ring Records

A ring R is represented by a record with the following entries.

`isDomain`
: is of course always the value `true`.

`isRing`
: is of course always the value `true`.

`isCommutativeRing`
: is `true` if the multiplication is known to be commutative, `false` if the multiplication is known to be noncommutative, and unbound otherwise.

`isIntegralRing`
: is `true` if R is known to be a commutative domain with 1 without zero divisor, `false` if R is known to lack one of these properties, and unbound otherwise.

`isUniqueFactorizationRing`
: is `true` if R is known to be a domain with unique factorization into primes, `false` if R is known to have a nonunique factorization, and unbound otherwise.

`isEuclideanRing`
: is `true` if R is known to be a Euclidean domain, `false` if it is known not to be a Euclidean domain, and unbound otherwise.

`zero`
: is the additive neutral element.

`units`
: is the list of units of the ring if it is known.

`size`
: is the size of the ring if it is known. If the ring is not finite this is the string "infinity".

`one`
: is the multiplicative neutral element, if the ring has one.

`integralBase`
: if the ring is, as additive group, isomorphic to the direct product of a finite number of copies of Z this contains a base.

As an example of a ring record, here is the definition of the ring record `Integers`.

```
rec(

    # category components
    isDomain                    := true,
    isRing                      := true,

    # identity components
    generators                  := [ 1 ],
    zero                        := 0,
    one                         := 1,
    name                        := "Integers",

    # knowledge components
    size                        := "infinity",
    isFinite                    := false,
    isCommutativeRing           := true,
    isIntegralRing              := true,
    isUniqueFactorizationRing   := true,
    isEuclideanRing             := true,
    units                       := [ -1, 1 ],

    # operations record
    operations                  := rec(
        ...
        IsPrime                 := function ( Integers, n )
            return IsPrimeInt( n );
        end,
        ...
        'mod'                   := function ( Integers, n, m )
            return n mod m;
        end,
        ... ) )
```

Chapter 6

Fields

Fields are important algebraic domains. Mathematically a **field** is a commutative ring F (see chapter 5), such that every element except 0 has a multiplicative inverse. Thus F has two operations `+` and `*` called addition and multiplication. $(F,+)$ must be an abelian group, whose identity is called 0_F. $(F-\{0_F\},*)$ must be an abelian group, whose identity element is called 1_F.

GAP3 supports the field of rationals (see 12), subfields of cyclotomic fields (see 15), and finite fields (see 18).

This chapter begins with sections that describe how to test whether a domain is a field (see 6.1), how to find the smallest field and the default field in which a list of elements lies (see 6.2 and 6.3), and how to view a field over a subfield (see 6.4).

The next sections describes the operation applicable to field elements (see 6.5 and 6.6).

The next sections describe the functions that are applicable to fields (see 6.7) and their elements (see 6.12, 6.10, 6.11, 6.9, and 6.8).

The following sections describe homomorphisms of fields (see 6.13, 6.14, 6.15, 6.16).

The last section describes how fields are represented internally (see 6.17).

Fields are domains, so all functions that are applicable to all domains are also applicable to fields (see chapter 4).

All functions for fields are in `LIBNAME/"field.g"`.

6.1 IsField

`IsField(` *D* `)`

`IsField` returns `true` if the object *D* is a field and `false` otherwise.

More precisely `IsField` tests whether *D* is a field record (see 6.17). So, for example, a matrix group may in fact be a field, yet `IsField` would return `false`.

```
gap> IsField( GaloisField(16) );
true
gap> IsField( CyclotomicField(9) );
```

```
true
gap> IsField( rec( isDomain := true, isField := true ) );
true     # it is possible to fool IsField
gap> IsField( AsRing( Rationals ) );
false     # though this ring is, as a set, still Rationals
```

6.2 Field

`Field( z,.. )` `Field( list )`

In the first form `Field` returns the smallest field that contains all the elements *z*,.. etc. In the second form `Field` returns the smallest field that contains all the elements in the list *list*. If any element is not an element of a field or the elements lie in no common field an error is raised.

```
gap> Field( Z(4) );
GF(2^2)
gap> Field( E(9) );
CF(9)
gap> Field( [ Z(4), Z(9) ] );
Error, CharFFE: <z> must be a finite field element, vector, or matrix
gap> Field( [ E(4), E(9) ] );
CF(36)
```

`Field` differs from `DefaultField` (see 6.3) in that it returns the smallest field in which the elements lie, while `DefaultField` may return a larger field if that makes sense.

6.3 DefaultField

`DefaultField( z,.. )` `DefaultField( list )`

In the first form `DefaultField` returns the default field that contains all the elements *z*,.. etc. In the second form `DefaultField` returns the default field that contains all the elements in the list *list*. If any element is not an element of a field or the elements lie in no common field an error is raised.

The field returned by `DefaultField` need not be the smallest field in which the elements lie. For example for elements from cyclotomic fields `DefaultField` may return the smallest cyclotomic field in which the elements lie, which need not be the smallest field overall, because the elements may in fact lie in a smaller number field which is not a cyclotomic field.

For the exact definition of the default field of a certain type of elements read the chapter describing this type (see 18 and 15).

`DefaultField` is used by `Conjugates`, `Norm`, `Trace`, `CharPol`, and `MinPol` (see 6.12, 6.10, 6.11, 6.9, and 6.8) if no explicit field is given.

```
gap> DefaultField( Z(4) );
GF(2^2)
gap> DefaultField( E(9) );
CF(9)
gap> DefaultField( [ Z(4), Z(9) ] );
```

```
Error, CharFFE: <z> must be a finite field element, vector, or matrix
gap> DefaultField( [ E(4), E(9) ] );
CF(36)
```

`Field` (see 6.2) differs from `DefaultField` in that it returns the smallest field in which the elements lie, while `DefaultField` may return a larger field if that makes sense.

6.4 Fields over Subfields

F / *G*

The quotient operator `/` evaluates to a new field H. This field has the same elements as F, i.e., is a domain equal to F. However H is viewed as a field over the field G, which must be a subfield of F.

What subfield a field is viewed over determines its Galois group. As described in 6.7 the Galois group is the group of field automorphisms that leave the subfield fixed. It also influences the results of 6.10, 6.11, 6.9, and 6.8, because they are defined in terms of the Galois group.

```
gap> F := GF(2^12);
GF(2^12)
gap> G := GF(2^2);
GF(2^2)
gap> Q := F / G;
GF(2^12)/GF(2^2)
gap> Norm( F, Z(2^6) );
Z(2)^0
gap> Norm( Q, Z(2^6) );
Z(2^2)^2
```

The operator `/` calls *G*`.operations./(` *F*`,` *G* `)`.

The default function called this way is `FieldOps./`, which simply makes a copy of F and enters G into the record component *F*`.field` (see 6.17).

6.5 Comparisons of Field Elements

f = *g*
f <> *g*

The equality operator `=` evaluates to `true` if the two field elements f and g are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the two field elements f and g are not equal, and to `false` otherwise. Note that any two field elements can be compared, even if they do not lie in compatible fields. In this case they cn, of course, never be equal. For each type of fields the equality of those field elements is given in the respective chapter.

Note that you can compare field elements with elements of other types; of course they are never equal.

f < *g*
f <= *g*

f `>` *g*
f `>=` *g*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the field element f is less than, less than or equal to, greater than, or greater than or equal to the field element g. For each type of fields the definition of the ordering of those field elements is given in the respective chapter. The ordering of field elements is as follows. Rationals are smallest, next are cyclotomics, followed by finite field elements.

Note that you can compare field elements with elements of other types; they are smaller than everything else.

6.6 Operations for Field Elements

The following operations are always available for field elements. Of course the operands must lie in compatible fields, i.e., the fields must be equal, or at least have a common superfield.

f `+` *g*

The operator `+` evaluates to the sum of the two field elements f and g, which must lie in compatible fields.

f `-` *g*

The operator `-` evaluates to the difference of the two field elements f and g, which must lie in compatible fields.

f `*` *g*

The operator `*` evaluates to the product of the two field elements f and g, which must lie in compatible fields.

f `/` *g*

The operator `/` evaluates to the quotient of the two field elements f and g, which must lie in compatible fields. If the divisor is 0 an error is signalled.

f `^` *n*

The operator `^` evaluates to the n-th power of the field element f. If n is a positive integer then *f*`^`*n* is *f*`*`*f*`*..*`*f* (n factors). If n is a negative integer *f*`^`*n* is defined as $1/f^{-n}$. If 0 is raised to a negative power an error is signalled. Any field element, even 0, raised to the 0-th power yields 1.

For the precedence of the operators see 2.10.

6.7 GaloisGroup

`GaloisGroup(` *F* `)`

`GaloisGroup` returns the Galois group of the field F as a group (see 7) of field automorphisms (see 6.13).

The Galois group of a field F over a subfield *F*`.field` is the group of automorphisms of F that leave the subfield *F*`.field` fixed. This group can be interpreted as a permutation group permuting the zeroes of the characteristic polynomial of a primitive element of F. The degree of this group is equal to the number of zeroes, i.e., to the dimension of F as a vector space over the subfield F.field. It operates transitively on those zeroes. The normal divisors of the Galois group correspond to the subfields between F and F.field.

```
gap> G := GaloisGroup( GF(4096)/GF(4) );;
gap> Size( G );
6
gap> IsCyclic( G );
true     # the Galois group of every finite field is
         # generated by the Frobenius automorphism
gap> H := GaloisGroup( CF(60) );;
gap> Size( H );
16
gap> IsAbelian( H );
true
```

The default function FieldOps.GaloisGroup just raises an error, since there is no general method to compute the Galois group of a field. This default function is overlaid by more specific functions for special types of domains (see 18.13 and 15.8).

6.8 MinPol

MinPol(z)
MinPol(F, z)

In the first form MinPol returns the coefficients of the minimal polynomial of the element z in its default field over its prime field (see 6.3). In the second form MinPol returns the coefficients of the minimal polynomial of the element z in the field F over the subfield F.field.

Let F/S be a field extension and L a minimal normal extension of S, containing F. The **minimal polynomial** of z in F over S is the squarefree polynomial whose roots are precisely the conjugates of z in L (see 6.12). Because the set of conjugates is fixed under the Galois group of L over S (see 6.7), so is the polynomial. Thus all the coefficients of the minimal polynomial lie in S.

```
gap> MinPol( Z(2^6) );
[ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ]
gap> MinPol( GF(2^12), Z(2^6) );
[ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ]
gap> MinPol( GF(2^12)/GF(2^2), Z(2^6) );
[ Z(2^2), Z(2)^0, Z(2)^0, Z(2)^0 ]
```

The default function FieldOps.MinPol, which works only for extensions with abelian Galois group, multiplies the linear factors $x - c$ with c ranging over the set of conjugates of z in F (see 6.12). For generic algebraic extensions, it is overlayed by solving a system of linear equations, given by the coefficients of powers of z in respect to a given base.

6.9 CharPol

CharPol(z)
CharPol(F, z)

In the first form `CharPol` returns the coefficients of the characteristic polynomial of the element z in its default field over its prime field (see 6.3). In the second form `CharPol` returns the coefficients of the characteristic polynomial of the element z in the field F over the subfield F`.field`. The characteristic polynomial is returned as a list of coefficients, the i-th entry is the coefficient of x^{i-1}.

The **characteristic polynomial** of an element z in a field F over a subfield S is the $\frac{[F:S]}{\deg\mu}$-th power of μ, where μ denotes the minimal polynomial of z in F over S. It is fixed under the Galois group of the normal closure of F. Thus all the coefficients of the characteristic polynomial lie in S. The constant term is $(-1)^{F.degree/S.degree} = (-1)^{[F:S]}$ times the norm of z (see 6.10), and the coefficient of the second highest degree term is the negative of the trace of z (see 6.11). The roots (including their multiplicities) in F of the characteristic polynomial of z in F are the conjugates (see 6.12) of z in F.

```
gap> CharPol( Z(2^6) );
[ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ]
gap> CharPol( GF(2^12), Z(2^6) );
[ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2),
  Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ]
gap> CharPol( GF(2^12)/GF(2^2), Z(2^6) );
[ Z(2^2)^2, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]
```

The default function `FieldOps.CharPol` multiplies the linear factors $x - c$ with c ranging over the conjugates of z in F (see 6.12). For nonabelian extensions, it is overlayed by a function, which computes the appropriate power of the minimal polynomial.

6.10 Norm

`Norm(` z `)`
`Norm(` F`,` z `)`

In the first form `Norm` returns the norm of the field element z in its default field over its prime field (see 6.3). In the second form `Norm` returns the norm of z in the field F over the subfield F`.field`.

The **norm** of an element z in a field F over a subfield S is $(-1)^{F.degree/S.degree} = (-1)^{[F:S]}$ times the constant term of the characteristic polynomial of z (see 6.9). Thus the norm lies in S. The norm is the product of all conjugates of z in the normal closure of F over S (see 6.12).

```
gap> Norm( Z(2^6) );
Z(2)^0
gap> Norm( GF(2^12), Z(2^6) );
Z(2)^0
gap> Norm( GF(2^12)/GF(2^2), Z(2^6) );
Z(2^2)^2
```

The default function `FieldOps.Norm` multiplies the conjugates of z in F (see 6.12). For nonabelian extensions, it is overlayed by a function, which obtains the norm from the characteristic polynomial.

6.11 Trace

`Trace( z )`
`Trace( F, z )`

In the first form `Trace` returns the trace of the field element z in its default field over its prime field (see 6.3). In the second form `Trace` returns the trace of the element z in the field F over the subfield F`.field`.

The **trace** of an element z in a field F over a subfield S is the negative of the coefficient of the second highest degree term of the characteristic polynomial of z (see 6.9). Thus the trace lies in S. The trace is the sum over all conjugates of z in the normal closure of F over S (see 6.12).

```
gap> Trace( Z(2^6) );
0*Z(2)
gap> Trace( GF(2^12), Z(2^6) );
0*Z(2)
gap> Trace( GF(2^12)/GF(2^2), Z(2^6) );
0*Z(2)
```

The default function `FieldOps.Trace` adds the conjugates of z in F (see 6.12). For non-abelian extensions, this is overlayed by a function, which obtains the trace from the characteristic polynomial.

6.12 Conjugates

`Conjugates( z )`
`Conjugates( F, z )`

In the first form `Conjugates` returns the list of conjugates of the field element z in its default field over its prime field (see 6.3). In the second form `Conjugates` returns the list of conjugates of the field element z in the field F over the subfield F`.field`. In either case the list may contain duplicates if z lies in a proper subfield of its default field, respectively of F.

The **conjugates** of an element z in a field F over a subfield S are the roots in F of the characteristic polynomial of z in F (see 6.9). If F is a normal extension of S, then the conjugates of z are the images of z under all elements of the Galois group of F over S (see 6.7), i.e., under those automorphisms of F that leave S fixed. The number of different conjugates of z is given by the degree of the smallest extension of S in which z lies.
For a normal extension F, `Norm` (see 6.10) computes the product, `Trace` (see 6.11) the sum of all conjugates. `CharPol` (see 6.9) computes the polynomial that has precisely the conjugates with their corresponding multiplicities as roots, `MinPol` (see 6.8) the squarefree polynomial that has precisely the conjugates as roots.

```
gap> Conjugates( Z(2^6) );
[ Z(2^6), Z(2^6)^2, Z(2^6)^4, Z(2^6)^8, Z(2^6)^16, Z(2^6)^32 ]
gap> Conjugates( GF(2^12), Z(2^6) );
[ Z(2^6), Z(2^6)^2, Z(2^6)^4, Z(2^6)^8, Z(2^6)^16, Z(2^6)^32, Z(2^6),
  Z(2^6)^2, Z(2^6)^4, Z(2^6)^8, Z(2^6)^16, Z(2^6)^32 ]
gap> Conjugates( GF(2^12)/GF(2^2), Z(2^6) );
```

```
[ Z(2^6), Z(2^6)^4, Z(2^6)^16, Z(2^6), Z(2^6)^4, Z(2^6)^16 ]
```

The default function `FieldOps.Conjugates` applies the automorphisms of the Galois group of F (see 6.7) to z and returns the list of images. For nonabelian extensions, this is overlayed by a factorization of the characteristic polynomial.

6.13 Field Homomorphisms

Field homomorphisms are an important class of homomorphisms in GAP3 (see chapter 44).

A **field homomorphism** ϕ is a mapping that maps each element of a field F, called the source of ϕ, to an element of another field G, called the range of ϕ, such that for each pair $x, y \in F$ we have $(x+y)^\phi = x^\phi + y^\phi$ and $(xy)^\phi = x^\phi y^\phi$. We also require that ϕ maps the one of F to the one of G (that ϕ maps the zero of F to the zero of G is implied by the above relations).

An Example of a field homomorphism is the Frobinius automorphism of a finite field (see 18.11). Look under **field homomorphisms** in the index for a list of all available field homomorphisms.

Since field homomorphisms are just a special case of homomorphisms, all functions described in chapter 44 are applicable to all field homomorphisms, e.g., the function to test if a homomorphism is a an automorphism (see 44.6). More general, since field homomorphisms are just a special case of mappings all functions described in chapter 43 are also applicable, e.g., the function to compute the image of an element under a homomorphism (see 43.8).

The following sections describe the functions that test whether a mapping is a field homomorphism (see 6.14), compute the kernel of a field homomorphism (see 6.15), and how the general mapping functions are implemented for field homomorphisms.

6.14 IsFieldHomomorphism

`IsFieldHomomorphism(` *map* `)`

`IsFieldHomomorphism` returns `true` if the mapping *map* is a field homomorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping *map* is a field homomorphism if its source F and range G are both fields and if for each pair of elements $x, y \in F$ we have $(x+y)^{map} = x^{map} + y^{map}$ and $(xy)^{map} = x^{map} y^{map}$. We also require that $1_F^{map} = 1_G$.

```
gap> f := GF( 16 );
GF(2^4)
gap> fun := FrobeniusAutomorphism( f );
FrobeniusAutomorphism( GF(2^4) )
gap> IsFieldHomomorphism( fun );
true
```

`IsFieldHomomorphism` first tests if the flag *map*`.isFieldHomomorphism` is bound. If the flag is bound, `IsFieldHomomorphism` returns its value. Otherwise it calls *map*`.source.operations.IsFieldHomomorphism(` *map* `)`, remembers the returned value in *map*`.isFieldHomomorphism`, and returns it. Note that of course all functions that create field homomorphism set the flag *map*`.isFieldHomomorphism` to `true`, so that no function is called for those field homomorphisms.

The default function called this way is `MappingOps.IsFieldHomomorphism`. It computes all the elements of the source of *map* and for each pair of elements x, y tests whether $(x + y)^{map} = x^{map} + y^{map}$ and $(xy)^{map} = x^{map}y^{map}$. Look under **IsHomomorphism** in the index to see for which mappings this function is overlaid.

6.15 KernelFieldHomomorphism

`KernelFieldHomomorphism(` *hom* `)`

`KernelFieldHomomorphism` returns the kernel of the field homomorphism *hom*.

Because the kernel must be a ideal in the source and it can not be the full source (because we require that the one of the source is mapped to the one of the range), it must be the trivial ideal. Therefor the kernel of every field homomorphism is the set containing only the zero of the source.

6.16 Mapping Functions for Field Homomorphisms

This section describes how the mapping functions defined in chapter 43 are implemented for field homomorphisms. Those functions not mentioned here are implemented by the default functions described in the respective sections.

`IsInjective(` *hom* `)`

Always returns `true` (see 6.15).

`IsSurjective(` *hom* `)`

The field homomorphism *hom* is surjective if the size of the image `Size( Image(` *hom* `) )` is equal to the size of the range `Size(` *hom*`.range )`.

hom1 `=` *hom2*

The two field homomorphism *hom1* and *hom2* are are equal if the have the same source and range and if the images of the generators of the source under *hom1* and *hom2* are equal.

`Image(` *hom* `)`
`Image(` *hom*`,` *H* `)`
`Images(` *hom*`,` *H* `)`

The image of a subfield under a field homomorphism is computed by computing the images of a set of generators of the subfield, and the result is the subfield generated by those images.

`PreImage(` *hom* `)`
`PreImage(` *hom*`,` *H* `)`
`PreImages(` *hom*`,` *H* `)`

The preimages of a subfield under a field homomorphism are computed by computing the preimages of all the generators of the subfield, and the result is the subfield generated by those elements.

Look in the index under **IsInjective**, **IsSurjective**, **Image**, **Images**, **PreImage**, **PreImages**, and **equality** to see for which field homomorphisms these functions are overlaid.

6.17 Field Records

A field is represented by a record that contains important information about this field. The GAP3 library predefines some field records, for example `Rationals` (see 12). Field constructors construct others, for example `Field` (see 6.2), and `GaloisField` (see 18.10). Of course you may also create such a record by hand.

All field records contain the components `isDomain`, `isField`, `char`, `degree`, `generators`, `zero`, `one`, `field`, `base`, and `dimension`. They may also contain the optional components `isFinite`, `size`, `galoisGroup`. The contents of all components of a field F are described below.

`isDomain`
: is always `true`. This indicates that F is a domain.

`isField`
: is always `true`. This indicates that F is a field.

`char`
: is the characteristic of F. For finite fields this is always a prime, for infinite fields this is 0.

`degree`
: is the degree of F **as extension of the prime field**, not as extension of the subfield S. For finite fields the order of F is given by *F*`.char^`*F*`.degree`.

`generators`
: a list of elements that together generate F. That is F is the smallest field over the prime field given by *F*`.char` that contains the elements of *F*`.generators`.

`zero`
: is the additive neutral element of the finite field.

`one`
: is the multiplicative neutral element of the finite field.

`field`
: is the subfield S over which F was constructed. This is either a field record for S, or the same value as *F*`.char`, denoting the prime field (see 6.4).

`base`
: is a list of elements of F forming a base for F as vector space over the subfield S.

`dimension`
: is the dimension of F as vector space over the subfield S.

`isFinite`
: if present this is `true` if the field F is finite and `false` otherwise.

`size`
: if present this is the size of the field F. If F is infinite this holds the string "infinity".

`galoisGroup`
: if present this holds the Galois group of F (see 6.7).

Chapter 7

Groups

Finitely generated groups and their subgroups are important domains in GAP3. They are represented as permutation groups, matrix groups, ag groups or even more complicated constructs as for instance automorphism groups, direct products or semi-direct products where the group elements are represented by records.

Groups are created using `Group` (see 7.9), they are represented by records that contain important information about the groups. Subgroups are created as subgroups of a given group using `Subgroup`, and are also represented by records. See 7.6 for details about the distinction between groups and subgroups.

Because this chapter is very large it is split into several parts. Each part consists of several sections.

Note that some functions will only work if the elements of a group are represented in an unique way. This is not true in finitely presented groups, see 23.3 for a list of functions applicable to finitely presented groups.

The first part describes the operations and functions that are available for group elements, e.g., `Order` (see 7.1). The next part tells your more about the distinction of parent groups and subgroups (see 7.6). The next parts describe the functions that compute subgroups, e.g., `SylowSubgroup` (7.14), and series of subgroups, e.g., `DerivedSeries` (see 7.36). The next part describes the functions that compute and test properties of groups, e.g., `AbelianInvariants` and `IsSimple` (see 7.45), and that identify the isomorphism type. The next parts describe conjugacy classes of elements and subgroups (see 7.67) and cosets (see 7.84). The next part describes the functions that create new groups, e.g., `DirectProduct` (see 7.98). The next part describes group homomorphisms, e.g., `NaturalHomomorphism` (see 7.106). The last part tells you more about the implementation of groups, e.g., it describes the format of group records (see 7.114).

The functions described in this chapter are implemented in the following library files. `LIBNAME/"grpelms.g"` contains the functions for group elements, `LIBNAME/"group.g"` contains the dispatcher and default group functions, `LIBNAME/"grpcoset.g"` contains the functions for cosets and factor groups, `LIBNAME/"grphomom.g"` implements the group homomorphisms, and `LIBNAME/"grpprods.g"` implements the group constructions.

7.1 Group Elements

The following sections describe the operations and functions available for group elements (see 7.2, 7.3, 7.4, and 7.5).

Note that group elements usually exist independently of a group, e.g., you can write down two permutations and compute their product without ever defining a group that contains them.

7.2 Comparisons of Group Elements

g `=` *h*
g `<>` *h*

The equality operator `=` evaluates to `true` if the group elements g and h are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the group elements g and h are not equal and to `false` otherwise.

You can compare group elements with objects of other types. Of course they are never equal. Standard group elements are permutations, ag words and matrices. For examples of generic group elements see for instance 7.99.

g `<` *h*
g `<=` *h*
g `>=` *h*
g `>` *h*

The operators `<`, `<=`, `>=` and `>` evaluate to `true` if the group element g is strictly less than, less than or equal to, greater than or equal to and strictly greater than the group element h. There is no general ordering on group elements.

Standard group elements may be compared with objects of other types while generic group elements may disallow such a comparison.

7.3 Operations for Group Elements

g `*` *h*
g `/` *h*

The operators `*` and `/` evaluate to the product and quotient of the two group elements g and h. The operands must of course lie in a common parent group, otherwise an error is signaled.

g `^` *h*

The operator `^` evaluates to the conjugate $h^{-1} * g * h$ of g under h for two group elements elements g and h. The operands must of course lie in a common parent group, otherwise an error is signaled.

g `^` *i*

The powering operator ^ returns the i-th power of a group element g and an integer i. If i is zero the identity of a parent group of g is returned.

list `*` *g*
g `*` *list*

In this form the operator `*` returns a new list where each entry is the product of g and the corresponding entry of *list*. Of course multiplication must be defined between g and each entry of *list*.

list `/` *g*

In this form the operator `/` returns a new list where each entry is the quotient of g and the corresponding entry of *list*. Of course division must be defined between g and each entry of *list*.

`Comm(` *g* `,` *h* `)`

`Comm` returns the commutator $g^{-1} * h^{-1} * g * h$ of two group elements g and h. The operands must of course lie in a common parent group, otherwise an error is signaled.

`LeftNormedComm(` *g1* `, ...,` *gn* `)`

`LeftNormedComm` returns the left normed commutator `Comm( LeftNormedComm(` *g1* `, ...,` *gn-1* `),` *gn* `)` of group elements *g1*, ..., *gn*. The operands must of course lie in a common parent group, otherwise an error is signaled.

`RightNormedComm(` *g1* `,` *g2* `, ...,` *gn* `)`

`RightNormedComm` returns the right normed commutator `Comm(` *g1* `, RightNormedComm(` *g2* `, ...,` *gn* `) )` of group elements *g1*, ..., *gn*. The operands must of course lie in a common parent group, otherwise an error is signaled.

`LeftQuotient(` *g* `,` *h* `)`

`LeftQuotient` returns the left quotient $g^{-1} * h$ of two group elements g and h. The operands must of course lie in a common parent group, otherwise an error is signaled.

7.4 IsGroupElement

`IsGroupElement(` *obj* `)`

`IsGroupElement` returns `true` if *obj*, which may be an object of arbitrary type, is a group element, and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsGroupElement( 10 );
false
gap> IsGroupElement( (11,10) );
true
gap> IsGroupElement( IdWord );
true
```

7.5 Order

`Order(` G`,` g `)`

`Order` returns the order of a group element g in the group G.

The **order** is the smallest positive integer i such that g^i is the identity. The order of the identity is one.

```
gap> Order( Group( (1,2), (1,2,3,4) ), (1,2,3) );
3
gap> Order( Group( (1,2), (1,2,3,4) ), () );
1
```

7.6 More about Groups and Subgroups

GAP3 distinguishs between parent groups and subgroups of parent groups. Each subgroup belongs to a unique parent group. We say that this parent group is the **parent** of the subgroup. We also say that a parent group is its own parent.

Parent groups are constructed by `Group` and subgroups are constructed by `Subgroup`. The first argument of `Subgroup` must be a parent group, i.e., it must not be a subgroup of a parent group, and this parent group will be the parent of the constructed subgroup.

Those group functions that take more than one argument require that the arguments have a common parent. Take for instance `CommutatorSubgroup`. It takes two arguments, a group G and a group H, and returns the commutator subgroup of H with G. So either G is a parent group, and H is a subgroup of this parent group, or G and H are subgroups of a common parent group P.

```
gap> s4 := Group( (1,2), (1,2,3,4) );
Group( (1,2), (1,2,3,4) )
gap> c3 := Subgroup( s4, [ (1,2,3) ] );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2,3) ] )
gap> CommutatorSubgroup( s4, c3 );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,3,2), (1,2,4) ] )
# ok, c3 is a subgroup of the parent group s4
gap> a4 := Subgroup( s4, [ (1,2,3), (2,3,4) ] );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2,3), (2,3,4) ] )
gap> CommutatorSubgroup( a4, c3 );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,4)(2,3), (1,3)(2,4) ] )
# also ok, c3 and a4 are subgroups of the parent group s4
gap> x3 := Group( (1,2,3) );
Group( (1,2,3) )
gap> CommutatorSubgroup( s4, x3 );
Error, <G> and <H> must have the same parent group
# not ok, s4 is its own parent and x3 is its own parent
```

Those functions that return new subgroups, as with `CommutatorSubgroup` above, return this subgroup as a subgroup of the common parent of their arguments. Note especially that the commutator subgroup of `c3` with `a4` is returned as a subgroup of their common parent group `s4`, not as a subgroup of `a4`. It can not be a subgroup of `a4`, because subgroups must be subgroups of parent groups, and `a4` is not a parent group. Of course, mathematically the commutator subgroup is a subgroup of `a4`.

Note that a subgroup of a parent group need not be a proper subgroup, as can be seen in the following example.

```
gap> s4 := Group( (1,2), (1,2,3,4) );
Group( (1,2), (1,2,3,4) )
gap> x4 := Subgroup( s4, [ (1,2,3,4), (3,4) ] );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2,3,4), (3,4) ] )
gap> Index( s4, x4 );
1
```

One exception to the rule are functions that construct new groups such as `DirectProduct`. They accept groups with different parents. If you want rename the function `DirectProduct` to `OuterDirectProduct`.

Another exception is `Intersection` (see 4.12), which allows groups with different parent groups, it computes the intersection in such cases as if the groups were sets of elements. This is because `Intersection` is not a group function, but a domain function, i.e., it accepts two (or more) arbitrary domains as arguments.

Whenever you have two subgroups which have different parent groups but have a common supergroup G you can use `AsSubgroup` (see 7.13) in order to construct new subgroups which have a common parent group G.

```
gap> s4 := Group( (1,2), (1,2,3,4) );
Group( (1,2), (1,2,3,4) )
gap> x3 := Group( (1,2,3) );
Group( (1,2,3) )
gap> CommutatorSubgroup( s4, x3 );
Error, <G> and <H> must have the same parent group
#  not ok, s4 is its own parent and x3 is its own parent
gap> c3 := AsSubgroup( s4, x3 );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2,3) ] )
gap> CommutatorSubgroup( s4, c3 );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,3,2), (1,2,4) ] )
```

The following sections describe the functions related to this concept (see 7.7, 7.8, 7.9, 7.10, 7.11, 7.12, 7.13).

7.7 IsParent

`IsParent(` G `)`

`IsParent` returns `true` if G is a parent group, and `false` otherwise (see 7.6).

7.8 Parent

`Parent(` U_1`, ...,` U_n `)`

`Parent` returns the common parent group of its subgroups and parent group arguments.

In case more than one argument is given, all groups must have the same parent group. Otherwise an error is signaled. This can be used to ensure that a collection of given subgroups have a common parent group.

7.9 Group

`Group(` U `)`

Let U be a parent group or a subgroup. `Group` returns a new parent group G which is isomorphic to U. The generators of G need not be the same elements as the generators of U. The default group function uses the same generators, while the ag group function may create new generators along with a new collector.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s3 := Subgroup( s4, [ (1,2,3), (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,3), (1,2) ] )
gap> Group( s3 ); # same elements
Group( (1,2,3), (1,2) )
gap> s4.1 * s3.1;
(1,3,4,2)
gap> s4 := AgGroup( s4 );
Group( g1, g2, g3, g4 )
gap> a4 := DerivedSubgroup( s4 );
Subgroup( Group( g1, g2, g3, g4 ), [ g2, g3, g4 ] )
gap> a4 := Group( a4 ); # different elements
Group( g1, g2, g3 )
gap> s4.1 * a4.1;
Error, AgWord op: agwords have different groups
```

`Group(` *list*, *id* `)`

`Group` returns a new parent group G generated by group elements $g_1, ..., g_n$ of *list*. *id* must be the identity of this group.

`Group(` g_1, ..., g_n `)`

`Group` returns a new parent group G generated by group elements g_1, ..., g_n.

The generators of this new parent group need not be the same elements as $g_1, ..., g_n$. The default group function however returns a group record with generators $g_1, ..., g_n$ and identity *id*, while the ag group function may create new generators along with a new collector.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> z4 := Group( s4.1 ); # same element
Group( (1,2,3,4) )
gap> s4.1 * z4.1;
(1,3)(2,4)
gap> s4 := AgGroup( s4 );
Group( g1, g2, g3, g4 )
gap> z4 := Group( s4.1 * s4.3 ); # different elements
Group( g1, g2 )
gap> s4.1 * z4.1;
Error, AgWord op: agwords have different groups
```

Let $g_{i_1}, ..., g_{i_m}$ be the set of nontrivial generators in all four cases. `Groups` sets record components G`.1`, ..., G`.m` to these generators.

7.10 AsGroup

`AsGroup( D )`

Let D be a domain. `AsGroup` returns a group G such that the set of elements of D is the same as the set of elements of G if this is possible.

If D is a list of group elements these elements must form a group. Otherwise an error is signaled.

Note that this function returns a parent group or a subgroup of a parent group depending on D. In order to convert a subgroup into a parent group you must use `Group` (see 7.9).

```
gap> s4 := AgGroup( Group( (1,2,3,4), (2,3) ) );
Group( g1, g2, g3, g4 )
gap> Elements( last );
[ IdAgWord, g4, g3, g3*g4, g2, g2*g4, g2*g3, g2*g3*g4, g2^2, g2^2*g4,
  g2^2*g3, g2^2*g3*g4, g1, g1*g4, g1*g3, g1*g3*g4, g1*g2, g1*g2*g4,
  g1*g2*g3, g1*g2*g3*g4, g1*g2^2, g1*g2^2*g4, g1*g2^2*g3,
  g1*g2^2*g3*g4 ]
gap> AsGroup( last );
Group( g1, g2, g3, g4 )
```

The default function `GroupOps.AsGroup` for a group D returns a copy of D. If D is a subgroup then a subgroup is returned. The default function `GroupElementsOps.AsGroup` expects a list D of group elements forming a group and uses successively `Closure` in order to compute a reduced generating set.

7.11 IsGroup

`IsGroup( obj )`

`IsGroup` returns `true` if *obj*, which can be an object of arbitrary type, is a parent group or a subgroup and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsGroup( Group( (1,2,3) ) );
true
gap> IsGroup( 1/2 );
false
```

7.12 Subgroup

`Subgroup( G, L )`

Let G be a parent group and L be a list of elements $g_1, ..., g_n$ of G. `Subgroup` returns the subgroup U generated by $g_1, ..., g_n$ with parent group G.

Note that this function is the only group function in which the name `Subgroup` does not refer to the mathematical terms subgroup and supergroup but to the implementation of groups as subgroups and parent groups. `IsSubgroup` (see 7.62) is not the negation of `IsParent` (see 7.7) but decides subgroup and supergroup relations.

`Subgroup` always binds a copy of L to U`.generators`, so it is safe to modify L after calling `Subgroup` because this will not change the entries in U.

Let $g_{i_1}, ..., g_{i_m}$ be the nontrivial generators. `Subgroups` binds these generators to U`.1`, ..., U`.m`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> v4 := Subgroup( s4, [ (1,2), (1,2)(3,4) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2), (1,2)(3,4) ] )
gap> IsParent( v4 );
false
```

7.13 AsSubgroup

`AsSubgroup(` G`,` U `)`

Let G be a parent group and U be a parent group or a subgroup with a possibly different parent group, such that the generators $g_1, ..., g_n$ of U are elements of G. `AsSubgroup` returns a new subgroup S such that S has parent group G and is generated by $g_1, ..., g_n$.

```
gap> d8 := Group( (1,2,3,4), (1,2)(3,4) );
Group( (1,2,3,4), (1,2)(3,4) )
gap> z := Centre( d8 );
Subgroup( Group( (1,2,3,4), (1,2)(3,4) ), [ (1,3)(2,4) ] )
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> Normalizer( s4, AsSubgroup( s4, z ) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (2,4), (1,2,3,4), (1,3)(2,4)
 ] )
```

7.14 Subgroups

The following sections describe functions that compute certain subgroups of a given group, e.g., `SylowSubgroup` computes a Sylow subgroup of a group (see 7.16, 7.17, 7.18, 7.19, 7.20, 7.21, 7.22, 7.23, 7.24, 7.25, 7.26, 7.27, 7.28, 7.29, 7.30, 7.31, 7.32).

They return group records as described in 7.118 for the computed subgroups. Some functions may not terminate if the given group has an infinite set of elements, while other functions may signal an error in such cases.

Here the term "subgroup" is used in a mathematical sense. But in GAP3, every group is either a parent group or a subgroup of a unique parent group. If you compute a Sylow subgroup S of a group U with parent group G then S is a subgroup of U but its parent group is G (see 7.6).

Further sections describe functions that return factor groups of a given group (see 7.33 and 7.35).

7.15 Agemo

`Agemo(` G`,` p `)`

G must be a p-group. `Agemo` returns the subgroup of G generated by the p.th powers of the elements of G.

```
gap> d8 := Group( (1,3)(2,4), (1,2) );
Group( (1,3)(2,4), (1,2) )
gap> Agemo( d8, 2 );
Subgroup( Group( (1,3)(2,4), (1,2) ), [ (1,2)(3,4) ] )
```

The default function `GroupOps.Agemo` computes the subgroup of G generated by the p.th powers of the generators of G if G is abelian. Otherwise the function computes the normal closure of the p.th powers of the representatives of the conjugacy classes of G.

7.16 Centralizer

`Centralizer(` G, x `)`

`Centralizer` returns the centralizer of an element x in G where x must be an element of the parent group of G.

The **centralizer** of an element x in G is defined as the set C of elements c of G such that c and x commute.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> v4 := Centralizer( s4, (1,2) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (3,4), (1,2) ] )
```

The default function `GroupOps.Centralizer` uses `Stabilizer` (see 8.24) in order to compute the centralizer of x in G acting by conjugation.

`Centralizer(` G, U `)`

`Centralizer` returns the centralizer of a group U in G as group record. Note that G and U must have a common parent group.

The **centralizer** of a group U in G is defined as the set C of elements c of C such c commutes with every element of U.

If G is the parent group of U then `Centralizer` will set and test the record component U`.centralizer`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> v4 := Centralizer( s4, (1,2) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (3,4), (1,2) ] )
gap> c2 := Subgroup( s4, [ (1,3) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,3) ] )
gap> Centralizer( v4, c2 );
Subgroup( Group( (1,2,3,4), (1,2) ), [  ] )
```

The default function `GroupOps.Centralizer` uses `Stabilizer` in order to compute successively the stabilizer of the generators of U.

7.17 Centre

`Centre(` G `)`

`Centre` returns the centre of G.

The **centre** of a group G is defined as the centralizer of G in G.

Note that `Centre` sets and tests the record component G`.centre`.

```
gap> d8 := Group( (1,2,3,4), (1,2)(3,4) );
Group( (1,2,3,4), (1,2)(3,4) )
gap> Centre( d8 );
Subgroup( Group( (1,2,3,4), (1,2)(3,4) ), [ (1,3)(2,4) ] )
```

The default group function `GroupOps.Centre` uses `Centralizer` (see 7.16) in order to compute the centralizer of G in G.

7.18 Closure

`Closure(` U`,` g `)`

Let U be a group with parent group G and let g be an element of G. Then `Closure` returns the closure C of U and g as subgroup of G. The closure C of U and g is the subgroup generated by U and g.

```
gap> s4 := Group( (1,2,3,4), (1,2 ) );
Group( (1,2,3,4), (1,2) )
gap> s2 := Subgroup( s4, [ (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2) ] )
gap> Closure( s2, (3,4) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2), (3,4) ] )
```

The default function `GroupOps.Closure` returns U if U is a parent group, or if g or its inverse is a generator of U, or if the set of elements is known and g is in this set, or if g is trivial. Otherwise the function constructs a new subgroup C which is generated by the generators of U and the element g.

Note that if the set of elements of U is bound to U`.elements` then `GroupOps.Closure` computes the set of elements for C and binds it to C`.elements`.

If U is known to be non-abelian or infinite so is C. If U is known to be abelian the function checks whether g commutes with every generator of U.

`Closure(` U`,` S `)`

Let U and S be two group with a common parent group G. Then `Closure` returns the subgroup of G generated by U and S.

```
gap> s4 := Group( (1,2,3,4), (1,2 ) );
Group( (1,2,3,4), (1,2) )
gap> s2 := Subgroup( s4, [ (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2) ] )
gap> z3 := Subgroup( s4, [ (1,2,3) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,3) ] )
gap> Closure( z3, s2 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,3), (1,2) ] )
```

The default function `GroupOps.Closure` returns the parent of U and S if U or S is a parent group. Otherwise the function computes the closure of U under all generators of S.

Note that if the set of elements of U is bound to U`.elements` then `GroupOps.Closure` computes the set of elements for the closure C and binds it to C`.elements`.

7.19 CommutatorSubgroup

`CommutatorSubgroup(` G`,` H `)`

Let G and H be groups with a common parent group. `CommutatorSubgroup` returns the commutator subgroup $[G, H]$.

The **commutator subgroup** of G and H is the group generated by all commutators $[g, h]$ with $g \in G$ and $h \in H$.

See also `DerivedSubgroup` (7.22).

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> d8 := Group( (1,2,3,4), (1,2)(3,4) );
Group( (1,2,3,4), (1,2)(3,4) )
gap> CommutatorSubgroup( s4, AsSubgroup( s4, d8 ) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,3)(2,4), (1,3,2) ] )
```

Let G be generated by $g_1, ..., g_n$ and H be generated by $h_1, ..., h_m$. The normal closure of the subgroup S generated by $Comm(g_i, h_j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ under G and H is the commutator subgroup of G and H (see [Hup67]). The default function `GroupOps.CommutatorSubgroup` returns the normal closure of S under the closure of G and H.

7.20 ConjugateSubgroup

`ConjugateSubgroup(` U`,` g `)`

`ConjugateSubgroup` returns the subgroup U^g conjugate to U under g, which must be an element of the parent group of G.

If present, the flags U`.isAbelian`, U`.isCyclic`, U`.isElementaryAbelian`, U`.isFinite`, U`.isNilpotent`, U`.isPerfect`, U`.isSimple`, U`.isSolvable`, and U`.size` are copied to U^g.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> c2 := Subgroup( s4, [ (1,2)(3,4) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2)(3,4) ] )
gap> ConjugateSubgroup( c2, (1,3) );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,4)(2,3) ] )
```

The default function `GroupOps.ConjugateSubgroup` returns U if the set of elements of U is known and g is an element of this set or if g is a generator of U. Otherwise it conjugates the generators of U with g.

If the set of elements of U is known the default function also conjugates and binds it to the conjugate subgroup.

7.21 Core

`Core(` S`,` U `)`

Let S and U be groups with a common parent group G. Then `Core` returns the core of U under conjugation of S.

The **core** of a group U under a group S $Core_S(U)$ is the intersection $\bigcap_{s \in S} U^s$ of all groups conjugate to U under conjugation by elements of S.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s4.name := "s4";;
gap> d8 := Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] );
Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] )
gap> Core( s4, d8 );
Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] )
gap> Core( d8, s4 );
s4
```

The default function `GroupOps.Core` starts with U and replaces U with the intersection of U and a conjugate subgroup of U under a generator of G until the subgroup is normalized by G.

7.22 DerivedSubgroup

`DerivedSubgroup(` G `)`

`DerivedSubgroup` returns the derived subgroup $G' = [G, G]$ of G.

The **derived subgroup** of G is the group generated by all commutators $[g, h]$ with $g, h \in G$.

Note that `DerivedSubgroup` sets and tests G`.derivedSubgroup`. `CommutatorSubgroup` (see 7.19) allows you to compute the commutator group of two subgroups.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> DerivedSubgroup( s4 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,3,2), (2,4,3) ] )
```

Let G be generated by $g_1, ..., g_n$. Then the default function `GroupOps.DerivedSubgroup` returns the normal closure of S under G where S is the subgroup of G generated by $Comm(g_i, g_j)$ for $1 \le j < i \le n$.

7.23 FittingSubgroup

`FittingSubgroup(` G `)`

`FittingSubgroup` returns the Fitting subgroup of G.

The **Fitting subgroup** of a group G is the biggest nilpotent normal subgroup of G.

```
gap> s4;
Group( (1,2,3,4), (1,2) )
gap> FittingSubgroup( s4 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,3)(2,4), (1,4)(2,3) ] )
gap> IsNilpotent( last );
true
```

Let G be a finite group. Then the default group function `GroupOps.FittingSubgroup` computes the subgroup of G generated by the cores of the Sylow subgroups in G.

7.24 FrattiniSubgroup

`FrattiniSubgroup(` G `)`

`FrattiniSubgroup` returns the Frattini subgroup of group G.

The **Frattini subgroup** of a group G is the intersection of all maximal subgroups of G.

```
gap> s4 := SymmetricGroup( AgWords, 4 );;
gap> ss4 := SpecialAgGroup( s4 );;
gap> FrattiniSubgroup( ss4 );
Subgroup( Group( g1, g2, g3, g4 ), [  ] )
```

The generic method computes the Frattini subgroup as intersection of the cores (see 7.21) of the representatives of the conjugacy classes of maximal subgroups (see 7.80).

7.25 NormalClosure

`NormalClosure(` S`,` U `)`

Let S and U be groups with a common parent group G. Then `NormalClosure` returns the normal closure of U under S as a subgroup of G.

The **normal closure** N of a group U under the action of a group S is the smallest subgroup in G that contains U and is invariant under conjugation by elements of S. Note that N is independent of G.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s4.name := "s4";;
gap> d8 := Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] );
Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] )
gap> NormalClosure( s4, d8 );
Subgroup( s4, [ (1,2,3,4), (1,2)(3,4), (1,3,4,2) ] )
gap> last = s4;
true
```

7.26 NormalIntersection

`NormalIntersection(` N`,` U `)`

Let N and U be two subgroups with a common parent group. `NormalIntersection` returns the intersection in case U normalizes N.

Depending on the domain this may be faster than the general intersection algorithm (see 4.12). The default function `GroupOps.NormalIntersection` however uses `Intersection`.

7.27 Normalizer

`Normalizer(` S`,` U `)`

Let S and U be groups with a common parent group G. Then `Normalizer` returns the normalizer of U in S.

The **normalizer** $N_S(U)$ of U in S is the biggest subgroup of S which leaves U invariant under conjugation.

If S is the parent group of U then `Normalizer` sets and tests U`.normalizer`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> c2 := Subgroup( s4, [ (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2) ] )
gap> Normalizer( s4, c2 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (3,4), (1,2) ] )
```

The default function `GroupOps.Normalizer` uses `Stabilizer` (see 8.24) in order to compute the stabilizer of U in S acting by conjugation (see 7.20).

7.28 PCore

`PCore(` G`,` p `)`

`PCore` returns the p-core of the finite group G for a prime p.

The p-core is the largest normal subgroup whose size is a power of p. This is the core of the Sylow-p-subgroups (see 7.21 and 7.31).

Note that `PCore` sets and tests G`.pCores[` p `]`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> PCore( s4, 2 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,4)(2,3), (1,3)(2,4) ] )
gap> PCore( s4, 3 );
Subgroup( Group( (1,2,3,4), (1,2) ), [  ] )
```

The default function `GroupOps.PCore` computes the p-core as the core of a Sylow-p-subgroup (see 7.21 and 7.31).

7.29 PrefrattiniSubgroup

`PrefrattiniSubgroup(` G `)`

`PrefrattiniSubgroup` returns a Prefrattini subgroup of the group G.

A factor M/N of G is called a Frattini factor if $M/N \leq \phi(G/N)$ holds. The group P is a Prefrattini subgroup of G if P covers each Frattini chief factor of G, and if for each maximal subgroup of G there exists a conjugate maximal subgroup, which contains P.

```
gap> s4 := SymmetricGroup( AgWords, 4 );;
gap> ss4 := SpecialAgGroup( s4 );;
gap> PrefrattiniSubgroup( ss4 );
Subgroup( Group( g1, g2, g3, g4 ), [  ] )
```

Currently `PrefrattiniSubgroup` can only be applied to special Ag groups (see 26).

7.30 Radical

`Radical( G )`

`Radical` returns the radical of the finite group G.

The radical is the largest normal solvable subgroup of G.

```
gap> g := Group( (1,5), (1,5,6,7,8)(2,3,4) );
Group( (1,5), (1,5,6,7,8)(2,3,4) )
gap> Radical( g );
Subgroup( Group( (1,5), (1,5,6,7,8)(2,3,4) ), [ ( 2, 3, 4) ] )
```

The default function `GroupOps.Radical` tests if G is solvable and signals an error if not.

7.31 SylowSubgroup

`SylowSubgroup( G, p )`

`SylowSubgroup` returns a Sylow-p-subgroup of the finite group G for a prime p.

Let p be a prime and G be a finite group of order $p^n m$ where m is relative prime to p. Then by Sylow's theorem there exists at least one subgroup S of G of order p^n.

Note that `SylowSubgroup` sets and tests G`.sylowSubgroups[` p `]`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> SylowSubgroup( s4, 2 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (3,4), (1,2), (1,3)(2,4) ] )
gap> SylowSubgroup( s4, 3 );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (2,3,4) ] )
```

The default function `GroupOps.SylowSubgroup` computes the set of elements of p power order of G, starts with such an element of maximal order and computes the closure (see 7.18) with normalizing elements of p power order until a Sylow group is found.

7.32 TrivialSubgroup

`TrivialSubgroup( U )`

Let U be a group with parent group G. Then `TrivialSubgroup` returns the trivial subgroup T of U. Note that the parent group of T is G not U (see 7.14).

The default function `GroupOps.TrivialSubgroup` binds the set of elements of U, namely [U.*identity*], to T`.elements`,

7.33 FactorGroup

`FactorGroup( G, N )`

`FactorGroup` returns the factor group G/N where N must be a normal subgroup of G (see 7.58). This is the same as G `/` N (see 7.117).

`NaturalHomomorphism` returns the natural homomorphism from G (or a subgroup thereof) onto the factor group (see 7.110).

It is not specified how the factor group N is represented.

```
gap> a4 := Group( (1,2,3), (2,3,4) );;  a4.name := "a4";
"a4"
gap> v4 := Subgroup(a4,[(1,2)(3,4),(1,3)(2,4)]);;  v4.name := "v4";
"v4"
gap> f := FactorGroup( a4, v4 );
(a4 / v4)
gap> Size( f );
3
gap> Elements( f );
[ FactorGroupElement( v4, () ), FactorGroupElement( v4, (2,3,4) ),
  FactorGroupElement( v4, (2,4,3) ) ]
```

If G is the parent group of N, `FactorGroup` first checks for the knowledge component N`.factorGroup`. If this component is bound, `FactorGroup` returns its value. Otherwise, `FactorGroup` calls G`.operations.FactorGroup(` G`,` N `)`, remembers the returned value in N`.factorGroup`, and returns it. If G is not the parent group of N, `FactorGroup` calls G`.operations.FactorGroup(` G`,` N `)` and returns this value.

The default function called this way is `GroupOps.FactorGroup`. It returns the factor group as a group of factor group elements (see 7.34). Look under **FactorGroup** in the index to see for which groups this function is overlaid.

7.34 FactorGroupElement

`FactorGroupElement(` N`,` g `)`

`FactorGroupElement` returns the coset N `*` g as a group element. It is not tested whether g normalizes N, but g must be an element of the parent group of N.

Factor group elements returned by `FactorGroupElement` are represented by records. Those records contain the following components.

`isGroupElement`
: contains `true`.

`isFactorGroupElement`
: contains `true`.

`element`
: contains a right coset of N (see 7.86).

`domain`
: contains `FactorGroupElements` (see 4.5).

`operations`
: contains the operations record `FactorGroupElementOps`.

All operations for group elements (see 7.3) are available for factor group elements, e.g., two factor group elements can be multiplied (provided that they have the same subgroup N).

```
gap> a4 := Group( (1,2,3), (2,3,4) );;  a4.name := "a4";;
gap> v4 := Subgroup(a4,[(1,2)(3,4),(1,3)(2,4)]);;  v4.name := "v4";;
gap> x := FactorGroupElement( v4, (1,2,3) );
FactorGroupElement( v4, (2,4,3) )
```

```
gap> y := FactorGroupElement( v4, (2,3,4) );
FactorGroupElement( v4, (2,3,4) )
gap> x * y;
FactorGroupElement( v4, () )
```

7.35 CommutatorFactorGroup

`CommutatorFactorGroup(` G `)`

`CommutatorFactorGroup` returns a group isomorphic to G/G' where G' is the derived subgroup of G (see 7.22).

```
gap> s4 := AgGroup( Group( (1,2,3,4), (1,2) ) );
Group( g1, g2, g3, g4 )
gap> CommutatorFactorGroup( s4 );
Group( g1 )
```

The default group function `GroupOps.CommutatorFactorGroup` uses `DerivedSubgroup` (see 7.22) and `FactorGroup` (see 7.33) in order to compute the commutator factor group.

7.36 Series of Subgroups

The following sections describe functions that compute and return series of subgroups of a given group (see 7.37, 7.41, 7.43, and 7.44). The series are returned as lists of subgroups of the group (see 7.6).

These functions print warnings if the argument is an infinite group, because they may run forever.

7.37 DerivedSeries

`DerivedSeries(` G `)`

`DerivedSeries` returns the derived series of G.

The **derived series** is the series of iterated derived subgroups. The group G is solvable if and only if this series reaches $\{1\}$ after finitely many steps.

Note that this function does not terminate if G is an infinite group with derived series of infinite length.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> DerivedSeries( s4 );
[ Group( (1,2,3,4), (1,2) ), Subgroup( Group( (1,2,3,4), (1,2) ),
    [ (1,3,2), (1,4,3) ] ), Subgroup( Group( (1,2,3,4), (1,2) ),
    [ (1,4)(2,3), (1,3)(2,4) ] ),
  Subgroup( Group( (1,2,3,4), (1,2) ), [  ] ) ]
```

The default function `GroupOps.DerivedSeries` uses `DerivedSubgroup` (see 7.22) in order to compute the derived series of G.

7.38 CompositionSeries

`CompositionSeries(` G `)`

`CompositionSeries` returns a composition series of G as list of subgroups.

```
gap> s4 := SymmetricGroup( 4 );
Group( (1,4), (2,4), (3,4) )
gap> s4.name := "s4";;
gap> CompositionSeries( s4 );
[ Subgroup( s4, [ (1,2), (1,3,2), (1,3)(2,4), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,3,2), (1,3)(2,4), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,3)(2,4), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,2)(3,4) ] ), Subgroup( s4, [  ] ) ]
gap> d8 := SylowSubgroup( s4, 2 );
Subgroup( s4, [ (1,2), (3,4), (1,3)(2,4) ] )
gap> CompositionSeries( d8 );
[ Subgroup( s4, [ (1,3)(2,4), (1,2), (3,4) ] ),
  Subgroup( s4, [ (1,2), (3,4) ] ), Subgroup( s4, [ (3,4) ] ),
  Subgroup( s4, [  ] ) ]
```

Note that there is no default function. `GroupOps.CompositionSeries` signals an error if called.

7.39 ElementaryAbelianSeries

`ElementaryAbelianSeries(` G `)`

Let G be a solvable group (see 7.61). Then the functions returns a normal series $G = E_0, E_1, ..., E_n = \{1\}$ of G such that the factor groups E_i/E_{i+1} are elementary abelian groups.

```
gap> s5 := SymmetricGroup( 5 );; s5.name := "s5";;
gap> s4 := Subgroup( s5, [ (2,3,4,5), (2,3) ] );
Subgroup( s5, [ (2,3,4,5), (2,3) ] )
gap> ElementaryAbelianSeries( s4 );
[ Subgroup( s5, [ (2,3), (2,4,3), (2,5)(3,4), (2,3)(4,5) ] ),
  Subgroup( s5, [ (2,4,3), (2,5)(3,4), (2,3)(4,5) ] ),
  Subgroup( s5, [ (2,5)(3,4), (2,3)(4,5) ] ), Subgroup( s5, [  ] ) ]
```

The default function `GroupOps.ElementaryAbelianSeries` uses `AgGroup` (see 25.25) in order to convert G into an isomorphic ag group and computes the elementary abelian series in this group. (see 25.9).

7.40 JenningsSeries

`JenningsSeries(` G, p `)`

`JenningsSeries` returns the Jennings series of a p-group G.

The **Jennings series** of a p-group G is defined as follows. $S_1 = G$ and $S_n = [S_{n-1}, G]{S_i}^p$ where i is the smallest integer equal or greater than n/p. The length l of S is the smallest integer such that $S_l = \{1\}$.

Note that $S_n = S_{n+1}$ is possible.

```
gap> G := CyclicGroup( AgWords, 27 );
Group( c27_1, c27_2, c27_3 )
gap> G.name := "G";;
gap> JenningsSeries( G );
[ G, Subgroup( G, [ c27_2, c27_3 ] ), Subgroup( G, [ c27_2, c27_3 ] ),
  Subgroup( G, [ c27_3 ] ), Subgroup( G, [ c27_3 ] ),
  Subgroup( G, [ c27_3 ] ), Subgroup( G, [ c27_3 ] ),
  Subgroup( G, [ c27_3 ] ), Subgroup( G, [ c27_3 ] ),
  Subgroup( G, [  ] ) ]
```

7.41 LowerCentralSeries

`LowerCentralSeries(` G `)`

`LowerCentralSeries` returns the lower central series of G as a list of group records.

The **lower central series** is the series defined by $S_1 = G$ and $S_i = [G, S_{i-1}]$. The group G is nilpotent if this series reaches $\{1\}$ after finitely many steps.

Note that this function may not terminate if G is an infinite group. `LowerCentralSeries` sets and tests the record component G`.lowerCentralSeries` in the group record of G.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> LowerCentralSeries( s4 );
[ Group( (1,2,3,4), (1,2) ), Subgroup( Group( (1,2,3,4), (1,2) ),
    [ (1,3,2), (2,4,3) ] ) ]
```

The default group function `GroupOps.LowerCentralSeries` uses `CommutatorSubgroup` (see 7.19) in order to compute the lower central series of G.

7.42 PCentralSeries

`PCentralSeries(` G`,` p `)`

`PCentralSeries` returns the p-central series of a group G for a prime p.

The **p-central series** of a group G is defined as follows. $S_1 = G$ and S_{i+1} is set to $[G, S_i] * S_i^p$. The length of this series is n, where $n = max\{i; S_i > S_{i+1}\}$.

```
gap> s4 := Group( (1,2,3,4), (1,2) );; s4.name := "s4";;
gap> PCentralSeries( s4, 3 );
[ s4 ]
gap> PCentralSeries( s4, 2 );
[ s4, Subgroup( s4, [ (1,2,3), (1,3,4) ] ) ]
```

7.43 SubnormalSeries

`SubnormalSeries(` G`,` U `)`

Let U be a subgroup of G, then `SubnormalSeries` returns a subnormal series $G = G_1 > ... > G_n$ of groups such that U is contained in G_n and there exists no proper subgroup V between G_n and U which is normal in G_n.

G_n is equal to U if and only if U is **subnormal** in G.

Note that this function may not terminate if G is an infinite group.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> c2 := Subgroup( s4, [ (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2) ] )
gap> SubnormalSeries( s4, c2 );
[ Group( (1,2,3,4), (1,2) ) ]
gap> IsSubnormal( s4, c2 );
false
gap> c2 := Subgroup( s4, [ (1,2)(3,4) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2)(3,4) ] )
gap> SubnormalSeries( s4, c2 );
[ Group( (1,2,3,4), (1,2) ), Subgroup( Group( (1,2,3,4), (1,2) ),
    [ (1,2)(3,4), (1,3)(2,4) ] ),
  Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2)(3,4) ] ) ]
gap> IsSubnormal( s4, c2 );
true
```

The default function `GroupOps.SubnormalSeries` constructs the subnormal series as follows. $G_1 = G$ and G_{i+1} is set to the normal closure (see 7.25) of U under G_i. The length of the series is n, where $n = max\{i; G_i > G_{i+1}\}$.

7.44 UpperCentralSeries

`UpperCentralSeries(` *G* `)`

`UpperCentralSeries` returns the upper central series of G as a list of subgroups.

The **upper central series** is the series $S_n, ..., S_0$ defined by $S_0 = \{1\} < G$ and $S_i/S_{i-1} = Z(G/S_{i-1})$ where $n = min\{i; S_i = S_{i+1}\}$

Note that this function may not terminate if G is an infinite group. `UpperCentralSeries` sets and tests *G*`.upperCentralSeries` in the group record of G.

```
gap> d8 := AgGroup( Group( (1,2,3,4), (1,2)(3,4) ) );
Group( g1, g2, g3 )
gap> UpperCentralSeries( d8 );
[ Group( g1, g2, g3 ), Subgroup( Group( g1, g2, g3 ), [ g3 ] ),
  Subgroup( Group( g1, g2, g3 ), [  ] ) ]
```

7.45 Properties and Property Tests

The following sections describe the functions that computes or test properties of groups (see 7.46, 7.47, 7.48, 7.49, 7.50, 7.51, 7.52, 7.53, 7.54, 7.55, 7.56, 7.57, 7.58, 7.59, 7.60, 7.61, 7.62, 7.63, 7.64, 7.65, 7.66).

All tests expect a parent group or subgroup and return `true` if the group has the property and `false` otherwise. Some functions may not terminate if the given group has an infinite set of elements. A warning may be printed in such cases.

In addition the set theoretic functions `Elements`, `Size` and `IsFinite`, which are described in chapter 4, can be used for groups. `Size` (see 4.10) returns the order of a group, this is either a positive integer or the string "infinity". `IsFinite` (see 4.9) returns `true` if a group is finite and `false` otherwise.

7.46 AbelianInvariants

`AbelianInvariants(` G `)`

Let G be an abelian group. Then `AbelianInvariants` returns the abelian invariants of G as a list of integers. If G is not abelian then the abelian invariants of the commutator factor group of G are returned.

Let G be a finitely generated abelian group. Then there exist n nontrivial subgroups A_i of prime power order $p_i^{e_i}$ and m infinite cyclic subgroups Z_j such that $G = A_1 \times ... \times A_n \times Z_1 ... \times Z_m$. The **invariants** of G are the integers $p_1^{e_1}, ..., p_n^{e_n}$ together with m zeros.

Note that `AbelianInvariants` tests and sets G`.abelianInvariants`.

```
gap> AbelianInvariants( AbelianGroup( AgWords, [2,3,4,5,6,9] ) );
[ 2, 2, 3, 3, 4, 5, 9 ]
```

The default function `GroupOps.AbelianInvariants` requires that G is finite.

Let G be a finite abelian group of order $p_1^{e_1} ... p_n^{e_n}$ where p_i are distinct primes. The default function constructs for every prime p_i the series $G, G^{p_i}, G^{p_i^2}, ...$ and computes the abelian invariants using the indices of these groups.

7.47 DimensionsLoewyFactors

`DimensionsLoewyFactors(` G `)`

Let G be p-group. Then `DimensionsLoewyFactors` returns the dimensions c_i of the Loewy factors of F_pG.

The **Loewy series** of F_pG is defined as follows. Let R be the Jacobson radical of the group ring F_pG. The series $R^0 = F_pG > R^1 > ... > R^{l+1} = \{1\}$ is the Loewy series. The dimensions c_i are the dimensions of R^i/R^{i+1}.

```
gap> f6 := FreeGroup( 6, "f6" );;
gap> g := f6 / [ f6.1^3, f6.2^3, f6.3^3, f6.4^3, f6.5^3, f6.6^3,
>          Comm(f6.3,f6.2)/f6.6^2,  Comm(f6.3,f6.1)/(f6.6*f6.5),
>          Comm(f6.2,f6.1)/(f6.5*f6.4^2) ];;
gap> a := AgGroupFpGroup(g);
Group( f6.1, f6.2, f6.3, f6.4, f6.5, f6.6 )
gap> DimensionsLoewyFactors(a);
[ 1, 3, 9, 16, 30, 42, 62, 72, 87, 85, 87, 72, 62, 42, 30, 16, 9, 3,
  1 ]
```

The default function `GroupOps.DimensionsLoewyFactors` computes the Jennings series of G and uses Jennings thereom in order to calculate the dimensions of the Loewy factors.

Let $G = X_1 \geq X_2 \geq ... \geq X_l > X_{l+1} = \{1\}$ be the Jennings series of G (see 7.40) and let d_i be the dimensions of X_i/X_{i+1}. Then the Jennings polynomial is

$$\sum_{i=0}^{l} c_i x^i = \prod_{k=1}^{l} (1 + x^k + x^{2k} + ... + x^{(p-1)k})^{d_k}.$$

7.48 EulerianFunction

`EulerianFunction(` G`,` n `)`

`EulerianFunction` returns the number of n-tuples $(g_1, g_2, \ldots g_n)$ of elements of the group G that generate the whole group G. The elements of a tuple need not be different.

```
gap> s4 := SymmetricGroup( AgWords, 4 );;
gap> ss4 := SpecialAgGroup( s4 );;
gap> EulerianFunction( ss4, 1 );
0
gap> EulerianFunction( ss4, 2 );
216
gap> EulerianFunction( ss4, 3 );
10080
```

Currently `EulerianFunction` can only be applied to special Ag groups (see 26).

7.49 Exponent

`Exponent(` G `)`

Let G be a finite group. Then `Exponent` returns the exponent of G.

Note that `Exponent` tests and sets G`.exponent`.

```
gap> Exponent( Group( (1,2,3,4), (1,2) ) );
12
```

The default function `GroupOps.Exponent` computes all elements of G and their orders.

7.50 Factorization

`Factorization(` G`,` g `)`

Let G be a group with generators $g_1, ..., g_n$ and let g be an element of G. `Factorization` returns a representation of g as word in the generators of G.

The group record of G must have a component G`.abstractGenerators` which contains a list of n abstract words $h_1, ..., h_n$. Otherwise a list of n abstract generators is bound to G`.abstractGenerators`. The function returns an abstract word $h = h_{i_1}^{e_1} * ... * h_{i_m}^{e_m}$ such that $g_{i_1}^{e_1} * ... * g_{i_m}^{e_m} = g$.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> Factorization( s4, (1,2,3) );
x1^3*x2*x1*x2
gap> (1,2,3,4)^3 * (1,2) * (1,2,3,4) * (1,2);
(1,2,3)
```

The default group function `GroupOps.Factorization` needs a finite group G. It computes the set of elements of G using a Dimino algorithm, together with a representation of these elements as words in the generators of G.

7.51 Index

`Index(` G`,` U `)`

Let U be a subgroup of G. Then `Index` returns the index of U in G as an integer.

Note that `Index` sets and checks U`.index` if G is the parent group of U.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> Index( s4, DerivedSubgroup( s4 ) );
2
```

The default function `GroupOps.Index` needs a finite group G. It returns the quotient of `Size(` G `)` and `Size(` U `)`.

7.52 IsAbelian

`IsAbelian(` G `)`

`IsAbelian` returns `true` if the group G is abelian and `false` otherwise.

A group G is **abelian** if and only if for every $g, h \in G$ the equation $g * h = h * g$ holds.

Note that `IsAbelian` sets and tests the record component G`.isAbelian`. If G is abelian it also sets G`.centre`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> IsAbelian( s4 );
false
gap> IsAbelian( Subgroup( s4, [ (1,2) ] ) );
true
```

The default group function `GroupOps.IsAbelian` returns `true` for a group G generated by $g_1, ..., g_n$ if g_i commutes with g_j for $i > j$.

7.53 IsCentral

`IsCentral(` G`,` U `)`

`IsCentral` returns `true` if the group G centralizes the group U and `false` otherwise.

A group G **centralizes** a group U if and only if for all $g \in G$ and for all $u \in U$ the equation $g * u = u * g$ holds. Note that U need not to be a subgroup of G but they must have a common parent group.

Note that `IsCentral` sets and tests U`.isCentral` if G is the parent group of U.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> d8 := Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] );;
gap> c2 := Subgroup( s4, [ (1,3)(2,4) ] );;
gap> IsCentral( s4, c2 );
false
gap> IsCentral( d8, c2 );
true
```

The default function `GroupOps.IsCentral` tests whether G centralizes U by testing whether the generators of G commutes with the generators of U.

7.54 IsConjugate

`IsConjugate(` G`,` x`,` y `)`

Let x and y be elements of the parent group of G. Then `IsConjugate` returns `true` if x is conjugate to y under an element g of G and `false` otherwise.

```
gap> s5 := Group( (1,2,3,4,5), (1,2) );
Group( (1,2,3,4,5), (1,2) )
gap> a5 := Subgroup( s5, [ (1,2,3), (2,3,4), (3,4,5) ] );
Subgroup( Group( (1,2,3,4,5), (1,2) ), [ (1,2,3), (2,3,4), (3,4,5) ] )
gap> IsConjugate( a5, (1,2,3,4,5), (1,2,3,4,5)^2 );
false
gap> IsConjugate( s5, (1,2,3,4,5), (1,2,3,4,5)^2 );
true
```

The default function `GroupOps.IsConjugate` uses `Representative` (see 4.15) in order to check whether x is conjugate to y under G.

7.55 IsCyclic

`IsCyclic(` G `)`

`IsCyclic` returns `true` if G is cyclic and `false` otherwise.

A group G is **cyclic** if and only if there exists an element $g \in G$ such that G is generated by g.

Note that `IsCyclic` sets and tests the record component G`.isCyclic`.

```
gap> z6 := Group( (1,2,3), (4,5) );;
gap> IsCyclic( z6 );
true
gap> z36 := AbelianGroup( AgWords, [ 9, 4 ] );;
gap> IsCyclic( z36 );
true
```

The default function `GroupOps.IsCyclic` returns `false` if G is not an abelian group. Otherwise it computes the abelian invariants (see 7.46) if G is infinite. If G is finite of order $p_1^{e_1}...p_n^{e_n}$, where p_i are distinct primes, then G is cyclic if and only if each G^{p_i} has index p_i in G.

7.56 IsElementaryAbelian

`IsElementaryAbelian(` G `)`

`IsElementaryAbelian` returns `true` if the group G is an elementary abelian p-group for a prime p and `false` otherwise.

A p-group G is **elementary abelian** if and only if for every $g, h \in G$ the equations $g * h = h * g$ and $g^p = 1$ hold.

Note that the `IsElementaryAbelian` sets and tests G`.isElementaryAbelian`.

```
gap> z4 := Group( (1,2,3,4) );;
```

```
gap> IsElementaryAbelian( z4 );
false
gap> v4 := Group( (1,2)(3,4), (1,3)(2,4) );;
gap> IsElementaryAbelian( v4 );
true
```

The default function `GroupOps.IsElementaryAbelian` returns `true` if G is abelian and for some prime p each generator is of order p.

7.57 IsNilpotent

`IsNilpotent(` G `)`

`IsNilpotent` returns `true` if the group G is nilpotent and `false` otherwise.

A group G is **nilpotent** if and only if the lower central series of G is of finite length and reaches $\{1\}$.

Note that `IsNilpotent` sets and tests the record component G`.isNilpotent`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> IsNilpotent( s4 );
false
gap> v4 := Group( (1,2)(3,4), (1,3)(2,4) );;
gap> IsNilpotent( v4 );
true
```

The default group function `GroupOps.IsNilpotent` computes the lower central series using `LowerCentralSeries` (see 7.41) in order to check whether G is nilpotent.

If G has an infinite set of elements a warning is given, as this function does not stop if G has a lower central series of infinite length.

7.58 IsNormal

`IsNormal(` G, U `)`

`IsNormal` returns `true` if the group G normalizes the group U and `false` otherwise.

A group G **normalizes** a group U if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of U. Note that U need not be a subgroup of G but they must have a common parent group.

Note that `IsNormal` tests and sets U`.isNormal` if G is the parent group of U.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> d8 := Subgroup( s4, [ (1,2,3,4), (1,2)(3,4) ] );;
gap> c2 := Subgroup( s4, [ (1,3)(2,4) ] );;
gap> IsNormal( s4, c2 );
false
gap> IsNormal( d8, c2 );
true
```

Let G be a finite group. Then the default function `GroupOps.IsNormal` checks whether the conjugate of each generator of U under each generator of G is an element of U.

If G is an infinite group, then the default function `GroupOps.IsNormal` checks whether the conjugate of each generator of U under each generator of G and its inverse is an element of U.

7.59 IsPerfect

`IsPerfect(` G `)`

`IsPerfect` returns `true` if G is a perfect group and `false` otherwise.

A group G is **perfect** if G is equal to its derived subgroup. See 7.22.

Note that `IsPerfect` sets and tests G`.isPerfect`.

```
gap> a4 := Group( (1,2,3), (2,3,4) );
Group( (1,2,3), (2,3,4) )
gap> IsPerfect( a4 );
false
gap> a5 := Group( (1,2,3), (2,3,4), (3,4,5) );
Group( (1,2,3), (2,3,4), (3,4,5) )
gap> IsPerfect( a5 );
true
```

The default group function `GroupOps.IsPerfect` checks for a finite group G the index of G' (see 7.22) in G. For an infinite group it computes the abelian invariants of the commutator factor group (see 7.46 and 7.35).

7.60 IsSimple

`IsSimple(` G `)`

`IsSimple` returns `true` if G is simple and `false` otherwise.

A group G is **simple** if and only if G and the trivial subgroup are the only normal subgroups of G.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> IsSimple( DerivedSubgroup( s4 ) );
false
gap> s5 := Group( (1,2,3,4,5), (1,2) );
Group( (1,2,3,4,5), (1,2) )
gap> IsSimple( DerivedSubgroup( s5 ) );
true
```

7.61 IsSolvable

`IsSolvable(` G `)`

`IsSolvable` returns `true` if the group G is solvable and `false` otherwise.

A group G is **solvable** if and only if the derived series of G is of finite length and reaches $\{1\}$.

Note that `IsSolvable` sets and tests G`.isSolvable`.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> IsSolvable( s4 );
true
```

The default function `GroupOps.IsSolvable` computes the derived series using the function `DerivedSeries` (see 7.37) in order to see whether G is solvable.

If G has an infinite set of elements a warning is given, as this function does not stop if G has a derived series of infinite length.

7.62 IsSubgroup

`IsSubgroup(` G, U `)`

`IsSubgroup` returns `true` if U is a subgroup of G and `false` otherwise.

Note that G and U must have a common parent group. This function returns `true` if and only if the set of elements of U is a subset of the set of elements of G, it is not the inverse of `IsParent` (see 7.7).

```
gap> s6  := Group( (1,2,3,4,5,6), (1,2) );;
gap> s4 := Subgroup( s6, [ (1,2,3,4), (1,2) ] );;
gap> z2 := Subgroup( s6, [ (5,6) ] );;
gap> IsSubgroup( s4, z2 );
false
gap> v4 := Subgroup( s6, [ (1,2)(3,4), (1,3)(2,4) ] );;
gap> IsSubgroup( s4, v4 );
true
```

If the elements of G are known, then the default function `GroupOps.IsSubgroup` checks whether the set of generators of U is a subset of the set of elements of G. Otherwise the function checks whether each generator of U is an element of G using `in`.

7.63 IsSubnormal

`IsSubnormal(` G, U `)`

`IsSubnormal` returns `true` if the subgroup U of G is subnormal in G and `false` otherwise.

A subgroup U of G is subnormal if and only if there exists a series of subgroups $G = G_0 > G_1 > ... > G_n = U$ such that G_i is normal in G_{i-1} for all $i \in \{1, ..., n\}$.

Note that U must be a subgroup of G. The function sets and checks U`.isSubnormal` if G is the parent group of G.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> c2 := Subgroup( s4, [ (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2) ] )
gap> IsSubnormal( s4, c2 );
false
gap> c2 := Subgroup( s4, [ (1,2)(3,4) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2)(3,4) ] )
gap> IsSubnormal( s4, c2 );
true
```

The default function `GroupOps.IsSubnormal` uses `SubnormalSeries` (see 7.43) in order to check if U is subnormal in G.

7.64 IsTrivial for Groups

`GroupOps.IsTrivial(` G `)`

`GroupOps.IsTrivial` returns `true` if G is the trivial group and `false` otherwise.

Note that G is trivial if and only if the component `generators` of the group record of G is the empty list. It is faster to check this than to call `IsTrivial`.

7.65 GroupId

`GroupId(` G `)`

For certain small groups the function returns a record which will identify the isomorphism type of G with respect to certain classifications. This record contains the components described below.

The function will work for all groups of order at most 100 or whose order is a product of at most three primes. Moreover if the ANU pq is installed and loaded (see 57.1 and 57.2) you can also use `GroupId` to identify groups of order 128, 256, 243 and 729. In this case a standard presentation for G is computed (see 58.6) and the returned record will only contain the components `size`, `pGroupId`, and possibly `abelianInvariants`. For 2- or 3-groups of order at most 100 `GroupId` will return the `pGroupId` identifier even if the ANU pq is not installed.

`catalogue`
: a pair $[o, n]$ where o is the size of G and n is the catalogue number of G following the catalogue of groups of order at most 100. See 38.7 for further details. This catalogue uses the Neubueser list for groups of order at most 100, excluding groups of orders 64 and 96 (see [Neu67]). It uses the lists developed by [HS64] and [Lau82] for orders 64 and 96 respectively.

  Note that there are minor discrepancies between n and the number in [Neu67] for abelian groups and groups of type `D(p,q)xr`. However, a solvable group G is isomorphic to `SolvableGroup(o, n)`, i.e., `GroupId(SolvableGroup(o,n)).catalogue` will be `[o,n]`.

  If G is a 2- or 3-group of order at most 100, its number in the appropriate p-group library is also returned. Note that, for such groups, the number n usually differs from the p-group identifier returned in `pGroupId` (see below).

`3primes`
: if G is non-abelian and its size is a product of at most three primes then `3primes` holds an identifier for G. The following isomorphisms are returned in `3primes`:
  `["A",p]` = `A(p^3)`, `["B",p]` = `B(p^3)`, `["D",p,q,r]` = `D(p,q)xr`,
  `["D",p,q]` = `D(p,q)`, `["G",p,q]` = `G(p^2,q)`, `["G",p,q,r,s]` = `G(p,q,r,s)`,
  `["H",p,q]` = `H(p^2,q)`, `["H",p,q,r]` = `H(p,q,r)`, `["K",p,q]` = `K(p,q^2)`,
  `["L",p,q,s]` = `L(p,q^2,s)`, `["M",p,q]` = `M(p,q^2)`, `["N",p,q]` = `N(p,q^2)`
  (see `names` below for a definition of `A` ... `N`).

`pGroupId`
: if G is a 2- or 3-group, this will be the number of G in the list of 2-groups of order at most 256, prepared by Newman and O'Brien, or 3-groups of order at most 729,

prepared by O'Brien and Rhodes. In particular, for an integer n and for o a power of 2 at most 256, `GroupId(TwoGroup(o,n)).pGroupId` is always n (and similarly for 3-groups). See 38.8 and 38.9 for details about the libraries of 2- and 3-groups. Note that if G is a 2- or 3-group of order at most 100 its `pGroupId` usually **differs** from its GAP solvable library number returned in `catalogue`.

`abelianInvariants`
: if G is abelian, this is a list of abelian invariants.

`names`
: a list of names of G. For non-abelian groups of order 96 this name is that used in the Laue catalogue (see [Lau82]). For the other groups the following symbols are used. Note that this list of names is neither complete, i.e., most of the groups of order 64 do not have a name even if they are of one of the types described below, nor does it uniquely determine the group up to isomorphism in some cases.

`m` is the cyclic group of order m,
`Dm` is the dihedral group of order m,
`Qm` is the quaternion group of order m,
`QDm` is the quasi-dihedral group of order m,
`Sm` is the symmetric group on m points,
`Am` is the alternating group on m points,
`SL(d,q)` is the special linear group,
`GL(d,q)` is the general linear group,
`PSL(d,q)` is the projective special linear group,
`K^n` is the direct power of m copies of K,
`K$H` is a wreath product of K and H,
`K:H` is a split extension of K by H,
`K.H` is a non-split extension of K and H,
`K+H` is a subdirect product with identified factor groups of K and H,
`KYH` is a central amalgamated product of the groups K and H,
`KxH` is the direct product of K and H,

`A(p^3)` is $\langle A,B,C; A^p=B^p=C^p=[A,B]=[A,C]=1, [B,C]=A\rangle$,

`B(p^3)` is $\langle A,B,C; B^p=C^p=A, A^p=[A,B]=[A,C]=1, [B,C]=A\rangle$,

`D(p,q)` is $\langle A,B; A^q=B^p=1, A^B=A^x\rangle$ such that $p|q-1$, $x\neq 1 \bmod q$, and $x^p=1 \bmod q$,

`G(p^2,q)` is $\langle A,B,C; A^p=B^q=1, C^p=A, [A,B]=[A,C]=1, B^C=B^x\rangle$ such that $p|q-1$, $x\neq 1 \bmod q$, and $x^p=1 \bmod q$,

`G(p,q,r,s)` is $\langle A,B,C; A^r=B^q=C^p=[A,B]=1, A^C=A^x, B^C=B^{(y^s)}\rangle$ such that $p|q-1$, $p|r-1$, x minimal with $x\neq 1 \bmod r$ and $x^p=1 \bmod r$, y minimal with $y\neq 1 \bmod q$ and $y^p=1 \bmod q$, and $0<s<p$,

`H(p^2,q)` is $\langle A,B; A^q=B^{(p^2)}=1, A^B=A^x\rangle$ such that $p^2|q-1$, $x^p\neq 1 \bmod q$, and $x^{(p^2)}=1 \bmod q$,

`H(p,q,r)` is $\langle A,B; A^r=B^{pq}=1, A^B=A^x\rangle$ such that $pq|r-1$, $x^p\neq 1 \bmod r$, $x^q\neq 1 \bmod r$, and $x^{pq}=1 \bmod r$,

`K(p,q^2)` is $\langle A,B,C; A^q=B^q=C^p=[A,B]=1, A^C=A^x, B^C=B^x\rangle$ such that $p|q-1$, $x\neq 1 \bmod q$, and $x^p=1 \bmod q$,

`L(p,q^2,s)` is $\langle A,B,C; A^q=B^q=C^p=[A,B]=1, A^C=A^x, B^C=B^{(x^s)}\rangle$ such that $p|q-1$, $x \neq 1 \bmod q$, $x^p = 1 \bmod q$, and $1 < s < p$, note that `L(q,p^2,s)` $\cong$ `L(q,p^2,t)` iff $st = 1 \bmod p$,

`M(p,q^2)` is $\langle A, B; A^{(q^2)} = B^p = 1, A^B = A^x\rangle$ such that $p|q-1$, $x \neq 1 \bmod q^2$, and $x^p = 1 \bmod q^2$,

`N(p,q^2)` is $\langle A, B, C; A^q = B^q = C^p = [A, B] = 1, A^C = A^{-1}B, B^C = A^{-1}B^{x^q+x-1}\rangle$ such that $2 < p$, $p|q+1$, x is an element of order p mod q^2,

`^` has the strongest, `x` the weakest binding.

```
gap> q8 := SolvableGroup( 8, 5 );;
gap> s4 := SymmetricGroup(4);;
gap> d8 := SylowSubgroup( s4, 2 );;
gap> GroupId(q8);
rec(
  catalogue := [ 8, 5 ],
  names := [ "Q8" ],
  3primes := [ "B", 2 ],
  size := 8,
  pGroupId := 4 )
gap> GroupId(d8);
rec(
  catalogue := [ 8, 4 ],
  names := [ "D8" ],
  3primes := [ "A", 2 ],
  size := 8,
  pGroupId := 3 )
gap> GroupId(s4);
rec(
  catalogue := [ 24, 15 ],
  names := [ "S4" ],
  size := 24 )
gap> GroupId(DirectProduct(d8,d8));
rec(
  catalogue := [ 64, 154 ],
  names := [ "D8xD8" ],
  size := 64,
  pGroupId := 226 )
gap> GroupId(DirectProduct(q8,d8));
rec(
  catalogue := [ 64, 155 ],
  names := [ "D8xQ8" ],
  size := 64,
  pGroupId := 230 )
gap> GroupId( WreathProduct( CyclicGroup(2), CyclicGroup(4) ) );
rec(
  catalogue := [ 64, 250 ],
  names := [  ],
  size := 64,
  pGroupId := 32 )
```

```
gap> f := FreeGroup("c","b","a");; a:=f.3;;b:=f.2;;c:=f.1;;
gap> r := [ c^5, b^31, a^31, Comm(b,c)/b^7, Comm(a,c)/a, Comm(a,b) ];;
gap> g := AgGroupFpGroup( f / r );
Group( c, b, a )
gap> GroupId(g);
rec(
  3primes := [ "L", 5, 31, 2 ],
  names := [ "L(5,31^2,2)" ],
  size := 4805 )
gap> RequirePackage("anupq");
gap> g := TwoGroup(256,4);
Group( a1, a2, a3, a4, a5, a6, a7, a8 )
gap> GroupId(g);
rec(
  size := 256,
  pGroupId := 4 )
gap> g := TwoGroup(256,232);
Group( a1, a2, a3, a4, a5, a6, a7, a8 )
gap> GroupId(g);
rec(
  size := 256,
  pGroupId := 232 )
```

7.66 PermutationCharacter

`PermutationCharacter(` G, U `)`

computes the permutation character of the operation of G on the cosets of U. The permutation character is returned as list of integers such that the i.th position contains the value of the permutation character on the i.th conjugacy class of G (see 7.68).

The value of the **permutation character** of U in G on a class c of G is the number of right cosets invariant under the action of an element of c.

```
gap> G := SymmetricPermGroup(5);;
gap> PermutationCharacter( G, SylowSubgroup(G,2) );
[ 15, 3, 3, 0, 0, 1, 0 ]
```

For small groups the default function `GroupOps.PermutationCharacter` calculates the permutation character by inducing the trivial character of U. For large groups it counts the fixed points by examining double cosets of U and the subgroup generated by a class element.

7.67 Conjugacy Classes

The following sections describe how one can compute conjugacy classes of elements and subgroups in a group (see 7.68 and 7.74). Further sections describe how conjugacy classes of elements are created (see 7.69 and 7.71), and how they are implemented (see 7.72 and 7.73). Further sections describe how classes of subgroups are created (see 7.76 and 7.77), and how they are implemented (see 7.78 and 7.79). Another section describes the function that returns a conjugacy class of subgroups as a list of subgroups (see 7.83).

7.68 ConjugacyClasses

`ConjugacyClasses( G )`

`ConjugacyClasses` returns a list of the conjugacy classes of elements of the group G. The elements in the list returned are conjugacy class domains as created by `ConjugacyClass` (see 7.69). Because conjugacy classes are domains, all set theoretic functions can be applied to them (see 4).

```
gap> a5 := Group( (1,2,3), (3,4,5) );;  a5.name := "a5";;
gap> ConjugacyClasses( a5 );
[ ConjugacyClass( a5, () ), ConjugacyClass( a5, (3,4,5) ),
  ConjugacyClass( a5, (2,3)(4,5) ), ConjugacyClass( a5, (1,2,3,4,5) ),
  ConjugacyClass( a5, (1,2,3,5,4) ) ]
```

`ConjugacyClasses` first checks if G`.conjugacyClasses` is bound. If the component is bound, it returns that value. Otherwise it calls G`.operations.ConjugacyClasses(` G `)`, remembers the returned value in G`.conjugacyClasses`, and returns it.

The default function called this way is `GroupOps.ConjugacyClasses`. This function takes random elements in G and tests whether such a random element g lies in one of the already known classes. If it does not it adds the new class `ConjugacyClass(` G`,` g `)` (see 7.69). Also after adding a new class it tests whether any power of the representative gives rise to a new class. It returns the list of classes when the sum of the sizes is equal to the size of G.

7.69 ConjugacyClass

`ConjugacyClass( G, g )`

`ConjugacyClass` returns the conjugacy class of the element g in the group G. Signals an error if g is not an element in G. The conjugacy class is returned as a domain, so that all set theoretic functions are applicable (see 4).

```
gap> a5 := Group( (1,2,3), (3,4,5) );;  a5.name := "a5";;
gap> c := ConjugacyClass( a5, (1,2,3,4,5) );
ConjugacyClass( a5, (1,2,3,4,5) )
gap> Size( c );
12
gap> Representative( c );
(1,2,3,4,5)
gap> Elements( c );
[ (1,2,3,4,5), (1,2,4,5,3), (1,2,5,3,4), (1,3,5,4,2), (1,3,2,5,4),
  (1,3,4,2,5), (1,4,3,5,2), (1,4,5,2,3), (1,4,2,3,5), (1,5,4,3,2),
  (1,5,2,4,3), (1,5,3,2,4) ]
```

`ConjugacyClass` calls G`.operations.ConjugacyClass(` G`,` g `)` and returns that value.

The default function called this way is `GroupOps.ConjugacyClass`, which creates a conjugacy class record (see 7.73) with the operations record `ConjugacyClassOps` (see 7.72). Look in the index under **ConjugacyClass** to see for which groups this function is overlaid.

7.70 PositionClass

`PositionClass( G, g )`

G must be a domain for which `ConjugacyClasses` is defined and *g* must be an element of *G*. This functions returns a positive integer `i` such that `g in ConjugacyClasses( G )[i]`.

```
gap> G := Group( (1,2)(3,4), (1,2,3,4,5) );;
gap> ConjugacyClasses( G );
[ ConjugacyClass( Group( (1,2)(3,4), (1,2,3,4,5) ), () ),
  ConjugacyClass( Group( (1,2)(3,4), (1,2,3,4,5) ), (3,4,5) ),
  ConjugacyClass( Group( (1,2)(3,4), (1,2,3,4,5) ), (2,3)(4,5) ),
  ConjugacyClass( Group( (1,2)(3,4), (1,2,3,4,5) ), (1,2,3,4,5) ),
  ConjugacyClass( Group( (1,2)(3,4), (1,2,3,4,5) ), (1,2,3,5,4) ) ]
gap>  g := Random( G );
(1,2,5,4,3)
gap> PositionClass( G, g );
5
```

7.71 IsConjugacyClass

`IsConjugacyClass( obj )`

`IsConjugacyClass` returns `true` if *obj* is a conjugacy class as created by `ConjugacyClass` (see 7.69) and `false` otherwise.

```
gap> a5 := Group( (1,2,3), (3,4,5) );;  a5.name := "a5";;
gap> c := ConjugacyClass( a5, (1,2,3,4,5) );
ConjugacyClass( a5, (1,2,3,4,5) )
gap> IsConjugacyClass( c );
true
gap> IsConjugacyClass(
>       [ (1,2,3,4,5), (1,2,4,5,3), (1,2,5,3,4), (1,3,5,4,2),
>         (1,3,2,5,4), (1,3,4,2,5), (1,4,3,5,2), (1,4,5,2,3),
>         (1,4,2,3,5), (1,5,4,3,2), (1,5,2,4,3), (1,5,3,2,4) ] );
false     # even though this is as a set equal to c
```

7.72 Set Functions for Conjugacy Classes

As mentioned above, conjugacy classes are domains, so all domain functions are applicable to conjugacy classes (see 4). This section describes the functions that are implemented especially for conjugacy classes. Functions not mentioned here inherit the default functions mentioned in the respective sections.

In the following let *C* be the conjugacy class of the element *g* in the group *G*.

`Elements( C )`

The elements of the conjugacy class *C* are computed as the orbit of *g* under *G*, where *G* operates by conjugation.

`Size(` C `)`

The size of the conjugacy class C is computed as the index of the centralizer of g in G.

h `in` C

To test whether an element h lies in C, `in` tests whether there is an element of G that takes h to g. This is done by calling `RepresentativeOperation(`G`,`h`,`g`)` (see 8.25).

`Random(` C `)`

A random element of the conjugacy class C is computed by conjugating g with a random element of G.

7.73 Conjugacy Class Records

A conjugacy class C of an element g in a group G is represented by a record with the following components.

`isDomain`
: always `true`.

`isConjugacyClass`
: always `true`.

`group`
: holds the group G.

`representative`
: holds the representative g.

The following component is optional. It is computed and assigned when the size of a conjugacy class is computed.

`centralizer`
: holds the centralizer of g in G.

7.74 ConjugacyClassesSubgroups

`ConjugacyClassesSubgroups(` G `)`

`ConjugacyClassesSubgroups` returns a list of all conjugacy classes of subgroups of the group G. The elements in the list returned are conjugacy class domains as created by `ConjugacyClassSubgroups` (see 7.76). Because conjugacy classes are domains, all set theoretic functions can be applied to them (see 4).

In fact, `ConjugacyClassesSubgroups` computes much more than it returns, for it calls (indirectly via the function G`.operations.ConjugacyClassesSubgroups(` G `))` the `Lattice` command (see 7.75), constructs the whole subgroup lattice of G, stores it in the record component G`.lattice`, and finally returns the list G`.lattice.classes`. This means, in particular, that it will fail if G is non-solvable and its maximal perfect subgroup is not in the built-in catalogue of perfect groups (see the description of the `Lattice` command 7.75 for details).

```
gap> # Conjugacy classes of subgroups of S4
```

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> s4.name := "s4";;
gap> cl := ConjugacyClassesSubgroups( s4 );
[ ConjugacyClassSubgroups( s4, Subgroup( s4, [  ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4, [ (1,2)(3,4) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4, [ (3,4) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4, [ (2,3,4) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4)
     ] ) ), ConjugacyClassSubgroups( s4, Subgroup( s4,
    [ (3,4), (1,2) ] ) ), ConjugacyClassSubgroups( s4, Subgroup( s4,
    [ (1,2)(3,4), (1,4,2,3) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4, [ (2,3,4), (3,4) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4,
    [ (3,4), (1,2), (1,3)(2,4) ] ) ),
  ConjugacyClassSubgroups( s4, Subgroup( s4,
    [ (1,2)(3,4), (1,3)(2,4), (2,3,4) ] ) ),
  ConjugacyClassSubgroups( s4, s4 ) ]
```

Each entry of the resulting list is a domain. As an example, let us take the seventh class in the above list of conjugacy classes of S_4.

```
gap> # Conjugacy classes of subgroups of S4 (continued)
gap> class7 := cl[7];;
gap> # Print the class representative subgroup.
gap> rep7 := Representative( class7 );
Subgroup( s4, [ (1,2)(3,4), (1,4,2,3) ] )
gap> # Print the order of the class representative subgroup.
gap> Size( rep7 );
4
gap> # Print the number of conjugates.
gap> Size( class7 );
3
```

7.75 Lattice

`Lattice(` *G* `)`

`Lattice` returns the lattice of subgroups of the group G in the form of a record L, say, which contains certain lists with some appropriate information on the subgroups of G and their conjugacy classes. In particular, in its component L`.classes`, L provides the same list of all conjugacy classes of all subgroups of G as is returned by the `ConjugacyClassesSubgroups` command (see 7.74).

The construction of the subgroup lattice record L of a group G may be very time consuming. Therefore, as soon as L has been computed for the first time, it will be saved as a component G`.lattice` in the group record G to avoid any duplication of that effort.

The underlying routines are a reimplementation of the subgroup lattice routines which have been developed since 1958 by several people in Kiel and Aachen under the supervision of Joachim Neubüser. Their final version, written by Volkmar Felsch in 1984, has been available since then in Cayley (see [BC92]) and has also been used in SOGOS (see [Leh89a]). The current implementation in GAP3 by Jürgen Mnich is described in [Mni92], a summary of the method and references to all predecessors can be found in [FS84].

The `Lattice` command invokes the following procedure. In a first step, the solvable residuum P, say, of G is computed and looked up in a built-in catalogue of perfect groups which is given in the file `LIBNAME/"lattperf.g"`. A list of subgroups is read off from that catalogue which contains just one representative of each conjugacy class of perfect subgroups of P and hence at least one representative of each conjugacy class of perfect subgroups of G. Then, starting from the identity subgroup and the conjugacy classes of perfect subgroups, the so called **cyclic extension method** is used to compute the non-perfect subgroups of G by forming for each class representative all its not yet involved cyclic extensions of prime number index and adding their conjugacy classes to the list.

It is clear that this procedure cannot work if the catalogue of perfect groups does not contain a group isomorphic to P. At present, it contains only all perfect groups of order less than 5000 and, in addition, the groups $PSL(3,3)$, M_{11}, and A_8. If the `Lattice` command is called for a group G with a solvable residuum P not in the catalogue, it will provide an error message. As an example we handle the group $SL(2,19)$ of order 6840.

```
gap> s := [ [4,0], [0,5] ] * Z( 19 )^0;;
gap> t := [ [4,4], [-9,-4] ] * Z(19)^0;;
gap> G := Group( s, t );;
gap> Size( G );
6840
gap> Lattice( G );
Error, sorry, can' t identify the group's solvable residuum
```

However, if you know the perfect subgroups of G, you can use the `Lattice` command to compute the whole subgroup lattice of G even if the solvable residuum of G is not in the catalogue. All you have to do in such a case is to create a list of subgroups of G which contains at least one representative of each conjugacy class of proper perfect subgroups of G, attach this list to the group record as a new component G`.perfectSubgroups`, and then call the `Lattice` command. The existence of that record component will prevent GAP3 from looking up the solvable residuum of G in the catalogue. Instead, it will insert the given subgroups into the lattice, leaving it to you to guarantee that in fact all conjugacy classes of proper perfect subgroups are involved.

If you miss classes, the resulting lattice will be incomplete, but you will not get any warning. As long as you are aware of this fact, you may use this possibility to compute a sublattice of the subgroup lattice of G without getting the above mentioned error message even if the solvable residuum of G is not in the catalogue. In particular, you will get at least the classes of all proper solvable subgroups of G if you define G`.perfectSubgroups` to be an empty list.

As an example for the computation of the complete lattice of subgroups of a group which is not covered by the catalogue, we handle the Mathieu group M_{12}.

```
gap> # Define the Mathieu group M12.
gap> a := (2,3,5,7,11,9,8,12,10,6,4);;
gap> b := (3,6)(5,8)(9,11)(10,12);;
gap> c := (1,2)(3,4)(5,9)(6,8)(7,12)(10,11);;
gap> M12 := Group( a, b, c );;
```

```
gap> Print( "#I  M12 has order ", Size( M12 ), "\n" );
#I  M12 has order 95040
gap> # Define a list of proper perfect subgroups of M_12 and attach
gap> # it to the group record M12 as component M12.perfectSubgroups.
gap> L2_11a := Subgroup( M12, [ a, b ] );;
gap> M11a   := Subgroup( M12, [ a, b, c*a^-1*b*a*c ] );;
gap> M11b   := Subgroup( M12, [ a, b, c*a*b*a^-1*c ] );;
gap> x      := a*b*a^2;;
gap> y      := a*c*a^-1*b*a*c*a^6;;
gap> A6a    := Subgroup( M12, [ x, y ] );;
gap> A5c    := Subgroup( M12, [ x*y, x^3*y^2*x^2*y ] );;
gap> x      := a^2*b*a;;
gap> y      := a^6*c*a*b*a^-1*c*a;;
gap> A6b    := Subgroup( M12, [ x, y ] );;
gap> A5d    := Subgroup( M12, [ x*y, x^3*y^2*x^2*y ] );;
gap> x      := a;;
gap> y      := b*c*b;;
gap> z      := c;;
gap> L2_11b := Subgroup( M12, [ x, y, z ] );;
gap> A5b    := Subgroup( M12, [ y, x*z ] );;
gap> x      := c;;
gap> y      := b*a^-1*c*a*b;;
gap> z      := a^2*b*a^-1*c*a*b*a^-2;;
gap> A5a    := Subgroup( M12, [ (x*z)^2, (y*z)^2 ] );;
gap> M12.perfectSubgroups := [
>   L2_11a, L2_11b, M11a, M11b, A6a, A6b, A5a, A5b, A5c, A5d ];;
gap> # Now compute the subgroup lattice of M12.
gap> lat := Lattice( M12 );
LatticeSubgroups( Group( ( 2, 3, 5, 7,11, 9, 8,12,10, 6, 4), ( 3, 6)
( 5, 8)( 9,11)(10,12), ( 1, 2)( 3, 4)( 5, 9)( 6, 8)( 7,12)(10,11) ) )
```

The `Lattice` command returns a record which represents a very complicated structure.

```
gap> # Subgroup lattice of M12 (continued)
gap> RecFields( lat );
[ "isLattice", "classes", "group", "printLevel", "operations" ]
```

Probably the most important component of the lattice record is the list *lat*`.classes`. Its elements are domains. They are described in section 7.74. We can use this list, for instance, to print the number of conjugacy classes of subgroups and the number of subgroups of M_{12}.

```
gap> # Subgroup lattice of M12 (continued)
gap> n1 := Length( lat.classes );;
gap> n2 := Sum( [ 1 .. n1 ], i -> Size( lat.classes[i] ) );;
gap> Print( "#I  M12 has ", n1, " classes of altogether ", n2,
>   " subgroups\n" );
#I  M12 has 147 classes of altogether 214871 subgroups
```

It would not make sense to get all components of a subgroup lattice record printed in full detail whenever we ask GAP3 to print the lattice. Therefore, as you can see in the above example, the default printout is just an expression of the form "`Lattice(` *group* `)`". However, you can ask GAP3 to display some additional information in any subsequent printout of the lattice by increasing its individual print level. This print level is stored (in the form of a list of several print flags) in the lattice record and can be changed by an appropriate call of the `SetPrintLevel` command described below.

The following example demonstrates the effect of the subgroup lattice print level.

```
gap> # Subgroup lattice of S4
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> lat := Lattice( s4 );
LatticeSubgroups( Group( (1,2,3,4), (1,2) ) )
```

The default subgroup lattice print level is 0. In this case, the print command provides just the expression mentioned above.

```
gap> # Subgroup lattice of S4 (continued)
gap> SetPrintLevel( lat, 1 );
gap> lat;
#I  class 1, size 1, length 1
#I  class 2, size 2, length 3
#I  class 3, size 2, length 6
#I  class 4, size 3, length 4
#I  class 5, size 4, length 1
#I  class 6, size 4, length 3
#I  class 7, size 4, length 3
#I  class 8, size 6, length 4
#I  class 9, size 8, length 3
#I  class 10, size 12, length 1
#I  class 11, size 24, length 1
LatticeSubgroups( Group( (1,2,3,4), (1,2) ) )
```

If the print level is set to a value greater than 0, you get, in addition, for each class a kind of heading line. This line contains the position number and the length of the respective class as well as the order of the subgroups in the class.

```
gap> # Subgroup lattice of S4 (continued)
gap> SetPrintLevel( lat, 2 );
gap> lat;
#I  class 1, size 1, length 1
#I    representative [  ]
#I      maximals
#I  class 2, size 2, length 3
#I    representative [ (1,2)(3,4) ]
#I      maximals [ 1, 1 ]
#I  class 3, size 2, length 6
#I    representative [ (3,4) ]
#I      maximals [ 1, 1 ]
#I  class 4, size 3, length 4
#I    representative [ (2,3,4) ]
#I      maximals [ 1, 1 ]
#I  class 5, size 4, length 1
#I    representative [ (1,2)(3,4), (1,3)(2,4) ]
```

```
#I      maximals [ 2, 1 ] [ 2, 2 ] [ 2, 3 ]
#I  class 6, size 4, length 3
#I    representative [ (3,4), (1,2) ]
#I      maximals [ 3, 1 ] [ 3, 4 ] [ 2, 1 ]
#I  class 7, size 4, length 3
#I    representative [ (1,2)(3,4), (1,4,2,3) ]
#I      maximals [ 2, 1 ]
#I  class 8, size 6, length 4
#I    representative [ (2,3,4), (3,4) ]
#I      maximals [ 4, 1 ] [ 3, 1 ] [ 3, 2 ] [ 3, 3 ]
#I  class 9, size 8, length 3
#I    representative [ (3,4), (1,2), (1,3)(2,4) ]
#I      maximals [ 7, 1 ] [ 6, 1 ] [ 5, 1 ]
#I  class 10, size 12, length 1
#I    representative [ (1,2)(3,4), (1,3)(2,4), (2,3,4) ]
#I      maximals [ 5, 1 ] [ 4, 1 ] [ 4, 2 ] [ 4, 3 ] [ 4, 4 ]
#I  class 11, size 24, length 1
#I    representative [ (1,2,3,4), (1,2) ]
#I      maximals [ 10, 1 ] [ 9, 1 ] [ 9, 2 ] [ 9, 3 ] [ 8, 1 ]
[ 8, 2 ] [ 8, 3 ] [ 8, 4 ]
LatticeSubgroups( Group( (1,2,3,4), (1,2) ) )
gap> PrintClassSubgroupLattice( lat, 8 );
#I  class 8, size 6, length 4
#I    representative [ (2,3,4), (3,4) ]
#I      maximals [ 4, 1 ] [ 3, 1 ] [ 3, 2 ] [ 3, 3 ]
```

If the subgroup lattice print level is at least 2, GAP3 prints, in addition, for each class representative subgroup a set of generators and a list of its maximal subgroups, where each maximal subgroup is represented by a pair of integers consisting of its class number and its position number in that class. As this information blows up the output, it may be convenient to restrict it to a particular class. We can do this by calling the `PrintClassSubgroupLattice` command described below.

```
gap> # Subgroup lattice of S4 (continued)
gap> SetPrintLevel( lat, 3 );
gap> PrintClassSubgroupLattice( lat, 8 );
#I  class 8, size 6, length 4
#I    representative [ (2,3,4), (3,4) ]
#I      maximals [ 4, 1 ] [ 3, 1 ] [ 3, 2 ] [ 3, 3 ]
#I    conjugate 2 by (1,4,3,2) is [ (1,2,3), (2,3) ]
#I    conjugate 3 by (1,2) is [ (1,3,4), (3,4) ]
#I    conjugate 4 by (1,3)(2,4) is [ (1,2,4), (1,2) ]
```

If the subgroup lattice print level has been set to at least 3, GAP3 displays, in addition, for each non-representative subgroup of a class its number in the class, an element which transforms the class representative subgroup into that subgroup, and a set of generators.

```
gap> # Subgroup lattice of S4 (continued)
gap> SetPrintLevel( lat, 4 );
gap> PrintClassSubgroupLattice( lat, 8 );
#I  class 8, size 6, length 4
```

```
#I    representative [ (2,3,4), (3,4) ]
#I      maximals [ 4, 1 ] [ 3, 1 ] [ 3, 2 ] [ 3, 3 ]
#I    conjugate 2 by (1,4,3,2) is [ (1,2,3), (2,3) ]
#I      maximals [ 4, 2 ] [ 3, 2 ] [ 3, 4 ] [ 3, 5 ]
#I    conjugate 3 by (1,2) is [ (1,3,4), (3,4) ]
#I      maximals [ 4, 3 ] [ 3, 1 ] [ 3, 5 ] [ 3, 6 ]
#I    conjugate 4 by (1,3)(2,4) is [ (1,2,4), (1,2) ]
#I      maximals [ 4, 4 ] [ 3, 4 ] [ 3, 6 ] [ 3, 3 ]
```

A subgroup lattice print level value of at least 4 causes GAP3 to list the maximal subgroups not only for the class representatives, but also for the other subgroups.

```
gap> # Subgroup lattice of S4 (continued)
gap> SetPrintLevel( lat, 5 );
gap> PrintClassSubgroupLattice( lat, 8 );
#I  class 8, size 6, length 4
#I    representative [ (2,3,4), (3,4) ]
#I      maximals [ 4, 1 ] [ 3, 1 ] [ 3, 2 ] [ 3, 3 ]
#I      minimals [ 11, 1 ]
#I    conjugate 2 by (1,4,3,2) is [ (1,2,3), (2,3) ]
#I      maximals [ 4, 2 ] [ 3, 2 ] [ 3, 4 ] [ 3, 5 ]
#I      minimals [ 11, 1 ]
#I    conjugate 3 by (1,2) is [ (1,3,4), (3,4) ]
#I      maximals [ 4, 3 ] [ 3, 1 ] [ 3, 5 ] [ 3, 6 ]
#I      minimals [ 11, 1 ]
#I    conjugate 4 by (1,3)(2,4) is [ (1,2,4), (1,2) ]
#I      maximals [ 4, 4 ] [ 3, 4 ] [ 3, 6 ] [ 3, 3 ]
#I      minimals [ 11, 1 ]
```

The maximal valid value of the subgroup lattice print level is 5. If it is set, GAP3 displays not only the maximal subgroups, but also the minimal supergroups of each subgroup. This is the most extensive output of a subgroup lattice record which you can get with the `Print` command, but of course you can use the `RecFields` command (see 46.13) to list all record components and then print them out individually in full detail.

If the computation of some subgroup lattice is very time consuming (as in the above example of the Mathieu group M_{12}), you might wish to see some intermediate printout which informs you about the progress of the computation. In fact, you can get such messages by activating a print mechanism which has been inserted into the subgroup lattice routines for diagnostic purposes. All you have to do is to replace the call

```
lat := Lattice( M12 );
```

by the three calls

```
InfoLattice1 := Print;
lat := Lattice( M12 );
InfoLattice1 := Ignore;
```

Note, however, that the final numbering of the conjugacy classes of subgroups will differ from the order in which they occur in the intermediate listing because they will be reordered by increasing subgroup orders at the end of the construction.

`PrintClassSubgroupLattice(` *lattice*, *n* `)`

`PrintClassSubgroupLattice` prints information on the nth conjugacy class of subgroups in the subgroup lattice *lattice*. The amount of this information depends on the current value of the subgroup lattice print level of *lattice*. Note that the default of that print level is zero which means that you will not get any output from the `PrintClassSubgroupLattice` command without increasing it (see `SetPrintLevel` below). Examples are given in the above description of the `Lattice` command.

`SetPrintLevel(` *lattice*, *level* `)`

`SetPrintLevel` changes the subgroup lattice print level of the subgroup lattice *lattice* to the specified value *level* by an appropriate alteration of the list of print flags which is stored in *lattice*`.printLevel`. The argument *level* is expected to be an integer between 0 and 5.

Examples of the effect of the subgroup lattice print level are given in the above description of the `Lattice` command.

7.76 ConjugacyClassSubgroups

`ConjugacyClassSubgroups(` G, U `)`

`ConjugacyClassSubgroups` returns the conjugacy class of the subgroup U in the group G. Signals an error if U is not a subgroup of G. The conjugacy class is returned as a domain, so all set theoretic functions are applicable (see 4).

```
gap> s5 := Group( (1,2), (1,2,3,4,5) );;  s5.name := "s5";;
gap> a5 := DerivedSubgroup( s5 );
Subgroup( s5, [ (1,2,3), (2,3,4), (3,4,5) ] )
gap> C := ConjugacyClassSubgroups( s5, a5 );
ConjugacyClassSubgroups( s5, Subgroup( s5,
[ (1,2,3), (2,3,4), (3,4,5) ] ) )
gap> Size( C );
1
```

Another example of such domains is given in section 7.74.

`ConjugacyClassSubgroups` calls
G`.operations.ConjugacyClassSubgroups(` G, U `)` and returns this value.

The default function called is `GroupOps.ConjugacyClassSubgroups`, which creates a conjugacy class record (see 7.79) with the operations record `ConjugacyClassSubgroupsOps` (see 7.78). Look in the index under **ConjugacyClassSubgroups** to see for which groups this function is overlaid.

7.77 IsConjugacyClassSubgroups

`IsConjugacyClassSubgroups(` *obj* `)`

`IsConjugacyClassSubgroups` returns `true` if *obj* is a conjugacy class of subgroups as created by `ConjugacyClassSubgroups` (see 7.76) and `false` otherwise.

```
gap> s5 := Group( (1,2), (1,2,3,4,5) );;  s5.name := "s5";;
gap> a5 := DerivedSubgroup( s5 );
```

```
Subgroup( s5, [ (1,2,3), (2,3,4), (2,4)(3,5) ] )
gap> c := ConjugacyClassSubgroups( s5, a5 );
ConjugacyClassSubgroups( s5, Subgroup( s5,
[ (1,2,3), (2,3,4), (2,4)(3,5) ] ) )
gap> IsConjugacyClassSubgroups( c );
true
gap> IsConjugacyClassSubgroups( [ a5 ] );
false     # even though this is as a set equal to c
```

7.78 Set Functions for Subgroup Conjugacy Classes

As mentioned above, conjugacy classes of subgroups are domains, so all set theoretic functions are also are applicable to conjugacy classes (see 4). This section describes the functions that are implemented especially for conjugacy classes. Functions not mentioned here inherit the default functions mentioned in the respective sections.

`Elements(` *C* `)`

The elements of the conjugacy class *C* with representative *U* in the group *G* are computed by first finding a right transversal of the normalizer of *U* in *G* and by computing the conjugates of *U* with the elements in the right transversal.

V `in` *C*

Membership of a group *V* is tested by comparing the set of contained cyclic subgroups of prime power order of *V* with those of the groups in *C*.

`Size(` *C* `)`

The size of the conjugacy class *C* with representative *U* in the group *G* is computed as the index of the normalizer of *U* in *G*.

7.79 Subgroup Conjugacy Class Records

Each conjugacy class of subgroups *C* is represented as a record with at least the following components.

`isDomain`
: always `true`, because conjugacy classes of subgroups are domains.

`isConjugacyClassSubgroups`
: as well, this entry is always set to `true`.

`group`
: The group in which the members of this conjugacy class lie. This is not necessarily a parent group; it may also be a subgroup.

`representative`
: The representative of the conjugacy class of subgroups as domain.

The following components are optional and may be bound by some functions which compute or make use of their value.

`normalizer`
: The normalizer of C`.representative` in C`.group`.

`normalizerLattice`
: A special entry that is used when the conjugacy classes of subgroups are computed by `ConjugacyClassesSubgroups`. It determines the normalizer of the subgroup C`.representative`. It is a list of length 2. The first element is another conjugacy class D (in the same group), the second is an element g in C`.group`. The normalizer of C`.representative` is then D`.representative ^` g.

`conjugands`
: A right transversal of the normalizer of C`.representative` in C`.group`. Thus the elements of the class C can be computed by conjugating C`.representative` with those elements.

7.80 ConjugacyClassesMaximalSubgroups

`ConjugacyClassesMaximalSubgroups(` G `)`

`ConjugacyClassesMaximalSubgroups` returns a list of conjugacy classes of maximal subgroups of the group G.

A subgroup H of G is **maximal** if H is a proper subgroup and for all subgroups I of G with $H < I \leq G$ the equality $I = G$ holds.

```
gap> s4 := SymmetricGroup( AgWords, 4 );;
gap> ss4 := SpecialAgGroup( s4 );;
gap> ConjugacyClassesMaximalSubgroups( ss4 );
[ ConjugacyClassSubgroups( Group( g1, g2, g3, g4 ), Subgroup( Group(
    g1, g2, g3, g4 ), [ g2, g3, g4 ] ) ),
  ConjugacyClassSubgroups( Group( g1, g2, g3, g4 ), Subgroup( Group(
    g1, g2, g3, g4 ), [ g1, g3, g4 ] ) ),
  ConjugacyClassSubgroups( Group( g1, g2, g3, g4 ), Subgroup( Group(
    g1, g2, g3, g4 ), [ g1, g2 ] ) ) ) ]
```

The generic method computes the entire lattice of conjugacy classes of subgroups (see 7.75) and returns the maximal ones.

`MaximalSubgroups` (see 7.81) computes the list of all maximal subgroups.

7.81 MaximalSubgroups

`MaximalSubgroups(` G `)`

MaximalSubgroups calculates all maximal subroups of the special ag group G.

```
gap> s4 := SymmetricGroup( AgWords, 4 );;
gap> ss4 := SpecialAgGroup( s4 );;
gap> MaximalSubgroups( ss4 );
[ Subgroup( Group( g1, g2, g3, g4 ), [ g2, g3, g4 ] ),
  Subgroup( Group( g1, g2, g3, g4 ), [ g1, g3, g4 ] ),
  Subgroup( Group( g1, g2, g3, g4 ), [ g1*g2^2, g3, g4 ] ),
  Subgroup( Group( g1, g2, g3, g4 ), [ g1*g2, g3, g4 ] ),
  Subgroup( Group( g1, g2, g3, g4 ), [ g1, g2 ] ),
```

```
    Subgroup( Group( g1, g2, g3, g4 ), [ g1, g2*g3*g4 ] ),
    Subgroup( Group( g1, g2, g3, g4 ), [ g1*g4, g2*g4 ] ),
    Subgroup( Group( g1, g2, g3, g4 ), [ g1*g4, g2*g3 ] ) ]
```

`ConjugacyClassesMaximalSubgroups` (see 7.80) computes the list of conjugacy classes of maximal subgroups.

7.82 NormalSubgroups

`NormalSubgroups(` *G* `)`

`NormalSubgroups` returns a list of all normal subgroups of G. The subgroups are sorted according to their sizes.

```
gap> s4 := Group( (1,2,3,4), (1,2) );; s4.name := "s4";;
gap> NormalSubgroups( s4 );
[ Subgroup( s4, [  ] ), Subgroup( s4, [ (1,2)(3,4), (1,4)(2,3) ] ),
  Subgroup( s4, [ (2,3,4), (1,3,4) ] ),
  Subgroup( s4, [ (3,4), (1,4), (1,2,4) ] ) ]
```

The default function `GroupOps.NormalSubgroups` uses the conjugacy classes of G and normal closures in order to compute the normal subgroups.

7.83 ConjugateSubgroups

`ConjugateSubgroups(` *G*`,` *U* `)`

`ConjugateSubgroups` returns the orbit of U under G acting by conjugation (see 7.20) as list of subgroups. U and G must have a common parent group.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s3 := Subgroup( s4, [ (1,2,3), (1,2) ] );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,3), (1,2) ] )
gap> ConjugateSubgroups( s4, s3 );
[ Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,3), (1,2) ] ),
  Subgroup( Group( (1,2,3,4), (1,2) ), [ (2,3,4), (2,3) ] ),
  Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,3,4), (3,4) ] ),
  Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,2,4), (1,4) ] ) ]
```

7.84 Cosets of Subgroups

The following sections describe how one can compute the right, left, and double cosets of subgroups (see 7.85, 7.90, 7.93). Further sections describe how cosets are created (see 7.86, 7.87, 7.91, 7.92, 7.94, and 7.95), and their implementation (see 7.88, 7.89, 7.96, and 7.97).

A coset is a GAP3 domain, which is different from a group. Altough the set of elements of a group and its trivial coset are equal, the group functions do not take trivial cosets as arguments. A trivial coset must be convert into a group using `AsGroup` (see 7.10) in order to be used as group.

7.85 RightCosets

`Cosets(` *G*`,` *U* `)`
`RightCosets(` *G*`,` *U* `)`

`Cosets` and `RightCosets` return a list of the right cosets of the subgroup U in the group G. The list is not sorted, i.e., the right cosets may appear in any order. The right cosets are domains as constructed by `RightCoset` (see 7.86).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> G.name := "G";;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> RightCosets( G, U );
[ (Subgroup( G, [ (1,2), (3,4) ] )*()),
  (Subgroup( G, [ (1,2), (3,4) ] )*(2,4,3)),
  (Subgroup( G, [ (1,2), (3,4) ] )*(2,3)),
  (Subgroup( G, [ (1,2), (3,4) ] )*(1,2,4,3)),
  (Subgroup( G, [ (1,2), (3,4) ] )*(1,2,3)),
  (Subgroup( G, [ (1,2), (3,4) ] )*(1,3)(2,4)) ]
```

If G is the parent of U, the dispatcher `RightCosets` first checks whether U has a component `rightCosets`. If U has this component, it returns that value. Otherwise it calls *G*`.operations.RightCosets(`*G*`,`*U*`)`, remembers the returned value in *U*`.rightCosets` and returns it. If G is not the parent of U, `RightCosets` directly calls the function *G*`.operations.RightCosets(`*G*`,`*U*`)` and returns that value.

The default function called this way is `GroupOps.RightCosets`, which calls `Orbit(` *G*`, RightCoset(` *U* `), OnRight )`. Look up `RightCosets` in the index, to see for which groups this function is overlaid.

7.86 RightCoset

U `*` *u*
`Coset(` *U*`,` *u* `)`
`RightCoset(` *U*`,` *u* `)`
`Coset(` *U* `)`
`RightCoset(` *U* `)`

The first three forms return the right coset of the subgroup U with the representative u. u must lie in the parent group of U, otherwise an error is signalled. In the last two forms the right coset of U with the identity element of the parent of U as representative is returned. In each case the right coset is returned as a domain, so all domain functions are applicable to right cosets (see chapter 4 and 7.88).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> U * (1,2,3);
(Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2), (3,4) ] )*(1,2,3))
```

`RightCosets` (see 7.85) computes the set of all right cosets of a subgroup in a group. `LeftCoset` (see 7.91) constructs left cosets.

`RightCoset` calls *U*`.operations.RightCoset(` *U*`,` *u* `)` and returns that value.

The default function called this way is `GroupOps.RightCoset`, which creates a right coset record (see 7.89) with the operations record `RightCosetGroupOps` (see 7.88). Look up the entries for `RightCoset` in the index to see for which groups this function is overlaid.

7.87 IsRightCoset

`IsRightCoset(` *obj* `)`
`IsCoset(` *obj* `)`

`IsRightCoset` and `IsCoset` return `true` if the object *obj* is a right coset, i.e., a record with the component `isRightCoset` with value `true`, and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> C := Subgroup( Group( (1,2), (1,2,3) ), [ (1,2,3) ] ) * (1,2);;
gap> IsRightCoset( C );
true
gap> D := (1,2) * Subgroup( Group( (1,2), (1,2,3) ), [ (1,2,3) ] );;
gap> IsCoset( D );
false     # note that D is a left coset record,
gap> C = D;
true      # though as a set, it is of course also a right coset
gap> IsCoset( 17 );
false
```

7.88 Set Functions for Right Cosets

Right cosets are domains, thus all set theoretic functions are applicable to cosets (see chapter 4). The following describes the functions that are implemented especially for right cosets. Functions not mentioned here inherit the default function mentioned in the respective sections.

More technically speaking, all right cosets of generic groups have the operations record `RightCosetGroupOps`, which inherits its functions from `DomainOps` and overlays the components mentioned below with more efficient functions.

In the following let C be the coset U `*` u.

`Elements(` C `)`

To compute the proper set of elements of a right coset C the proper set of elements of the subgroup U is computed, each element is multiplied by u, and the result is sorted.

`IsFinite(` C `)`

This returns the result of applying `IsFinite` to the subgroup U.

`Size(` C `)`

This returns the result of applying `Size` to the subgroup U.

C `=` D

If C and D are both right cosets of the same subgroup, `=` returns `true` if the quotient of the representatives lies in the subgroup U, otherwise the test is delegated to `DomainOps.=`.

h `in` U

If h is an element of the parent group of U, this returns `true` if the quotient h `/` u lies in the subgroup U, otherwise the test is delegated to `DomainOps.in`.

`Intersection(` C`,` D `)`

If C and D are both right cosets of subgroups U and V with the same parent group the result is a right coset of the intersection of U and V. The representative is found by a random search for a common element. In other cases the computation of the intersection is delegated to `DomainOps.Intersection`.

`Random(` C `)`

This takes a random element of the subgroup U and returns the product of this element by the representative u.

`Print(` C `)`

A right coset C is printed as `(`U `*` u`)` (the parenthesis are used to avoid confusion about the precedence, which could occur if the coset is part of a larger object).

C `*` v

If v is an element of the parent group of the subgroup U, the result is a new right coset of U with representative u `*` v. Otherwise the result is obtained by multiplying the proper set of elements of C with the element v, which may signal an error.

v `*` C

The result is obtained by multiplying the proper set of elements of the coset C with the element v, which may signal an error.

7.89 Right Cosets Records

A right coset is represented by a domain record with the following tag components.

`isDomain`
: always `true`.

`isRightCoset`
: always `true`.

The right coset is determined by the following identity components, which every right coset record has.

`group`
: the subgroup U of which this right coset is a right coset.

`representative`
: an element of the right coset. It is unspecified which element.

In addition, a right coset record may have the following optional information components.

`elements`
: if present the proper set of elements of the coset.

`isFinite`
: if present this is `true` if the coset is finite, and `false` if the coset is infinite. If not present it is not known whether the coset is finite or infinite.

`size`
: if present the size of the coset. Is "infinity" if the coset is infinite. If not present the size of the coset is not known.

7.90 LeftCosets

`LeftCosets(` G`,` U `)`

`LeftCosets` returns a list of the left cosets of the subgroup U in the group G. The list is not sorted, i.e., the left cosets may appear in any order. The left cosets are domains as constructed by `LeftCosets` (see 7.90).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> G.name := "G";;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> LeftCosets( G, U );
[ (()*Subgroup( G, [ (1,2), (3,4) ] )),
  ((2,3,4)*Subgroup( G, [ (1,2), (3,4) ] )),
  ((2,3)*Subgroup( G, [ (1,2), (3,4) ] )),
  ((1,3,4,2)*Subgroup( G, [ (1,2), (3,4) ] )),
  ((1,3,2)*Subgroup( G, [ (1,2), (3,4) ] )),
  ((1,3)(2,4)*Subgroup( G, [ (1,2), (3,4) ] )) ]
```

If G is the parent of U, the dispatcher `LeftCosets` first checks whether U has a component `leftCosets`. If U has this component, it returns that value. Otherwise `LeftCosets` calls G`.operations.LeftCosets(`G`,`U`)`, remembers the returned value in U`.leftCosets` and returns it. If G is not the parent of U, `LeftCosets` calls G`.operations.LeftCosets(`G`,`U`)` directly and returns that value.

The default function called this way is `GroupOps.LeftCosets`, which calls `RightCosets(` G`,` U `)` and turns each right coset U `*` u into the left coset u`^-1 *` U. Look up the entries for `LeftCosets` in the index, to see for which groups this function is overlaid.

7.91 LeftCoset

u `*` U
`LeftCoset(` U`,` u `)`
`LeftCoset(` U `)`

`LeftCoset` is exactly like `RightCoset`, except that it constructs left cosets instead of right cosets. So everything that applies to `RightCoset` applies also to `LeftCoset`, with **right** replaced by **left** (see 7.86, 7.88, 7.89).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> (1,2,3) * U;
((1,2,3)*Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2), (3,4) ] ))
```

`LeftCosets` (see 7.90) computes the set of all left cosets of a subgroup in a group.

7.92 IsLeftCoset

`IsLeftCoset(` *obj* `)`

`IsLeftCoset` returns `true` if the object *obj* is a left coset, i.e., a record with the component `isLeftCoset` with value `true`, and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> C := (1,2) * Subgroup( Group( (1,2), (1,2,3) ), [ (1,2,3) ] );;
gap> IsLeftCoset( C );
true
gap> D := Subgroup( Group( (1,2), (1,2,3) ), [ (1,2,3) ] ) * (1,2);;
gap> IsLeftCoset( D );
false     # note that D is a right coset record,
gap> C = D;
true      # though as a set, it is of course also a left coset
gap> IsLeftCoset( 17 );
false
```

`IsRightCoset` (see 7.87) tests if an object is a right coset.

7.93 DoubleCosets

`DoubleCosets(` G`,` U`,` V `)`

`DoubleCosets` returns a list of the double cosets of the subgroups U and V in the group G. The three groups G, U and V must have a common parent. The list is not sorted, i.e., the double cosets may appear in any order. The double cosets are domains as constructed by `DoubleCoset` (see 7.94).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;  U.name := "U";;
gap> DoubleCosets( G, U, U );
[ DoubleCoset( U, (), U ), DoubleCoset( U, (2,3), U ),
  DoubleCoset( U, (1,3)(2,4), U ) ]
```

`DoubleCosets` calls G`.operations.DoubleCoset(` G`,` U`,` V `)` and returns that value.

The default function called this way is `GroupOps.DoubleCosets`, which takes random elements from G, tests if this element lies in one of the already found double cosets, adds the double coset if this is not the case, and continues this until the sum of the sizes of the found double cosets equals the size of G. Look up `DoubleCosets` in the index, to see for which groups this function is overlaid.

7.94 DoubleCoset

`DoubleCoset(` *U*`,` *u*`,` *V* `)`

`DoubleCoset` returns the double coset with representative u and left group U and right group V. U and V must have a common parent and u must lie in this parent, otherwise an error is signaled. Double cosets are domains, so all domain function are applicable to double cosets (see chapter 4 and 7.96).

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> D := DoubleCoset( U, (1,2,3), U );
DoubleCoset( Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2), (3,4) ] ),
(1,2,3), Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2), (3,4) ] ) )
gap> Size( D );
16
```

`DoubleCosets` (see 7.93) computes the set of all double cosets of two subgroups in a group.

`DoubleCoset` calls *U*`.operations.DoubleCoset(`*U*`,`*u*`,`*V*`)` and returns that value.

The default function called this way is `GroupOps.DoubleCoset`, which creates a double coset record (see 7.97) with the operations record `DoubleCosetGroupOps` (see 7.96). Look up `DoubleCosets` in the index to see for which groups this function is overlaid.

7.95 IsDoubleCoset

`IsDoubleCoset(` *obj* `)`

`IsDoubleCoset` returns `true` if the object *obj* is a double coset, i.e., a record with the component `isDoubleCoset` with value `true`, and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> G := Group( (1,2), (1,2,3,4) );;
gap> U := Subgroup( G, [ (1,2), (3,4) ] );;
gap> D := DoubleCoset( U, (1,2,3), U );;
gap> IsDoubleCoset( D );
true
```

7.96 Set Functions for Double Cosets

Double cosets are domains, thus all set theoretic functions are applicable to double cosets (see chapter 4). The following describes the functions that are implemented especially for double cosets. Functions not mentioned here inherit the default functions mentioned in the respective sections.

More technically speaking, double cosets of generic groups have the operations record `DoubleCosetGroupOps`, which inherits its functions from `DomainOps` and overlays the components mentioned below with more efficient functions.

Most functions below use the component *D*`.rightCosets` that contains a list of right cosets of the left group U whose union is this double coset. If this component is unbound they will compute it by computing the orbit of the right group V on the right coset *U* `*` *u*, where u is the representative of the double coset (see 7.97).

`Elements(` D `)`

To compute the proper set of elements the union of the right cosets D`.rightCosets` is computed.

`IsFinite(` D `)`

This returns the result of `IsFinite(` U `) and IsFinite(` V `)`.

`Size(` D `)`

This returns the size of the left group U times the number of cosets in D`.rightCosets`.

C `=` D

If C and D are both double cosets with the same left and right groups this returns the result of testing whether the representative of C lies in D. In other cases the test is delegated to `DomainOps.=`.

g `in` D

If g is an element of the parent group of the left and right group of D, this returns `true` if g lies in one of the right cosets in D`.rightCosets`. In other cases the the test is delegated to `DomainOps.in`.

`Intersection(` C`,` D `)`

If C and D are both double cosets that are equal, this returns C. If C and D are both double cosets with the same left and right groups that are not equal, this returns `[]`. In all other cases the computation is delegated to `DomainsOps.Intersection`.

`Random(` D `)`

This takes a random right coset from D`.rightCosets` and returns the result of applying `Random` to this right coset.

`Print(` D `)`

This prints the double coset in the form `DoubleCoset(` U`,` u`,` V `)`.

D `*` g
g `*` D

Those returns the result of multiplying the proper set of element of D with the element g, which may signal an error.

7.97 Double Coset Records

A double coset is represented by a domain record with the following tag components.

`isDomain`
: always `true`.

`isDoubleCoset`
: always `true`.

The double coset is determined by the following identity components, which every double coset must have.

`leftGroup`
: the left subgroup U.

`rightGroup`
: the right subgroup V.

`representative`
: an element of the double coset. It is unspecified which element.

In addition, a double coset record may have the following optional information components.

`rightCosets`
: a list of disjoint right cosets of the left subgroup U, whose union is the double coset.

`elements`
: if present the proper set of elements of the double coset.

`isFinite`
: if present this is `true` if the double coset is finite and `false` if the double coset is infinite. If not present it is not known whether the double coset is finite or infinite.

`size`
: if present the size of the double coset. Is "infinity" if the coset is infinite. If not present the size of the double coset is not known.

7.98 Group Constructions

The following functions construct new parent groups from given groups (see 7.99, 7.101, 7.103 and 7.104).

7.99 DirectProduct

`DirectProduct(` G_1`, ...,` G_n `)`

`DirectProduct` returns a group record of the direct product D of the groups G_1, ..., G_n which need not to have a common parent group, it is even possible to construct the direct product of an ag group with a permutation group.

Note that the elements of the direct product may be just represented as records. But more complicate constructions, as for instance installing a new collector, may be used. The choice of method strongly depends on the type of group arguments.

`Embedding(` U`,` D`,` i `)`

Let U be a subgroup of G_i. `Embedding` returns a homomorphism of U into D which describes the embedding of U in D.

`Projection(` D, U, i `)`

Let U be a supergroup of G_i. `Projection` returns a homomorphism of D into U which describes the projection of D onto G_i.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> S4 := AgGroup( s4 );
Group( g1, g2, g3, g4 )
gap> D := DirectProduct( s4, S4 );
Group( DirectProductElement(
(1,2,3,4), IdAgWord ), DirectProductElement(
(1,2), IdAgWord ), DirectProductElement( (),
g1 ), DirectProductElement( (), g2 ), DirectProductElement( (),
g3 ), DirectProductElement( (), g4 ) )
gap> pr := Projection( D, s4, 1 );;
gap> Image( pr );
Group( (1,2,3,4), (1,2) )
```

7.100 DirectProduct for Groups

`GroupOps.DirectProduct(` L `)`

Let L be a list of groups $G_1, ..., G_n$. Then a group element g of the direct product D is represented as record containing the following components.

`element`
: a list $g_1 \in G_1, ..., g_n \in G_n$ describing g.

`domain`
: contains `GroupElements`.

`isGroupElement`
: contains `true`.

`isDirectProductElement`
: contains `true`.

`operations`
: contains the operations record `DirectProductElementOps` (see 4.5).

7.101 SemidirectProduct

`SemidirectProduct(` G, a, H `)`

`SemidirectProduct` returns the semidirect product of G with H. a must be a homomorphism that from G onto a group A that operates on H via the caret (`^`) operator. A may either be a subgroup of the parent group of H that normalizes H, or a subgroup of the automorphism group of H, i.e., a group of automorphisms (see 7.106).

The semidirect product of G and H is a the group of pairs (g, h) with $g \in G$ and $h \in H$, where the product of $(g_1, h_1)(g_2, h_2)$ is defined as $(g_1 g_2, h_1^{g_2^a} h_2)$. Note that the elements $(1_G, h)$ form a normal subgroup in the semidirect product.

`Embedding(` U, S, `1 )`

Let U be a subgroup of G. `Embedding` returns the homomorphism of U into the semidirect product S where u is mapped to `(`u`,1)`.

`Embedding(` U, S, `2 )`

Let U be a subgroup of H. `Embedding` returns the homomorphism of U into the semidirect product S where u is mapped to `(1,`u`)`.

`Projection(` S, G, `1 )`

`Projection` returns the homomorphism of S onto G, where `(`g`,`h`)` is mapped to g.

`Projection(` S, H, `2 )`

`Projection` returns the homomorphism of S onto H, where `(`g`,`h`)` is mapped to h.

It is not specified how the elements of the semidirect product are represented. Thus `Embedding` and `Projection` are the only general possibility to relate G and H with the semidirect product.

```
gap> s4 := Group( (1,2), (1,2,3,4) );;  s4.name := "s4";;
gap> s3 := Subgroup( s4, [ (1,2), (1,2,3) ] );; s3.name := "s3";;
gap> a4 := Subgroup( s4, [ (1,2,3), (2,3,4) ] );;  a4.name := "a4";;
gap> a := IdentityMapping( s3 );;
gap> s := SemidirectProduct( s3, a, a4 );
Group( SemidirectProductElement( (1,2),
(1,2), () ), SemidirectProductElement( (1,2,3),
(1,2,3), () ), SemidirectProductElement( (), (),
(1,2,3) ), SemidirectProductElement( (), (), (2,3,4) ) )
gap> Size( s );
72
```

Note that the three arguments of `SemidirectProductElement` are the element g, its image under a, and the element h.

`SemidirectProduct` calls the function G`.operations.SemidirectProduct` with the arguments G, a, and H, and returns the result.

The default function called this way is `GroupOps.SemidirectProduct`. This function constructs the semidirect product as a group of semidirect product elements (see 7.102). Look in the index under **SemidirectProduct** to see for which groups this function is overlaid.

7.102 SemidirectProduct for Groups

The function `GroupOps.SemidirectProduct` constructs the semidirect product as a group of semidirect product elements. In the following let G, a, and H be the arguments of `SemidirectProduct`.

Each such element `(`g`,`h`)` is represented by a record with the following components.

`element`
: the list `[` g, h `]`.

`automorphism`
: contains the image of g under a.

`isGroupElement`
: always `true`.

`isSemidirectProductElement`
: always `true`.

`domain`
: contains `GroupElements`.

`operations`
: contains the operations record `SemidirectProductOps`.

The operations of semidirect product elements in done in the obvious way.

7.103 SubdirectProduct

`SubdirectProduct(` *G1* `,` *G2* `,` *h1* `,` *h2* `)`

`SubdirectProduct` returns the subdirect product of the groups *G1* and *G2*. *h1* and *h2* must be homomorphisms from *G1* and *G2* into a common group H.

The subdirect product of G_1 and G_2 is the subgroup of the direct product of G_1 and G_2 of those elements (g_1, g_2) with $g_1^{h_1} = g_2^{h_2}$. This subgroup is generated by the elements (g_1, x_{g_1}), where g_1 loops over the generators of G_1 and $x_{g_1} \in G_2$ is an arbitrary element such that $g_1^{h_1} = x_{g_1}^{h_2}$ together with the element $(1_G, k_2)$ where k_2 loops over the generators of the kernel of h_2.

`Projection(` *S* `,` *G1* `, 1 )`

`Projection` returns the projection of S onto *G1*, where `(`*g1* `,`*g2*`)` is mapped to *g1*.

`Projection(` *S* `,` *G2* `, 2 )`

`Projection` returns the projection of S onto *G2*, where `(`*g1* `,`*g2*`)` is mapped to *g2*.

It is not specified how the elements of the subdirect product are represented. Therefor `Projection` is the only general possibility to relate *G1* and *G2* with the subdirect product.

```
gap> s3 := Group( (1,2,3), (1,2) );;
gap> c3 := Subgroup( s3, [ (1,2,3) ] );;
gap> x1 := Operation( s3, Cosets( s3, c3 ), OnRight );;
gap> h1 := OperationHomomorphism( s3, x1 );;
gap> d8 := Group( (1,2,3,4), (2,4) );;
gap> c4 := Subgroup( d8, [ (1,2,3,4) ] );;
gap> x2 := Operation( d8, Cosets( d8, c4 ), OnRight );;
gap> h2 := OperationHomomorphism( d8, x2 );;
gap> s := SubdirectProduct( s3, d8, h1, h2 );
Group( (1,2,3), (1,2)(5,7), (4,5,6,7) )
gap> Size( s );
24
```

`SubdirectProduct` calls the function *G1*`.operations.SubdirectProduct` with the arguments *G1*, *G2*, *h1*, and *h2*.

The default function called this way is `GroupOps.SubdirectProduct`. This function constructs the subdirect product as a subgroup of the direct product. The generators for this subgroup are computed as described above.

7.104 WreathProduct

`WreathProduct(` G`,` H `)`
`WreathProduct(` G`,` H`,` α `)`

In the first form of `WreathProduct` the right regular permutation representation of H on its elements is used as the homomorphism α. In the second form α must be a homomorphism of H into a permutation group. Let d be the degree of the range of α. Then `WreathProduct` returns the wreath product of G by H with respect to α, that is the semi-direct product of the direct product of d copies of G which are permuted by H through application of α to H.

```
gap> s3 := Group( (1,2,3), (1,2) );
Group( (1,2,3), (1,2) )
gap> z2 := CyclicGroup( AgWords, 2 );
Group( c2 )
gap> f := IdentityMapping( s3 );
IdentityMapping( Group( (1,2,3), (1,2) ) )
gap> w := WreathProduct( z2, s3, f );
Group( WreathProductElement(
c2, IdAgWord, IdAgWord, (), () ), WreathProductElement( IdAgWord,
c2, IdAgWord, (), () ), WreathProductElement( IdAgWord, IdAgWord,
c2, (), () ), WreathProductElement( IdAgWord, IdAgWord, IdAgWord,
(1,2,3),
(1,2,3) ), WreathProductElement( IdAgWord, IdAgWord, IdAgWord, (1,2),
(1,2) ) )
gap> Factors( Size( w ) );
[ 2, 2, 2, 2, 3 ]
```

7.105 WreathProduct for Groups

`GroupOps.WreathProduct(` G`,` H`,` α `)`

Let d be the degree of α`.range`. A group element of the wreath product W is represented as a record containing the following components.

`element`
: a list of d elements of G followed by an element h of H.

`permutation`
: the image of h under α.

`domain`
: contains `GroupElements`.

`isGroupElement`
: contains `true`.

`isWreathProductElement`
: contains `true`.

`operations`
: contains the operations record `WreathProductElementOps` (see 4.5).

7.106 Group Homomorphisms

Since groups is probably the most important category of domains in GAP3 group homomorphisms are probably the most important homomorphisms (see chapter 44)

A **group homomorphism** ϕ is a mapping that maps each element of a group G, called the source of ϕ, to an element of another group H, called the range of ϕ, such that for each pair $x, y \in G$ we have $(xy)^\phi = x^\phi y^\phi$.

Examples of group homomorphisms are the natural homomorphism of a group into a factor group (see 7.110) and the homomorphism of a group into a symmetric group defined by an operation (see 8.21). Look under **group homomorphisms** in the index for a list of all available group homomorphisms.

Since group homomorphisms are just a special case of homomorphisms, all functions described in chapter 44 are applicable to all group homomorphisms, e.g., the function to test if a homomorphism is an automorphism (see 44.6). More general, since group homomorphisms are just a special case of mappings all functions described in chapter 43 are also applicable, e.g., the function to compute the image of an element under a group homomorphism (see 43.8).

The following sections describe the functions that test whether a mapping is a group homomorphism (see 7.107), compute the kernel of a group homomorphism (see 7.108), how the general mapping functions are implemented for group homomorphisms (see 7.109), the natural homomorphism of a group onto a factor group (see 7.110), homomorphisms by conjugation (see 7.111, 7.112), and the most general group homomorphism, which is defined by simply specifying the images of a set of generators (see 7.113).

7.107 IsGroupHomomorphism

`IsGroupHomomorphism(` *map* `)`

`IsGroupHomomorphism` returns `true` if the function *map* is a group homomorphism and `false` otherwise. Signals an error if *map* is a multi value mapping.

A mapping *map* is a group homomorphism if its source G and range H are both groups and if for every pair of elements $x, y \in G$ it holds that $(xy)^{map} = x^{map} y^{map}$.

```
gap> s4 := Group( (1,2), (1,2,3,4) );;
gap> v4 := Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] );;
gap> phi := NaturalHomomorphism( s4, s4/v4 );;
gap> IsGroupHomomorphism( phi );
true
gap> IsHomomorphism( phi );
true     # since the source is a group this is equivalent to the above
gap> IsGroupHomomorphism( FrobeniusAutomorphism( GF(16) ) );
false    # it is a field automorphism
```

`IsGroupHomomorphism` first tests if the flag *map*`.isGroupHomomorphism` is bound. If the flag is bound, `IsGroupHomomorphism` returns its value. Otherwise it calls *map*`.source.operations.IsGroupHomomorphism(` *map* `)`, remembers the returned value in *map*`.isGroupHomomorphism`, and returns it. Note that of course all functions that create group homomorphisms set the flag *map*`.isGroupHomomorphism` to `true`, so that no function is called for those group homomorphisms.

The default function called this way is `MappingOps.IsGroupHomomorphism`. It computes all the elements of the source of *map* and for each such element x and each generator y tests whether $(xy)^{map} = x^{map}y^{map}$. Look under **IsHomomorphism** in the index to see for which mappings this function is overlaid.

7.108 KernelGroupHomomorphism

`KernelGroupHomomorphism(` *hom* `)`

`KernelGroupHomomorphism` returns the kernel of the group homomorphism *hom* as a subgroup of the group *hom*`.source`.

The **kernel** of a group homomorphism *hom* is the subset of elements x of the source G that are mapped to the identity of the range H, i.e., $x^{hom} = H.identity$.

```
gap> s4 := Group( (1,2), (1,2,3,4) );;
gap> v4 := Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] );;
gap> phi := NaturalHomomorphism( s4, s4/v4 );;
gap> KernelGroupHomomorphism( phi );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2)(3,4), (1,3)(2,4) ] )
gap> Kernel( phi );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2)(3,4), (1,3)(2,4) ] )
    # since the source is a group this is equivalent to the above
gap> rho := GroupHomomorphismByImages( s4, Group( (1,2) ),
>                             [ (1,2), (1,2,3,4) ], [ (1,2), (1,2) ] );;
gap> Kernel( rho );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (2,4,3), (1,4,3) ] )
```

`KernelGroupHomomorphism` first tests if *hom*`.kernelGroupHomomorphism` is bound. If it is bound, `KernelGroupHomomorphisms` returns that value. Otherwise it calls *hom*`.operations.KernelGroupHomomorphism(` *hom* `)`, remembers the returned value in *hom*`.kernelGroupHomomorphism`, and returns it.

The default function for this is `MappingOps.KernelGroupHomomorphism`, which simply tries random elements of the source of *hom*, until the subgroup generated by those that map to the identity has the correct size, i.e., `Size(` *hom*`.source ) / Size( Image( hom ) )`. Note that this implies that the image of *hom* and its size are computed. Look under **Kernel** in the index to see for which group homomorphisms this function is overlaid.

7.109 Mapping Functions for Group Homomorphisms

This section describes how the mapping functions defined in chapter 43 are implemented for group homomorphisms. Those functions not mentioned here are implemented by the default functions described in the respective sections.

`IsInjective(` *hom* `)`

The group homomorphism *hom* is injective if the kernel of *hom* `KernelGroupHomomorphism(` *hom* `)` (see 7.108) is trivial.

`IsSurjective(` *hom* `)`

The group homomorphism *hom* is surjective if the size of the image `Size( Image(` *hom* `) )` (see 43.8 and below) is equal to the size of the range `Size(` *hom*`.range )`.

hom1 `=` *hom2*

The two group homomorphisms *hom1* and *hom2* are equal if the have the same source and range and if the images of the generators of the source under *hom1* and *hom2* are equal.

hom1 `<` *hom2*

By definition *hom1* is smaller than *hom2* if either the source of *hom1* is smaller than the source of *hom2*, or, if the sources are equal, if the range of *hom1* is smaller than the range of *hom2*, or, if sources and ranges are equal, the image of the smallest element x of the source for that the images are not equal under *hom1* is smaller than the image under *hom2*. Therefor `GroupHomomorphismOps.<` first compares the sources and the ranges. For group homomorphisms with equal sources and ranges only the images of the smallest irredundant generating system are compared. A generating system $g_1, g_2, ..., g_n$ is called irredundant if no g_i lies in the subgroup generated by $g_1, ..., g_{i-1}$. The smallest irredundant generating system is simply the smallest such generating system with respect to the lexicographical ordering.

`Image(` *hom* `)`
`Image(` *hom*`,` *H* `)`
`Images(` *hom*`,` *H* `)`

The image of a subgroup under a group homomorphism is computed by computing the images of a set of generators of the subgroup, and the result is the subgroup generated by those images.

`PreImages(` *hom*`,` *elm* `)`

The preimages of an element under a group homomorphism are computed by computing a representative with `PreImagesRepresentative(` *hom*`,` *elm* `)` and the result is the coset of `Kernel(` *hom* `)` containing this representative.

`PreImage(` *hom* `)`
`PreImage(` *hom*`,` *H* `)`
`PreImages(` *hom*`,` *H* `)`

The preimages of a subgroup under a group homomorphism are computed by computing representatives of the preimages of all the generators of the subgroup, adding the generators of the kernel of *hom*, and the result is the subgroup generated by those elements.

Look under **IsInjective**, **IsSurjective**, **equality**, **ordering**, **Image**, **Images**, **PreImage**, and **PreImages** in the index to see for which group homomorphisms these functions are overlaid.

7.110 NaturalHomomorphism

`NaturalHomomorphism(` *G*`,` *F* `)`

`NaturalHomomorphism` returns the natural homomorphism of the group G into the factor group F. F must be a factor group, i.e., the result of `FactorGroup(`*H*`,`*N*`)` (see 7.33) or *H*`/`*N* (see 7.117), and G must be a subgroup of H.

Mathematically the factor group H/N consists of the cosets of N, and the natural homomorphism ϕ maps each element h of H to the coset Nh. Note that in GAP3 the representation of factor group elements is unspecified, but they are **never** cosets (see 7.87), because cosets are domains and not group elements in GAP3. Thus the natural homomorphism is the only connection between a group and one of its factorgroups.

G is the source of the natural homomorphism ϕ, F is its range. Note that because G may be a proper subgroup of the group H of which F is a factor group ϕ need not be surjective, i.e., the image of ϕ may be a proper subgroup of F. The kernel of ϕ is of course the intersection of N and G.

```
gap> s4 := Group( (1,2), (1,2,3,4) );;
gap> v4 := Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] );;
gap> v4.name := "v4";;
gap> phi := NaturalHomomorphism( s4, s4/v4 );;
gap> (1,2,3) ^ phi;
FactorGroupElement( v4, (2,4,3) )
gap> PreImages( phi, last );
(v4*(2,4,3))
gap> (1,2,3) in last;
true
gap> rho :=
>  NaturalHomomorphism( Subgroup( s4, [ (1,2), (1,2,3) ] ), s4/v4 );;
gap> Kernel( rho );
Subgroup( Group( (1,2), (1,2,3,4) ), [  ] )
gap> IsIsomorphism( rho );
true
```

`NaturalHomomorphism` calls
F`.operations.NaturalHomomorphism(` *G*`,` *F* `)` and returns that value.

The default function called this way is `GroupOps.NaturalHomomorphism`. The homomorphism constructed this way has the operations record `NaturalHomomorphismOps`. It computes the image of an element g of G by calling `FactorGroupElement(` *N*`,` *g* `)`, the preimages of an factor group element f as `Coset( Kernel(`*phi*`),` *f*`.element.representative )`, and the kernel by computing `Intersection(` *G*`,` *N* `)`. Look under **NaturalHomomorphism** in the index to see for which groups this function is overlaid.

7.111 ConjugationGroupHomomorphism

`ConjugationGroupHomomorphism(` *G*`,` *H*`,` *x* `)`

`ConjugationGroupHomomorphism` returns the homomorphism from G into H that takes each element g in G to the element g `^` x. G and H must have a common parent group P and x must lie in this parent group. Of course G `^` x must be a subgroup of H.

```
gap> d12 := Group( (1,2,3,4,5,6), (2,6)(3,5) );; d12.name := "d12";;
gap> c2 := Subgroup( d12, [ (2,6)(3,5) ] );
Subgroup( d12, [ (2,6)(3,5) ] )
gap> v4 := Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6) ] );
Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6) ] )
gap> x := ConjugationGroupHomomorphism( c2, v4, (1,3,5)(2,4,6) );
ConjugationGroupHomomorphism( Subgroup( d12,
[ (2,6)(3,5) ] ), Subgroup( d12, [ (1,2)(3,6)(4,5), (1,4)(2,5)(3,6)
 ] ), (1,3,5)(2,4,6) )
gap> IsSurjective( x );
false
gap> Image( x );
Subgroup( d12, [ (1,5)(2,4) ] )
```

`ConjugationGroupHomomorphism` calls
G`.operations.ConjugationGroupHomomorphism(` G, H, x `)` and returns that value.

The default function called is `GroupOps.ConjugationGroupHomomorphism`. It just creates a homomorphism record with range G, source H, and the component `element` with the value x. It computes the image of an element g of G as g `^` x. If the sizes of the range and the source are equal the inverse of such a homomorphism is computed as a conjugation homomorphism from H to G by x`^-1`. To multiply two such homomorphisms their elements are multiplied. Look under **ConjugationGroupHomomorphism** in the index to see for which groups this default function is overlaid.

7.112 InnerAutomorphism

`InnerAutomorphism(` G, g `)`

`InnerAutomorphism` returns the automorphism on the group G that takes each element h to h `^` g. g must be an element in the parent group of G (but need not actually be in G) that normalizes G.

```
gap> s5 := Group( (1,2), (1,2,3,4,5) );;  s5.name := "s5";;
gap> i := InnerAutomorphism( s5, (1,2) );
InnerAutomorphism( s5, (1,2) )
gap> (1,2,3,4,5) ^ i;
(1,3,4,5,2)
```

`InnerAutomorphism(` G, g `)` calls `ConjugationGroupHomomorphism(` G, G, g `)` (see 7.111).

7.113 GroupHomomorphismByImages

`GroupHomomorphismByImages(` G, H, *gens*, *imgs* `)`

`GroupHomomorphismByImages` returns the group homomorphism with source G and range H that is defined by mapping the list *gens* of generators of G to the list *imgs* of images in H.

```
gap> g := Group( (1,2,3,4), (1,2) );;
gap> h := Group( (2,3), (1,2) );;
```

```
gap> m := GroupHomomorphismByImages(g,h,g.generators,h.generators);
GroupHomomorphismByImages( Group( (1,2,3,4), (1,2) ), Group( (2,3),
(1,2) ), [ (1,2,3,4), (1,2) ], [ (2,3), (1,2) ] )
gap> Image( m, (1,3,4) );
(1,3,2)
gap> Kernel( m );
Subgroup( Group( (1,2,3,4), (1,2) ), [ (1,4)(2,3), (1,2)(3,4) ] )
```

Note that the result need not always be a **single value** mapping, even though the name seems to imply this. Namely if the elements in *imgs* do not satisfy all relations that hold for the generators *gens*, no element of G has a unique image under the mapping. This is demonstrated in the following example.

```
gap> g := Group( (1,2,3,4,5,6,7,8,9,10) );;
gap> h := Group( (1,2,3,4,5,6) );;
gap> m := GroupHomomorphismByImages(g,h,g.generators,h.generators);
GroupHomomorphismByImages( Group( ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10
 ) ), Group( (1,2,3,4,5,6) ), [ ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10) ],
[ (1,2,3,4,5,6) ] )
gap> IsMapping( m );
false
gap> Images( m, () );
(Subgroup( Group( (1,2,3,4,5,6) ), [ ( 1, 3, 5)( 2, 4, 6) ] )*())
gap> g.1^10;
()      # the generator of g satisfies this relation
gap> h.1^10;
(1,5,3)(2,6,4)      # but its image does not
```

The set of images of the identity returned by `Images` is the set of elements `h.1^`*n* such that `g.1^`*n* is the identity in `g`.

The test whether a mapping constructed by `GroupHomomorphismByImages` is a single valued mapping, is usually quite expensive. Note that this test is automatically performed the first time that you apply a function that expects a single valued mapping, e.g., `Image` or `Images`. There are two possibilities to avoid this test. When you know that the mapping constructed is really a single valued mapping, you can set the flag *map*`.isMapping` to `true`. Then the functions assume that *map* is indeed a mapping and do not test it again. On the other hand if you are not certain whether the mapping is single valued, you can use `ImagesRepresentative` instead of `Image` (see 43.10). `ImagesRepresentative` returns just one possible image, without testing whether there might actually be more than one possible image.

`GroupHomomorphismByImages` calls
G`.operations.GroupHomomorphismByImages(` *G*`,` *H*`,` *gens*`,` *imgs* `)`
and returns this value.

The default function called this way is `GroupOps.GroupHomomorphismByImages`. Below we describe how the mapping functions are implemented for such a mapping. The functions not mentioned below are implemented by the default functions described in 7.109.

All the function below first compute the list of elements of G with an orbit algorithm, sorts this list, and stores this list in *hom*`.elements`. In parallel they computes and sort a list of images, and store this list in *hom*`.images`.

`IsMapping(` *map* `)`

The mapping constructed by `GroupHomomorphismByImages` is a single valued mapping if for each i and for each k the following equation holds
map`.images[Position(`*map*`.elements,`*map*`.elements[`i`]*`*gens*`[`k`])]`
`=` *map*`.images[`i`] *` *imgs*`[`k`]`.

`Image(` *map*`,` *elm* `)`

If the mapping *map* is a single valued mapping, the image of an element *elm* is computed as *map*`.images[ Position(`*map*`.elements,`*elm*`) ]`.

`ImagesRepresentative(` *map*`,` *elm* `)`

The representative of the images of an element *elm* under the mapping *map* is computed as *map*`.images[ Position(`*map*`.elements,`*elm*`) ]`.

`InverseMapping(` *map* `)`

The inverse of the mapping *map* is constructed as `GroupHomomorphismByImages(` H`,` G`,` *imgs*`,` *gens* `)`.

`CompositionMapping(` *map1*`,` *map2* `)`

If *map2* is a mapping constructed by `GroupHomomorphismByImages` the composition is constructed by making a copy of *map2* and replacing every element in *map2*`.images` with its image under *map1*.

Look under **GroupHomomorphismByImages** in the index to see for which groups this function is overlaid.

7.114 Set Functions for Groups

As already mentioned in the introduction of the chapter, groups are domains. Thus all set theoretic functions, for example `Intersection` and `Size` can be applied to groups. This and the following sections give further comments on the definition and implementations of those functions for groups. All set theoretic functions not mentioned here not treated specially for groups. The last section describes the format of the records that describe groups (see 7.118).

`Elements(` G `)`

The elements of a group G are constructed using a Dimino algorithm. See 7.115.

`IsSubset(` G`,` H `)`

If G and H are groups then `IsSubset` tests whether the generators of H are elements of G. Otherwise `DomainOps.IsSubset` is used.

`Intersection(` G`,` H `)`

The intersection of groups G and H is computed using an orbit algorithm. See 7.116.

7.115 Elements for Groups

`GroupOps.Elements(` G `)`

`GroupOps.Elements` returns the sets of elements of G (see 4.6). The function starts with the trivial subgroup of G, for which the set of elements is known and constructs the successive closures with the generators of G using `GroupOps.Closure` (see 7.18).

Note that this function neither checks nor sets the record component G`.elements`. It recomputes the set of elements even it is bound to G`.elements`.

7.116 Intersection for Groups

`GroupOps.Intersection(` *G*, *H* `)`

`GroupOps.Intersection` returns the intersection of G and H either as set of elements or as a group record (see 4.12).

If one argument, say G, is a set and the other a group, say H, then `GroupOps.Intersection` returns the subset of elements of G which lie in H.

If G and H have different parent groups then `GroupOps.Intersection` uses the function `DomainOps.Intersection` in order to compute the intersection.

Otherwise `GroupOps.Intersection` computes the stabilizer of the trivial coset of the bigger group in the smaller group using `Stabilizer` and `Coset`.

7.117 Operations for Groups

G `^` *s*

The operator `^` evaluates to the subgroup conjugate to G under a group element s of the parent group of G. See 7.20.

```
gap> s4 := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> s4.name := "s4";;
gap> v4 := Subgroup( s4, [ (1,2), (1,2)(3,4) ] );
Subgroup( s4, [ (1,2), (1,2)(3,4) ] )
gap> v4 ^ (2,3);
Subgroup( s4, [ (1,3), (1,3)(2,4) ] )
gap> v4 ^ (2,5);
Error, <g> must be an element of the parent group of <G>
```

s `in` *G*

The operator `in` evaluates to `true` if s is an element of G and `false` otherwise. s must be an element of the parent group of G.

```
gap> (1,2,3,4) in v4;
false
gap> (2,4) in v4^(2,3);
true
```

G `*` *s*

The operator `*` evaluates to the right coset of G with representative s. s must be an element of the parent group of G. See 7.86 for details about right cosets.

s `*` *G*

The operator `*` evaluates to the left coset of G with representative s. s must be an element of the parent group of G. See 7.91 for details about left cosets.

```
gap> v4 * (1,2,3,4);
(Subgroup( s4, [ (1,2), (1,2)(3,4) ] )*(1,2,3))
gap> (1,2,3,4) * v4;
((1,2,3,4)*Subgroup( s4, [ (1,2), (1,2)(3,4) ] ))
```

G / *N*

The operator / evaluates to the factor group G/N where N must be a normal subgroup of G. This is the same as `FactorGroup(`G`,`N`)` (see 7.33).

7.118 Group Records

As for all domains (see 4 and 4.1) groups and their subgroups are represented by records that contain important information about groups. Most of the following functions return such records. Of course it is possible to create a group record by hand but generally `Group` (see 7.9) and `Subgroup` (see 7.12) should be used for such tasks.

Once a group record is created you may add record components to it but you must not alter informations already present, especially not `generators` and `identity`.

Group records must always contain the components `generators`, `identity`, `isDomain` and `isGroup`. Subgroups contain an additional component `parent`. The contents of all components of a group G are described below.

The following two components are the so-called **category components** used to identify the category this domain belongs to.

`isDomain`
: is always `true` as a group is a domain.

`isGroup`
: is of course `true` as G is a group.

The following three components determine a group domain. These are the so-called **identification components**.

`generators`
: is a list group generators. Duplicate generators are allowed but none of the generators may be the group identity. The group G is the trivial group if and only if `generators` is the empty list. Note that once created this entry must never be changed, as most of the other entries depend on `generators`.

`identity`
: is the group identity of G.

`parent`
: if present this contains the group record of the parent group of a subgroup G, otherwise G itself is a parent group.

The following components are optional and contain **knowledge** about the group G.

`abelianInvariants`
: a list of integers containing the abelian invariants of an abelian group G.

`centralizer`
: contains the centralizer of G in its parent group.

`centre`
: contains the centre of G. See 7.17.

`commutatorFactorGroup`
: contains the commutator factor group of G. See 7.35 for details.

`conjugacyClasses`
: contains a list of the conjugacy classes of G. See 7.68 for details.

`core`
: contains the core of G under the action of its parent group. See 7.21 for details.

`derivedSubgroup`
: contains the derived subgroup of G. See 7.22.

`elements`
: is the set of all elements of G. See 4.6.

`fittingSubgroup`
: contains the Fitting subgroup of G. See 7.23.

`frattiniSubgroup`
: contains the Frattini subgroup of G. See 7.24.

`index`
: contains the index of G in its parent group. See 7.51.

`lowerCentralSeries`
: contains the lower central series of G as list of subgroups. See 7.41.

`normalizer`
: contains the normalizer of G in its parent group. See 7.27 for details.

`normalClosure`
: contains the normal closure of G in its parent group. See 7.25 for details.

`upperCentralSeries`
: contains the upper central series of G as list of subgroups. See 7.44.

`subnormalSeries`
: contains a subnormal series from the parent of G down to G. See 7.43 for details.

`sylowSubgroups`
: contains a list of Sylow subgroups of G. See 7.31 for details.

`size`
: is either an integer containing the size of a finite group or the string "infinity" if the group is infinite. See 4.10.

`perfectSubgroups`
: contains the a list of subgroups which includes at least one representative of each class of conjugate proper perfect subgroups of G. See 7.75.

`lattice`
: contains the subgroup lattice of G. See 7.75.

`conjugacyClassesSubgroups`
: identical to the list G`.lattice.classes`, contains the conjugacy classes of subgroups of G. See 7.74.

`tableOfMarks`
: contains the table of narks of G. See 48.4.

The following components are `true` if the group G has the property, `false` if not, and are not present if it is unknown whether the group has the property or not.

`isAbelian`
: is `true` if the group G is abelian. See 7.52.

`isCentral`
: is `true` if the group G is central in its parent group. See 7.53.

`isCyclic`
: is `true` if the group G is cyclic. See 7.55.

`isElementaryAbelian`
: is `true` if the group G is elementary abelian. See 7.56.

`isFinite`
: is `true` if the group G is finite. If you know that a group for which you want to use the generic low level group functions is infinite, you should set this component to `false`. This will avoid attempts to compute the set of elements.

`isNilpotent`
: is `true` if the group G is nilpotent. See 7.57.

`isNormal`
: is `true` if the group G is normal in its parent group. See 7.58.

`isPerfect`
: is `true` if the group G is perfect. See 7.59.

`isSimple`
: is `true` if the group G is simple. See 7.60.

`isSolvable`
: is `true` if the group G is solvable. See 7.61.

`isSubnormal`
: is `true` if the group G is subnormal in its parent group. See 7.63.

The component `operations` contains the **operations record** (see 4.1 and 4.2).

Chapter 8

Operations of Groups

One of the most important tools in group theory is the **operation** or **action** of a group on a certain set.

We say that a group G operates on a set D if we have a function that takes each $d \in D$ and each $g \in G$ to another element $d^g \in D$, which we call the image of d under g, such that $d^{identity} = d$ and $(d^g)^h = d^{gh}$ for each $d \in D$ and $g, h \in G$.

This is equivalent to saying that an operation is a homomorphism of the group G into the full symmetric group on D. We usually call D the **domain** of the operation and its elements **points**.

An example of the usage of the functions in this package can be found in the introduction to GAP3 (see 1.19).

In GAP3 group elements usually operate through the power operator, which is denoted by the caret `^`. It is possible however to specify other operations (see 8.1).

First this chapter describes the functions that take a single element of the group and compute cycles of this group element and related information (see 8.2, 8.3, 8.4, and 8.5), and the function that describes how a group element operates by a permutation that operates the same way on `[1..`*n*`]` (see 8.8).

Next come the functions that test whether an orbit has minimal or maximal length and related functions (see 8.9, 8.10, 8.11, 8.12, and 8.13).

Next this chapter describes the functions that take a group and compute orbits of this group and related information (see 8.16, 8.17, 8.18, and 8.19).

Next are the functions that compute the permutation group P that operates on `[ 1 .. Length(`*D*`) ]` in the same way that G operates on D, and the corresponding homomorphism from G to P (see 8.20, 8.21).

Next is the functions that compute block systems, i.e., partitions of D such that G operates on the sets of the partition (see 8.22), and the function that tests whether D has such a nontrivial partitioning under the operation of G (see 8.23).

Finally come the functions that relate an orbit of G on D with the subgroup of G that fixes the first point in the orbit (see 8.24), and the cosets of this subgroup in G (see 8.25 and 8.26).

All functions described in this chapter are in `LIBNAME/"operatio.g"`.

8.1 Other Operations

The functions in the operation package generally compute with the operation of group elements defined by the canonical operation that is denoted with the caret (^) in GAP3. However they also allow you to specify other operations. Such operations are specified by functions, which are accepted as optional argument by all the operations package functions.

This function must accept two arguments. The first argument will be the point and the second will be the group element. The function must return the image of the point under the group element.

As an example, the function `OnPairs` that specifies the operation on pairs could be defined as follows

```
OnPairs := function ( pair, g )
    return [ pair[1] ^ g, pair[2] ^ g ];
end;
```

The following operations are predefined.

`OnPoints`
: specifies the canonical default operation. Passing this function is equivalent to specifying no operation. This function exists because there are places where the operation in not an option.

`OnPairs`
: specifies the componentwise operation of group elements on pairs of points, which are represented by lists of length 2.

`OnTuples`
: specifies the componentwise operation of group elements on tuples of points, which are represented by lists. `OnPairs` is the special case of `OnTuples` for tuples with two elements.

`OnSets`
: specifies the operation of group elements on sets of points, which are represented by sorted lists of points without duplicates (see 28).

`OnRight`
: specifies that group elements operate by multiplication from the right.

`OnLeftInverse`
: specifies that group elements operate by multiplication by their inverses from the left. This is an operation, unlike `OnLeftAntiOperation` (see below).

`OnRightCosets`
: specifies that group elements operate by multiplication from the right on sets of points, which are represented by sorted lists of points without duplicates (see 28).

`OnLeftCosets`
: specifies that group elements operate by multiplication from the left on sets of points, which are represented by sorted lists of points without duplicates (see 28).

`OnLines`
: specifies that group elements, which must be matrices, operate on lines, which are represented by vectors with first nonzero coefficient one. That is, `OnLines` multiplies

the vector by the group element and then divides the vector by the first nonzero coefficient.

Note that it is your responsibility to make sure that the elements of the domain D on which you are operating are already in normal form. The reason is that all functions will compare points using the `=` operation. For example, if you are operating on sets with `OnSets`, you will get an error message it not all elements of the domain are sets.

```
gap> Cycle( (1,2), [2,1], OnSets );
Error, OnSets: <tuple> must be a set
```

The former function `OnLeft` which operated by mulitplication from the left has been renamed `OnLeftAntiOperation`, to emphasise the point that it does not satisify the axioms of an operation, and may cause errors if supplied where an operation is expected.

8.2 Cycle

`Cycle(` *g*`,` *d* `)`
`Cycle(` *g*`,` *d*`,` *operation* `)`

`Cycle` returns the orbit of the point d, which may be an object of arbitrary type, under the group element g as a list of points.

The points e in the cycle of d under the group element g are those for which a power g^i exists such that $d^{g^i} = e$.

The first point in the list returned by `Cycle` is the point d itself, the ordering of the other points is such that each point is the image of the previous point.

`Cycle` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the element g operates (see 8.1).

```
gap> Cycle( (1,5,3,8)(4,6,7), 3 );
[ 3, 8, 1, 5 ]
gap> Cycle( (1,5,3,8)(4,6,7), [3,4], OnPairs );
[ [ 3, 4 ], [ 8, 6 ], [ 1, 7 ], [ 5, 4 ], [ 3, 6 ], [ 8, 7 ],
  [ 1, 4 ], [ 5, 6 ], [ 3, 7 ], [ 8, 4 ], [ 1, 6 ], [ 5, 7 ] ]
```

`Cycle` calls
`Domain([`*g*`]).operations.Cycle(` *g*`,` *d*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for the functions called this way.

The default function called this way is `GroupElementsOps.Cycle`, which starts with d and applies g to the last point repeatedly until d is reached again. Special categories of group elements overlay this default function with more efficient functions.

8.3 CycleLength

`CycleLength(` *g*`,` *d* `)`
`CycleLength(` *g*`,` *d*`,` *operation* `)`

`CycleLength` returns the length of the orbit of the point d, which may be an object of arbitrary type, under the group elements g. See 8.2 for the definition of cycles.

`CycleLength` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the group element g operates (see 8.1).

```
gap> CycleLength( (1,5,3,8)(4,6,7), 3 );
4
gap> CycleLength( (1,5,3,8)(4,6,7), [3,4], OnPairs );
12
```

`CycleLength` calls
`Domain([`g`]).operations.CycleLength(` g, d, *operation* `)`
and returns the value. Note that the third argument is not optional for the functions called this way.

The default function called this way is `GroupElementsOps.CycleLength`, which starts with d and applies g to the last point repeatedly until d is reached again. Special categories of group elements overlay this default function with more efficient functions.

8.4 Cycles

`Cycles(` g, D `)`
`Cycles(` g, D, *operation* `)`

`Cycles` returns the set of cycles of the group element g on the domain D, which must be a list of points of arbitrary type, as a set of lists of points. See 8.2 for the definition of cycles.

It is allowed that D is a proper subset of a domain, i.e., that D is not invariant under the operation of g. In this case D is silently replaced by the smallest superset of D which is invariant.

The first point in each cycle is the smallest point of D in this cycle. The ordering of the other points is such that each point is the image of the previous point. If D is invariant under g, then because `Cycles` returns a set of cycles, i.e., a sorted list, and because cycles are compared lexicographically, and because the first point in each cycle is the smallest point in that cycle, the list returned by `Cycles` is in fact sorted with respect to the smallest point in the cycles.

`Cycles` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the element g operates (see 8.1).

```
gap> Cycles( (1,5,3,8)(4,6,7), [3,5,7] );
[ [ 3, 8, 1, 5 ], [ 7, 4, 6 ] ]
gap> Cycles( (1,5,3,8)(4,6,7), [[1,3],[4,6]], OnPairs );
[ [ [ 1, 3 ], [ 5, 8 ], [ 3, 1 ], [ 8, 5 ] ],
  [ [ 4, 6 ], [ 6, 7 ], [ 7, 4 ] ] ]
```

`Cycles` calls
`Domain([`g`]).operations.Cycles(` g, D, *operation* `)`
and returns the value. Note that the third argument is not optional for the functions called this way.

The default function called this way is `GroupElementsOps.Cycles`, which takes elements from D, computes their orbit, removes all points in the orbit from D, and repeats this until D has been emptied. Special categories of group elements overlay this default function with more efficient functions.

8.5 CycleLengths

`CycleLengths(` *g*, *D* `)`
`CycleLengths(` *g*, *D*, *operation* `)`

`CycleLengths` returns a list of the lengths of the cycles of the group element g on the domain D, which must be a list of points of arbitrary type. See 8.2 for the definition of cycles.

It is allowed that D is a proper subset of a domain, i.e., that D is not invariant under the operation of g. In this case D is silently replaced by the smallest superset of D which is invariant.

The ordering of the lengths of cycles in the list returned by `CycleLengths` corresponds to the list of cycles returned by `Cycles`, which is ordered with respect to the smallest point in each cycle.

`CycleLengths` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the element g operates (see 8.1).

```
gap> CycleLengths( (1,5,3,8)(4,6,7), [3,5,7] );
[ 4, 3 ]
gap> CycleLengths( (1,5,3,8)(4,6,7), [[1,3],[4,6]], OnPairs );
[ 4, 3 ]
```

`CycleLengths` calls
`Domain([`*g*`]).operations.CycleLengths(` *g*, *D*, *operation* `)`
and returns the value. Note that the third argument is not optional for the functions called this way.

The default function called this way is `GroupElementsOps.CycleLengths`, which takes elements from D, computes their orbit, removes all points in the orbit from D, and repeats this until D has been emptied. Special categories of group elements overlay this default function with more efficient functions.

8.6 MovedPoints

`MovedPoints(` *g* `)`

```
gap> MovedPoints( (1,7)(2,3,8) );
[ 1, 2, 3, 7, 8 ]
```

8.7 NrMovedPoints

`NrMovedPoints(` *p* `)`

`NrMovedPoints` returns the number of points moved by the permutation g, the group element g, or the group g.

```
gap> NrMovedPoints( (1,7)(2,3,8) );
 5
```

8.8 Permutation

`Permutation(` *g*`,` *D* `)`
`Permutation(` *g*`,` *D*`,` *operation* `)`

`Permutation` returns a permutation that operates on the points `[1..Length(D)]` in the same way that the group element *g* operates on the domain *D*, which may be a list of arbitrary type.

It is not allowed that *D* is a proper subset of a domain, i.e., *D* must be invariant under the element *g*.

`Permutation` accepts a function *operation* of two arguments *d* and *g* as optional third argument, which specifies how the element *g* operates (see 8.1).

```
gap> Permutation( (1,5,3,8)(4,6,7), [4,7,6] );
(1,3,2)
gap> D := [ [1,4], [1,6], [1,7], [3,4], [3,6], [3,7],
>           [4,5], [5,6], [5,7], [4,8], [6,8], [7,8] ];;
gap> Permutation( (1,5,3,8)(4,6,7), D, OnSets );
( 1, 8, 6,10, 2, 9, 4,11, 3, 7, 5,12)
```

`Permutation` calls
`Domain([`*g*`]).operations.Permutation(` *g*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for the functions called this way.

The default function called this way is `GroupElementsOps.Permutation`, which simply applies *g* to all the points of *D*, finds the position of the image in *D*, and finally applies `PermList` (see 20.9) to the list of those positions. Actually this is not quite true. Because finding the position of an image in a sorted list is so much faster than finding it in *D*, `GroupElementsOps.Permutation` first sorts a copy of *D* and remembers how it had to rearrange the elements of *D* to achieve this. Special categories of group elements overlay this default function with more efficient functions.

8.9 IsFixpoint

`IsFixpoint(` *G*`,` *d* `)`
`IsFixpoint(` *G*`,` *d*`,` *operation* `)`

`IsFixpoint` returns `true` if the point *d* is a fixpoint under the operation of the group *G*.

We say that *d* is a **fixpoint** under the operation of *G* if every element *g* of *G* maps *d* to itself. This is equivalent to saying that each generator of *G* maps *d* to itself.

As a special case it is allowed that the first argument is a single group element, though this does not make a lot of sense, since in this case `IsFixpoint` simply has to test *d*`^`*g* `=` *d*.

`IsFixpoint` accepts a function *operation* of two arguments *d* and *g* as optional third argument, which specifies how the elements of *G* operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> IsFixpoint( g, 1 );
false
gap> IsFixpoint( g, [6,7,8], OnSets );
```

```
true
```

`IsFixpoint` is so simple that it does all the work by itself, and, unlike the other functions described in this chapter, does not dispatch to another function.

8.10 IsFixpointFree

`IsFixpointFree(` G, D `)`
`IsFixpointFree(` G, D, *operation* `)`

`IsFixpointFree` returns `true` if the group G operates without a fixpoint (see 8.9) on the domain D, which must be a list of points of arbitrary type.

We say that G operates **fixpoint free** on the domain D if each point of D is moved by at least one element of G. This is equivalent to saying that each point of D is moved by at least one generator of G. This definition also applies in the case that D is a proper subset of a domain, i.e., that D is not invariant under the operation of G.

As a special case it is allowed that the first argument is a single group element.

`IsFixpointFree` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> IsFixpointFree( g, [1..8] );
true
gap> sets := Combinations( [1..8], 3 );;  Length( sets );
56    # a list of all three element subsets of [1..8]
gap> IsFixpointFree( g, sets, OnSets );
false
```

`IsFixpointFree` calls
G`.operations.IsFixpointFree(` G, D, *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.IsFixpointFree`, which simply loops over the elements of D and applies to each all generators of G, and tests whether each is moved by at least one generator. This function is seldom overlaid, because it is very difficult to improve it.

8.11 DegreeOperation

`DegreeOperation(` G, D `)`
`DegreeOperation(` G, D, *operation* `)`

`DegreeOperation` returns the degree of the operation of the group G on the domain D, which must be a list of points of arbitrary type.

The **degree** of the operation of G on D is defined as the number of points of D that are properly moved by at least one element of G. This definition also applies in the case that D is a proper subset of a domain, i.e., that D is not invariant under the operation of G.

`DegreeOperation` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> DegreeOperation( g, [1..10] );
8
gap> sets := Combinations( [1..8], 3 );;  Length( sets );
56    # a list of all three element subsets of [1..8]
gap> DegreeOperation( g, sets, OnSets );
55
```

`DegreeOperation` calls
G`.operations.DegreeOperation(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.DegreeOperation`, which simply loops over the elements of D and applies to each all generators of G, and counts those that are moved by at least one generator. This function is seldom overlaid, because it is very difficult to improve it.

8.12 IsTransitive

`IsTransitive(` *G*`,` *D* `)`
`IsTransitive(` *G*`,` *D*`,` *operation* `)`

`IsTransitive` returns `true` if the group G operates transitively on the domain D, which must be a list of points of arbitrary type.

We say that a group G acts **transitively** on a domain D if and only if for every pair of points d and e there is an element g of G such that $d^g = e$. An alternative characterization of this property is to say that D as a set is equal to the orbit of every single point.

It is allowed that D is a proper subset of a domain, i.e., that D is not invariant under the operation of G. In this case `IsTransitive` checks whether for every pair of points d, e of D there is an element g of G, such that $d^g = e$. This can also be characterized by saying that D is a subset of the orbit of every single point.

`IsTransitive` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> IsTransitive( g, [1..8] );
false
gap> IsTransitive( g, [1,6] );
false    # note that the domain need not be invariant
gap> sets := Combinations( [1..5], 3 );;  Length( sets );
10    # a list of all three element subsets of [1..5]
gap> IsTransitive( g, sets, OnSets );
true
```

`IsTransitive` calls
G`.operations.IsTransitive(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.IsTransitive`, which tests whether D is a subset of the orbit of the first point in D. This function is seldom overlaid, because it is difficult to improve it.

8.13 Transitivity

`Transitivity(` G`,` D `)`
`Transitivity(` G`,` D`,` *operation* `)`

`Transitivity` returns the degree of transitivity of the group G on the domain D, which must be a list of points of arbitrary type. If G does not operate transitively on D then `Transitivity` returns 0.

The **degree of transitivity** of the operation of G on D is the largest k such that G operates k-fold transitively on D. We say that G operates k-**fold transitively** on D if it operates transitively on D (see 8.12) and the stabilizer of one point d of D operates k`-1`-fold transitively on `Difference(`D`,[`d`])`. Because the stabilizers of the points of D are conjugate this is equivalent to saying that the stabilizer of **each** point d of D operates k`-1`-fold transitively on `Difference(`D`,[`d`])`.

It is not allowed that D is a proper subset of a domain, i.e., D must be invariant under the operation of G.

`Transitivity` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> Transitivity( g, [1..8] );
0
gap> Transitivity( g, [1..5] );
3
gap> sets := Combinations( [1..5], 3 );;  Length( sets );
10    # a list of all three element subsets of [1..5]
gap> Transitivity( g, sets, OnSets );
1
```

`Transitivity` calls
G`.operations.Transitivity(` G`,` D`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.Transitivity`, which first tests whether G operates transitively on D. If so, it returns
`Transitivity(Stabilizer(`G`,Difference(`D`,[`D`[1]]),`*operation*`)+1`;
if not, it simply returns 0. Special categories of groups overlay this default function with more efficient functions.

8.14 IsRegular

`IsRegular(` G`,` D `)` `IsRegular(` G`,` D`,` *operation* `)`

`IsRegular` returns `true` if the group G operates regularly on the domain D, which must be a list of points of arbitrary type, and `false` otherwise.

A group G operates **regularly** on a domain D if it operates transitively and no element of G other than the idenity leaves a point of D fixed. An equal characterisation is that G operates transitively on D and the stabilizer of any point of D is trivial. Yet another characterisation is that the operation of G on D is equivalent to the operation of G on its elements by multiplication from the right.

It is not allowed that D is a proper subset of a domain, i.e., D must be invariant under the operation of G.

`IsRegular` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> IsRegular( g, [1..5] );
false
gap> IsRegular( g, Elements(g), OnRight );
true
gap> g := Group( (1,2,3), (3,4,5) );;
gap> IsRegular( g, Orbit( g, [1,2,3], OnTuples ), OnTuples );
true
```

`IsRegular` calls
G`.operations.IsRegular(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.IsRegular`, which tests if G operates transitively and semiregularly on D (see 8.12 and 8.15).

8.15 IsSemiRegular

`IsSemiRegular(` *G*`,` *D* `)`
`IsSemiRegular(` *G*`,` *D*`,` *operation* `)`

`IsSemiRegular` returns `true` if the group G operates semiregularly on the domain D, which must be a list of points of arbitrary type, and `false` otherwise.

A group G operates **semiregularly** on a domain D if no element of G other than the idenity leaves a point of D fixed. An equal characterisation is that the stabilizer of any point of D is trivial. Yet another characterisation is that the operation of G on D is equivalent to multiple copies of the operation of G on its elements by multiplication from the right.

It is not allowed that D is a proper subset of a domain, i.e., D must be invariant under the operation of G.

`IsSemiRegular` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2)(3,4)(5,7)(6,8), (1,3)(2,4)(5,6)(7,8) );;
gap> IsSemiRegular( g, [1..8] );
true
gap> g := Group( (1,2)(3,4)(5,7)(6,8), (1,3)(2,4)(5,6,7,8) );;
gap> IsSemiRegular( g, [1..8] );
false
```

```
gap> IsSemiRegular( g, Orbit( g, [1,5], OnSets ), OnSets );
true
```

`IsSemiRegular` calls
G`.operations.IsSemiRegular(` G, D, *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.IsSemiRegular`, which computes a permutation group P that operates on `[1..Length(`D`)]` in the same way that G operates on D (see 8.20) and then checks if this permutation group operations semiregularly. This of course only works because this default function is overlaid for permutation groups (see 21.22).

8.16 Orbit

`Orbit(` G, d `)`
`Orbit(` G, d, *operation* `)`

`Orbit` returns the orbit of the point d, which may be an object of arbitrary type, under the group G as a list of points.

The points e in the orbit of d under the group G are those points for which a group element g of G exists such that $d^g = e$.

Suppose G has n generators. First we order the words of the free monoid with n abstract generators according to length and for words with equal length lexicographically. So if G has two generators called a and b the ordering is $identity, a, b, a^2, ab, ba, b^2, a^3, ...$ Next we order the elements of G that can be written as a product of the generators, i.e., without inverses of the generators, according to the first occurrence of a word representing the element in the above ordering. Then the ordering of points in the orbit returned by `Orbit` is according to the order of the first representative of each point e, i.e., the smallest g such that $d^g = e$. Note that because the orbit is finite there is for every point in the orbit at least one representative that can be written as a product in the generators of G.

`Orbit` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> Orbit( g, 1 );
[ 1, 2, 3, 4, 5 ]
gap> Orbit( g, 2 );
[ 2, 3, 1, 4, 5 ]
gap> Orbit( g, [1,6], OnPairs );
[ [ 1, 6 ], [ 2, 7 ], [ 3, 6 ], [ 2, 8 ], [ 1, 7 ], [ 4, 6 ],
  [ 3, 8 ], [ 2, 6 ], [ 1, 8 ], [ 4, 7 ], [ 5, 6 ], [ 3, 7 ],
  [ 5, 8 ], [ 5, 7 ], [ 4, 8 ] ]
```

`Orbit` calls
G`.operations.Orbit(` G, d, *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.Orbit`, which performs an ordinary orbit algorithm. Special categories of groups overlay this default function with more efficient functions.

8.17 OrbitLength

`OrbitLength(` G`,` d `)`
`OrbitLength(` G`,` d`,` *operation* `)`

`OrbitLength` returns the length of the orbit of the point d, which may be an object of arbitrary type, under the group G. See 8.16 for the definition of orbits.

`OrbitLength` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> OrbitLength( g, 1 );
5
gap> OrbitLength( g, 10 );
1
gap> OrbitLength( g, [1,6], OnPairs );
15
```

`OrbitLength` calls
G`.operations.OrbitLength(` G`,` d`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.OrbitLength`, which performs an ordinary orbit algorithm. Special categories of groups overlay this default function with more efficient functions.

8.18 Orbits

`Orbits(` G`,` D `)`
`Orbits(` G`,` D`,` *operation* `)`

`Orbits` returns the orbits of the group G on the domain D, which must be a list of points of arbitrary type, as a set of lists of points. See 8.16 for the definition of orbits.

It is allowed that D is a proper subset of a domain, i.e., that D is not invariant under the operation of G. In this case D is silently replaced by the smallest superset of D which is invariant.

The first point in each orbit is the smallest point, the other points of each orbit are ordered in the standard order defined for orbits (see 8.16). Because `Orbits` returns a set of orbits, i.e., a sorted list, and because those orbits are compared lexicographically, and because the first point in each orbit is the smallest point in that orbit, the list returned by `Orbits` is in fact sorted with respect to the smallest points the orbits.

`Orbits` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
```

```
gap> Orbits( g, [1..8] );
[ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8 ] ]
gap> Orbits( g, [1,6] );
[ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8 ] ]     # the domain is not invariant
gap> sets := Combinations( [1..8], 3 );; Length( sets );
56    # a list of all three element subsets of [1..8]
gap> Orbits( g, sets, OnSets );
[ [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 2, 3, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ],
      [ 2, 4, 5 ], [ 2, 3, 5 ], [ 1, 4, 5 ], [ 3, 4, 5 ], [ 1, 3, 5 ]
     ],
  [ [ 1, 2, 6 ], [ 2, 3, 7 ], [ 1, 3, 6 ], [ 2, 4, 8 ], [ 1, 2, 7 ],
      [ 1, 4, 6 ], [ 3, 4, 8 ], [ 2, 5, 7 ], [ 2, 3, 6 ],
      [ 1, 2, 8 ], [ 2, 4, 7 ], [ 1, 5, 6 ], [ 1, 4, 8 ],
      [ 4, 5, 7 ], [ 3, 5, 6 ], [ 2, 3, 8 ], [ 1, 3, 7 ],
      [ 2, 4, 6 ], [ 3, 4, 6 ], [ 2, 5, 8 ], [ 1, 5, 7 ],
      [ 4, 5, 6 ], [ 3, 5, 8 ], [ 1, 3, 8 ], [ 3, 4, 7 ],
      [ 2, 5, 6 ], [ 1, 4, 7 ], [ 1, 5, 8 ], [ 4, 5, 8 ], [ 3, 5, 7 ]
     ],
  [ [ 1, 6, 7 ], [ 2, 6, 7 ], [ 1, 6, 8 ], [ 3, 6, 7 ], [ 2, 6, 8 ],
      [ 2, 7, 8 ], [ 4, 6, 8 ], [ 3, 7, 8 ], [ 3, 6, 8 ],
      [ 4, 7, 8 ], [ 5, 6, 7 ], [ 1, 7, 8 ], [ 4, 6, 7 ],
      [ 5, 7, 8 ], [ 5, 6, 8 ] ], [ [ 6, 7, 8 ] ] ]
```

`Orbits` calls
G`.operations.Orbits(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.Orbits`, which takes an element from *D*, computes its orbit, removes all points in the orbit from *D*, and repeats this until *D* has been emptied. Special categories of groups overlay this default function with more efficient functions.

8.19 OrbitLengths

`OrbitLengths(` *G*`,` *D* `)`
`OrbitLengths(` *G*`,` *D*`,` *operation* `)`

`OrbitLengths` returns a list of the lengths of the orbits of the group *G* on the domain *D*, which may be a list of points of arbitrary type. See 8.16 for the definition of orbits.

It is allowed that *D* is proper subset of a domain, i.e., that *D* is not invariant under the operation of *G*. In this case *D* is silently replaced by the smallest superset of *D* which is invariant.

The ordering of the lengths of orbits in the list returned by `OrbitLengths` corresponds to the list of cycles returned by `Orbits`, which is ordered with respect to the smallest point in each orbit.

`OrbitLengths` accepts a function *operation* of two arguments *d* and *g* as optional third argument, which specifies how the elements of *G* operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
```

```
gap> OrbitLengths( g, [1..8] );
[ 5, 3 ]
gap> sets := Combinations( [1..8], 3 );; Length( sets );
56    # a list of all three element subsets of [1..8]
gap> OrbitLengths( g, sets, OnSets );
[ 10, 30, 15, 1 ]
```

`OrbitLengths` calls
G`.operations.OrbitLenghts(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.OrbitLengths`, which takes an element from D, computes its orbit, removes all points in the orbit from D, and repeats this until D has been emptied. Special categories of groups overlay this default function with more efficient functions.

8.20 Operation

`Operation(` *G*`,` *D* `)`
`Operation(` *G*`,` *D*`,` *operation* `)`

`Operation` returns a permutation group with the same number of generators as G, such that each generator of the permutation group operates on the set `[1..Length(D)]` in the same way that the corresponding generator of the group G operates on the domain D, which may be a list of arbitrary type.

It is not allowed that D is a proper subset of a domain, i.e., D must be invariant under the element g.

`Operation` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

The function `OperationHomomorphism` (see 8.21) can be used to compute the homomorphism that maps G onto the new permutation group. Of course if you are only interested in mapping single elements of G into the new permutation group you may also use `Permutation` (see 8.8).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> Operation( g, [1..5] );
Group( (1,2,3), (3,4,5) )
gap> Operation( g, Orbit( g, [1,6], OnPairs ), OnPairs );
Group( ( 1, 2, 3, 5, 8,12)( 4, 7, 9)( 6,10)(11,14), ( 2, 4)( 3, 6,11)
( 5, 9)( 7,10,13,12,15,14) )
```

`Operation` calls
G`.operations.Operation(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.Operation`, which simply applies each generator of G to all the points of D, finds the position of the image in D, and finally applies `PermList` (see 20.9) to the list of those positions. Actually this is not quite true. Because finding the position on an image in a sorted list is so much faster than finding it in D, `GroupElementsOps.Operation` first sorts a copy of D and remembers how it had to rearrange the elements of D to achieve this. Special categories of groups overlay this default function with more efficient functions.

8.21 OperationHomomorphism

`OperationHomomorphism(` G`,` P `)`

`OperationHomomorphism` returns the group homomorphism (see 7.106) from the group G to the permutation group P, which must be the result of a prior call to `Operation` (see 8.20) with G or a group of which G is a subgroup (see 7.62) as first argument.

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> h := Operation( g, [1..5] );
Group( (1,2,3), (3,4,5) )
gap> p := OperationHomomorphism( g, h );
OperationHomomorphism( Group( (1,2,3)(6,7), (3,4,5)(7,8) ), Group(
(1,2,3), (3,4,5) ) )
gap> (1,4,2,5,3)(6,7,8) ^ p;
(1,4,2,5,3)
gap> h := Operation( g, Orbit( g, [1,6], OnPairs ), OnPairs );
Group( ( 1, 2, 3, 5, 8,12)( 4, 7, 9)( 6,10)(11,14), ( 2, 4)( 3, 6,11)
( 5, 9)( 7,10,13,12,15,14) )
gap> p := OperationHomomorphism( g, h );;
gap> s := SylowSubgroup( g, 2 );
Subgroup( Group( (1,2,3)(6,7), (3,4,5)(7,8) ),
[ (7,8), (7,8), (2,5)(3,4), (2,3)(4,5) ] )
gap> Images( p, s );
Subgroup( Group( ( 1, 2, 3, 5, 8,12)( 4, 7, 9)( 6,10)(11,14), ( 2, 4)
( 3, 6,11)( 5, 9)( 7,10,13,12,15,14) ),
[ ( 2, 4)( 5, 9)( 7,12)(10,15)(13,14),
  ( 2, 4)( 5, 9)( 7,12)(10,15)(13,14),
  ( 2,14)( 3, 6)( 4,13)( 7,15)( 8,11)(10,12),
  ( 2,12)( 3, 8)( 4, 7)( 6,11)(10,14)(13,15) ] )
gap> OperationHomomorphism( g, Group( (1,2,3), (3,4,5) ) );
Error, Record: element 'operation' must have an assigned value
```

`OperationHomomorphism` calls
P`.operations.OperationHomomorphism(` G`,` P `)`
and returns the value.

The default function called this way is `GroupOps.OperationHomomorphism`, which uses the fields P`.operationGroup`, P`.operationDomain`, and P`.operationOperation` (the arguments to the `Operation` call that created P) to construct a generic homomorphism h. This homomorphism uses
`Permutation(`g`,`h`.range.operationDomain,`h`.range.operationOperation)`
to compute the image of an element g of G under h. It uses `Representative` to compute the preimages of an element p of P under h. And it computes the kernel by intersecting the cores (see 7.21) of the stabilizers (see 8.24) of representatives of the orbits of G. Look under **OperationHomomorphism** in the index to see for which groups and operations this function is overlaid.

8.22 Blocks

`Blocks(` *G*`,` *D*`,` *seed* `)`
`Blocks(` *G*`,` *D*`,` *seed*`,` *operation* `)`

In this form `Blocks` returns a block system of the domain D, which may be a list of points of arbitrary type, under the group G, such that the points in the list *seed* all lie in the same block. If no such nontrivial block system exists, `Blocks` returns `[` *D* `]`. G must operate transitively on D, otherwise an error is signalled.

`Blocks(` *G*`,` *D* `)`
`Blocks(` *G*`,` *D*`,` *operation* `)`

In this form `Blocks` returns a minimal block system of the domain D, which may be a list of points of arbitrary type, under the group G. If no nontrivial block system exists, `Blocks` returns `[` *D* `]`. G must operate transitively on D, otherwise an error is signalled.

A **block system** B is a list of blocks with the following properties. Each block b of B is a subset of D. The blocks are pairwise disjoint. The union of blocks is D. The image of each block under each element g of G is as a set equal to some block of the block system. Note that this implies that all blocks contain the same number of elements as G operates transitive on D. Put differently a block system B of D is a partition of D such that G operates with `OnSets` (see 8.1) on B. The block system that consists of only singleton sets and the block system consisting only of D are called **trivial**. A block system B is called **minimal** if there is no nontrivial block system whose blocks are all subsets of the blocks of B and whose number of blocks is larger than the number of blocks of B.

`Blocks` accepts a function *operation* of two arguments d and g as optional third, resp. fourth, argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> Blocks( g, [1..5] );
[ [ 1 .. 5 ] ]
gap> Blocks( g, Orbit( g, [1,2], OnPairs ), OnPairs );
[ [ [ 1, 2 ], [ 3, 2 ], [ 4, 2 ], [ 5, 2 ] ],
  [ [ 1, 3 ], [ 2, 3 ], [ 4, 3 ], [ 5, 3 ] ],
  [ [ 1, 4 ], [ 2, 4 ], [ 3, 4 ], [ 5, 4 ] ],
  [ [ 1, 5 ], [ 2, 5 ], [ 3, 5 ], [ 4, 5 ] ],
  [ [ 2, 1 ], [ 3, 1 ], [ 4, 1 ], [ 5, 1 ] ] ]
```

`Blocks` calls
G`.operations.Blocks(` *G*`,` *D*`,` *seed*`,` *operation* `)`
and returns the value. If no seed was given as argument to `Blocks` it passes the empty list. Note that the fourth argument is not optional for functions called this way.

The default function called this way is `GroupOps.Blocks`, which computes a permutation group P that operates on `[1..Length(`*D*`)]` in the same way that G operates on D (see 8.20) and leaves it to this permutation group to find the blocks. This of course works only because this default function is overlaid for permutation groups (see 21.22).

8.23 IsPrimitive

`IsPrimitive(` *G*`,` *D* `)`
`IsPrimitive(` *G*`,` *D*`,` *operation* `)`

`IsPrimitive` returns `true` if the group G operates primitively on the domain D, which may be a list of points of arbitrary type, and `false` otherwise.

A group G operates **primitively** on a domain D if and only if D operates transitively (see 8.12) and has only the trivial block systems (see 8.22).

`IsPrimitive` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> IsPrimitive( g, [1..5] );
true
gap> IsPrimitive( g, Orbit( g, [1,2], OnPairs ), OnPairs );
false
```

`IsPrimitive` calls
G`.operations.IsPrimitive(` *G*`,` *D*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.IsPrimitive`, which simply calls `Blocks(` *G*`,` *D*`,` *operation* `)` and tests whether the returned block system is `[` *D* `]`. This function is seldom overlaid, because all the important work is done in `Blocks`.

8.24 Stabilizer

`Stabilizer(` *G*`,` *d* `)`
`Stabilizer(` *G*`,` *d*`,` *operation* `)`

`Stabilizer` returns the stabilizer of the point d under the operation of the group G.

The **stabilizer** S of d in G is the subgroup of those elements g of G that fix d, i.e., for which $d^g = d$. The right cosets of S correspond in a canonical way to the points p in the orbit O of d under G; namely all elements from a right coset Sg map d to the same point $d^g \in O$, and elements from different right cosets Sg and Sh map d to different points d^g and d^h. Thus the index of the stabilizer S in G is equal to the length of the orbit O. `RepresentativesOperation` (see 8.26) computes a system of representatives of the right cosets of S in G.

`Stabilizer` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> g.name := "G";;
gap> Stabilizer( g, 1 );
Subgroup( G, [ (3,4,5)(7,8), (2,5,3)(6,7) ] )
gap> Stabilizer( g, [1,2,3], OnSets );
Subgroup( G, [ (7,8), (6,8), (2,3)(4,5)(6,7,8), (1,2)(4,5)(6,7,8) ] )
```

`Stabilizer` calls
G`.operations.Stabilizer(` *G*`,` *d*`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.Stabilizer`, which computes the orbit of d under G, remembers a representative r_e for each point e in the orbit, and uses Schreier's theorem, which says that the stabilizer is generated by the elements $r_e g r_{e^g}^{-1}$. Special categories of groups overlay this default function with more efficient functions.

8.25 RepresentativeOperation

`RepresentativeOperation(` G`,` d`,` e `)`
`RepresentativeOperation(` G`,` d`,` e`,` *operation* `)`

`RepresentativeOperation` returns a representative of the point e in the orbit of the point d under the group G. If $d = e$ then `RepresentativeOperation` returns G`.identity`, otherwise it is not specified which group element `RepresentativeOperation` will return if there are several that map d to e. If e is not in the orbit of d under G, `RepresentativeOperation` returns `false`.

An element g of G is called a **representative** for the point e in the orbit of d under G if g maps d to e, i.e., $d^g = e$. Note that the set of such representatives that map d to e forms a right coset of the stabilizer of d in G (see 8.24).

`RepresentativeOperation` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> RepresentativeOperation( g, 1, 5 );
(1,5,4,3,2)(6,8,7)
gap> RepresentativeOperation( g, 1, 6 );
false
gap> RepresentativeOperation( g, [1,2,3], [3,4,5], OnSets );
(1,3,5,2,4)
gap> RepresentativeOperation( g, [1,2,3,4], [3,4,5,2], OnTuples );
false
```

`RepresentativeOperation` calls
G`.operations.RepresentativeOperation(` G`,` d`,` e`,` *operation* `)`
and returns the value. Note that the fourth argument is not optional for functions called this way.

The default function called this way is `GroupOper.RepresentativeOperation`, which starts a normal orbit calculation to compute the orbit of d under G, and remembers for each point how it was obtained, i.e., which generator of G took which orbit point to this new point. When the point e appears this information can be traced back to write down the representative of e as a word in the generators. Special categories of groups overlay this default function with more efficient functions.

8.26 RepresentativesOperation

`RepresentativesOperation(` G`,` d `)`
`RepresentativesOperation(` G`,` d`,` *operation* `)`

`RepresentativesOperation` returns a list of representatives of the points in the orbit of the point d under the group G.

The ordering of the representatives corresponds to the ordering of the points in the orbit as returned by `Orbit` (see 8.16). Therefore `List( RepresentativesOperation(`G`,`d`), r->`d`^r ) = Orbit(`G`,`d`)`.

An element g of G is called a **representative** for the point e in the orbit of d under G if g maps d to e, i.e., $d^g = e$. Note that the set of such representatives that map d to e forms a right coset of the stabilizer of d in G (see 8.24). The set of all representatives of the orbit of d under G thus forms a system of representatives of the right cosets of the stabilizer of d in G.

`RepresentativesOperation` accepts a function *operation* of two arguments d and g as optional third argument, which specifies how the elements of G operate (see 8.1).

```
gap> g := Group( (1,2,3)(6,7), (3,4,5)(7,8) );;
gap> RepresentativesOperation( g, 1 );
[ (), (1,2,3)(6,7), (1,3,2), (1,4,5,3,2)(7,8), (1,5,4,3,2) ]
gap> Orbit( g, [1,2], OnSets );
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 2, 4 ], [ 1, 4 ], [ 3, 4 ],
  [ 2, 5 ], [ 1, 5 ], [ 4, 5 ], [ 3, 5 ] ]
gap> RepresentativesOperation( g, [1,2], OnSets );
[ (), (1,2,3)(6,7), (1,3,2), (1,2,4,5,3)(6,8,7), (1,4,5,3,2)(7,8),
  (1,3,2,4,5)(6,8), (1,2,5,4,3)(6,7), (1,5,4,3,2), (1,4,3,2,5)(6,7,8),
  (1,3,2,5,4) ]
```

`RepresentativesOperation` calls
G`.operations.RepresentativesOperation(` G`,` d`,` *operation* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `GroupOps.RepresentativesOperation`, which computes the orbit of d with the normal algorithm, but remembers for each point e in the orbit a representative r_e. When a generator g of G takes an old point e to a point f not yet in the orbit, the representative r_f for f is computed as $r_e g$. Special categories of groups overlay this default function with more efficient functions.

8.27 IsEquivalentOperation

`IsEquivalentOperation(` G`,` D`,` H`,` E `)`
`IsEquivalentOperation(` G`,` D`,` H`,` E`,` *operationH* `)`
`IsEquivalentOperation(` G`,` D`,` *operationG*`,` H`,` E `)`
`IsEquivalentOperation(` G`,` D`,` *operationG*`,` H`,` E`,` *operationH* `)`

`IsEquivalentOperation` returns `true` if G operates on D in like H operates on E, and `false` otherwise.

The operations of G on D and H on E are equivalent if they have the same number of generators and there is a permutation F of the elements of E such that for every generator g of G and the corresponding generator h of H we have $Position(D, D_i^g) = Position(F, F_i^h)$. Note that this assumes that the mapping defined by mapping $G.generators$ to $H.generators$ is a homomorphism (actually an isomorphism of factor groups of G and H represented by the respective operation).

`IsEquivalentOperation` accepts functions *operationG* and *operationH* of two arguments d and g as optional third and sixth arguments, which specify how the elements of G and H operate (see 8.1).

```
gap> g := Group( (1,2)(4,5), (1,2,3)(4,5,6) );;
gap> h := Group( (2,3)(4,5), (1,2,3)(4,5,6) );;
gap> IsEquivalentOperation( g, [1..6], h, [1..6] );
true
gap> h := Group( (1,2), (1,2,3) );;
gap> IsEquivalentOperation(g,[[1,4],[2,5],[3,6]],OnPairs,h,[1..3]);
true
gap> h := Group( (1,2), (1,2,3)(4,5,6) );;
gap> IsEquivalentOperation( g, [1..6], h, [1..6] );
false
gap> h := Group( (1,2,3)(4,5,6), (1,2)(4,5) );;
gap> IsEquivalentOperation( g, [1..6], h, [1..6] );
false    # the generators must correspond
```

`IsEquivalentOperation` calls
G`.operations.IsEquivalentOperation(`G`,`D`,`*oprG*`,`H`,`E`,`*oprH*`)` and returns the value. Note that the third and sixth argument are not optional for functions called this way.

The default function called this way is `GroupOps.IsEquivalentOperation`, which tries to rearrange E so that the above condition is satisfied. This is done one orbit of G at a time, and for each such orbit all the orbits of H of the same length are tried to see if there is one which can be rearranged as necessary. Special categories of groups overlay this function with more efficient ones.

Chapter 9

Vector Spaces

The material described in this chapter is subject to change.

Vector spaces form another important domain in GAP3. They may be given in any representation whenever the underlying set of elements forms a vector space in terms of linear algebra. Thus, for example, one may construct a vector space by defining generating matrices over a field or by using the base of a field extension as generators. More complex constructions may fake elements of a vector space by specifying records with appropriate operations. A special type of vector space, that is implemented in the GAP3 library, handles the case where the elements are lists over a field. This type is the so called `RowSpace` (see 33 for details).

General vector spaces are created using the function `VectorSpace` (see 9.1) and they are represented as records that contain all necessary information to deal with the vector space. The components listed in 9.3 are common for all vector spaces, but special types of vector spaces, such as the row spaces, may use additional entries to store specific data.

The following sections contain descriptions of functions and operations defined for vector spaces.

The next sections describe functions to compute a base (see 9.6) and the dimension (see 9.8) of a vector space over its field.

The next sections describe how to calculate linear combinations of the elements of a base (see 9.9) and how to find the coefficients of an element of a vector space when expressed as a linear combination in the current base (see 9.10).

The functions described in this chapter are implemented in the file `LIBNAME/"vecspace.g"`.

9.1 VectorSpace

`VectorSpace(` *generators* `,` *field* `)`

Let *generators* be a list of objects generating a vector space over the field *field*. Then `VectorSpace` returns this vector space represented as a GAP3 record.

```
gap> f := GF( 3^2 );
GF(3^2)
```

```
gap> m := [ [ f.one, f.one ], [ f.zero, f.zero ] ];
[ [ Z(3)^0, Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ]
gap> n := [ [ f.one, f.zero ], [ f.zero, f.one ] ];
[ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
gap> VectorSpace( [ m, n ], f );
VectorSpace( [ [ [ Z(3)^0, Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] ], GF(3^2) )
```

`VectorSpace(` *generators*, *field*, *zero* `)`

`VectorSpace` returns the vector space generated by *generators* over the field *field* having *zero* as the uniquely determined neutral element. This call of `VectorSpace` always is requested if *generators* is the empty list.

```
gap> VectorSpace( [], f, [ [ f.zero, f.zero ], [ f.zero, f.zero ] ] );
VectorSpace( [  ], GF(3^2), [ [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ]
 ] )
```

9.2 IsVectorSpace

`IsVectorSpace(` *obj* `)`

`IsVectorSpace` returns `true` if *obj*, which can be an object of arbitrary type, is a vector space and `false` otherwise.

9.3 Vector Space Records

A vector space is represented as a GAP3 record having several entries to hold some necessary information about the vector space.

Basically a vector space record is constructed using the function `VectorSpace` although one may create such a record by hand. Furthermore vector space records may be returned by functions described here or somewhere else in this manual.

Once a vector space record is created you are free to add components, but you should never alter existing entries, especially `generators`, `field` and `zero`.

The following list mentions all components that are requested for a vector space V.

`generators`
: a list of elements generating the vector space V.

`field`
: the field over which the vector space V is written.

`zero`
: the zero element of the vector space.

`isDomain`
: always `true`, because vector spaces are domains.

`isVectorSpace`
: always `true`, for obvious reasons.

There are as well some optional components for a vector space record.

`base`
: a base for V, given as a list of elements of V.

`dimension`
: the dimension of V which is the length of a base of V.

9.4 Set Functions for Vector Spaces

As mentioned before, vector spaces are domains. So all functions that exist for domains may also be applied to vector spaces. This and the following chapters give further information on the implementation of these functions for vector spaces, as far as they differ in their implementation from the general functions.

`Elements(` V `)`

The elements of a vector space V are computed by producing all linear combinations of the generators of V.

`Size(` V `)`

The size of a vector space V is determined by calculating the dimension of V and looking at the field over which it is written.

`IsFinite(` V `)`

A vector space in GAP3 is finite if it contains only its zero element or if the field over which it is written is finite. This characterisation is true here, as in GAP3 all vector spaces have a finite dimension.

`Intersection(` V`,` W `)`

The intersection of vector spaces is computed by finding a base for the intersection of the sets of their elements. One may consider the algorithm for finding a base of a vector space V as another way to write `Intersection(` V`,` V `)`.

9.5 IsSubspace

`IsSubspace(` V`,` W `)`

`IsSubspace` tests whether the vector space W is a subspace of V. It returns `true` if W lies in V and `false` if it does not.

The answer to the question is obtained by testing whether all the generators of W lie in V, so that, for the general case of vector space handling, a list of all the elements of V is constructed.

9.6 Base

`Base(` V `)`

`Base` computes a base of the given vector space V. The result is returned as a list of elements of the vector space V.

The base of a vector space is defined to be a minimal generating set. It can be shown that for a given vector space V each base has the same number of elements, which is called the dimension of V (see 9.8).

Unfortunately, no better algorithm is known to compute a base in general than to browse through the list of all elements of the vector space. So be careful when using this command on plain vector spaces.

```
gap> f := GF(3);
GF(3)
gap> m1 := [[ f.one, f.one, f.zero, f.zero ]];
[ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ]
gap> m2 := [[ f.one, f.one, f.one, f.zero ]];
[ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ]
gap> V := VectorSpace( [ m1, m2, m1+m2 ], GF(3) );
VectorSpace( [ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ],
  [ [ Z(3), Z(3), Z(3)^0, 0*Z(3) ] ] ], GF(3) )
gap> Base( V );
[ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ] ]
gap> Dimension( V );
2
```

9.7 AddBase

`AddBase(` *V* `,` *base* `)`

`AddBase` attaches a user-supplied base for the vector space V to the record that represents V.

Most of the functions for vector spaces make use of a base (see 9.9, 9.10). These functions get access to a base using the function `Base`, which normally computes a base for the vector space using an appropriate algorithm. Once a base is computed it will always be reused, no matter whether there is a more interesting base available or not.

`AddBase` installs a given *base* for V by overwriting any other base of the vector space that has been installed before. So after `AddBase` has successfully been used, *base* will be used whenever `Base` is called with V as argument.

Calling `AddBase` with a *base* which is not a base for V might produce unpredictable results in following computations.

```
gap> f := GF(3);
GF(3)
gap> m1 := [[ f.one, f.one, f.zero, f.zero ]];
[ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ]
gap> m2 := [[ f.one, f.one, f.one, f.zero ]];
[ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ]
gap> V := VectorSpace( [ m1, m2, m1+m2 ], GF(3) );
VectorSpace( [ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ],
```

```
  [ [ Z(3), Z(3), Z(3)^0, 0*Z(3) ] ] ], GF(3) )
gap> Base( V );
[ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ] ] ]
gap> AddBase( V, [ m1, m1+m2 ] );
gap> Base( V );
[ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3), Z(3), Z(3)^0, 0*Z(3) ] ] ]
```

9.8 Dimension

`Dimension(` *V* `)`

`Dimension` computes the dimension of the given vector space *V* over its field.

The dimension of a vector space *V* is defined to be the length of a minimal generating set of *V*, which is called a base of *V* (see 9.6).

The implementation of `Dimension` strictly follows its above definition, so that this function will always determine a base of *V*.

```
gap> f := GF( 3^4 );
GF(3^4)
gap> f.base;
[ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ]
gap> V := VectorSpace( f.base, GF( 3 ) );
VectorSpace( [ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ], GF(3) )
gap> Dimension( V );
4
```

9.9 LinearCombination

`LinearCombination(` *V*`,` *cf* `)`

`LinearCombination` computes the linear combination of the base elements of the vector space *V* with coefficients *cf*.

cf has to be a list of elements of *V*.field, the field over which the vector space is written. Its length must be equal to the dimension of *V* to make sure that one coefficient is specified for each element of the base.

`LinearCombination` will use that base of *V* which is returned when applying the function `Base` to *V* (see 9.6). To perform linear combinations of different bases use `AddBase` to specify which base should be used (see 9.7).

```
gap> f := GF( 3^4 );
GF(3^4)
gap> f.base;
[ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ]
gap> V := VectorSpace( f.base, GF( 3 ) );
VectorSpace( [ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ], GF(3) )
gap> LinearCombination( V, [ Z(3), Z(3)^0, Z(3), 0*Z(3) ] );
Z(3^4)^16
gap> Coefficients( V, f.root ^ 16 );
[ Z(3), Z(3)^0, Z(3), 0*Z(3) ]
```

9.10 Coefficients

`Coefficients(` V, v `)`

`Coefficients` computes the coefficients that have to be used to write v as a linear combination in the base of V.

To make sure that this function produces the correct result, v has to be an element of V. If v does not lie in V the result is unpredictable.

The result of `Coefficients` is returned as a list of elements of the field over which the vector space V is written. Of course, the length of this list equals the dimension of V.

```
gap> f := GF( 3^4 );
GF(3^4)
gap> f.base;
[ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ]
gap> V := VectorSpace( f.base, GF( 3 ) );
VectorSpace( [ Z(3)^0, Z(3^4), Z(3^4)^2, Z(3^4)^3 ], GF(3) )
gap> Dimension( V );
4
gap> Coefficients( V, f.root ^ 16 );
[ Z(3), Z(3)^0, Z(3), 0*Z(3) ]
```

Chapter 10

Integers

One of the most fundamental datatypes in every programming language is the integer type. GAP3 is no exception.

GAP3 integers are entered as a sequence of digits optionally preceded by a `+` sign for positive integers or a `-` sign for negative integers. The size of integers in GAP3 is only limited by the amount of available memory, so you can compute with integers having thousands of digits.

```
gap> -1234;
-1234
gap> 123456789012345678901234567890123456789012345678901234567890;
123456789012345678901234567890123456789012345678901234567890
```

The first sections in this chapter describe the operations applicable to integers (see 10.1, 10.2, 10.3 and 10.4).

The next sections describe the functions that test whether an object is an integer (see 10.5) and convert objects of various types to integers (see 10.6).

The next sections describe functions related to the ordering of integers (see 10.7, 10.8).

The next section describes the function that computes a Chinese remainder (see 10.11).

The next sections describe the functions related to the ordering of integers, logarithms, and roots (10.12, 10.13, 10.14).

The GAP3 object `Integers` is the ring domain of all integers. So all set theoretic functions are also applicable to this domain (see chapter 4 and 10.15). The only serious use of this however seems to be the generation of random integers.

Since the integers form a Euclidean ring all the ring functions are applicable to integers (see chapter 5, 10.16, 10.17, 10.18, 10.19, 10.20, 10.21, 10.22, 10.23, 10.24, 10.25, and 10.26).

Since the integers are naturally embedded in the field of rationals all the field functions are applicable to integers (see chapter 6 and 12.9).

Many more functions that are mainly related to the prime residue group of integers modulo an integer are described in chapter 11.

The external functions are in the file `LIBNAME/"integer.g"`.

10.1 Comparisons of Integers

n1 `=` *n2*
n1 `<>` *n2*

The equality operator `=` evaluates to `true` if the integer *n1* is equal to the integer *n2* and `false` otherwise. The inequality operator `<>` evaluates to `true` if *n1* is not equal to *n2* and `false` otherwise.

Integers can also be compared to objects of other types; of course, they are never equal.

```
gap> 1 = 1;
true
gap> 1 <> 0;
true
gap> 1 = (1,2);       # (1,2) is a permutation
false
```

n1 `<` *n2*
n1 `<=` *n2*
n1 `>` *n2*
n1 `>=` *n2*

The operators `<`, `<=`, `>`, and `=>` evaluate to `true` if the integer *n1* is less than, less than or equal to, greater than, or greater than or equal to the integer *n2*, respectively.

Integers can also be compared to objects of other types, they are considered smaller than any other object, except rationals, where the ordering reflects the ordering of the rationals (see 12.6).

```
gap> 1 < 2;
true
gap> 1 < -1;
false
gap> 1 < 3/2;
true
gap> 1 < false;
true
```

10.2 Operations for Integers

n1 `+` *n2*

The operator `+` evaluates to the sum of the two integers *n1* and *n2*.

n1 `-` *n2*

The operator `-` evaluates to the difference of the two integers *n1* and *n2*.

n1 `*` *n2*

The operator `*` evaluates to the product of the two integers *n1* and *n2*.

n1 `/` *n2*

The operator `/` evaluates to the quotient of the two integers *n1* and *n2*. If the divisor does not divide the dividend the quotient is a rational (see 12). If the divisor is 0 an error is signalled. The integer part of the quotient can be computed with `QuoInt` (see 10.3).

n1 `mod` *n2*

The operator `mod` evaluates to the smallest positive representative of the residue class of the left operand modulo the right, i.e., *i* `mod` *k* is the unique *m* in the range `[0 .. AbsInt(`*k*`)-1]` such that *k* divides *i* `-` *m*. If the right operand is 0 an error is signalled. The remainder of the division can be computed with `RemInt` (see 10.4).

n1 `^` *n2*

The operator `^` evaluates to the *n2*-th power of the integer *n1*. If *n2* is a positive integer then *n1*`^`*n2* is *n1*`*`*n1*`*..*`*n1* (*n2* factors). If *n2* is a negative integer *n1*`^`*n2* is defined as $1/n1^{-n2}$. If 0 is raised to a negative power an error is signalled. Any integer, even 0, raised to the zeroth power yields 1.

Since integers embed naturally into the field of rationals all the rational operations are available for integers too (see 12.7).

For the precedence of the operators see 2.10.

```
gap> 2 * 3 + 1;
7
```

10.3 QuoInt

`QuoInt(` *n1*`,` *n2* `)`

`QuoInt` returns the integer part of the quotient of its integer operands.

If *n1* and *n2* are positive `QuoInt(` *n1*`,` *n2* `)` is the largest positive integer *q* such that *q*`*`*n2* `<=` *n1*. If *n1* or *n2* or both are negative the absolute value of the integer part of the quotient is the quotient of the absolute values of *n1* and *n2*, and the sign of it is the product of the signs of *n1* and *n2*.

`RemInt` (see 10.4) can be used to compute the remainder.

```
gap> QuoInt(5,2);  QuoInt(-5,2);  QuoInt(5,-2);  QuoInt(-5,-2);
2
-2
-2
2
```

10.4 RemInt

`RemInt(` *n1*`,` *n2* `)`

`RemInt` returns the remainder of its two integer operands.

If *n2* is not equal to zero `RemInt(` *n1*`,` *n2* `) =` *n1* `-` *n2*`*QuoInt(` *n1*`,` *n2* `)`. Note that the rules given for `QuoInt` (see 10.3) imply that `RemInt(` *n1*`,` *n2* `)` has the same sign as *n1* and its absolute value is strictly less than the absolute value of *n2*. Dividing by 0 signals an error.

```
gap> RemInt(5,2);  RemInt(-5,2);  RemInt(5,-2);  RemInt(-5,-2);
1
-1
1
-1
```

10.5 IsInt

`IsInt( obj )`

`IsInt` returns `true` if *obj*, which can be an arbitrary object, is an integer and `false` otherwise. `IsInt` will signal an error if *obj* is an unbound variable.

```
gap> IsInt( 1 );
true
gap> IsInt( IsInt );
false         # IsInt is a function, not an integer
```

10.6 Int

`Int( obj )`

`Int` converts an object *obj* to an integer. If *obj* is an integer `Int` will simply return *obj*.

If *obj* is a rational number (see 12) `Int` returns the unique integer that has the same sign as *obj* and the largest absolute value not larger than the absolute value of *obj*.

If *obj* is an element of the prime field of a finite field F, `Int` returns the least positive integer n such that n`*`F`.one = `*obj* (see 18.8).

If *obj* is not of one of the above types an error is signalled.

```
gap> Int( 17 );
17
gap> Int( 17 / 3 );
5
gap> Int( Z(5^3)^62 );
4   #  Z(5^3)^62 = (Z(5^3)^(124/4))^2 = Z(5)^2 = PrimitiveRoot(5)^2 = 2^2
```

10.7 AbsInt

`AbsInt( n )`

`AbsInt` returns the absolute value of the integer n, i.e., n if n is positive, -n if n is negative and 0 if n is 0 (see 10.8).

```
gap> AbsInt( 33 );
33
gap> AbsInt( -214378 );
214378
gap> AbsInt( 0 );
0
```

10.8 SignInt

`SignInt( obj )`

`SignInt` returns the sign of the integer *obj*, i.e., 1 if *obj* is positive, -1 if *obj* is negative and 0 if *obj* is 0 (see 10.7).

```
gap> SignInt( 33 );
```

```
1
gap> SignInt( -214378 );
-1
gap> SignInt( 0 );
0
```

10.9 IsOddInt

`IsOddInt(` *i* `)`

Determines whether *i* is odd.

```
gap> IsOddInt(3);IsOddInt(4);
 true
 false
```

10.10 IsEvenInt

`IsEvenInt(` *i* `)`

Determines whether *i* is even.

```
gap> IsEvenInt(3);IsEvenInt(4);
 false
 true
```

10.11 ChineseRem

`ChineseRem(` *moduli*`,` *residues* `)`

`ChineseRem` returns the combination of the *residues* modulo the *moduli*, i.e., the unique integer *c* from `[0..Lcm(`*moduli*`)-1]` such that *c* `=` *residues*`[i]` modulo *moduli*`[i]` for all *i*, if it exists. If no such combination exists `ChineseRem` signals an error.

Such a combination does exist if and only if
residues`[`*i*`]=`*residues*`[`*k*`]` mod `Gcd(`*moduli*`[`*i*`],`*moduli*`[`*k*`])` for every pair *i*, *k*. Note that this implies that such a combination exists if the moduli are pairwise relatively prime. This is called the Chinese remainder theorem.

```
gap> ChineseRem( [ 2, 3, 5, 7 ], [ 1, 2, 3, 4 ] );
53
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 3, 5 ] );
103
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 2, 3 ] );
Error, the residues must be equal modulo 2
```

10.12 LogInt

`LogInt(` *n*`,` *base* `)`

`LogInt` returns the integer part of the logarithm of the positive integer *n* with respect to the positive integer *base*, i.e., the largest positive integer *exp* such that $base^{exp} <= n$. `LogInt` will signal an error if either *n* or *base* is not positive.

```
gap> LogInt( 1030, 2 );
10          # 2^10 = 1024
gap> LogInt( 1, 10 );
0
```

10.13 RootInt

`RootInt( n )`
`RootInt( n, k )`

`RootInt` returns the integer part of the kth root of the integer n. If the optional integer argument k is not given it defaults to 2, i.e., `RootInt` returns the integer part of the square root in this case.

If n is positive `RootInt` returns the largest positive integer r such that $r^k <= n$. If n is negative and k is odd `RootInt` returns `-RootInt( -n, k )`. If n is negative and k is even `RootInt` will cause an error. `RootInt` will also cause an error if k is 0 or negative.

```
gap> RootInt( 361 );
19
gap> RootInt( 2 * 10^12 );
1414213
gap> RootInt( 17000, 5 );
7         # 7^5 = 16807
```

10.14 SmallestRootInt

`SmallestRootInt( n )`

`SmallestRootInt` returns the smallest root of the integer n.

The smallest root of an integer n is the integer r of smallest absolute value for which a positive integer k exists such that $n = r^k$.

```
gap> SmallestRootInt( 2^30 );
2
gap> SmallestRootInt( -(2^30) );
-4          # note that (-2)^30 = +(2^30)
gap> SmallestRootInt( 279936 );
6
gap> LogInt( 279936, 6 );
7
gap> SmallestRootInt( 1001 );
1001
```

`SmallestRootInt` can be used to identify and decompose powers of primes as is demonstrated in the following example (see 10.19)

```
p := SmallestRootInt( q );  n := LogInt( q, p );
if not IsPrimeInt(p) then Error("GF: <q> must be a primepower"); fi;
```

10.15 Set Functions for Integers

As already mentioned in the first section of this chapter, `Integers` is the domain of all integers. Thus in principle all set theoretic functions, for example `Intersection`, `Size`, and so on can be applied to this domain. This seems generally of little use.

```
gap> Intersection( Integers, [ 0, 1/2, 1, 3/2 ] );
[ 0, 1 ]
gap> Size( Integers );
"infinity"
```

`Random( Integers )`

This seems to be the only useful domain function that can be applied to the domain `Integers`. It returns pseudo random integers between -10 and 10 distributed according to a binomial distribution.

```
gap> Random( Integers );
1
gap> Random( Integers );
-4
```

To generate uniformly distributed integers from a range, use the construct `Random( [` *low* `..` *high* `] )`.

10.16 Ring Functions for Integers

As was already noted in the introduction to this chapter the integers form a Euclidean ring, so all ring functions (see chapter 5) are applicable to the integers. This section comments on the implementation of those functions for the integers and tells you how you can call the corresponding functions directly, for example to save time.

`IsPrime( Integers,` *n* `)`

This is implemented by `IsPrimeInt`, which you can call directly to save a little bit of time (see 10.18).

`Factors( Integers,` *n* `)`

This is implemented as by `FactorsInt`, which you can call directly to save a little bit of time (see 10.22).

`EuclideanDegree( Integers,` *n* `)`

The Euclidean degree of an integer is of course simply the absolute value of the integer. Calling `AbsInt` directly will be a little bit faster.

`EuclideanRemainder( Integers,` *n*`,` *m* `)`

This is implemented as `RemInt(` *n*`,` *m* `)`, which you can use directly to save a lot of time.

`EuclideanQuotient( Integers,` *n*`,` *m* `)`

This is implemented as `QuoInt(` *n*`,` *m* `)`, which you can use directly to save a lot of time.

`QuotientRemainder( Integers,` *n*`,` *m* `)`

This is implemented as `[ QuoInt(`*n*`,`*m*`), RemInt(`*n*`,`*m*`) ]`, which you can use directly to save a lot of time.

`QuotientMod( Integers,` *n1* `,` *n2* `,` *m* `)`

This is implemented as `(`*n1* `/` *n2*`) mod` *m*, which you can use directly to save a lot of time.

`PowerMod( Integers,` *n* `,` *e* `,` *m* `)`

This is implemented by `PowerModInt`, which you can call directly to save a little bit of time. Note that using *n* `^` *e* `mod` *m* will generally be slower, because it can not reduce intermediate results like `PowerMod`.

`Gcd( Integers,` *n1* `,` *n2* `.. )`

This is implemented by `GcdInt`, which you can call directly to save a lot of time. Note that `GcdInt` takes only two arguments, not several as `Gcd` does.

`Gcdex(` *n1* `,` *n2* `)`

`Gcdex` returns a record. The component `gcd` is the gcd of *n1* and *n2*.

The components `coeff1` and `coeff2` are integer cofactors such that
g`.gcd =` *g*`.coeff1*`*n1* `+` *g*`.coeff2*`*n2*.
If *n1* and *n2* both are nonzero, `AbsInt(` *g*`.coeff1 )` is less than or equal to `AbsInt(`*n2*`) / (2*`*g*`.gcd)` and `AbsInt(` *g*`.coeff2 )` is less than or equal to `AbsInt(`*n1*`) / (2*`*g*`.gcd)`.

The components `coeff3` and `coeff4` are integer cofactors such that
`0 =` *g*`.coeff3*`*n1* `+` *g*`.coeff4*`*n2*.
If *n1* or *n2* or are both nonzero `coeff3` is `-`*n2* `/` *g*`.gcd` and `coeff4` is *n1* `/` *g*`.gcd`.

The coefficients always form a unimodular matrix, i.e., the determinant
g`.coeff1*`*g*`.coeff4 -` *g*`.coeff3*`*g*`.coeff2`
is 1 or -1.

```
gap> Gcdex( 123, 66 );
rec(
  gcd := 3,
  coeff1 := 7,
  coeff2 := -13,
  coeff3 := -22,
  coeff4 := 41 )
      # 3 = 7*123 - 13*66, 0 = -22*123 + 41*66
gap> Gcdex( 0, -3 );
rec(
  gcd := 3,
  coeff1 := 0,
  coeff2 := -1,
  coeff3 := 1,
  coeff4 := 0 )
gap> Gcdex( 0, 0 );
rec(
  gcd := 0,
  coeff1 := 1,
  coeff2 := 0,
  coeff3 := 0,
  coeff4 := 1 )
```

`Lcm( Integers,` *n1* `,` *n2* `.. )`

This is implemented as `LcmInt`, which you can call directly to save a little bit of time. Note that `LcmInt` takes only two arguments, not several as `Lcm` does.

10.17 Primes

`Primes[` n `]`

`Primes` is a set, i.e., a sorted list, of the 168 primes less than 1000.

`Primes` is used in `IsPrimeInt` (see 10.18) and `FactorsInt` (see 10.22) to cast out small prime divisors quickly.

```
gap> Primes[1];
2
gap> Primes[100];
541
```

10.18 IsPrimeInt

`IsPrimeInt(` n `)`

`IsPrimeInt` returns `false` if it can prove that n is composite and `true` otherwise. By convention `IsPrimeInt(0) = IsPrimeInt(1) = false` and we define `IsPrimeInt(` $-n$ `) = IsPrimeInt(` n `)`.

`IsPrimeInt` will return `true` for all prime n. `IsPrimeInt` will return `false` for all composite $n < 10^{13}$ and for all composite n that have a factor $p < 1000$. So for integers $n < 10^{13}$, `IsPrimeInt` is a proper primality test. It is conceivable that `IsPrimeInt` may return `true` for some composite $n > 10^{13}$, but no such n is currently known. So for integers $n > 10^{13}$, `IsPrimeInt` is a probable-primality test. If composites that fool `IsPrimeInt` do exist, they would be extremly rare, and finding one by pure chance is less likely than finding a bug in GAP3.

`IsPrimeInt` is a deterministic algorithm, i.e., the computations involve no random numbers, and repeated calls will always return the same result. `IsPrimeInt` first does trial divisions by the primes less than 1000. Then it tests that n is a strong pseudoprime w.r.t. the base 2. Finally it tests whether n is a Lucas pseudoprime w.r.t. the smallest quadratic nonresidue of n. A better description can be found in the comment in the library file `integer.g`.

The time taken by `IsPrimeInt` is approximately proportional to the third power of the number of digits of n. Testing numbers with several hundreds digits is quite feasible.

```
gap> IsPrimeInt( 2^31 - 1 );
true
gap> IsPrimeInt( 10^42 + 1 );
false
```

10.19 IsPrimePowerInt

`IsPrimePowerInt(` n `)`

`IsPrimePowerInt` returns `true` if the integer n is a prime power and `false` otherwise.

n is a **prime power** if there exists a prime p and a positive integer i such that $p^i = n$. If n is negative the condition is that there must exist a negative prime p and an odd positive integer i such that $p^i = n$. 1 and -1 are not prime powers.

Note that `IsPrimePowerInt` uses `SmallestRootInt` (see 10.14) and a probable-primality test (see 10.18).

```
gap> IsPrimePowerInt( 31^5 );
true
gap> IsPrimePowerInt( 2^31-1 );
true         # 2^31 - 1 is actually a prime
gap> IsPrimePowerInt( 2^63-1 );
false
gap> Filtered( [-10..10], IsPrimePowerInt );
[ -8, -7, -5, -3, -2, 2, 3, 4, 5, 7, 8, 9 ]
```

10.20 NextPrimeInt

`NextPrimeInt(` n `)`

`NextPrimeInt` returns the smallest prime which is strictly larger than the integer n.

Note that `NextPrimeInt` uses a probable-primality test (see 10.18).

```
gap> NextPrimeInt( 541 );
547
gap> NextPrimeInt( -1 );
2
```

10.21 PrevPrimeInt

`PrevPrimeInt(` n `)`

`PrevPrimeInt` returns the largest prime which is strictly smaller than the integer n.

Note that `PrevPrimeInt` uses a probable-primality test (see 10.18).

```
gap> PrevPrimeInt( 541 );
523
gap> PrevPrimeInt( 1 );
-2
```

10.22 FactorsInt

`FactorsInt(` n `)`

`FactorsInt` returns a list of the prime factors of the integer n. If the ith power of a prime divides n this prime appears i times. The list is sorted, that is the smallest prime factors come first. The first element has the same sign as n, the others are positive. For any integer n it holds that `Product( FactorsInt(` n `) ) =` n.

Note that `FactorsInt` uses a probable-primality test (see 10.18). Thus `FactorsInt` might return a list which contains composite integers.

The time taken by `FactorsInt` is approximately proportional to the square root of the second largest prime factor of n, which is the last one that `FactorsInt` has to find, since the largest factor is simply what remains when all others have been removed. Thus the time is roughly bounded by the fourth root of n. `FactorsInt` is guaranteed to find all factors less than 10^6 and will find most factors less than 10^{10}. If n contains multiple factors larger than that `FactorsInt` may not be able to factor n and will then signal an error.

```
gap> FactorsInt( -Factorial(6) );
[ -2, 2, 2, 2, 3, 3, 5 ]
gap> Set( FactorsInt( Factorial(13)/11 ) );
[ 2, 3, 5, 7, 13 ]
gap> FactorsInt( 2^63 - 1 );
[ 7, 7, 73, 127, 337, 92737, 649657 ]
gap> FactorsInt( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
```

10.23 DivisorsInt

`DivisorsInt(` n `)`

`DivisorsInt` returns a list of all positive **divisors** of the integer n. The list is sorted, so it starts with 1 and ends with n. We define `DivisorsInt(` $-n$ `) = DivisorsInt(` n `)`. Since the set of divisors of 0 is infinite calling `DivisorsInt( 0 )` causes an error.

`DivisorsInt` calls `FactorsInt` (see 10.22) to obtain the prime factors. `Sigma` (see 10.24) computes the sum, `Tau` (see 10.25) the number of positive divisors.

```
gap> DivisorsInt( 1 );
[ 1 ]
gap> DivisorsInt( 20 );
[ 1, 2, 4, 5, 10, 20 ]
gap> DivisorsInt( 541 );
[ 1, 541 ]
```

10.24 Sigma

`Sigma(` n `)`

`Sigma` returns the sum of the positive divisors (see 10.23) of the integer n.

`Sigma` is a multiplicative arithmetic function, i.e., if n and m are relatively prime we have $\sigma(nm) = \sigma(n)\sigma(m)$. Together with the formula $\sigma(p^e) = (p^{e+1} - 1)/(p - 1)$ this allows you to compute $\sigma(n)$.

Integers n for which $\sigma(n) = 2n$ are called perfect. Even perfect integers are exactly of the form $2^{n-1}(2^n - 1)$ where $2^n - 1$ is prime. Primes of the form $2^n - 1$ are called **Mersenne primes**, the known ones are obtained for n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, and 859433. It is not known whether odd perfect integers exist, however [BC89] show that any such integer must have at least 300 decimal digits.

`Sigma` usually spends most of its time factoring n (see 10.22).

```
gap> Sigma( 0 );
Error, Sigma: <n> must not be 0
```

```
gap> Sigma( 1 );
1
gap> Sigma( 1009 );
1010        # thus 1009 is a prime
gap> Sigma( 8128 ) = 2*8128;
true        # thus 8128 is a perfect number
```

10.25 Tau

`Tau(` n `)`

`Tau` returns the number of the positive divisors (see 10.23) of the integer n.

`Tau` is a multiplicative arithmetic function, i.e., if n and m are relatively prime we have $\tau(nm) = \tau(n)\tau(m)$. Together with the formula $\tau(p^e) = e + 1$ this allows us to compute $\tau(n)$.

`Tau` usually spends most of its time factoring n (see 10.22).

```
gap> Tau( 0 );
Error, Tau: <n> must not be 0
gap> Tau( 1 );
1
gap> Tau( 1013 );
2        # thus 1013 is a prime
gap> Tau( 8128 );
14
gap> Tau( 36 );
9        # τ(n) is odd if and only if n is a perfect square
```

10.26 MoebiusMu

`MoebiusMu(` n `)`

`MoebiusMu` computes the value of the **Moebius function** for the integer n. This is 0 for integers which are not squarefree, i.e., which are divisible by a square r^2. Otherwise it is 1 if n has an even number and -1 if n has an odd number of prime factors.

The importance of μ stems from the so called inversion formula. Suppose $f(n)$ is a function defined on the positive integers and let $g(n) = \sum_{d|n} f(d)$. Then $f(n) = \sum_{d|n} \mu(d)g(n/d)$. As a special case we have $\phi(n) = \sum_{d|n} \mu(d)n/d$ since $n = \sum_{d|n} \phi(d)$ (see 11.2).

`MoebiusMu` usually spends all of its time factoring n (see 10.22).

```
gap> MoebiusMu( 60 );
0
gap> MoebiusMu( 61 );
-1
gap> MoebiusMu( 62 );
1
```

Chapter 11

Number Theory

The integers relatively prime to m form a group under multiplication modulo m, called the **prime residue group**. This chapter describes the functions that deal with this group.

The first section describes the function that computes the set of representatives of the group (see 11.1).

The next sections describe the functions that compute the size and the exponent of the group (see 11.2 and 11.3).

The next section describes the function that computes the order of an element in the group (see 11.4).

The next section describes the functions that test whether a residue generates the group or computes a generator of the group, provided it is cyclic (see 11.5, 11.6).

The next section describes the functions that test whether an element is a square in the group (see 11.7 and 11.8).

The next sections describe the functions that compute general roots in the group (see 11.9 and 11.10).

All these functions are in the file `LIBNAME/"numtheor.g"`.

11.1 PrimeResidues

`PrimeResidues(` *m* `)`

`PrimeResidues` returns the set of integers from the range $0..Abs(m)-1$ that are relatively prime to the integer m.

$Abs(m)$ must be less than 2^{28}, otherwise the set would probably be too large anyhow.

The integers relatively prime to m form a group under multiplication modulo m, called the **prime residue group**. $\phi(m)$ (see 11.2) is the order of this group, $\lambda(m)$ (see 11.3) the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called primitive roots (see 11.5, 11.6).

```
gap> PrimeResidues( 0 );
```

```
[  ]
gap> PrimeResidues( 1 );
[ 0 ]
gap> PrimeResidues( 20 );
[ 1, 3, 7, 9, 11, 13, 17, 19 ]
```

11.2 Phi

`Phi( m )`

`Phi` returns the value of the **Euler totient function** $\phi(m)$ for the integer m. $\phi(m)$ is defined as the number of positive integers less than or equal to m that are relatively prime to m.

Suppose that $m = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$. Then $\phi(m)$ is $p_1^{e_1-1}(p_1-1)p_2^{e_2-1}(p_2-1)...p_k^{e_k-1}(p_k-1)$. It follows that m is a prime if and only if $\phi(m) = m-1$.

The integers relatively prime to m form a group under multiplication modulo m, called the **prime residue group**. It can be computed with `PrimeResidues` (see 11.1). $\phi(m)$ is the order of this group, $\lambda(m)$ (see 11.3) the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called primitive roots (see 11.5, 11.6).

`Phi` usually spends most of its time factoring m (see 10.22).

```
gap> Phi( 12 );
4
gap> Phi( 2^13-1 );
8190          # which proves that 2^13 - 1 is a prime
gap> Phi( 2^15-1 );
27000
```

11.3 Lambda

`Lambda( m )`

`Lambda` returns the exponent of the group of relatively prime residues modulo the integer m.

$\lambda(m)$ is the smallest positive integer l such that for every a relatively prime to m we have $a^l = 1 \bmod m$. Fermat's theorem asserts $a^{\phi(m)} = 1 \bmod m$, thus $\lambda(m)$ divides $\phi(m)$ (see 11.2).

Carmichael's theorem states that λ can be computed as follows $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^e) = 2^{e-2}$ if $3 <= e$, $\lambda(p^e) = (p-1)p^{e-1}$ $(= \phi(p^e))$ if p is an odd prime, and $\lambda(nm) = Lcm(\lambda(n), \lambda(m))$ if n, m are relatively prime.

Composites for which $\lambda(m)$ divides $m-1$ are called Carmichaels. If $6k+1$, $12k+1$ and $18k+1$ are primes their product is such a number. It is believed but unproven that there are infinitely many Carmichaels. There are only 1547 Carmichaels below 10^{10} but 455052511 primes.

The integers relatively prime to m form a group under multiplication modulo m, called the **prime residue group**. It can be computed with `PrimeResidues` (see 11.1). $\phi(m)$ (see

11.2) is the order of this group, $\lambda(m)$ the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called primitive roots (see 11.5, 11.6).

`Lambda` usually spends most of its time factoring m (see 10.22).

```
gap> Lambda( 10 );
4
gap> Lambda( 30 );
4
gap> Lambda( 561 );
80          # 561 is the smallest Carmichael number
```

11.4 OrderMod

`OrderMod(` n`,` m `)`

`OrderMod` returns the multiplicative order of the integer n modulo the positive integer m. If n is less than 0 or larger than m it is replaced by its remainder. If n and m are not relatively prime the order of n is not defined and `OrderMod` will return 0.

If n and m are relatively prime the multiplicative order of n modulo m is the smallest positive integer i such that $n^i = 1 \bmod m$. Elements of maximal order are called primitive roots (see 11.2).

`OrderMod` usually spends most of its time factoring m and $\phi(m)$ (see 10.22).

```
gap> OrderMod( 2, 7 );
3
gap> OrderMod( 3, 7 );
6          # 3 is a primitive root modulo 7
```

11.5 IsPrimitiveRootMod

`IsPrimitiveRootMod(` r`,` m `)`

`IsPrimitiveRootMod` returns `true` if the integer r is a primitive root modulo the positive integer m and `false` otherwise. If r is less than 0 or larger than m it is replaced by its remainder.

The integers relatively prime to m form a group under multiplication modulo m, called the prime residue group. It can be computed with `PrimeResidues` (see 11.1). $\phi(m)$ (see 11.2) is the order of this group, $\lambda(m)$ (see 11.3) the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called **primitive roots** (see also 11.6).

```
gap> IsPrimitiveRootMod( 2, 541 );
true
gap> IsPrimitiveRootMod( -539, 541 );
true          # same computation as above
gap> IsPrimitiveRootMod( 4, 541 );
false
gap> ForAny( [1..29], r -> IsPrimitiveRootMod( r, 30 ) );
false          # there does not exist a primitive root modulo 30
```

11.6 PrimitiveRootMod

`PrimitiveRootMod(` *m* `)`
`PrimitiveRootMod(` *m*`,` *start* `)`

`PrimitiveRootMod` returns the smallest primitive root modulo the positive integer m and `false` if no such primitive root exists. If the optional second integer argument *start* is given `PrimitiveRootMod` returns the smallest primitive root that is strictly larger than *start*.

The integers relatively prime to m form a group under multiplication modulo m, called the prime residue group. It can be computed with `PrimeResidues` (see 11.1). $\phi(m)$ (see 11.2) is the order of this group, $\lambda(m)$ (see 11.3) the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called **primitive roots** (see also 11.5).

```
gap> PrimitiveRootMod( 409 );
21          # largest primitive root for a prime less than 2000
gap> PrimitiveRootMod( 541, 2 );
10
gap> PrimitiveRootMod( 337, 327 );
false       # 327 is the largest primitive root mod 337
gap> PrimitiveRootMod( 30 );
false       # the exists no primitive root modulo 30
```

11.7 Jacobi

`Jacobi(` *n*`,` *m* `)`

`Jacobi` returns the value of the **Jacobi symbol** of the integer n modulo the integer m.

Suppose that $m = p_1 p_2 .. p_k$ as a product of primes, not necessarily distinct. Then for n relatively prime to m the Jacobi symbol is defined by $J(n/m) = L(n/p_1)L(n/p_2)..L(n/p_k)$, where $L(n/p)$ is the Legendre symbol (see 11.8). By convention $J(n/1) = 1$. If the gcd of n and m is larger than 1 we define $J(n/m) = 0$.

If n is an **quadratic residue** modulo m, i.e., if there exists an r such that $r^2 = n \bmod m$ then $J(n/m) = 1$. However $J(n/m) = 1$ implies the existence of such an r only if m is a prime.

`Jacobi` is very efficient, even for large values of n and m, it is about as fast as the Euclidean algorithm (see 5.26).

```
gap> Jacobi( 11, 35 );
1           # 9^2 = 11 mod 35
gap> Jacobi( 6, 35 );
-1          # thus there is no r such that r^2 = 6 mod 35
gap> Jacobi( 3, 35 );
1           # even though there is no r with r^2 = 3 mod 35
```

11.8 Legendre

`Legendre(` *n*`,` *m* `)`

`Legendre` returns the value of the **Legendre symbol** of the integer n modulo the positive integer m.

The value of the Legendre symbol $L(n/m)$ is 1 if n is a **quadratic residue** modulo m, i.e., if there exists an integer r such that $r^2 = n \bmod m$ and -1 otherwise.

If a root of n exists it can be found by `RootMod` (see 11.9).

While the value of the Legendre symbol usually is only defined for m a prime, we have extended the definition to include composite moduli too. The Jacobi symbol (see 11.7) is another generalization of the Legendre symbol for composite moduli that is much cheaper to compute, because it does not need the factorization of m (see 10.22).

```
gap> Legendre( 5, 11 );
1           # 4^2 = 5 mod 11
gap> Legendre( 6, 11 );
-1          # thus there is no r such that r^2 = 6 mod 11
gap> Legendre( 3, 35 );
-1          # thus there is no r such that r^2 = 3 mod 35
```

11.9 RootMod

`RootMod(` n`,` m `)`
`RootMod(` n`,` k`,` m `)`

In the first form `RootMod` computes a square root of the integer n modulo the positive integer m, i.e., an integer r such that $r^2 = n \bmod m$. If no such root exists `RootMod` returns `false`.

A root of n exists only if `Legendre(`n`,`m`) = 1` (see 11.8). If m has k different prime factors then there are 2^k different roots of n mod m. It is unspecified which one `RootMod` returns. You can, however, use `RootsUnityMod` (see 11.10) to compute the full set of roots.

In the second form `RootMod` computes a kth root of the integer n modulo the positive integer m, i.e., an integer r such that $r^k = n \bmod m$. If no such root exists `RootMod` returns `false`.

In the current implementation k must be a prime.

`RootMod` is efficient even for large values of m, actually most time is usually spent factoring m (see 10.22).

```
gap> RootMod( 64, 1009 );
1001          # note RootMod does not return 8 in this case but -8
gap> RootMod( 64, 3, 1009 );
518
gap> RootMod( 64, 5, 1009 );
656
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>           x -> x mod 1009 );
[ 1001, 8 ]          # set of all square roots of 64 mod 1009
```

11.10 RootsUnityMod

`RootsUnityMod(` m `)`
`RootsUnityMod(` k`,` m `)`

In the first form RootsUnityMod computes the square roots of 1 modulo the integer m, i.e., the set of all positive integers r less than n such that $r^2 = 1 \bmod m$.

In the second form RootsUnityMod computes the kth roots of 1 modulo the integer m, i.e., the set of all positive integers r less than n such that $r^k = 1 \bmod m$.

In general there are k^n such roots if the modulus m has n different prime factors p such that $p = 1 \bmod k$. If k^2 divides m then there are k^{n+1} such roots; and especially if $k = 2$ and 8 divides m there are 2^{n+2} such roots.

If you are interested in the full set of roots of another number instead of 1 use RootsUnityMod together with RootMod (see 11.9).

In the current implementation k must be a prime.

RootsUnityMod is efficient even for large values of m, actually most time is usually spent factoring m (see 10.22).

```
gap> RootsUnityMod(7*31);
[ 1, 92, 125, 216 ]
gap> RootsUnityMod(3,7*31);
[ 1, 25, 32, 36, 67, 149, 156, 191, 211 ]
gap> RootsUnityMod(5,7*31);
[ 1, 8, 64, 78, 190 ]
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>           x -> x mod 1009 );
[ 1001, 8 ]          # set of all square roots of 64 mod 1009
```

Chapter 12

Rationals

The **rationals** form a very important field. On the one hand it is the quotient field of the integers (see 10). On the other hand it is the prime field of the fields of characteristic zero (see 15).

The former comment suggests the representation actually used. A rational is represented as a pair of integers, called **numerator** and **denominator**. Numerator and denominator are **reduced**, i.e., their greatest common divisor is 1. If the denominator is 1, the rational is in fact an integer and is represented as such. The numerator holds the sign of the rational, thus the denominator is always positive.

Because the underlying integer arithmetic can compute with arbitrary size integers, the rational arithmetic is always exact, even for rationals whose numerators and denominators have thousands of digits.

```
gap> 2/3;
2/3
gap> 66/123;
22/41     # numerator and denominator are made relatively prime
gap> 17/-13;
-17/13     # the numerator carries the sign
gap> 121/11;
11     # rationals with denominator 1 (after cancelling) are integers
```

The first sections of this chapter describe the functions that test whether an object is a rational (see 12.1), and select the numerator and denominator of a rational (see 12.2, 12.3).

The next sections describe the rational operations (see 12.6, and 12.7).

The GAP3 object `Rationals` is the field domain of all rationals. All set theoretic functions are applicable to this domain (see chapter 4 and 12.8). Since `Rationals` is a field all field functions are also applicable to this domain and its elements (see chapter 6 and 12.9).

All external functions are defined in the file `"LIBNAME/rational.g"`.

12.1 IsRat

`IsRat(` *obj* `)`

IsRat returns true if *obj*, which can be an arbitrary object, is a rational and false otherwise. Integers are rationals with denominator 1, thus IsRat returns true for integers. IsRat will signal an error if *obj* is an unbound variable or a procedure call.

```
gap> IsRat( 2/3 );
true
gap> IsRat( 17/-13 );
true
gap> IsRat( 11 );
true
gap> IsRat( IsRat );
false     # IsRat is a function, not a rational
```

12.2 Numerator

Numerator(*rat*)

Numerator returns the numerator of the rational *rat*. Because the numerator holds the sign of the rational it may be any integer. Integers are rationals with denominator 1, thus Numerator is the identity function for integers.

```
gap> Numerator( 2/3 );
2
gap> Numerator( 66/123 );
22    # numerator and denominator are made relatively prime
gap> Numerator( 17/-13 );
-17    # the numerator holds the sign of the rational
gap> Numerator( 11 );
11    # integers are rationals with denominator 1
```

Denominator (see 12.3) is the counterpart to Numerator.

12.3 Denominator

Denominator(*rat*)

Denominator returns the denominator of the rational *rat*. Because the numerator holds the sign of the rational the denominator is always a positive integer. Integers are rationals with the denominator 1, thus Denominator returns 1 for integers.

```
gap> Denominator( 2/3 );
3
gap> Denominator( 66/123 );
41    # numerator and denominator are made relatively prime
gap> Denominator( 17/-13 );
13    # the denominator holds the sign of the rational
gap> Denominator( 11 );
1    # integers are rationals with denominator 1
```

Numerator (see 12.2) is the counterpart to Denominator.

12.4 Floor

`Floor(`*r*`)`

This function returns the largest integer smaller or equal to *r*.

```
gap> Floor(-2/3);
-1
gap> Floor(2/3);
0
```

12.5 Mod1

`Mod1(`*r*`)`

The argument should be a rational or a list. If *r* is a rational, it returns `(Numerator(`*r*`) mod Denominator(`*r*`))/Denominator(r)`. If *r* is a list, it returns `List(`*r*`,Mod1)`. This function is very useful for working in $\mathbb{Q}/\mathbb{Z}$.

```
gap> Mod1([-2/3,-1,7/4,3]);
[ 1/3, 0, 3/4, 0 ]
```

12.6 Comparisons of Rationals

q1 `=` *q2*
q1 `<>` *q2*

The equality operator `=` evaluates to `true` if the two rationals *q1* and *q2* are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the two rationals *q1* and *q2* are not equal and to `false` otherwise.

```
gap> 2/3 = -4/-6;
true
gap> 66/123 <> 22/41;
false
gap> 17/13 = 11;
false
```

q1 `<` *q2*
q1 `<=` *q2*
q1 `>` *q2*
q1 `>=` *q2*

The operators `<`, `<=`, `>`, and `=>` evaluate to `true` if the rational *q1* is less than, less than or equal to, greater than, and greater than or equal to the rational *q2* and to `false` otherwise. One rational $q_1 = n_1/d_1$ is less than another $q_2 = n_2/d_2$ if and only if $n_1 d_2 < n_2 d_2$. This definition is of course only valid because the denominator of rationals is always defined to be positive. This definition also extends to the comparison of rationals with integers, which are interpreted as rationals with denominator 1. Rationals can also be compared with objects of other types. They are smaller than objects of any other type by definition.

```
gap> 2/3 < 22/41;
false
gap> -17/13 < 11;
true
```

12.7 Operations for Rationals

q1 + *q2*
q1 - *q2*
q1 * *q2*
q1 / *q2*

The operators +, -, * and / evaluate to the sum, difference, product, and quotient of the two rationals *q1* and *q2*. For the quotient / *q2* must of course be nonzero, otherwise an error is signalled. Either operand may also be an integer *i*, which is interpreted as a rational with denominator 1. The result of those operations is always reduced. If, after the reduction, the denominator is 1, the rational is in fact an integer, and is represented as such.

```
gap> 2/3 + 4/5;
22/15
gap> 7/6 * 2/3;
7/9    # note how the result is cancelled
gap> 67/6 - 1/6;
11     # the result is an integer
```

q ^ *i*

The powering operator ^ returns the *i*-th power of the rational *q*. *i* must be an integer. If the exponent *i* is zero, *q*^*i* is defined as 1; if *i* is positive, *q*^*i* is defined as the *i*-fold product *q***q**..**q*; finally, if *i* is negative, *q*^*i* is defined as (1/*q*)^-*i*. In this case *q* must of course be nonzero.

```
gap> (2/3) ^ 3;
8/27
gap> (-17/13) ^ -1;
-13/17    # note how the sign switched
gap> (1/2) ^ -2;
4
```

12.8 Set Functions for Rationals

As was already mentioned in the introduction of this chapter the GAP3 object `Rationals` is the domain of all rationals. All set theoretic functions, e.g., `Intersection` and `Size`, are applicable to this domain.

```
gap> Intersection( Rationals, [ E(4)^0, E(4)^1, E(4)^2, E(4)^3 ] );
[ -1, 1 ]    # E(4) is the complex square root of -1
gap> Size( Rationals );
"infinity"
```

This does not seem to be very useful.

12.9 Field Functions for Rationals

As was already mentioned in the introduction of this chapter the GAP3 object `Rationals` is the field of all rationals. All field functions, e.g., `Norm` and `MinPol` are applicable to this domain and its elements. However, since the field of rationals is the prime field, all those functions are trivial. Therefore, `Conjugates( Rationals,` q `)` returns `[` q `]`, `Norm( Rationals,` q `)` and `Trace( Rationals,` q `)` return q, and `CharPol( Rationals,` q `)` and `MinPol( Rationals,` q `)` both return `[ -`q`, 1 ]`.

Chapter 13

Cyclotomics

GAP3 allows computations in abelian extension fields of the rational field Q, i.e., fields with abelian Galois group over Q. These fields are described in chapter 15. They are subfields of **cyclotomic fields** $Q_n = Q(e_n)$ where $e_n = e^{\frac{2\pi i}{n}}$ is a primitive n–th root of unity. Their elements are called **cyclotomics**.

The internal representation of a cyclotomic does not refer to the smallest number field but the smallest cyclotomic field containing it (the so–called **conductor**). This is because it is easy to embed two cyclotomic fields in a larger one that contains both, i.e., there is a natural way to get the sum or the product of two arbitrary cyclotomics as element of a cyclotomic field. The disadvantage is that the arithmetical operations are too expensive to do arithmetics in number fields, e.g., calculations in a matrix ring over a number field. But it suffices to deal with irrationalities in character tables (see 49). (And in fact, the comfortability of working with the natural embeddings is used there in many situations which did not actually afford it . . .)

All functions that take a field extension as —possibly optional— argument, e.g., `Trace` or `Coefficients` (see chapter 6), are described in chapter 15.

This chapter informs about

- the representation of cyclotomics in GAP3 (see 13.1),
- access to the internal data (see 13.7, 13.8)
- integral elements of number fields (see 13.2, 13.3, 13.4),
- characteristic functions (see 13.5, 13.6),
- comparison and arithmetical operations of cyclotomics (see 13.9, 13.10),
- functions concerning Galois conjugacy of cyclotomics (see 13.11, 13.14), or lists of them (see 13.16, 13.17),
- some special cyclotomics, as defined in [CCN+85] (see 13.13, 13.15)

The external functions are in the file `LIBNAME/"cyclotom.g"`.

13.1 More about Cyclotomics

Elements of number fields (see chapter 15), cyclotomics for short, are arithmetical objects like rationals and finite field elements; they are not implemented as records —like groups—

or e.g. with respect to a character table (although character tables may be the main interest for cyclotomic arithmetics).

`E( n )`

returns the primitive n-th root of unity $e_n = e^{\frac{2\pi i}{n}}$. Cyclotomics are usually entered as (and irrational cyclotomics are always displayed as) sums of roots of unity with rational coefficients. (For special cyclotomics, see 13.13.)

```
gap> E(9); E(9)^3; E(6); E(12) / 3;
-E(9)^4-E(9)^7      # the root needs not to be an element of the base
E(3)
-E(3)^2
-1/3*E(12)^7
```

For the representation of cyclotomics one has to recall that the cyclotomic field $Q_n = Q(e_n)$ is a vector space of dimension $\varphi(n)$ over the rationals where φ denotes Euler's phi-function (see 11.2).

Note that the set of all n-th roots of unity is linearly dependent for $n > 1$, so multiplication is not the multiplication of the group ring $Q\langle e_n\rangle$; given a Q-basis of Q_n the result of the multiplication (computed as multiplication of polynomials in e_n, using $(e_n)^n = 1$) will be converted to the base.

```
gap> E(5) * E(5)^2; ( E(5) + E(5)^4 ) * E(5)^2;
E(5)^3
E(5)+E(5)^3
gap> ( E(5) + E(5)^4 ) * E(5);
-E(5)-E(5)^3-E(5)^4
```

Cyclotomics are always represented in the smallest cyclotomic field they are contained in. Together with the choice of a fixed base this means that two cyclotomics are equal if and only if they are equally represented.

Addition and multiplication of two cyclotomics represented in Q_n and Q_m, respectively, is computed in the smallest cyclotomic field containing both: $Q_{\texttt{Lcm}(n,m)}$. Conversely, if the result is contained in a smaller cyclotomic field the representation is reduced to the minimal such field.

The base, the base conversion and the reduction to the minimal cyclotomic field are described in [Zum89], more about the base can be found in 15.9.

Since n must be a `short integer`, the maximal cyclotomic field implemented in GAP3 is not really the field Q^{ab}. The biggest allowed (though not very useful) n is 65535.

There is a global variable `Cyclotomics` in GAP3, a record that stands for the domain of all cyclotomics (see chapter 15).

13.2 Cyclotomic Integers

A cyclotomic is called **integral** or **cyclotomic integer** if all coefficients of its minimal polynomial are integers. Since the base used is an integral base (see 15.9), the subring of cyclotomic integers in a cyclotomic field is formed by those cyclotomics which have not only rational but integral coefficients in their representation as sums of roots of unity. For example, square roots of integers are cyclotomic integers (see 13.13), any root of unity is a cyclotomic integer, character values are always cyclotomic integers, but all rationals which are not integers are not cyclotomic integers. (See 13.6)

```
gap> ER( 5 );                   # The square root of 5 is a cyclotomic
E(5)-E(5)^2-E(5)^3+E(5)^4       # integer, it has integral coefficients.
gap> 1/2 * ER( 5 );             # This is not a cyclotomic integer, ...
1/2*E(5)-1/2*E(5)^2-1/2*E(5)^3+1/2*E(5)^4
gap> 1/2 * ER( 5 ) - 1/2;       # ... but this is one.
E(5)+E(5)^4
```

13.3 IntCyc

`IntCyc(` *z* `)`

returns the cyclotomic integer (see 13.2) with Zumbroich base coefficients (see 15.9) `List(` *zumb*`, x -> Int( x ) )` where *zumb* is the vector of Zumbroich base coefficients of the cyclotomic *z*; see also 13.4.

```
gap> IntCyc( E(5)+1/2*E(5)^2 ); IntCyc( 2/3*E(7)+3/2*E(4) );
E(5)
E(4)
```

13.4 RoundCyc

`RoundCyc(` *z* `)`

returns the cyclotomic integer (see 13.2) with Zumbroich base coefficients (see 15.9) `List(` *zumb*`, x -> Int( x+1/2 ) )` where *zumb* is the vector of Zumbroich base coefficients of the cyclotomic *z*; see also 13.3.

```
gap> RoundCyc( E(5)+1/2*E(5)^2 ); RoundCyc( 2/3*E(7)+3/2*E(4) );
E(5)+E(5)^2
-2*E(28)^3+E(28)^4-2*E(28)^11-2*E(28)^15-2*E(28)^19-2*E(28)^23
 -2*E(28)^27
```

13.5 IsCyc

`IsCyc(` *obj* `)`

returns `true` if *obj* is a cyclotomic, and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsCyc( 0 ); IsCyc( E(3) ); IsCyc( 1/2 * E(3) ); IsCyc( IsCyc );
true
true
true
false
```

`IsCyc` is an internal function.

13.6 IsCycInt

`IsCycInt(` *obj* `)`

returns `true` if *obj* is a cyclotomic integer (see 13.2), `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsCycInt( 0 ); IsCycInt( E(3) ); IsCycInt( 1/2 * E(3) );
true
true
false
```

`IsCycInt` is an internal function.

13.7 NofCyc

`NofCyc(` *z* `)`
`NofCyc(` *list* `)`

returns the smallest positive integer n for which the cyclotomic z is resp. for which all cyclotomics in the list *list* are contained in $Q_n = Q(e^{\frac{2\pi i}{n}}) = Q(\mathtt{E}(n))$.

```
gap> NofCyc( 0 ); NofCyc( E(10) ); NofCyc( E(12) );
1
5
12
```

`NofCyc` is an internal function.

13.8 CoeffsCyc

`CoeffsCyc(` *z*`,` *n* `)`

If z is a cyclotomic which is contained in Q_n, `CoeffsCyc(` *z*`,` *n* `)` returns a list *cfs* of length n where the entry at position i is the coefficient of $\mathtt{E}(n)^{i-1}$ in the internal representation of z as element of the cyclotomic field Q_n (see 13.1, 15.9): $z = cfs[1] + cfs[2]\ \mathtt{E}(n)^1 + \ldots + cfs[n]\ \mathtt{E}(n)^{n-1}$.

Note that all positions which do not belong to base elements of Q_n contain zeroes.

```
gap> CoeffsCyc( E(5), 5 ); CoeffsCyc( E(5), 15 );
[ 0, 1, 0, 0, 0 ]
[ 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, -1, 0 ]
gap> CoeffsCyc( 1+E(3), 9 ); CoeffsCyc( E(5), 7 );
[ 0, 0, 0, 0, 0, 0, -1, 0, 0 ]
Error, no representation of <z> in 7th roots of unity
```

`CoeffsCyc` calls the internal function `COEFFSCYC`:

`COEFFSCYC(` *z* `)`

is equivalent to `CoeffsCyc(` *z*`, NofCyc(` *z* `) )`, see 13.7.

13.9 Comparisons of Cyclotomics

To compare cyclotomics, the operators `<`, `<=`, `=`, `>=`, `>` and `<>` can be used, the result will be `true` if the first operand is smaller, smaller or equal, equal, larger or equal, larger, or inequal, respectively, and `false` otherwise.

Cyclotomics are ordered as follows: The relation between rationals is as usual, and rationals are smaller than irrational cyclotomics. For two irrational cyclotomics *z1*, *z2* which lie in different minimal cyclotomic fields, we have *z1* < *z2* if and only if `NofCyc`(*z1*) < `NofCyc`(*z2*)); if `NofCyc`(*z1*) = `NofCyc`(*z2*)), that one is smaller that has the smaller coefficient vector, i.e., *z1* ≤ *z2* if and only if `COEFFSCYC`(*z1*) ≤ `COEFFSCYC`(*z2*).

You can compare cyclotomics with objects of other types; all objects which are not cyclotomics are larger than cyclotomics.

```
gap> E(5) < E(6);       # the latter value lies in Q_3
false
gap> E(3) < E(3)^2;     # both lie in Q_3, so compare coefficients
false
gap> 3 < E(3); E(5) < E(7);
true
true
gap> E(728) < (1,2);
true
```

13.10 Operations for Cyclotomics

The operators `+`, `-`, `*`, `/` are used for addition, subtraction, multiplication and division of two cyclotomics; note that division by 0 causes an error.

`+` and `-` can also be used as unary operators;

`^` is used for exponentiation of a cyclotomic with an integer; this is in general **not** equal to Galois conjugation.

```
gap> E(5) + E(3); (E(5) + E(5)^4) ^ 2; E(5) / E(3); E(5) * E(3);
-E(15)^2-2*E(15)^8-E(15)^11-E(15)^13-E(15)^14
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(15)^13
E(15)^8
```

13.11 GaloisCyc

`GaloisCyc(` *z*`,` *k* `)`

returns the cyclotomic obtained on raising the roots of unity in the representation of the cyclotomic *z* to the *k*-th power. If *z* is represented in the field Q_n and *k* is a fixed integer relative prime to *n*, `GaloisCyc( .,` *k* `)` acts as a Galois automorphism of Q_n (see 15.8); to get Galois automorphisms as functions, use 6.7 `GaloisGroup`.

```
gap> GaloisCyc( E(5) + E(5)^4, 2 );
E(5)^2+E(5)^3
gap> GaloisCyc( E(5), -1 );           # the complex conjugate
E(5)^4
gap> GaloisCyc( E(5) + E(5)^4, -1 );  # this value is real
E(5)+E(5)^4
gap> GaloisCyc( E(15) + E(15)^4, 3 );
E(5)+E(5)^4
```

`GaloisCyc` is an internal function.

13.12 Galois

`Galois(` z`,` e `)`

This function is a kind of generalized version of `GaloisCyc`. If z is a list it returns the list of `Galois(x,e)` for each element x of z. If z is a cyclotomic, if e is an integer it is equivalent to `GaloisCyc(`z`,`e`)` and if e is a Galois element it is equivalent to `z^e`. Finally, if z is a record with a `.operations` field, it returns z`.operations.Galois(z,e)`. One such operations is predefined if z is a polynomial, it does `Galois(x,e)` on each coefficient of z.

```
gap> Galois(E(3),-1);
E(3)^2
gap> Galois(E(3),-1);
E(3)^2
gap> G:=GaloisGroup(CF(3));
Group( NFAutomorphism( CF(3) , 2 ) )
gap> E(3)^G.1;
E(3)^2
gap> Galois([E(3),E(5)],-1);
[ E(3)^2, E(5)^4 ]
gap> Galois(X(Cyclotomics)+E(3),-1);
X(Cyclotomics) + (E(3)^2)
```

13.13 ATLAS irrationalities

`EB(` N `)`, `EC(` N `)`, ..., `EH(` N `)`,
`EI(` N `)`, `ER(` N `)`,
`EJ(` N `)`, `EK(` N `)`, `EL(` N `)`, `EM(` N `)`,
`EJ(` N`,` d `)`, `EK(` N`,` d `)`, `EL(` N`,` d `)`, `EM(` N`,` d `)`,
`ES(` N `)`, `ET(` N `)`, ..., `EY(` N `)`,
`ES(` N`,` d `)`, `ET(` N`,` d `)`, ..., `EY(` N`,` d `)`,
`NK(` N`,` k`,` d `)`

For N a positive integer, let $z = \mathtt{E}(N) = e^{2\pi i/N}$. The following so-called atomic irrationalities (see [CCN+85, Chapter 7, Section 10]) can be entered by functions (Note that the values are not necessary irrational.):

$$
\begin{array}{lllll}
\mathtt{EB}(N) & = & b_N & = & \frac{1}{2}\sum_{j=1}^{N-1} z^{j^2} & (N \equiv 1 \bmod 2) \\
\mathtt{EC}(N) & = & c_N & = & \frac{1}{3}\sum_{j=1}^{N-1} z^{j^3} & (N \equiv 1 \bmod 3) \\
\mathtt{ED}(N) & = & d_N & = & \frac{1}{4}\sum_{j=1}^{N-1} z^{j^4} & (N \equiv 1 \bmod 4) \\
\mathtt{EE}(N) & = & e_N & = & \frac{1}{5}\sum_{j=1}^{N-1} z^{j^5} & (N \equiv 1 \bmod 5) \\
\mathtt{EF}(N) & = & f_N & = & \frac{1}{6}\sum_{j=1}^{N-1} z^{j^6} & (N \equiv 1 \bmod 6) \\
\mathtt{EG}(N) & = & g_N & = & \frac{1}{7}\sum_{j=1}^{N-1} z^{j^7} & (N \equiv 1 \bmod 7) \\
\mathtt{EH}(N) & = & h_N & = & \frac{1}{8}\sum_{j=1}^{N-1} z^{j^8} & (N \equiv 1 \bmod 8)
\end{array}
$$

(Note that in $c_N, \ldots, h_N$, N must be a prime.)

$$\begin{array}{lclcl} \texttt{ER}(N) & = & \sqrt{N} & & \\ \texttt{EI}(N) & = & i\sqrt{N} & = & \sqrt{-N} \end{array}$$

From a theorem of Gauss we know that

$$b_N = \left\{ \begin{array}{ll} \frac{1}{2}(-1+\sqrt{N}) & \text{if} \quad N \equiv 1 \mod 4 \\ \frac{1}{2}(-1+i\sqrt{N}) & \text{if} \quad N \equiv -1 \mod 4 \end{array} \right. ,$$

so $\sqrt{N}$ can be (and in fact is) computed from b_N. If N is a negative integer then `ER(`N`) = EI(-`N`)`.

For given N, let $n_k = n_k(N)$ be the first integer with multiplicative order exactly k modulo N, chosen in the order of preference

$$1, -1, 2, -2, 3, -3, 4, -4, \ldots \ .$$

We have

$$\begin{array}{lclcll} \texttt{EY}(N) & = & y_n & = & z + z^n & (n = n_2) \\ \texttt{EX}(N) & = & x_n & = & z + z^n + z^{n^2} & (n = n_3) \\ \texttt{EW}(N) & = & w_n & = & z + z^n + z^{n^2} + z^{n^3} & (n = n_4) \\ \texttt{EV}(N) & = & v_n & = & z + z^n + z^{n^2} + z^{n^3} + z^{n^4} & (n = n_5) \\ \texttt{EU}(N) & = & u_n & = & z + z^n + z^{n^2} + \ldots + z^{n^5} & (n = n_6) \\ \texttt{ET}(N) & = & t_n & = & z + z^n + z^{n^2} + \ldots + z^{n^6} & (n = n_7) \\ \texttt{ES}(N) & = & s_n & = & z + z^n + z^{n^2} + \ldots + z^{n^7} & (n = n_8) \end{array}$$

$$\begin{array}{lclcll} \texttt{EM}(N) & = & m_n & = & z - z^n & (n = n_2) \\ \texttt{EL}(N) & = & l_n & = & z - z^n + z^{n^2} - z^{n^3} & (n = n_4) \\ \texttt{EK}(N) & = & k_n & = & z - z^n + \ldots - z^{n^5} & (n = n_6) \\ \texttt{EJ}(N) & = & j_n & = & z - z^n + \ldots - z^{n^7} & (n = n_8) \end{array}$$

Let $n_k^{(d)} = n_k^{(d)}(N)$ be the $d+1$-th integer with multiplicative order exactly k modulo N, chosen in the order of preference defined above; we write $n_k = n_k^{(0)}, n'_k = n_k^{(1)}, n''_k = n_k^{(2)}$ and so on. These values can be computed as `NK(`N`,`k`,`d`)`$= n_k^{(d)}(N)$; if there is no integer with the required multiplicative order, `NK` will return `false`.

The algebraic numbers

$$y'_N = y_N^{(1)}, y''_N = y_N^{(2)}, \ldots, x'_N, x''_N, \ldots, j'_N, j''_N, \ldots$$

are obtained on replacing n_k in the above definitions by $n'_k, n''_k, \ldots$; they can be entered as

$$\begin{array}{lcl} \texttt{EY}(N,d) & = & y_N^{(d)} \\ \texttt{EX}(N,d) & = & x_N^{(d)} \\ & \vdots & \\ \texttt{EJ}(N,d) & = & j_n^{(d)} \end{array}$$

```
gap> EW(16,3); EW(17,2); ER(3); EI(3); EY(5); EB(9);
0
E(17)+E(17)^4+E(17)^13+E(17)^16
-E(12)^7+E(12)^11
E(3)-E(3)^2
E(5)+E(5)^4
1
```

13.14 StarCyc

`StarCyc(` *z* `)`

If z is an irrational element of a quadratic number field (i.e. if z is a quadratic irrationality), `StarCyc(` *z* `)` returns the unique Galois conjugate of z that is different from z; this is often called $z*$ (see 49.37). Otherwise `false` is returned.

```
gap> StarCyc( EB(5) ); StarCyc( E(5) );
E(5)^2+E(5)^3
false
```

13.15 Quadratic

`Quadratic(` *z* `)`

If z is a cyclotomic integer that is contained in a quadratic number field over the rationals, it can be written as $z = \frac{a+b\sqrt{n}}{d}$ with integers a, b, n and d, where d is either 1 or 2. In this case `Quadratic(` *z* `)` returns a record with fields `a`, `b`, `root`, `d` and `ATLAS` where the first four mean the integers mentioned above, and the last one is a string that is a (not necessarily shortest) representation of z by b_m, i_m or r_m for $m = |$`root`$|$ (see 13.13).

If z is not a quadratic irrationality or not a cyclotomic integer, `false` is returned.

```
gap> Quadratic( EB(5) ); Quadratic( EB(27) );
rec(
  a := -1,
  b := 1,
  root := 5,
  d := 2,
  ATLAS := "b5" )
rec(
  a := -1,
  b := 3,
  root := -3,
  d := 2,
  ATLAS := "1+3b3" )
gap> Quadratic(0); Quadratic( E(5) );
rec(
  a := 0,
  b := 0,
  root := 1,
  d := 1,
```

```
  ATLAS := "0" )
false
```

13.16 GaloisMat

`GaloisMat(` *mat* `)`

mat must be a matrix of cyclotomics (or possibly unknowns, see 17.1). The conjugate of a row in *mat* under a particular Galois automorphism is defined pointwise. If *mat* consists of full orbits under this action then the Galois group of its entries acts on *mat* as a permutation group, otherwise the orbits must be completed before.

`GaloisMat(` *mat* `)` returns a record with fields `mat`, `galoisfams` and `generators`:

`mat`
: a list with initial segment *mat* (**not** a copy of *mat*); the list consists of full orbits under the action of the Galois group of the entries of *mat* defined above. The last entries are those rows which had to be added to complete the orbits; so if they were already complete, *mat* and `mat` have identical entries.

`galoisfams`
: a list that has the same length as `mat`; its entries are either 1, 0, -1 or lists:
`galoisfams`[`i`] $= 1$ means that `mat[i]` consists of rationals, i.e. [`mat`[`i`]] forms an orbit.
`galoisfams`[`i`] $= -1$ means that `mat[i]` contains unknowns; in this case [`mat`[`i`]] is regarded as an orbit, too, even if `mat[i]` contains irrational entries.
If `galoisfams`[`i`] $= [l_1, l_2]$ is a list then `mat[i]` is the first element of its orbit in `mat`; l_1 is the list of positions of rows which form the orbit, and l_2 is the list of corresponding Galois automorphisms (as exponents, not as functions); so we have `mat`$[l_1[j]][k] =$ `GaloisCyc`(`mat`$[i][k], l_2[j]$).
`galoisfams`[`i`] $= 0$ means that `mat[i]` is an element of a nontrivial orbit but not the first element of it.

`generators`
: a list of permutations generating the permutation group corresponding to the action of the Galois group on the rows of `mat`.

Note that *mat* should be a set, i.e. no two rows should be equal. Otherwise only the first row of some equal rows is considered for the permutations, and a warning is printed.

```
gap> GaloisMat( [ [ E(3), E(4) ] ] );
rec(
  mat := [ [ E(3), E(4) ], [ E(3), -E(4) ], [ E(3)^2, E(4) ],
      [ E(3)^2, -E(4) ] ],
  galoisfams := [ [ [ 1, 2, 3, 4 ], [ 1, 7, 5, 11 ] ], 0, 0, 0 ],
  generators := [ (1,2)(3,4), (1,3)(2,4) ] )
gap> GaloisMat( [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ] ] );
rec(
  mat := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ],
  galoisfams := [ 1, [ [ 2, 3 ], [ 1, 2 ] ], 0 ],
  generators := [ (2,3) ] )
```

13.17 RationalizedMat

`RationalizedMat(` *mat* `)`

returns the set of rationalized rows of *mat*, i.e. the set of sums over orbits under the action of the Galois group of the elements of *mat* (see 13.16).

This may be viewed as a kind of trace operation for the rows.

Note that *mat* should be a set, i.e. no two rows should be equal.

```
gap> mat:= CharTable( "A5" ).irreducibles;
[ [ 1, 1, 1, 1, 1 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> RationalizedMat( mat );
[ [ 1, 1, 1, 1, 1 ], [ 6, -2, 0, 1, 1 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
```

Chapter 14

Gaussians

If we adjoin a square root of -1, usually denoted by i, to the field of rationals we obtain a field that is an extension of degree 2. This field is called the **Gaussian rationals** and its ring of integers is called the **Gaussian integers**, because C.F. Gauss was the first to study them.

In GAP3 Gaussian rationals are written in the form `a + b*E(4)`, where a and b are rationals, because `E(4)` is GAP3's name for i. Because 1 and i form an integral base the Gaussian integers are written in the form `a + b*E(4)`, where a and b are integers.

The first sections in this chapter describe the operations applicable to Gaussian rationals (see 14.1 and 14.2).

The next sections describe the functions that test whether an object is a Gaussian rational or integer (see 14.3 and 14.4).

The GAP3 object `GaussianRationals` is the field domain of all Gaussian rationals, and the object `GaussianIntegers` is the ring domain of all Gaussian integers. All set theoretic functions are applicable to those two domains (see chapter 4 and 14.5).

The Gaussian rationals form a field so all field functions, e.g., `Norm`, are applicable to the domain `GaussianRationals` and its elements (see chapter 6 and 14.6).

The Gaussian integers form a Euclidean ring so all ring functions, e.g., `Factors`, are applicable to `GaussianIntegers` and its elements (see chapter 5, 14.7, and 14.8).

The field of Gaussian rationals is just a special case of cyclotomic fields, so everything that applies to those fields also applies to it (see chapters 13 and 15).

All functions are in the library file `LIBNAME/"gaussian.g"`.

14.1 Comparisons of Gaussians

x `=` *y*
x `<>` *y*

The equality operator evaluates to `true` if the two Gaussians x and y are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the two Gaussians x and

y are not equal, and to `false` otherwise. It is also possible to compare a Gaussian with an object of another type, of course they are never equal.

Two Gaussians a `+` b`*E(4)` and c `+` d`*E(4)` are considered equal if a `=` c and b `=` d.

```
gap> 1 + E(4) = 2 / (1 - E(4));
true
gap> 1 + E(4) = 1 - E(4);
false
gap> 1 + E(4) = E(6);
false
```

x `<`y
x `<=` y
x `>` y
x `>=` y

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the Gaussian x is less than, less than or equal to, greater than, and greater than or equal to the Gaussian y, and to `false` otherwise. Gaussians can also be compared to objects of other types, they are smaller than anything else, except other cyclotomics (see 13.9).

A Gaussian a `+` b`*E(4)` is considered less than another Gaussian c `+` d`*E(4)` if a is less than c, or if a is equal to c and b is less than d.

```
gap> 1 + E(4) < 2 + E(4);
true
gap> 1 + E(4) < 1 - E(4);
false
gap> 1 + E(4) < 1/2;
false
```

14.2 Operations for Gaussians

x `+` y
x `-` y
x `*` y
x `/` y

The operators `+`, `-`, `*`, and `/` evaluate to the sum, difference, product, and quotient of the two Gaussians x and y. Of course either operand may also be an ordinary rational (see 12), because the rationals are embedded into the Gaussian rationals. On the other hand the Gaussian rationals are embedded into other cyclotomic fields, so either operand may also be a cyclotomic (see 13). Division by 0 is as usual an error.

x `^` n

The operator `^` evaluates to the n-th power of the Gaussian rational x. If n is positive, the power is defined as the n-fold product x`*`x`*...`x; if n is negative, the power is defined as `(1/`x`)^(-`n`)`; and if n is zero, the power is 1, even if x is 0.

```
gap> (1 + E(4)) * (E(4) - 1);
-2
```

14.3 IsGaussRat

`IsGaussRat( obj )`

`IsGaussRat` returns `true` if *obj*, which may be an object of arbitrary type, is a Gaussian rational and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsGaussRat( 1/2 );
true
gap> IsGaussRat( E(4) );
true
gap> IsGaussRat( E(6) );
false
gap> IsGaussRat( true );
false
```

`IsGaussInt` can be used to test whether an object is a Gaussian integer (see 14.4).

14.4 IsGaussInt

`IsGaussInt( obj )`

`IsGaussInt` returns `true` if *obj*, which may be an object of arbitrary type, is a Gaussian integer, and false otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsGaussInt( 1 );
true
gap> IsGaussInt( E(4) );
true
gap> IsGaussInt( 1/2 + 1/2*E(4) );
false
gap> IsGaussInt( E(6) );
false
```

`IsGaussRat` can be used to test whether an object is a Gaussian rational (see 14.3).

14.5 Set Functions for Gaussians

As already mentioned in the introduction of this chapter the objects `GaussianRationals` and `GaussianIntegers` are the domains of Gaussian rationals and integers respectively. All set theoretic functions, i.e., `Size` and `Intersection`, are applicable to these domains and their elements (see chapter 4). There does not seem to be an important use of this however. All functions not mentioned here are not treated specially, i.e., they are implemented by the default function mentioned in the respective section.

`in`

The membership test for Gaussian rationals is implemented via `IsGaussRat` (14.3). The membership test for Gaussian integers is implemented via `IsGaussInt` (see 14.4).

`Random`

A random Gaussian rational `a + b*E(4)` is computed by combining two random rationals a and b (see 12.8). Likewise a random Gaussian integer `a + b*E(4)` is computed by combining two random integers a and b (see 10.15).

```
gap> Size( GaussianRationals );
"infinity"
gap> Intersection( GaussianIntegers, [1,1/2,E(4),-E(6),E(4)/3] );
[ 1, E(4) ]
```

14.6 Field Functions for Gaussian Rationals

As already mentioned in the introduction of this chapter, the domain of Gaussian rationals is a field. Therefore all field functions are applicable to this domain and its elements (see chapter 6). This section gives further comments on the definitions and implementations of those functions for the the Gaussian rationals. All functions not mentioned here are not treated specially, i.e., they are implemented by the default function mentioned in the respective section.

`Conjugates`

The field of Gaussian rationals is an extension of degree 2 of the rationals, its prime field. Therefore there is one further conjugate of every element `a + b*E(4)`, namely `a - b*E(4)`.

`Norm`, `Trace`

According to the definition of conjugates above, the norm of a Gaussian rational `a + b*E(4)` is `a^2 + b^2` and the trace is `2*a`.

14.7 Ring Functions for Gaussian Integers

As already mentioned in the introduction to this chapter, the ring of Gaussian integers is a Euclidean ring. Therefore all ring functions are applicable to this ring and its elements (see chapter 5). This section gives further comments on the definitions and implementations of those functions for the Gaussian integers. All functions not mentioned here are not treated specially, i.e., they are implemented by the default function mentioned in the respective section.

`IsUnit`, `Units`, `IsAssociated`, `Associates`

The units of `GaussianIntegers` are `[ 1, E(4), -1, -E(4) ]`.

`StandardAssociate`

The standard associate of a Gaussian integer x is the associated element y of x that lies in the first quadrant of the complex plane. That is y is that element from `x * [1,-1,E(4),-E(4)]` that has positive real part and nonnegative imaginary part.

`EuclideanDegree`

The Euclidean degree of a Gaussian integer x is the product of x and its complex conjugate.

`EuclideanRemainder`

Define the integer part i of the quotient of x and y as the point of the lattice spanned by 1 and `E(4)` that lies next to the rational quotient of x and y, rounding towards the origin if there are several such points. Then `EuclideanRemainder(` x, y `)` is defined as x `-` i `*` y. With this definition the ordinary Euclidean algorithm for the greatest common divisor works, whereas it does not work if you always round towards the origin.

`EuclideanQuotient`

The Euclidean quotient of two Gaussian integers x and y is the quotient of w and y, where w is the difference between x and the Euclidean remainder of x and y.

`QuotientRemainder`

`QuotientRemainder` uses `EuclideanRemainder` and `EuclideanQuotient`.

`IsPrime`, `IsIrreducible`

Since the Gaussian integers are a Euclidean ring, primes and irreducibles are equivalent. The primes are the elements `1 + E(4)` and `1 - E(4)` of norm 2, the elements a `+` b`*E(4)` and a `-` b`*E(4)` of norm p `=` a`^2 +` b`^2` with p a rational prime congruent to 1 mod 4, and the elements p of norm p`^2` with p a rational prime congruent to 3 mod 4.

`Factors`

The list returned by `Factors` is sorted according to the norms of the primes, and among those of equal norm with respect to `<`. All elements in the list are standard associates, except the first, which is multiplied by a unit as necessary.

The above characterization already shows how one can factor a Gaussian integer. First compute the norm of the element, factor this norm over the rational integers and then split 2 and the primes congruent to 1 mod 4 with `TwoSquares` (see 14.8).

```
gap> Factors( GaussianIntegers, 30 );
[ -1-E(4), 1+E(4), 3, 1+2*E(4), 2+E(4) ]
```

14.8 TwoSquares

`TwoSquares(` n `)`

`TwoSquares` returns a list of two integers $x <= y$ such that the sum of the squares of x and y is equal to the nonnegative integer n, i.e., $n = x^2 + y^2$. If no such representation exists `TwoSquares` will return `false`. `TwoSquares` will return a representation for which the gcd of x and y is as small as possible. If there are several such representations, it is not specified which one `TwoSquares` returns.

Let a be the product of all maximal powers of primes of the form $4k + 3$ dividing n. A representation of n as a sum of two squares exists if and only if a is a perfect square. Let b

be the maximal power of 2 dividing n, or its half, whichever is a perfect square. Then the minimal possible gcd of x and y is the square root c of ab. The number of different minimal representations with $x <= y$ is 2^{l-1}, where l is the number of different prime factors of the form $4k+1$ of n.

```
gap> TwoSquares( 5 );
[ 1, 2 ]
gap> TwoSquares( 11 );
false          # no representation exists
gap> TwoSquares( 16 );
[ 0, 4 ]
gap> TwoSquares( 45 );
[ 3, 6 ]          # 3 is the minimal possible gcd because 9 divides 45
gap> TwoSquares( 125 );
[ 2, 11 ]          # not [ 5, 10 ] because this has not minimal gcd
gap> TwoSquares( 13*17 );
[ 5, 14 ]          # [10,11] would be the other possible representation
gap> TwoSquares( 848654483879497562821 );
[ 6305894639, 28440994650 ]          # 848654483879497562821 is prime
```

Chapter 15

Subfields of Cyclotomic Fields

The only **number fields** that GAP3 can handle at the moment are subfields of cyclotomic fields, e.g., $Q(\sqrt{5})$ is a number field that is not cyclotomic but contained in the cyclotomic field $Q_5 = Q(e^{\frac{2\pi i}{5}})$. Although this means that GAP3 does not know arbitrary algebraic number fields but only those with abelian Galois group, here we call these fields **number fields** for short. The elements of number fields are called **cyclotomics** (see chapter 13). Thus number fields are the domains (see chapter 4) related to cyclotomics; they are special field records (see 6.17) which are needed to specify the field extension with respect to which e.g. the trace of a cyclotomic shall be computed.

In many situations cyclotomic fields need not be treated in a special way, except that there may be more efficient algorithms for them than for arbitrary number fields. For that, there are the global variables `NumberFieldOps` and `CyclotomicFieldOps`, both records which contain the field operations stored in `FieldOps` (see chapter 6) and in which some functions are overlaid (see 15.13). If all necessary information about a function is already given in chapter 6, this function is not described here; this is the case e.g. for `Conjugates` and related functions, like `Trace` and `CharPol`. Some functions, however, need further explanation, e.g., 15.12 tells more about `Coefficients` for number fields.

There are some functions which are different for cyclotomic fields and other number fields, e.g., the field constructors `CF` resp. `NF`. In such a situation, the special case is described in a section immediately following the section about the general case.

Besides the single number fields, there is another domain in GAP3 related to number fields, the domain `Cyclotomics` of all cyclotomics. Although this is an abstract field, namely the field Q^{ab}, `Cyclotomics` is not a field record. It is used by `DefaultField`, `DefaultRing`, `Domain`, `Field` and `Ring` (see 6.3, 5.3, 4.5, 6.2, 5.2) which are mainly interested in the corresponding entries of `Cyclotomics.operations` since these functions know how to create fields resp. integral rings generated by some cyclotomics.

This chapter informs about

characteristic functions (see 15.1, 15.2),
field constructors (see 15.3, 15.4),
(default) fields of cyclotomics (see 15.5), and (default) rings of cyclotomic integers (see 15.6),
Galois groups of number fields (see 15.7, 15.8),

vector space bases (see 15.9, 15.10, 15.11) and coefficients (see 15.12) and overlaid functions in the operations records (see 15.13).

The external functions are in the file `LIBNAME/"numfield.g"`

15.1 IsNumberField

`IsNumberField(` *obj* `)`

returns `true` if *obj* is a field record (see 6.1, 6.17) of a field of characteristic zero where *F*`.generators` is a list of cyclotomics (see chapter 13), and `false` else.

```
gap> IsNumberField( CF(9) ); IsNumberField( NF( [ ER(3) ] ) );
true
true
gap> IsNumberField( GF( 2 ) );
false
```

15.2 IsCyclotomicField

`IsCyclotomicField(` *obj* `)`

returns `true` if *obj* is a number field record (see 15.1) where *obj*`.isCyclotomicField = true`, and `false` else.

```
gap> IsCyclotomicField( CF(9) );
true
gap> IsCyclotomicField( NF( [ ER(-3) ] ) );
true
gap> IsCyclotomicField( NF( [ ER(3) ] ) );
false
```

15.3 Number Field Records

`NumberField(` *gens* `)`
`NumberField(` *n*`,` *stab* `)`
`NumberField(` *subfield*`,` *poly* `)`
`NumberField(` *subfield*`,` *base* `)`

`NumberField` may be abbreviated `NF`; it returns number fields, namely

`NumberField(` *gens* `):`
the number field generated by the cyclotomics in the list *gens*,

`NumberField(` *n*`,` *stab* `):`
the fixed field of the prime residues in the list *stab* inside the cyclotomic field Q_n (see 15.4),

`NumberField(` *subfield*`,` *poly* `):`
the splitting field of the polynomial *poly* (which must have degree at most 2) over the number field *subfield*; *subfield* `= 0` is equivalent to *subfield* `= Rationals`,

`NumberField(` *subfield*`,` *base* `):`
the extension field of the number field *subfield* which is as vector space generated by the elements of the list *base* of cyclotomics; that means, *base* must be or at least contain a

vector space base of this extension, if *base* is a base it will be assigned to the `base` field of the cyclotomic field (see 15.12). *subfield* `= 0` is equivalent to *subfield* `= Rationals`.

```
gap> NF( [ EB(7), ER(3) ] );
NF(84,[ 1, 11, 23, 25, 37, 71 ])
gap> NF( 7, [ 1 ] );
CF(7)
gap> NF( NF( [ EB(7) ] ), [ 1, 1, 1 ] );
NF(NF(7,[ 1, 2, 4 ]),[ 1, E(3) ])
gap> F:= NF( 0, [ 1, E(4) ] ); G:= NF( 0, NormalBaseNumberField( F ) );
GaussianRationals
CF( Rationals,[ 1/2-1/2*E(4), 1/2+1/2*E(4) ])
gap> G.base; G.basechangemat; Coefficients( G, 1 );
[ 1/2-1/2*E(4), 1/2+1/2*E(4) ]
[ [ 1, 1 ], [ -1, 1 ] ]
[ 1, 1 ]
```

Number field records are field records (see 6.17) representing a number field. Besides the obligatory record components, a number field record F contains the component

`stabilizer`
: the list of prime residues modulo `NofCyc(` F`.generators )` which fix all elements of F

and possibly

`isIntegralBase`
: `true` if F`.base` is an integral vector space base of the field extension F `/` F`.field`, `false` else (used by 5.2 `Ring`); for the case that F`.field` is a cyclotomic field, 15.10 describes integral bases of the field extension;

`isNormalBase`
: `true` if F`.base` is a normal vector space base of the field extension F`/`F`.field`, `false` else;

`coeffslist`
: a list of integers used by 9.10 `Coefficients`; (see also 15.12);

`coeffsmat`
: a matrix of cyclotomics used by 9.10 `Coefficients`; bound only if F`.field` is not a cyclotomic field (see also 15.12);

`basechangemat`
: square matrix of dimension F`.dimension`, representing the basechange from the default base of F `/` F`.field` (see 15.12) to the base stored in F`.base` if these two are different; used by `Coefficients`.

Note: These fields and also the field `base` should not be changed by hand!

15.4 Cyclotomic Field Records

`CyclotomicField(` *n* `)`
`CyclotomicField(` *gens* `)`
`CyclotomicField(` *subfield*`,` *n* `)`
`CyclotomicField(` *subfield*`,` *base* `)`

`CyclotomicField` may be abbreviated `CF`; it returns cyclotomic fields, namely

`CyclotomicField(` *n* `)`
: the field Q_n (over the rationals),

`CyclotomicField(` *gens* `)`
: the smallest cyclotomic field containing the cyclotomics in the list *gens* (over the rationals),

`CyclotomicField(` *subfield*`,` *n* `)`
: the field Q_n over the number field *subfield*,

`CyclotomicField(` *subfield*`,` *base* `)`
: the cyclotomic extension field of the number field *subfield* which is as vector space generated by the elements of the list *base* of cyclotomics; that means, *base* must be or at least contain a vector space base of this extension, if *base* is a base it will be assigned to the `base` field of the cyclotomic field (see 15.12). *subfield* `= 0` is equivalent to *subfield* `= Rationals`.

```
gap> CF( 5 ); CF( [ EB(7), ER(3) ] ); CF( NF( [ ER(3) ] ), 24 );
CF(5)
CF(84)
CF(24)/NF(12,[ 1, 11 ])
gap> CF( CF(3), [ 1, E(4) ] );
CF(12)/CF(3)
```

A cyclotomic field record is a field record (see 6.17), in particular a number field record (see 15.3) that represents a cyclotomic field. Besides the obligatory record fields, a cyclotomic field record F contains the fields

`isCyclotomicField`
: always `true`; used by 15.2 `IsCyclotomicField`,

`zumbroichbase`
: a list containing `ZumbroichBase(` *n*`,` *m* `)` (see 15.9) if F represents the field extension Q_n/Q_m, and containing `Zumbroichbase(` *n*`, 1 )` if F is an extension of a number field that is not cyclotomic; used by 9.10 `Coefficients`, see 15.12

and possibly optional fields of number fields (see 15.3).

15.5 DefaultField and Field for Cyclotomics

For a set S of cyclotomics,

`DefaultField(` *S* `) = CF(` *S* `)` is the smallest cylotomic field containing S (see 6.3), the so–called **conductor** of S;

`Field(` *S* `) = NF(` *S* `)` is the smallest field containing S (see 6.2).

```
gap> DefaultField( [ E(5) ] ); DefaultField( [ E(3), ER(6) ] );
CF(5)
CF(24)
gap> Field( [ E(5) ] ); Field( [ E(3), ER(6) ] );
CF(5)
NF(24,[ 1, 19 ])
```

`DefaultField` and `Field` are used by functions that specify the field for which some cyclotomics are regarded as elements (see 6.3, 6.2), e.g., `Trace` with only one argument will compute the trace of this argument (which must be a cyclotomic) with respect to its default field.

15.6 DefaultRing and Ring for Cyclotomic Integers

For a set S of cyclotomic integers,

`DefaultRing(` S `)` is the ring of integers in `CF(` S `)` (see 5.3),

`Ring(` S `)` is the ring of integers in `NF(` S `)` (see 5.2).

```
gap> Ring( [ E(5) ] );
Ring( E(5) )
gap> Ring( [ EB(7) ] );
Ring( E(7)+E(7)^2+E(7)^4 )
gap> DefaultRing( [ EB(7) ] );
Ring( E(7) )
```

15.7 GeneratorsPrimeResidues

`GeneratorsPrimeResidues(` n `)`

returns a record with fields

`primes`
: the set of prime divisors of the integer n,

`exponents`
: the corresponding exponents in the factorization of n and

`generators`
: generators of the group of prime residues: For each odd prime p there is one generator, corresponding to a primitive root of the subgroup $(Z/p^{\nu_p})^*$ of $(Z/nZ)^*$, where ν_p is the exponent of p in the factorization of n; for $p = 2$, we have one generator in the case that 8 does not divide n, and a list of two generators (corresponding to $\langle *5, *(2^{\nu_2} - 1)\rangle = (Z/2^{\nu_2})^*$) else.

```
gap> GeneratorsPrimeResidues( 9 );        # 2 is a primitive root
rec(
  primes := [ 3 ],
  exponents := [ 2 ],
  generators := [ 2 ] )
gap> GeneratorsPrimeResidues( 24 );       # 8 divides 24
rec(
  primes := [ 2, 3 ],
  exponents := [ 3, 1 ],
  generators := [ [ 7, 13 ], 17 ] )
gap> GeneratorsPrimeResidues( 1155 );
rec(
  primes := [ 3, 5, 7, 11 ],
  exponents := [ 1, 1, 1, 1 ],
  generators := [ 386, 232, 661, 211 ] )
```

15.8 GaloisGroup for Number Fields

The **Galois automorphisms** of the cyclotomic field Q_n are given by linear extension of the maps $*k: \ e_n \mapsto e_n^k$ with $1 \leq k < n$ and `Gcd(` n, k `) = 1` (see 13.11). Note that this action is not equal to exponentiation of cyclotomics, i.e., in general z^{*k} is different from z^k:

```
gap> ( E(5) + E(5)^4 )^2; GaloisCyc( E(5) + E(5)^4, 2 );
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(5)^2+E(5)^3
```

For `Gcd(` n, k `)` $\neq 1$, the map $e_n \mapsto e_n^k$ is not a field automorphism but only a linear map:

```
gap> GaloisCyc( E(5)+E(5)^4, 5 ); GaloisCyc( ( E(5)+E(5)^4 )^2, 5 );
2
-6
```

The **Galois group** $Gal(Q_n, Q)$ of the field extension Q_n/Q is isomorphic to the group $(Z/nZ)^*$ of prime residues modulo n, via the isomorphism

$$\begin{array}{ccc} (Z/nZ)^* & \rightarrow & Gal(Q_n, Q) \\ k & \mapsto & (z \mapsto z^{*k}) \end{array} ,$$

thus the Galois group of the field extension Q_n/L with $L \subseteq Q_n$ which is simply the factor group of $Gal(Q_n, Q)$ modulo the stabilizer of L, and the Galois group of L/L' which is the subgroup in this group that stabilizes L', are easily described in terms of $(Z/nZ)^*$ (Generators of $(Z/nZ)^*$ can be computed using 15.7 `GeneratorsPrimeResidues`.).

The Galois group of a field extension can be computed using 6.7 `GaloisGroup`:

```
gap> f:= NF( [ EY(48) ] );
NF(48,[ 1, 47 ])
gap> g:= GaloisGroup( f );
Group( NFAutomorphism( NF(48,[ 1, 47 ]) , 17 ), NFAutomorphism( NF(48,
[ 1, 47 ]) , 11 ), NFAutomorphism( NF(48,[ 1, 47 ]) , 17 ) )
gap> Size( g ); IsCyclic( g ); IsAbelian( g );
8
false
true
gap> f.base[1]; g.1; f.base[1] ^ g.1;
E(24)-E(24)^11
NFAutomorphism( NF(48,[ 1, 47 ]) , 17 )
E(24)^17-E(24)^19
gap> Operation( g, NormalBaseNumberField( f ), OnPoints );
Group( (1,6)(2,4)(3,8)(5,7), (1,4,8,5)(2,3,7,6), (1,6)(2,4)(3,8)
(5,7) )
```

The number field automorphism `NFAutomorphism(` F, k `)` maps each element x of F to `GaloisCyc(` x, k `)`, see 13.11.

15.9 ZumbroichBase

`ZumbroichBase(` n, m `)`

returns the set of exponents i where e_n^i belongs to the base $\mathcal{B}_{n,m}$ of the field extension Q_n/Q_m; for that, n and m must be positive integers where m divides n.

$\mathcal{B}_{n,m}$ is defined as follows:

Let P denote the set of prime divisors of n, $n = \prod_{p\in P} p^{\nu_p}$, $m = \prod_{p\in P} p^{\mu_p}$ with $\mu_p \leq \nu_p$, and $\{e_{n_1}^j\}_{j\in J} \otimes \{e_{n_2}^k\}_{k\in K} = \{e_{n_1}^j \cdot e_{n_2}^k\}_{j\in J,k\in K}$.

Then

$$\mathcal{B}_{n,m} = \bigotimes_{p\in P} \bigotimes_{k=\mu_p}^{\nu_p-1} \{e_{p^{k+1}}^j\}_{j\in J_{k,p}} \quad \text{where} \quad J_{k,p} = \begin{cases} \{0\} & ; \quad k=0, p=2 \\ \{0,1\} & ; \quad k>0, p=2 \\ \{1,\ldots,p-1\} & ; \quad k=0, p\neq 2 \\ \{-\frac{p-1}{2},\ldots,\frac{p-1}{2}\} & ; \quad k>0, p\neq 2 \end{cases} .$$

$\mathcal{B}_{n,1}$ is equal to the base $\mathcal{B}(Q_n)$ of Q_n over the rationals given in [Zum89] (Note that the notation here is slightly different from that there.).

$\mathcal{B}_{n,m}$ consists of roots of unity, it is an integral base (that is, the integral elements in Q_n have integral coefficients, see 13.2), it is a normal base for squarefree n and closed under complex conjugation for odd n.

```
gap> ZumbroichBase( 15, 1 ); ZumbroichBase( 12, 3 );
[ 1, 2, 4, 7, 8, 11, 13, 14 ]
[ 0, 3 ]
gap> ZumbroichBase( 10, 2 ); ZumbroichBase( 32, 4 );
[ 2, 4, 6, 8 ]
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

15.10 Integral Bases for Number Fields

`LenstraBase(` *n*`,` *stabilizer*`,` *super* `)`

returns a list $[b_1, b_2, \ldots, b_m]$ of lists, each b_i consisting of integers such that the elements $\sum_{j\in b_i} \mathtt{E(n)}^j$ form an integral base of the number field `NF(` *n*`,` *stabilizer* `)`, see 15.3.

super is a list representing a supergroup of the group described by the list *stabilizer*; the base is chosen such that the group of *super* acts on it, as far as this is possible.

Note: The b_i are in general not sets, since for *stabilizer* `=` *super*, $b_i[1]$ is always an element of `ZumbroichBase(` *N*`, 1 )`; this is used by `NF` (see 15.3) and `Coefficients` (see 15.12).

stabilizer must not contain the stabilizer of a proper cyclotomic subfield of Q_n.

```
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 19 ] );           # a base of
[ [ 1, 19 ], [ 8 ], [ 11, 17 ], [ 16 ] ]                # Q_3(√6),
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 5, 19, 23 ] );    # another one
[ [ 1, 19 ], [ 5, 23 ], [ 8 ], [ 16 ] ]
gap> LenstraBase( 15, [ 1, 4 ], PrimeResidues( 15 ) ); # normal base of
[ [ 1, 4 ], [ 2, 8 ], [ 7, 13 ], [ 11, 14 ] ]           # Q_3(√5)
```

15.11 NormalBaseNumberField

`NormalBaseNumberField(` F `)`
`NormalBaseNumberField(` F`,` x `)`

returns a list of cyclotomics which form a normal base of the number field F (see 15.3), i.e. a vector space base of the field F over its subfield F`.field` which is closed under the action of the Galois group F`.galoisGroup` of the field extension.

The normal base is computed as described in [Art68]: Let Φ denote the polynomial of a field extension L/L', Φ' its derivative and α one of its roots; then for all except finitely many elements $z \in L'$, the conjugates of $\frac{\Phi(z)}{(z-\alpha)\cdot\Phi'(\alpha)}$ form a normal base of L/L'.

When `NormalBaseNumberField(` F `)` is called, z is chosen as integer, starting with 1, `NormalBaseNumberField(` F`,` x `)` starts with $z = x$, increasing by one, until a normal base is found.

```
gap> NormalBaseNumberField( CF( 5 ) );
[ -E(5), -E(5)^2, -E(5)^3, -E(5)^4 ]
gap> NormalBaseNumberField( CF( 8 ) );
[ 1/4-2*E(8)-E(8)^2-1/2*E(8)^3, 1/4-1/2*E(8)+E(8)^2-2*E(8)^3,
  1/4+2*E(8)-E(8)^2+1/2*E(8)^3, 1/4+1/2*E(8)+E(8)^2+2*E(8)^3 ]
```

15.12 Coefficients for Number Fields

`Coefficients(` z `)`
`Coefficients(` F`,` z `)`

return the coefficient vector *cfs* of z with respect to a particular base B, i.e., we have z `=` *cfs* `*` B. If z is the only argument, B is the default base of the default field of z (see 15.5), otherwise F must be a number field containing z, and we have B `=` F`.base`.

The **default base** of a number field is defined as follows:

For the field extension Q_n/Q_m (i.e. both F and F`.field` are cyclotomic fields), B is the base $\mathcal{B}_{n,m}$ described in 15.9. This is an integral base which is closely related to the internal representation of cyclotomics, thus the coefficients are easy to compute, using only the `zumbroichbase` fields of F and F`.field`.

For the field extension L/Q where L is not a cyclotomic field, B is the integral base described in 15.10 that consists of orbitsums on roots of unity. The computation of coefficients requires the field F`.coeffslist`.

in future: replace Q by Q_m

In all other cases, B `= NormalBaseNumberField(` F `)`. Here, the coefficients of z with respect to B are computed using F`.coeffslist` and F`.coeffsmat`.

If F`.base` is not the default base of F, the coefficients with respect to the default base are multiplied with F`.basechangemat`. The only possibility where it is allowed to prescribe a base is when the field is constructed (see 15.3, 15.4).

```
gap> F:= NF( [ ER(3), EB(7) ] ) / NF( [ ER(3) ] );
NF(84,[ 1, 11, 23, 25, 37, 71 ])/NF(12,[ 1, 11 ])
gap> Coefficients( F, ER(3) ); Coefficients( F, EB(7) );
```

```
[ -E(12)^7+E(12)^11, -E(12)^7+E(12)^11 ]
[ 11*E(12)^4+7*E(12)^7+11*E(12)^8-7*E(12)^11,
  -10*E(12)^4-7*E(12)^7-10*E(12)^8+7*E(12)^11 ]
gap> G:= CF( 8 ); H:= CF( 0, NormalBaseNumberField( G ) );
CF(8)
CF( 0,[ 1/4-2*E(8)-E(8)^2-1/2*E(8)^3, 1/4-1/2*E(8)+E(8)^2-2*E(8)^3,
  1/4+2*E(8)-E(8)^2+1/2*E(8)^3, 1/4+1/2*E(8)+E(8)^2+2*E(8)^3 ])
gap> Coefficients( G, ER(2) ); Coefficients( H, ER(2) );
[ 0, 1, 0, -1 ]
[ -1/3, 1/3, 1/3, -1/3 ]
```

15.13 Domain Functions for Number Fields

The following functions of `FieldOps` (see chapter 6) are overlaid in `NumberFieldOps`:

`/`, `Coefficients`, `Conjugates`, `GaloisGroup`, `in`, `Intersection`, `Norm`, `Order`, `Print`, `Random`, `Trace`.

The following functions of `NumberFieldOps` are overlaid in `CyclotomicFieldOps`:

`Coefficients`, `Conjugates`, `in`, `Norm`, `Print`, `Trace`.

Chapter 16

Algebraic extensions of fields

If we adjoin a root α of an irreducible polynomial $p \in K[x]$ to the field K we get an **algebraic extension** $K(\alpha)$, which is again a field. By Kronecker's construction, we may identify $K(\alpha)$ with the factor ring $K[x]/(p)$, an identification that also provides a method for computing in these extension fields.

Currently GAP3 only allows extension fields of fields K, when K itself is not an extension field.

As it is planned to modify the representation of field extensions to unify vector space structures and to speed up computations, **All information in this chapter is subject to change in future versions**.

16.1 AlgebraicExtension

`AlgebraicExtension(` *pol* `)`

constructs the algebraic extension L corresponding to the polynomial *pol*. *pol* must be an irreducible polynomial defined over a "defining"field K. The elements of K are embedded into L in the canonical way. As L is a field, all field functions are applicable to L. Similarly, all field element functions apply to the elements of L.

L is considered implicitely to be a field over the subfield K. This means, that functions like `Trace` and `Norm` relative to subfields are not supported.

```
gap> x:=X(Rationals);;x.name:="x";;
gap> p:=x^4+3*x^2+1;
x^4 + 3*x^2 + 1
gap> e:=AlgebraicExtension(p);
AlgebraicExtension(Rationals,x^4 + 3*x^2 + 1)
gap> e.name:="e";;
gap> IsField(e);
true
gap> y:=X(GF(2));;y.name:="y";;
gap> q:=y^2+y+1;
Z(2)^0*(y^2 + y + 1)
gap> f:=AlgebraicExtension(q);
AlgebraicExtension(GF(2),Z(2)^0*(y^2 + y + 1))
```

16.2 IsAlgebraicExtension

`IsAlgebraicExtension(` *D* `)`

`IsAlgebraicExtension` returns `true` if the object *D* is an algebraic field extension and `false` otherwise.

More precisely, `IsAlgebraicExtension` tests whether *D* is an algebraic field extension record (see 16.11). So, for example, a matrix ring may in fact be a field extension, yet `IsAlgebraicExtension` would return `false`.

```
gap> IsAlgebraicExtension(e);
true
gap> IsAlgebraicExtension(Rationals);
false
```

16.3 RootOf

`RootOf(` *pol* `)`

returns a root of the irreducible polynomial *pol* as element of the corresponding extension field `AlgebraicExtension(`*pol*`)`. This root is called the **primitive element** of this extension.

```
gap> r:=RootOf(p);
RootOf(x^4 + 3*x^2 + 1)
gap> r.name:="alpha";;
```

16.4 Algebraic Extension Elements

According to Kronecker's construction, the elements of an algebraic extension are considered to be polynomials in the primitive element. Unless they are already in the defining field (in which case they are represented as elements of this field), they are represented by records in GAP3 (see 16.12). These records contain a representation a polynomial in the primitive element. The extension corresponding to this primitive element is the default field for the algebraic element.

The usual field operations are applicable to algebraic elements.

```
gap> r^3/(r^2+1);
-1*alpha^3-1*alpha
gap> DefaultField(r^2);
e
```

16.5 Set functions for Algebraic Extensions

As algebraic extensions are fields, all set theoretic functions are applicable to algebraic elements. The following two routines are treated specially:

`in`

tests, whether a given object is contained in an algebraic extension. The base field is embedded in the natural way into the extension. Two extensions are considered to be distinct, even if the minimal polynomial of one has a root in the other one.

```
gap> r in e;5 in e;
true
true
gap> p1:=Polynomial(Rationals,MinPol(r^2));
x^2 + 3*x + 1
gap> r2:=RootOf(p1);
RootOf(x^2 + 3*x + 1)
gap> r2 in e;
false
```

`Random`

A random algebraic element is computed by taking a linear combination of the powers of the primitive element with random coefficients from the ground field.

```
gap> ran:=Random(e);
-1*alpha^3-4*alpha^2
```

16.6 IsNormalExtension

`IsNormalExtension(`*L*`)`

An algebraic extension field is called a **normal extension**, if it is a splitting field of the defining polynomial. The second version returns whether L is a normal extension of K. The first version returns whether L is a normal extension of its definition field.

```
gap> IsNormalExtension(e);
true
gap> p2:=x^4+x+1;;
gap> e2:=AlgebraicExtension(p2);
AlgebraicExtension(Rationals,x^4 + x + 1)
gap> IsNormalExtension(e2);
false
```

16.7 MinpolFactors

`MinpolFactors(` *L* `)`

returns the factorization of the defining polynomial of L over L.

```
gap> X(e).name:="X";;
gap> MinpolFactors(e);
[ X + (-1*alpha), X + (-1*alpha^3-3*alpha), X + (alpha),
  X + (alpha^3+3*alpha) ]
```

16.8 GaloisGroup for Extension Fields

`GaloisGroup(` *L* `)`

returns the Galois group of the field L if L is a normal extension and issues an error if not. The Galois group is a group of extension automorphisms (see 16.9).

The computation of a Galois group is computationally relatively hard, and can take significant time.

```
gap> g:=GaloisGroup(f);
Group( ExtensionAutomorphism(AlgebraicExtension(GF(2),Z(2)^0*(y^
2 + y + 1)),RootOf(Z(2)^0*(y^2 + y + 1))+Z(2)^0) )
gap> h:=GaloisGroup(e);
Group( ExtensionAutomorphism(e,alpha^3+
3*alpha), ExtensionAutomorphism(e,-1*alpha), ExtensionAutomorphism(e,
-1*alpha^3-3*alpha) )
gap> Size(h);
4
gap> AbelianInvariants(h);
[ 2, 2 ]
```

16.9 ExtensionAutomorphism

`ExtensionAutomorphism(` L, *img* `)`

is the automorphism of the extension L, that maps the primitive root of L to *img*. As it is a field automorphism, section 6.13 applies.

16.10 Field functions for Algebraic Extensions

As already mentioned, algebraic extensions are fields. Thus all field functions like `Norm` and `Trace` are applicable.

```
gap> Trace(r^4+2*r);
14
gap> Norm(ran);
305
```

`DefaultField` always returns the algebraic extension, which contains the primitive element by which the number is represented, see 16.4.

```
gap> DefaultField(r^2);
e
```

As subfields are not yet supported, `Field` will issue an error, if several elements are given, or if the element is not a primitive element for its default field.

You can create a polynomial ring over an algebraic extension to which all functions described in 19.22 can be applied, for example you can factor polynomials. Factorization is done — depending on the polynomial — by factoring the squarefree norem or using a hensel lift (with possibly added lattice reduction) as described in [Abb89], using bounds from [BTW93].

```
gap> X(e).name:="X";;
gap> p1:=EmbeddedPolynomial(PolynomialRing(e),p1);
X^2 + 3*X + 1
gap> Factors(p1);
[ X + (-1*alpha^2), X + (alpha^2+3) ]
```

16.11 Algebraic Extension Records

Since every algebraic extension is a field, it is represented as a record. This record contains all components, a field record will contain (see 6.17). Additionally, it contains the components `isAlgebraicExtension`, `minpol`, `primitiveElm` and may contain the components `isNormalExtension`, `minpolFactors` and `galoisType`.

`isAlgebraicExtension`
: is always `true`. This indicates that F is an algebraic extension.

`minpol`
: is the defining polynomial of F.

`primitiveElm`
: contains `RootOf(`F`.minpol)`.

`isNormalExtension`
: indicates, whether F is a normal extension field.

`minpolFactors`
: contains a factorization of F`.minpol` over F.

`galoisType`
: contains the Galois type of the normal closure of F. See section 16.16.

16.12 Extension Element Records

Elements of an algebraic extension are represented by a record. The record for the element e of L contains the components `isAlgebraicElement`, `domain` and `coefficients`:

`isAlgebraicElement`
: is always `true`, and indicates, that e is an algebraic element.

`domain`
: contains L.

`coefficients`
: contains the coefficients of e as a polynomial in the primitive root of L.

16.13 IsAlgebraicElement

`IsAlgebraicElement(` *obj* `)`

returns `true` if obj is an algebraic element, i.e., an element of an algebraic extension, that is not in the defining field, and `false` otherwise.

```
gap> IsAlgebraicElement(r);
true
gap> IsAlgebraicElement(3);
false
```

16.14 Algebraic extensions of the Rationals

The following sections describe functions that are specific to algebraic extensions of $\mathbb{Q}$.

16.15 DefectApproximation

`DefectApproximation(` *L* `)`

computes a multiple of the defect of the basis of L, given by the powers of the primitive element. The **defect** indicates, which denominator is necessary in the coefficients, to express algebraic integers in L as a linear combination of the base of L. `DefectApproximation` takes the maximal square in the discriminant as a first approximation, and then uses Berwicks and Hesses method (see [Bra89]) to improve this approximation. The number returned is not neccessarily the defect, but may be a proper multiple of it.

```
gap> DefectApproximation(e);
1
```

16.16 GaloisType

`GaloisType(` *L* `)`
`Galois(` *f* `)`

The first version returns the number of the permutation isomorphism type of the Galois group of the normal closure of L, considered as a transitive permutation group of the roots of the defining polynomial (see 38.6). The second version returns the Galois type of the splitting field of f. Identification is done by factoring appropriate Galois resolvents as proposed in [MS85]. This function is provided for rational polynomials of degree up to 15. However, it may be not feasible to call this function for polynomials of degree 14 or 15, as the involved computations may be enormous. For some polynomials of degree 14, a complete discrimination is not yet possible, as it would require computations, that are not feasible with current factoring methods.

```
gap> GaloisType(e);
2
gap> TransitiveGroup(e.degree,2);
E(4) = 2[x]2
```

16.17 ProbabilityShapes

`ProbabilityShapes(` *pol* `)`

returns a list of numbers, which contains most likely the isomorphism type of the galois group of *pol* (see 16.16). This routine only applies the cycle structure test according to Tschebotareff's theorem. Accordingly, it is very fast, but the result is not guaranteed to be correct.

```
gap> ProbabilityShapes(e.minpol);
[ 2 ]
```

16.18 DecomPoly

`DecomPoly(` *pol* `)`
`DecomPoly(` *pol*`, "all")`

returns an ideal decomposition of the polynomial *pol*. An ideal decomposition is given by two polynomials g and h, such that *pol* divides $(g \circ h)$. By the Galois correspondence any ideal decomposition corresponds to a block system of the Galois group. The polynomial g defines a subfield $K(\beta)$ of $K(\alpha)$ with $h(\alpha) = \beta$. The first form finds one ideal decomposition, while the second form finds all possible different ideal decompositions (i.e. all subfields).

```
gap> d:=DecomPoly(e.minpol);
[ x^2 + 5, x^3 + 4*x ]
gap> p:=x^6+108;;
gap> d:=DecomPoly(p,"all");
[ [ x^2 + 108, x^3 ], [ x^3 + 108, x^2 ],
  [ x^3 - 186624, x^5 + 6*x^2 ], [ x^3 + 186624, x^5 - 6*x^2 ] ]
gap> Value(d[1][1],d[1][2]);
x^6 + 108
```

Chapter 17

Unknowns

Sometimes the result of an operation does not allow further computations with it. In many cases, then an error is signalled, and the computation is stopped.

This is not appropriate for some applications in character theory. For example, if a character shall be induced up (see 51.22) but the subgroup fusion is only a parametrized map (see chapter 52), there are positions where the value of the induced character are not known, and other values which are determined by the fusion map:

```
gap> m11:= CharTable( "M11" );; m12:= CharTable( "M12" );;
gap> fus:= InitFusion( m11, m12 );
[ 1, [ 2, 3 ], [ 4, 5 ], [ 6, 7 ], 8, [ 9, 10 ], [ 11, 12 ],
  [ 11, 12 ], [ 14, 15 ], [ 14, 15 ] ]
gap> Induced(m11,m12,Sublist(m11.irreducibles,[ 6 .. 9 ]),fus);
#I Induced: subgroup order not dividing sum in character 1 at class 4
#I Induced: subgroup order not dividing sum in character 1 at class 5
#I Induced: subgroup order not dividing sum in character 1 at class 14
#I Induced: subgroup order not dividing sum in character 1 at class 15
#I Induced: subgroup order not dividing sum in character 2 at class 4
#I Induced: subgroup order not dividing sum in character 2 at class 5
#I Induced: subgroup order not dividing sum in character 2 at class 14
#I Induced: subgroup order not dividing sum in character 2 at class 15
#I Induced: subgroup order not dividing sum in character 3 at class 2
#I Induced: subgroup order not dividing sum in character 3 at class 3
#I Induced: subgroup order not dividing sum in character 3 at class 4
#I Induced: subgroup order not dividing sum in character 3 at class 5
#I Induced: subgroup order not dividing sum in character 3 at class 9
#I Induced: subgroup order not dividing sum in character 3 at class 10
#I Induced: subgroup order not dividing sum in character 4 at class 2
#I Induced: subgroup order not dividing sum in character 4 at class 3
#I Induced: subgroup order not dividing sum in character 4 at class 6
#I Induced: subgroup order not dividing sum in character 4 at class 7
#I Induced: subgroup order not dividing sum in character 4 at class 11
#I Induced: subgroup order not dividing sum in character 4 at class 12
```

```
#I Induced: subgroup order not dividing sum in character 4 at class 14
#I Induced: subgroup order not dividing sum in character 4 at class 15
[ [ 192, 0, 0, Unknown(9), Unknown(12), 0, 0, 2, 0, 0, 0, 0, 0,
      Unknown(15), Unknown(18) ],
  [ 192, 0, 0, Unknown(27), Unknown(30), 0, 0, 2, 0, 0, 0, 0, 0,
      Unknown(33), Unknown(36) ],
  [ 528, Unknown(45), Unknown(48), Unknown(51), Unknown(54), 0, 0,
      -2, Unknown(57), Unknown(60), 0, 0, 0, 0, 0 ],
  [ 540, Unknown(75), Unknown(78), 0, 0, Unknown(81), Unknown(84), 0,
      0, 0, Unknown(87), Unknown(90), 0, Unknown(93), Unknown(96) ] ]
```

For this and other situations, in GAP3 there is the data type **unknown**. Objects of this type, further on called **unknowns**, may stand for any cyclotomic (see 13).

Unknowns are parametrized by positive integers. When a GAP3 session is started, no unknowns do exist.

The only ways to create unknowns are to call 17.1 `Unknown` or a function that calls it, or to do arithmetical operations with unknowns (see 17.4).

Two properties should be noted:

Lists of cyclotomics and unknowns are no vectors, so cannot be added or multiplied like vectors; as a consequence, unknowns never occur in matrices.

GAP3 objects which are printed to files will contain fixed unknowns, i.e., function calls `Unknown(` *n* `)` instead of `Unknown()`, so be careful to read files printed in different sessions, since there may be the same unknown at different places.

The rest of this chapter contains informations about the unknown constructor (see 17.1), the characteristic function (see 17.2), and comparison of and arithmetical operations for unknowns (see 17.3, 17.4); more is not yet known about unknowns.

17.1 Unknown

`Unknown()`
`Unknown(` *n* `)`

`Unknown()` returns a new unknown value, i.e. the first one that is larger than all unknowns which exist in the actual GAP3 session.

`Unknown(` *n* `)` returns the *n*-th unknown; if it did not exist already, it is created.

```
gap> Unknown(); Unknown(2000); Unknown();
Unknown(97)       # There were created already 96 unknowns.
Unknown(2000)
Unknown(2001)
```

17.2 IsUnknown

`IsUnknown(` *obj* `)`

returns `true` if *obj* is an object of type unknown, and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsUnknown( Unknown ); IsUnknown( Unknown() );
```

```
false
true
gap> IsUnknown( Unknown(2) );
true
```

17.3 Comparisons of Unknowns

To compare unknowns with other objects, the operators `<`, `<=`, `=`, `>=`, `>` and `<>` can be used. The result will be `true` if the first operand is smaller, smaller or equal, equal, larger or equal, larger, or inequal, respectively, and `false` otherwise.

We have `Unknown(` *n* `) >= Unknown(` *m* `)` if and only if *n* `>=` *m* holds; unknowns are larger than cyclotomics and finite field elements, unknowns are smaller than all objects which are not cyclotomics, finite field elements or unknowns.

```
gap> Unknown() >= Unknown();
false
gap> Unknown(2) < Unknown(3);
true
gap> Unknown() > 3;
true
gap> Unknown() > Z(8);
false
gap> Unknown() > E(3);
true
gap>  Unknown() > [];
false
```

17.4 Operations for Unknowns

The operators `+`, `-`, `*` and `/` are used for addition, subtraction, multiplication and division of unknowns and cyclotomics. The result will be a new unknown except in one of the following cases:

Multiplication with zero yields zero, and multiplication with one or addition of zero yields the old unknown.

```
gap> Unknown() + 1; Unknown(2) + 0; last * 3; last * 1; last * 0;
Unknown(2010)
Unknown(2)
Unknown(2011)
Unknown(2011)
0
```

Note that division by an unknown causes an error, since an unknown might stand for zero.

Chapter 18

Finite Fields

Finite fields comprise an important algebraic domain. The elements in a field form an additive group and the nonzero elements form a multiplicative group. For every prime power q there exists a unique field of size q up to isomorphism. GAP3 supports finite fields of size at most 2^{16}.

The first section in this chapter describes how you can enter elements of finite fields and how GAP3 prints them (see 18.1).

The next sections describe the operations applicable to finite field elements (see 18.2 and 18.3).

The next section describes the function that tests whether an object is a finite field element (see 18.4).

The next sections describe the functions that give basic information about finite field elements (see 18.5, 18.6, and 18.7).

The next sections describe the functions that compute various other representations of finite field elements (see 18.8 and 18.9).

The next section describes the function that constructs a finite field (see 18.10).

Finite fields are domains, thus all set theoretic functions are applicable to them (see chapter 4 and 18.12).

Finite fields are of course fields, thus all field functions are applicable to them and to their elements (see chapter 6 and 18.13).

All functions are in `LIBNAME/"finfield.g"`.

18.1 Finite Field Elements

`Z(` *p*^*d* `)`

The function `Z` returns the designated generator of the multiplicative group of the finite field with *p*^*d* elements. p must be a prime and *p*^*d* must be less than or equal to $2^{16} = 65536$.

The root returned by `Z` is a generator of the multiplicative group of the finite field with p^d elements, which is cyclic. The order of the element is of course $p^d - 1$. The $p^d - 1$ different powers of the root are exactly the nonzero elements of the finite field.

Thus all nonzero elements of the finite field with *p*`^`*d* elements can be entered as `Z(`*p*`^`*d*`)^`*i*. Note that this is also the form that GAP3 uses to output those elements.

The additive neutral element is `0*Z(`*p*`)`. It is different from the integer `0` in subtle ways. First `IsInt( 0*Z(`*p*`) )` (see 10.5) is `false` and `IsFFE( 0*Z(`*p*`) )` (see 18.4) is `true`, whereas it is just the other way around for the integer 0.

The multiplicative neutral element is `Z(`*p*`)^0`. It is different from the integer `1` in subtle ways. First `IsInt( Z(`*p*`)^0 )` (see 10.5) is `false` and `IsFFE( Z(`*p*`)^0 )` (see 18.4) is `true`, whereas it is just the other way around for the integer 1. Also `1+1` is `2`, whereas, e.g., `Z(2)^0 + Z(2)^0` is `0*Z(2)`.

The various roots returned by `Z` for finite fields of the same characteristic are compatible in the following sense. If the field $GF(p^n)$ is a subfield of the field $GF(p^m)$, i.e., n divides m, then $Z(p^n) = Z(p^m)^{(p^m-1)/(p^n-1)}$. Note that this is the simplest relation that may hold between a generator of $GF(p^n)$ and $GF(p^m)$, since $Z(p^n)$ is an element of order $p^m - 1$ and $Z(p^m)$ is an element of order $p^n - 1$. This is achieved by choosing $Z(p)$ as the smallest primitive root modulo p and $Z(p^n)$ as a root of the n-th Conway polynomial of characteristic p. Those polynomials where defined by J.H. Conway and computed by R.A. Parker.

```
gap> z := Z(16);
Z(2^4)
gap> z*z;
Z(2^4)^2
```

18.2 Comparisons of Finite Field Elements

z1 `=` *z2*
z1 `<>` *z2*

The equality operator `=` evaluates to `true` if the two elements in a finite field *z1* and *z2* are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the two elements in a finite finite field *z1* and *z2* are not equal and to `false` otherwise.

Note that the integer 0 is not equal to the zero element in any finite field. There comparisons *z* `= 0` will always evaluate to `false`. Use *z* `= 0*`*z* instead, or even better *z* `=` *F*`.zero`, where *F* is the field record for a finite field of the same characteristic.

```
gap> Z(2^4)^10 = Z(2^4)^25;
true     # Z(2^4) has order 15
gap> Z(2^4)^10 = Z(2^2)^2;
true     # shows the embedding of GF(4) into GF(16)
gap> Z(2^4)^10 = Z(3);
false
```

z1 `<` *z2*
z1 `<=` *z2*
z1 `>` *z2*
z1 `>=` *z2*

The operators `<`, `<=`, `>`, and `=>` evaluate to `true` if the element in a finite field *z1* is less than, less than or equal to, greater than, and greater than or equal to the element in a finite field *z2*.

Elements in finite fields are ordered as follows. If the two elements lie in fields of different characteristics the one that lies in the field with the smaller characteristic is smaller. If the two elements lie in different fields of the same characteristic the one that lies in the smaller field is smaller. If the two elements lie in the same field and one is the zero and the other is not, the zero element is smaller. If the two elements lie in the same field and both are nonzero, and are represented as $Z(p^d)^{i_1}$ and $Z(p^d)^{i_2}$ respectively, then the one with the smaller i is smaller.

You can compare elements in a finite field with objects of other types. Integers, rationals, and cyclotomics are smaller than elements in finite fields, all other objects are larger. Note especially that the integer 0 is smaller than the zero in every finite field.

```
gap> Z(2) < Z(3);
true
gap> Z(2) < Z(4);
true
gap> 0*Z(2) < Z(2);
true
gap> Z(4) < Z(4)^2;
true
gap> 0 < 0*Z(2);
true
gap> Z(4) < [ Z(4) ];
true
```

18.3 Operations for Finite Field Elements

z1 + *z2*
z1 - *z2*
z1 * *z2*
z1 / *z2*

The operators `+`, `-`, `*` and `/` evaluate to the sum, difference, product, and quotient of the two finite field elements *z1* and *z2*, which must lie in fields of the same characteristic. For the quotient `/` *z2* must of course be nonzero. The result must of course lie in a finite field of size less than or equal to 2^{16}, otherwise an error is signalled.

Either operand may also be an integer i. If i is zero it is taken as the zero in the finite field, i.e., *F*`.zero`, where F is a field record for the finite field in which the other operand lies. If i is positive, it is taken as i-fold sum *F*`.one+`*F*`.one+..+`*F*`.one`. If i is negative it is taken as the additive inverse of `-`i.

```
gap> Z(8) + Z(8)^4;
Z(2^3)^2
gap> Z(8) - 1;
Z(2^3)^3
gap> Z(8) * Z(8)^6;
Z(2)^0
gap> Z(8) / Z(8)^6;
Z(2^3)^2
gap> -Z(9);
```

```
Z(3^2)^5
```

z ^ *i*

The powering operator ^ returns the i-th power of the element in a finite field z. i must be an integer. If the exponent i is zero, z^i is defined as the one in the finite field, even if z is zero; if i is positive, z^i is defined as the i-fold product z*z*..*z; finally, if i is negative, z^i is defined as (1/z)^-i. In this case z must of course be nonzero.

```
gap> Z(4)^2;
Z(2^2)^2
gap> Z(4)^3;
Z(2)^0      # is in fact 1
gap> (0*Z(4))^0;
Z(2)^0
```

18.4 IsFFE

`IsFFE(` *obj* `)`

`IsFFE` returns `true` if *obj*, which may be an object of an arbitrary type, is an element in a finite field and `false` otherwise. Will signal an error if *obj* is an unbound variable.

Note that integers, even though they can be multiplied with elements in finite fields, are not considered themselves elements in finite fields. Therefore `IsFFE` will return `false` for integer arguments.

```
gap> IsFFE( Z(2^4)^7 );
true
gap> IsFFE( 5 );
false
```

18.5 CharFFE

`CharFFE(` *z* `)` or `CharFFE(` *vec* `)` or `CharFFE(` *mat* `)`

`CharFFE` returns the characteristic of the finite field F containing the element z, respectively all elements of the vector *vec* over a finite field (see 32), or matrix *mat* over a finite field (see 34).

```
gap> CharFFE( Z(16)^7 );
2
gap> CharFFE( Z(16)^5 );
2
gap> CharFFE( [ Z(3), Z(27)^11, Z(9)^3 ] );
3
gap> CharFFE( [ [ Z(5), Z(125)^3 ], [ Z(625)^13, Z(5) ] ] );
Error, CharFFE: <z> must be a finite field element, vector, or matrix
# The smallest finite field which contains all four of these elements
# is too large for GAP3
```

18.6 DegreeFFE

`DegreeFFE( ` *z* ` )` or `DegreeFFE( ` *vec* ` )` or `DegreeFFE( ` *mat* ` )`

`DegreeFFE` returns the degree of the smallest finite field F containing the element z, respectively all elements of the vector *vec* over a finite field (see 32), or matrix *mat* over a finite field (see 34). For vectors and matrices, an error is signalled if the smallest finite field containing all elements of the vector or matrix has size larger than 2^{16}.

```
gap> DegreeFFE( Z(16)^7 );
4
gap> DegreeFFE( Z(16)^5 );
2
gap> DegreeFFE( [ Z(3), Z(27)^11, Z(9)^3 ] );
6
gap> DegreeFFE( [ [ Z(5), Z(125)^3 ], [ Z(625)^13, Z(5) ] ] );
Error, DegreeFFE: <z> must be a finite field element, vector, or matrix
# The smallest finite field which contains all four of these elements
# is too large for GAP3
```

18.7 OrderFFE

`OrderFFE( ` *z* ` )`

`OrderFFE` returns the order of the element z in a finite field. The order is the smallest positive integer i such that z^i is 1. The order of the zero in a finite field is defined to be 0.

```
gap> OrderFFE( Z(16)^7 );
15
gap> OrderFFE( Z(16)^5 );
3
gap> OrderFFE( Z(27)^11 );
26
gap> OrderFFE( Z(625)^13 );
48
gap> OrderFFE( Z(211)^0 );
1
```

18.8 IntFFE

`IntFFE( ` *z* ` )`

`IntFFE` returns the integer corresponding to the element z, which must lie in a finite prime field. That is `IntFFE` returns the smallest nonnegative integer i such that i `*` z`^0 =` z.

The correspondence between elements from a finite prime field of characteristic p and the integers between 0 and p`-1` is defined by choosing `Z(`p`)` the smallest primitive root mod p (see 11.6).

```
gap> IntFFE( Z(13) );
2
gap> PrimitiveRootMod( 13 );
```

```
2
gap> IntFFE( Z(409) );
21
gap> IntFFE( Z(409)^116 );
311
gap> 21^116 mod 409;
311
```

18.9 LogFFE

`LogFFE(` *z* `)`
`LogFFE(` *z*`,` *r* `)`

In the first form `LogFFE` returns the discrete logarithm of the element z in a finite field with respect to the root `FieldFFE(`*z*`).root`. An error is signalled if z is zero.

In the second form `LogFFE` returns the discrete logarithm of the element z in a finite field with respect to the root r. An error is signalled if z is zero, or if z is not a power of r.

The **discrete logarithm** of an element z with respect to a root r is the smallest nonnegative integer i such that $r^i = z$.

```
gap> LogFFE( Z(409)^116 );
116
gap> LogFFE( Z(409)^116, Z(409)^2 );
58
```

18.10 GaloisField

`GaloisField(` *p*`^`*d* `)`
`GF(` *p*`^`*d* `)`
`GaloisField(` *p*`|`*S*`,` *d*`|`*pol*`|`*bas* `)`
`GF(` *p*`|`*S*`,` *d*`|`*pol*`|`*bas* `)`

`GaloisField` returns a field record (see 6.17) for a finite field. It takes two arguments. The form `GaloisField(`*p*`,`*d*`)`, where p,d are integers, can also be given as `GaloisField(`*p*`^`*d*`)`. `GF` is an abbreviation for `GaloisField`.

The first argument specifies the subfield S over which the new field F is to be taken. It can be a prime or a finite field record. If it is a prime p, the subfield is the prime field of this characteristic. If it is a field record S, the subfield is the field described by this record.

The second argument specifies the extension. It can be an integer, an irreducible polynomial, or a base. If it is an integer d, the new field is constructed as the polynomial extension with the Conway polynomial of degree d over the subfield S. If it is an irreducible polynomial *pol*, in which case the elements of the list *pol* must all lie in the subfield S, the new field is constructed as polynomial extension of the subfield S with this polynomial. If it is a base *bas*, in which case the elements of the list *bas* must be linear independently over the subfield S, the new field is constructed as a linear vector space over the subfield S.

Note that the subfield over which a field was constructed determines over which field the Galois group, conjugates, norm, trace, minimal polynom, and characteristic polynom are computed (see 6.7, 6.12, 6.10, 6.11, 6.8, 6.9, and 18.13).

```
gap> GF( 2^4 );
GF(2^4)
gap> GF( GF(2^4), 2 );
GF(2^8)/GF(2^4)
```

18.11 FrobeniusAutomorphism

`FrobeniusAutomorphism(` *F* `)`

`FrobeniusAutomorphism` returns the Frobenius automorphism of the finite field F as a field homomorphism (see 6.13).

The Frobenius automorphism f of a finite field F of characteristic p is the function that takes each element z of F to its p-th power. Each automorphism of F is a power of the Frobenius automorphism. Thus the Frobenius automorphism is a generator for the Galois group of F (and an appropriate power of it is a generator of the Galois group of F over a subfield S) (see 6.7).

```
gap> f := GF(16);
GF(2^4)
gap> x := FrobeniusAutomorphism( f );
FrobeniusAutomorphism( GF(2^4) )
gap> Z(16) ^ x;
Z(2^4)^2
```

The image of an element z under the i-th power of the Frobenius automorphism f of a finite field F of characteristic p is simply computed by computing the p^i-th power of z. The product of the i-th power and the j-th power of f is the k-th power of f, where k is i`*`j `mod (Size(`F`)-1)`. The zeroth power of f is printed as `IdentityMapping(` *F* `)`.

18.12 Set Functions for Finite Fields

Finite fields are of course domains. Thus all set theoretic functions are applicable to finite fields (see chapter 4). This section gives further comments on the definitions and implementations of those functions for finite fields. All set theoretic functions not mentioned here are not treated specially for finite fields.

`Elements`

The elements of a finite field are computed using the fact that the finite field is a vector space over its prime field.

`in`

The membership test is of course very simple, we just have to test whether the element is a finite field element with `IsFFE`, whether it has the correct characteristic with `CharFFE`, and whether its degree divides the degree of the finite field with `DegreeFFE` (see 18.4, 18.5, and 18.6).

`Random`

A random element of $GF(p^n)$ is computed by computing a random integer i from $[0..p^n-1]$ and returning $0 * Z(p)$ if $i = 0$ and $Z(p^n)^{i-1}$ otherwise.

`Intersection`

The intersection of $GF(p^n)$ and $GF(p^m)$ is the finite field $GF(p^{Gcd(n,m)})$, and is returned as finite field record.

18.13 Field Functions for Finite Fields

Finite fields are, as the name already implies, fields. Thus all field functions are applicable to finite fields and their elements (see chapter 6). This section gives further comments on the definitions and implementations of those functions for finite fields. All domain functions not mentioned here are not treated specially for finite fields.

`Field` and `DefaultField`

Both `Field` and `DefaultField` return the smallest finite field containing the arguments as an extension of the prime field.

`GaloisGroup`

The Galois group of a finite field F of size p^m over a subfield S of size $q = p^n$ is a cyclic group of size m/n. It is generated by the **Frobenius automorphism** that takes every element of F to its q-th power. This automorphism of F leaves exactly the subfield S fixed.

`Conjugates`

According to the above theorem about the Galois group, each element of F has m/n conjugates, $z, z^q, z^{q^2}, ..., z^{q^{m/n-1}}$.

`Norm`

The norm is the product of the conjugates, i.e., z^{p^m-1/p^n-1}. Because we have $Z(p^n) = Z(p^m)^{p^m-1/p^n-1}$, it follows that $Norm(GF(p^m)/GF(p^n), Z(p^m)^i) = Z(p^n)^i$.

Chapter 19

Polynomials

Let R be a commutative ring-with-one. A **(univariate) Laurent polynomial** over R is a sequence $(..., c_{-1}, c_0, c_1, ...)$ of elements of R such that only finitely many are non-zero. For a ring element r of R and polynomials $f = (..., f_{-1}, f_0, f_1, ...)$ and $g = (..., g_{-1}, g_0, g_1, ...)$ we define $f + g = (..., f_{-1} + g_{-1}, f_0 + g_0, f_1 + g_1, ...)$, $r \cdot f = (..., rf_{-1}, rf_0, rf_1, ...)$, and $f * g = (..., s_{-1}, s_0, s_1, ...)$, where $s_k = ... + f_i g_{k-i} + ...$ Note that s_k is well-defined as only finitely many f_i and g_i are non-zero. We call the largest integers $d(f)$, such that $f_{d(f)}$ is non-zero, the **degree** of f, f_i the **i.th coefficient** of f, and $f_{d(f)}$ the leading coefficient of f. If the smallest integer $v(f)$, such that $f_{v(f)}$ is non-zero, is negative, we say that f has a pole of order v at 0, otherwise we say that f has a root of order v at 0. We call R the **base (or coefficient) ring** of f. If $f = (..., 0, 0, 0, ...)$ we set $d(f) = -1$ and $v(f) = 0$.

The set of all Laurent polynomials $L(R)$ over a ring R together with above definitions of $+$ and $*$ is again a ring, the **Laurent polynomial ring** over R, and R is called the **base ring** of $L(R)$. The subset of all polynomials f with non-negative $v(f)$ forms a subring $P(R)$ of $L(R)$, the **polynomial ring** over R. If R is indeed a field then both rings $L(R)$ and $P(R)$ are Euclidean. Note that $L(R)$ and $P(R)$ have different Euclidean degree functions. If f is an element of $P(R)$ then the Euclidean degree of f is simply the degree of f. If f is viewed as an element of $L(R)$ then the Euclidean degree is the difference between $d(f)$ and $v(f)$. The units of $P(R)$ are just the units of R, while the units of $L(R)$ are the polynomials f such that $v(f) = d(f)$ and $f_{d(f)}$ is a unit in R.

GAP3 uses the above definition of polynomials. This definition has some advantages and some drawbacks. First of all, the polynomial $(..., x_0 = 0, x_1 = 1, x_2 = 0, ...)$ is commonly denoted by x and is called an indeterminate over R, $(..., c_{-1}, c_0, c_1, ...)$ is written as $... + c_{-1}x^{-1} + c_0 + c_1 x + c_2 x^2 + ...$, and $P(R)$ as $R[x]$ (note that the way GAP3 outputs a polynomial resembles this definition). But if we introduce a second indeterminate y it is not obvious whether the product xy lies in $(R[x])[y]$, the polynomial ring in y over the polynomial ring in x, in $(R[y])[x]$, in $R[x, y]$, the polynomial ring in two indeterminates, or in $R[y, x]$ (which should be equal to $R[x, y]$). Using the first definition we would define y as indeterminate over $R[x]$ and we know then that xy lies in $(R[x])[y]$.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Rx := LaurentPolynomialRing(Rationals);;
gap> y := Indeterminate(Rx);; y.name := "y";;
```

```
gap> y^2 + x;
y^2 + (x)
gap> last^2;
y^4 + (2*x)*y^2 + (x^2)
gap> last + x;
y^4 + (2*x)*y^2 + (x^2 + x)
gap> (x^2 + x + 1) * y^2 + y + 1;
(x^2 + x + 1)*y^2 + y + (x^0)
gap> x * y;
(x)*y
gap> y * x;
(x)*y
gap> 2 * x;
2*x
gap> x * 2;
2*x
```

Note that GAP3 does **not** embed the base ring of a polynomial into the polynomial ring. The trivial polynomial and the zero of the base ring are always different.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Rx := LaurentPolynomialRing(Rationals);;
gap> y := Indeterminate(Rx);; y.name := "y";;
gap> 0 = 0*x;
false
gap> nx := 0*x;       # a polynomial over the rationals
0*x^0
gap> ny := 0*y;       # a polynomial over a polynomial ring
0*y^0
gap> nx = ny;         # different base rings
false
```

The result `0*x` $\neq$ `0*y` is probably not what you expect or want. In order to compute with two indeterminates over R you must embed x into $R[x][y]$.

```
gap> x  := Indeterminate(Rationals);; x.name := "x";;
gap> Rx := LaurentPolynomialRing(Rationals);;
gap> y  := Indeterminate(Rx);; y.name := "y";;
gap> x  := x * y^0;
x*y^0
gap> 0*x = 0*y;
true
```

The other point which might be startling is that we require the supply of a base ring for a polynomial. But this guarantees that `Factor` gives a predictable result.

```
gap> f5 := GF(5);; f5.name := "f5";;
gap> f25 := GF(25);; f25.name := "f25";;
gap> Polynomial( f5, [3,2,1]*Z(5)^0 );
Z(5)^0*(X(f5)^2 + 2*X(f5) + 3)
gap> Factors(last);
[ Z(5)^0*(X(f5)^2 + 2*X(f5) + 3) ]
```

```
gap> Polynomial( f25, [3,2,1]*Z(5)^0 );
X(f25)^2 + Z(5)*X(f25) + Z(5)^3
gap> Factors(last);
[ X(f25) + Z(5^2)^7, X(f25) + Z(5^2)^11 ]
```

The first sections describe how polynomials are constructed (see 19.2, 19.3, and 19.4).

The next sections describe the operations applicable to polynomials (see 19.5 and 19.6).

The next sections describe the functions for polynomials (see 19.7, 19.12 and 19.11).

The next sections describe functions that construct certain polynomials (see 19.16, 19.17).

The next sections describe the functions for constructing the Laurent polynomial ring $L(R)$ and the polynomial ring $P(R)$ (see 19.18 and 19.20).

The next sections describe the ring functions applicable to Laurent polynomial rings. (see 19.22 and 19.23).

19.1 Multivariate Polynomials

As explained above, each ring R has exactly one indeterminate associated with R. In order to construct a polynomial ring with two indeterminates over R you must first construct the polynomial ring $P(R)$ and then the polynomial ring over $P(R)$.

```
gap> x  := Indeterminate(Integers);; x.name := "x";;
gap> Rx := PolynomialRing(Integers);;
gap> y  := Indeterminate(Rx);; y.name := "y";;
gap> x  := y^0 * x;
x*y^0
gap> f := x^2*y^2 + 3*x*y + x + 4*y;
(x^2)*y^2 + (3*x + 4)*y + (x)
gap> Value( f, 4 );
16*x^2 + 13*x + 16
gap> Value( last, -2 );
54
gap> (-2)^2 * 4^2 + 3*(-2)*4 + (-2) + 4*4;
54
```

We plan to add support for (proper) multivariate polynomials in one of the next releases of GAP3.

19.2 Indeterminate

`Indeterminate(` *R* `)`
`X(` *R* `)`

`Indeterminate` returns the polynomial $(..., x_0 = 0, x_1 = 1, x_2 = 0, ...)$ over R, which must be a commutative ring-with-one or a field.

Note that you can assign a name to the indeterminate, in which case all polynomials over R are printed using this name. Keep in mind that for each ring there is exactly one indeterminate.

```
gap> x := Indeterminate( Integers );; x.name := "x";;
```

```
gap> f := x^10 + 3*x - x^-1;
x^10 + 3*x - x^(-1)
gap> y := Indeterminate( Integers );;     # this is x
gap> y.name := "y";;
gap> f;     # so f is also printed differently from now on
y^10 + 3*y - y^(-1)
```

19.3 Polynomial

`Polynomial(` *R*, *l* `)`
`Polynomial(` *R*, *l*, *v* `)`

l must be a list of coefficients of the polynomial f to be constructed, namely $(..., f_v = l[1], f_{v+1} = l[2], ...)$ over R, which must be a commutative ring-with-one or a field. The default for v is 0. `Polynomial` returns this polynomial f.

For interactive calculation it might by easier to construct the indeterminate over R and construct the polynomial using `^`, `+` and `*`.

```
gap> x := Indeterminate( Integers );;
gap> x.name := "x";;
gap> f := Polynomial( Integers, [1,2,0,0,4] );
4*x^4 + 2*x + 1
gap> g := 4*x^4 + 2*x + 1;
4*x^4 + 2*x + 1
```

19.4 IsPolynomial

`IsPolynomial(` *obj* `)`

`IsPolynomial` returns `true` if *obj*, which can be an object of arbitrary type, is a polynomial and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsPolynomial( 1 );
false
gap> IsPolynomial( Indeterminate(Integers) );
true
```

19.5 Comparisons of Polynomials

f `=` *g*
f `<>` *g*

The equality operator `=` evaluates to `true` if the polynomials f and g are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the polynomials f and g are not equal, and to `false` otherwise.

Note that polynomials are equal if and only if their coefficients **and** their base rings are equal. Polynomials can also be compared with objects of other types. Of course they are never equal.

```
gap> f := Polynomial( GF(5^3), [1,2,3]*Z(5)^0 );
Z(5)^3*X(GF(5^3))^2 + Z(5)*X(GF(5^3)) + Z(5)^0
```

```
gap> x := Indeterminate(GF(25));;
gap> g := 3*x^2 + 2*x + 1;
Z(5)^3*X(GF(5^2))^2 + Z(5)*X(GF(5^2)) + Z(5)^0
gap> f = g;
false
gap> x^0 = Z(25)^0;
false
```

f `<` *g*
f `<=` *g*
f `>` *g*
f `>=` *g*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the polynomial f is less than, less than or equal to, greater than, or greater than or equal to the polynomial g, and to `false` otherwise.

A polynomial f is less than g if $v(f)$ is less than $v(g)$, or if $v(f)$ and $v(g)$ are equal and $d(f)$ is less than $d(g)$. If $v(f)$ is equal to $v(g)$ and $d(f)$ is equal to $d(g)$ the coefficient lists of f and g are compared.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> (1 + x^2 + x^3)*x^3 < (2 + x^2 + x^3);
false
gap> (1 + x^2 + x^4) < (2 + x^2 + x^3);
false
gap> (1 + x^2 + x^3) < (2 + x^2 + x^3);
true
```

19.6 Operations for Polynomials

The following operations are always available for polynomials. The operands must have a common base ring, no implicit conversions are performed.

f `+` *g*

The operator `+` evaluates to the sum of the polynomials f and g, which must be polynomials over a common base ring.

```
gap> f := Polynomial( GF(2), [Z(2), Z(2)] );
Z(2)^0*(X(GF(2)) + 1)
gap> f + f;
0*X(GF(2))^0
gap> g := Polynomial( GF(4), [Z(2), Z(2)] );
X(GF(2^2)) + Z(2)^0
gap> f + g;
Error, polynomials must have the same ring
```

f `+` *scl*
scl `+` *f*

The operator `+` evaluates to the sum of the polynomial f and the scalar *scl*, which must lie in the base ring of f.

```
gap> x := Indeterminate( Integers );; x.name := "x";;
```

```
gap> h := Polynomial( Integers, [1,2,3,4] );
4*x^3 + 3*x^2 + 2*x + 1
gap> h + 1;
4*x^3 + 3*x^2 + 2*x + 2
gap> 1/2 + h;
Error, <l> must lie in the base ring of <r>
```

f - *g*

The operator - evaluates to the difference of the polynomials *f* and *g*, which must be polynomials over a common base ring.

```
gap> x := Indeterminate( Integers );; x.name := "x";;
gap> h := Polynomial( Integers, [1,2,3,4] );
4*x^3 + 3*x^2 + 2*x + 1
gap> h - 2*h;
-4*x^3 - 3*x^2 - 2*x - 1
```

f - *scl*
scl - *f*

The operator - evaluates to the difference of the polynomial *f* and the scalar *scl*, which must lie in the base ring of *f*.

```
gap> x := Indeterminate( Integers );; x.name := "x";;
gap> h := Polynomial( Integers, [1,2,3,4] );
4*x^3 + 3*x^2 + 2*x + 1
gap> h - 1;
4*x^3 + 3*x^2 + 2*x
gap> 1 - h;
-4*x^3 - 3*x^2 - 2*x
```

f * *g*

The operator * evaluates to the product of the two polynomials *f* and *g*, which must be polynomial over a common base ring.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> h := 4*x^3 + 3*x^2 + 2*x + 1;
4*x^3 + 3*x^2 + 2*x + 1
gap> h * h;
16*x^6 + 24*x^5 + 25*x^4 + 20*x^3 + 10*x^2 + 4*x + 1
```

f * *scl*
scl * *f*

The operator * evaluates to the product of the polynomial *f* and the scalar *scl*, which must lie in the base ring of *f*.

```
gap> f := Polynomial( GF(2), [Z(2), Z(2)] );
Z(2)^0*(X(GF(2)) + 1)
gap> f - Z(2);
X(GF(2))
gap> Z(4) - f;
Error, <l> must lie in the base ring of <r>
```

f ^ *n*

The operator ^ evaluates the the n-th power of the polynomial f. If n is negative ^ will try to invert f in the Laurent polynomial ring ring.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> k := x - 1 + x^-1;
x - 1 + x^(-1)
gap> k ^ 3;
x^3 - 3*x^2 + 6*x - 7 + 6*x^(-1) - 3*x^(-2) + x^(-3)
gap> k^-1;
Error, cannot invert <l> in the laurent polynomial ring
```

f / *scl*

The operator / evaluates to the product of the polynomial f and the inverse of the scalar *scl*, which must be invertable in its default ring.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> h := 4*x^3 + 3*x^2 + 2*x + 1;
4*x^3 + 3*x^2 + 2*x + 1
gap> h / 3;
(4/3)*x^3 + x^2 + (2/3)*x + (1/3)
```

scl / *f*

The operator / evaluates to the product of the scalar *scl* and the inverse of the polynomial f, which must be invertable in its Laurent ring.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> 30 / x;
30*x^(-1)
gap> 3 / (1+x);
Error, cannot invert <l> in the laurent polynomial ring
```

f / *g*

The operator / evaluates to the quotient of the two polynomials f and g, if such quotient exists in the Laurent polynomial ring. Otherwise / signals an error.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> f := (1+x+x^2) * (3-x-2*x^2);
-2*x^4 - 3*x^3 + 2*x + 3
gap> f / (1+x+x^2);
-2*x^2 - x + 3
gap> f / (1+x);
Error, cannot divide <l> by <r>
```

19.7 Degree

Degree(*f*)

Degree returns the degree d_f of f (see 19).

Note that this is only equal to the Euclidean degree in the polynomial ring $P(R)$. It is not equal in the Laurent polynomial ring $L(R)$.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
```

```
gap> Degree( x^10 + x^2 + 1 );
10
gap> EuclideanDegree( x^10 + x^2 + 1 );
10      # the default ring is the polynomial ring
gap> Degree( x^-10 + x^-11 );
-10
gap> EuclideanDegree( x^-10 + x^-11 );
1       # the default ring is the Laurent polynomial ring
```

19.8 Valuation

`Valuation(` *f* `)`

`Valuation` returns the valuation *f* (see 19).

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Valuation( x^10 + x^2 + 1 );
0
gap> Valuation( x^10 + x^2);
2
gap> Valuation( x^-10 + x^-11 );
-11
```

19.9 LeadingCoefficient

`LeadingCoefficient(` *f* `)`

`LeadingCoefficient` returns the last non-zero coefficient of *f* (see 19).

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> LeadingCoefficient( 3*x^2 + 2*x + 1);
3
```

19.10 Coefficient

`Coefficient(` *f*`,` *i*`)`

`Coefficient` returns the *i*-th coefficient of *f* (see 19).

for other objects the function looks if *f* has a `Coefficient` method in its operations record and then returns `f.operations.Coefficient(f,i)`.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Coefficient(3*x^2 + 2*x, 2);
3
gap> Coefficient(3*x^2 + 2*x, 1);
2
gap> Coefficient(3*x^2 + 2*x, 0);
0
```

19.11 Value

`Value(` f, w `)`

Let f be a Laurent polynomial $(..., f_{-1}, f_0, f_1, ...)$. Then `Value` returns the finite sum $... + f_{-1}w^{-1} + f_0w^0 + f_1w + ...$

Note that x need not be contained in the base ring of f.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> k := -x + 1;
-x + 1
gap> Value( k, 2 );
-1
gap> Value( k, [[1,1],[0,1]] );
[ [ 0, -1 ], [ 0, 0 ] ]
```

19.12 Derivative

`Derivative(` f `)`

If f is a Laurent polynomial $(..., f_{-1}, f_0, f_1, ...)$, then `Derivative` returns the polynomial $g = (..., g_{i-1} = i * f_i, ...)$.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Derivative( x^10 + x^-11 );
10*x^9 - 11*x^(-12)
gap> y := Indeterminate(GF(5));; y.name := "y";;
gap> Derivative( y^10 + y^-11 );
Z(5)^2*y^(-12)
```

In a second form f is a list interpreted as the coefficients of a polynomial; then `Derivative` returns the coefficients of the derivative polynomial, that is the list `[f[2],2*f[3],..]`.

```
gap> Derivative([1,2,1,2,1,2]);
[ 2, 2, 6, 4, 10 ]
```

19.13 Resultant

`Resultant(` f, g `)`

f and g must be polynomials, not Laurent polynomials. The function returns their resultant.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Resultant(x^3+1,3*x^2);
27
```

19.14 Discriminant

`Discriminant(` f `)`

f must be a polynomial, not a Laurent polynomial. The function returns its discriminant.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> Discriminant(x^3+1);
-27
```

19.15 InterpolatedPolynomial

`InterpolatedPolynomial(` R, x, y `)`

`InterpolatedPolynomial` returns the unique polynomial of degree less than n which has value $y[i]$ at $x[i]$ for all $i = 1, ..., n$, where x and y must be lists of elements of the ring or field R, if such a polynomial exists. Note that the elements in x must be distinct.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> p := InterpolatedPolynomial( Rationals, [1,2,3,4], [3,2,4,1] );
(-4/3)*x^3 + (19/2)*x^2 + (-121/6)*x + 15
gap> List( [1,2,3,4], x -> Value(p,x) );
[ 3, 2, 4, 1 ]
gap> Unbind( x.name );
```

19.16 ConwayPolynomial

`ConwayPolynomial(` p, n `)`

returns the Conway polynomial of the finite field $GF(p^n)$ as polynomial over the Rationals.

The **Conway polynomial** $\Phi_{n,p}$ of $GF(p^n)$ is defined by the following properties.

First define an ordering of polynomials of degree n over $GF(p)$ as follows.

$f = \sum_{i=0}^{n}(-1)^i f_i x^i$ is smaller than $g = \sum_{i=0}^{n}(-1)^i g_i x^i$ if and only if there is an index $m \leq n$ such that $f_i = g_i$ for all $i > m$, and $\tilde{f_m} < \tilde{g_m}$, where $\tilde{c}$ denotes the integer value in $\{0, 1, \ldots, p-1\}$ that is mapped to $c \in GF(p)$ under the canonical epimorphism that maps the integers onto $GF(p)$.

$\Phi_{n,p}$ is primitive over $GF(p)$, that is, it is irreducible, monic, and is the minimal polynomial of a primitive element of $GF(p^n)$ over $GF(p)$.

For all divisors d of n the compatibility condition $\Phi_{d,p}(x^{\frac{p^n-1}{p^m-1}}) \equiv 0 \pmod{\Phi_{n,p}(x)}$ holds.

With respect to the ordering defined above, $\Phi_{n,p}$ shall be minimal.

```
gap> ConwayPolynomial( 7, 3 );
X(Rationals)^3 + 6*X(Rationals)^2 + 4
gap> ConwayPolynomial( 41, 3 );
X(Rationals)^3 + X(Rationals) + 35
```

The global list `CONWAYPOLYNOMIALS` contains Conway polynomials for small values of p and n. Note that the computation of Conway polynomials may be very expensive, especially if n is not a prime.

19.17 CyclotomicPolynomial

`CyclotomicPolynomial(` R, n `)`

returns the n-th cyclotomic polynomial over the field R.

```
gap> CyclotomicPolynomial( GF(2), 6 );
Z(2)^0*(X(GF(2))^2 + X(GF(2)) + 1)
gap> CyclotomicPolynomial( Rationals, 5 );
X(Rationals)^4 + X(Rationals)^3 + X(Rationals)^2 + X(Rationals) + 1
```

In every GAP3 session the computed cyclotomic polynomials are stored in the global list `CYCLOTOMICPOLYNOMIALS`.

19.18 PolynomialRing

`PolynomialRing(` *R* `)`

`PolynomialRing` returns the ring of all polynomials over a field R or ring-with-one R.

```
gap> f2 := GF(2);;
gap> R := PolynomialRing( f2 );
PolynomialRing( GF(2) )
gap> Z(2) in R;
false
gap> Polynomial( f2, [Z(2),Z(2)] ) in R;
true
gap> Polynomial( GF(4), [Z(2),Z(2)] ) in R;
false
gap> R := PolynomialRing( GF(2) );
PolynomialRing( GF(2) )
```

19.19 IsPolynomialRing

`IsPolynomialRing(` *domain* `)`

`IsPolynomialRing` returns `true` if the object *domain* is a ring record, representing a polynomial ring (see 19.18), and `false` otherwise.

```
gap> IsPolynomialRing( Integers );
false
gap> IsPolynomialRing( PolynomialRing( Integers ) );
true
gap> IsPolynomialRing( LaurentPolynomialRing( Integers ) );
false
```

19.20 LaurentPolynomialRing

`LaurentPolynomialRing(` *R* `)`

`LaurentPolynomialRing` returns the ring of all Laurent polynomials over a field R or ring-with-one R.

```
gap> f2 := GF(2);;
gap> R := LaurentPolynomialRing( f2 );
LaurentPolynomialRing( GF(2) )
gap> Z(2) in R;
false
gap> Polynomial( f2, [Z(2),Z(2)] ) in R;
true
gap> Polynomial( GF(4), [Z(2),Z(2)] ) in R;
false
gap> Indeterminate(f2)^-1 in R;
true
```

19.21 IsLaurentPolynomialRing

`IsLaurentPolynomialRing(` *domain* `)`

`IsLaurentPolynomialRing` returns `true` if the object *domain* is a ring record, representing a Laurent polynomial ring (see 19.20), and `false` otherwise.

```
gap> IsPolynomialRing( Integers );
false
gap> IsLaurentPolynomialRing( PolynomialRing( Integers ) );
false
gap> IsLaurentPolynomialRing( LaurentPolynomialRing( Integers ) );
true
```

19.22 Ring Functions for Polynomial Rings

As was already noted in the introduction to this chapter polynomial rings are rings, so all ring functions (see chapter 5) are applicable to polynomial rings. This section comments on the implementation of those functions.

Let R be a commutative ring-with-one or a field and let P be the polynomial ring over R.

`EuclideanDegree(` P, f `)`

P is an Euclidean ring if and only if R is field. In this case the Euclidean degree of f is simply the degree of f. If R is not a field then the function signals an error.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> EuclideanDegree( x^10 + x^2 + 1 );
10
gap> EuclideanDegree( x^0 );
0
```

`EuclideanRemainder(` P, f, g `)`

P is an Euclidean ring if and only if R is field. In this case it is possible to divide f by g with remainder.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> EuclideanRemainder( (x+1)*(x+2)+5, x+1 );
5*x^0
```

`EuclideanQuotient(` P, f, g `)`

P is an Euclidean ring if and only if R is field. In this case it is possible to divide f by g with remainder.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> EuclideanQuotient( (x+1)*(x+2)+5, x+1 );
x + 2
```

`QuotientRemainder(` P, f, g `)`

P is an Euclidean ring if and only if R is field. In this case it is possible to divide f by g with remainder.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> QuotientRemainder( (x+1)*(x+2)+5, x+1 );
[ x + 2, 5*x^0 ]
```

`Gcd(` P, f, g `)`

P is an Euclidean ring if and only if R is field. In this case you can compute the greatest common divisor of f and g using `Gcd`.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> g := x^2 + 2*x + 1;;
gap> h := x^2 - 1;;
gap> Gcd( g, h );
x + 1
gap> GcdRepresentation( g, h );
[ 1/2*x^0, -1/2*x^0 ]
gap> g * (1/2) * x^0 - h * (1/2) * x^0;
x + 1
```

`Factors(` P, f `)`

This method is implemented for polynomial rings P over a domain R, where R is either a finite field, the rational numbers, or an algebraic extension of either one. If char R is a prime, f is factored using a Cantor-Zassenhaus algorithm.

```
gap> f5 := GF(5);; f5.name := "f5";;
gap> x  := Indeterminate(f5);; x.name := "x";;
gap> g := x^20 + x^8 + 1;
Z(5)^0*(x^20 + x^8 + 1)
gap> Factors(g);
[ Z(5)^0*(x^8 + 4*x^4 + 2), Z(5)^0*(x^12 + x^8 + 4*x^4 + 3) ]
```

If char R is 0, a quadratic Hensel lift is used.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> f:=x^105-1;
x^105 - 1
gap> Factors(f);
[ x - 1, x^2 + x + 1, x^4 + x^3 + x^2 + x + 1,
  x^6 + x^5 + x^4 + x^3 + x^2 + x + 1,
  x^8 - x^7 + x^5 - x^4 + x^3 - x + 1,
  x^12 - x^11 + x^9 - x^8 + x^6 - x^4 + x^3 - x + 1,
  x^24 - x^23 + x^19 - x^18 + x^17 - x^16 + x^14 - x^13 + x^12 - x^
    11 + x^10 - x^8 + x^7 - x^6 + x^5 - x + 1,
  x^48 + x^47 + x^46 - x^43 - x^42 - 2*x^41 - x^40 - x^39 + x^36 + x^
    35 + x^34 + x^33 + x^32 + x^31 - x^28 - x^26 - x^24 - x^22 - x^
    20 + x^17 + x^16 + x^15 + x^14 + x^13 + x^12 - x^9 - x^8 - 2*x^
    7 - x^6 - x^5 + x^2 + x + 1 ]
```

As f is an element of P if and only if the base ring of f is R you must embed the polynomial into the polynomial ring P if it is written as polynomial over a subring.

```
gap> f25 := GF(25);; Indeterminate(f25).name := "y";;
gap> l := Factors( EmbeddedPolynomial( PolynomialRing(f25), g ) );
[ y^4 + Z(5^2)^13, y^4 + Z(5^2)^17, y^6 + Z(5)^3*y^2 + Z(5^2)^3,
  y^6 + Z(5)^3*y^2 + Z(5^2)^15 ]
gap> l[1] * l[2];
y^8 + Z(5)^2*y^4 + Z(5)
gap> l[3] * l[4];
y^12 + y^8 + Z(5)^2*y^4 + Z(5)^3
```

`StandardAssociate(` P, f `)`

For a ring R the standard associate a of f is a multiple of f such that the leading coefficient of a is the standard associate in R. For a field R the standard associate a of f is a multiple of f such that the leading coefficient of a is 1.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> StandardAssociate( -2 * x^3 - x );
2*x^3 + x
```

19.23 Ring Functions for Laurent Polynomial Rings

As was already noted in the introduction to this chapter Laurent polynomial rings are rings, so all ring functions (see chapter 5) are applicable to polynomial rings. This section comments on the implementation of those functions.

Let R be a commutative ring-with-one or a field and let P be the polynomial ring over R.

`EuclideanDegree(` P, f `)`

P is an Euclidean ring if and only if R is field. In this case the Euclidean degree of f is the difference of $d(f)$ and $v(f)$ (see 19). If R is not a field then the function signals an error.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> LR := LaurentPolynomialRing(Rationals);;
gap> EuclideanDegree( LR, x^10 + x^2 );
8
gap> EuclideanDegree( LR, x^7 );
0
gap> EuclideanDegree( x^7 );
7
gap> EuclideanDegree( LR, x^2 + x^-2 );
4
gap> EuclideanDegree( x^2 + x^-2 );
4
```

`Gcd(` P, f, g `)`

P is an Euclidean ring if and only if R is field. In this case you can compute the greatest common divisor of f and g using `Gcd`.

```
gap> x := Indeterminate(Rationals);; x.name := "x";;
gap> LR := LaurentPolynomialRing(Rationals);;
gap> g := x^3 + 2*x^2 + x;;
gap> h := x^3 - x;;
gap> Gcd( g, h );
x^2 + x
gap> Gcd( LR, g, h );
x + 1      # x is a unit in LR
gap> GcdRepresentation( LR, g, h );
[ (1/2)*x^(-1), (-1/2)*x^(-1) ]
```

`Factors(` *P*, *f* `)`

This method is only implemented for a Laurent polynomial ring P over a finite field R. In this case f is factored using a Cantor-Zassenhaus algorithm. As f is an element of P if and only if the base ring of f is R you must embed the polynomial into the polynomial ring P if it is written as polynomial over a subring.

```
gap> f5 := GF(5);; f5.name := "f5";;
gap> x  := Indeterminate(f5);; x.name := "x";;
gap> g := x^10 + x^-2 + x^-10;
Z(5)^0*(x^10 + x^(-2) + x^(-10))
gap> Factors(g);
[ Z(5)^0*(x^(-2) + 4*x^(-6) + 2*x^(-10)),
  Z(5)^0*(x^12 + x^8 + 4*x^4 + 3) ]
gap> f25 := GF(25);; Indeterminate(f25).name := "y";;
gap> gg := EmbeddedPolynomial( LaurentPolynomialRing(f25), g );
y^10 + y^(-2) + y^(-10)
gap> l := Factors( gg );
[ y^(-6) + Z(5^2)^13*y^(-10), y^4 + Z(5^2)^17,
  y^6 + Z(5)^3*y^2 + Z(5^2)^3, y^6 + Z(5)^3*y^2 + Z(5^2)^15 ]
gap> l[1] * l[2];
y^(-2) + Z(5)^2*y^(-6) + Z(5)*y^(-10)
gap> l[3]*[4];
[ Z(5)^2*y^6 + Z(5)*y^2 + Z(5^2)^15 ]
```

`StandardAssociate(` *P*, *f* `)`

For a ring R the standard associate a of f is a multiple of f such that the leading coefficient of a is the standard associate in R and $v(a)$ is zero. For a field R the standard associate a of f is a multiple of f such that the leading coefficient of a is 1 and $v(a)$ is zero.

```
gap> x := Indeterminate(Integers);; x.name := "x";;
gap> LR := LaurentPolynomialRing(Integers);;
gap> StandardAssociate( LR, -2 * x^3 - x );
2*x^2 + 1
```

Chapter 20

Permutations

GAP3 is a system especially designed for the computations in groups. Permutation groups are a very important class of groups and GAP3 offers a data type **permutation** to describe the elements of permutation groups.

Permutations in GAP3 operate on **positive integers**. Whenever group elements operate on a domain we call the elements of this domain **points**. Thus in this chapter we often call positive integers points, if we want to emphasize that a permutation operates on them. An integer i is said to be **moved** by a permutation p if the image i^p of i under p is not i. The largest integer moved by any permutation may not be larger than $2^{28} - 1$.

Note that permutations do not belong to a specific group. That means that you can work with permutations without defining a permutation group that contains them. This is just like it is with integers, with which you can compute without caring about the domain `Integers` that contains them. It also means that you can multiply any two permutations.

Permutations are entered and displayed in cycle notation.

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (2,3,4);
(1,3)(2,4)
```

The first sections in this chapter describe the operations that are available for permutations (see 20.1 and 20.2). The next section describes the function that tests whether an object is a permutation (see 20.3). The next sections describe the functions that find the largest and smallest point moved by a permutation (see 20.4 and 20.5). The next section describes the function that computes the sign of a permutation (see 20.6). The next section describes the function that computes the smallest permutation that generates the same cyclic subgroup as a given permutation (see 20.7). The final sections describe the functions that convert between lists and permutations (see 20.8, 20.9, 20.10, and 20.11).

Permutations are elements of groups operating on positive integers in a natural way, thus see chapter 7 and chapter 2.10 for more functions.

The external functions are in the file `LIBNAME/"permutat.g"`.

20.1 Comparisons of Permutations

p1 = *p2*
p1 <> *p2*

The equality operator = evaluates to `true` if the two permutations *p1* and *p2* are equal, and to `false` otherwise. The inequality operator <> evaluates to `true` if the two permutations *p1* and *p2* are not equal, and to `false` otherwise. You can also compare permutations with objects of other types, of course they are never equal.

Two permutations are considered equal if and only if they move the same points and if the images of the moved points are the same under the operation of both permutations.

```
gap> (1,2,3) = (2,3,1);
true
gap> (1,2,3) * (2,3,4) = (1,3)(2,4);
true
```

p1 <*p2*
p1 <= *p2*
p1 > *p2*
p1 >= *p2*

The operators <, <=, >, and >= evaluate to `true` if the permutation *p1* is less than, less than or equal to, greater than, or greater than or equal to the permutation *p2*, and to `false` otherwise.

Let p_1 and p_2 be two permutations that are not equal. Then there exists at least one point i such that $i^{p_1} <> i^{p_2}$. Let k be the smallest such point. Then p_1 is considered smaller than p_2 if and only if $k^{p_1} < k^{p_2}$. Note that this implies that the identity permutation is the smallest permutation.

You can also compare permutations with objects of other types. Integers, rationals, cyclotomics, unknowns, and finite field elements are smaller than permutations. Everything else is larger.

```
gap> (1,2,3) < (1,3,2);
true     # 1^(1,2,3) = 2 < 3 = 1^(1,3,2)
gap> (1,3,2,4) < (1,3,4,2);
false    # 2^(1,3,2,4) = 4 > 1 = 2^(1,3,4,2)
```

20.2 Operations for Permutations

p1 * *p2*

The operator * evaluates to the product of the two permutations *p1* and *p2*.

p1 / *p2*

The operator / evaluates to the quotient $p1 * p2^{-1}$ of the two permutations *p1* and *p2*.

`LeftQuotient(` *p1* `,` *p2* `)`

`LeftQuotient` returns the left quotient $p1^{-1} * p2$ of the two permutations *p1* and *p2*. (This can also be written *p1* `mod` *p2*.)

p ^ *i*

The operator ^ evaluates to the i-th power of the permutation p.

p1 ^ *p2*

The operator ^ evaluates to the conjugate $p2^{-1} * p1 * p2$ of the permutation *p1* by the permutation *p2*.

`Comm(` *p1* `,` *p2* `)`

`Comm` returns the commutator $p1^{-1} * p2^{-1} * p1 * p2$ of the two permutations *p1* and *p2*.

i ^ *p*

The operator ^ evaluates to the image i^p of the positive integer i under the permutation p.

i / *p*

The operator / evaluates to the preimage $i^{p^{-1}}$ of the integer i under the permutation p.

list * *p*
p * *list*

The operator * evaluates to the list of products of the permutations in *list* with the permutation p. That means that the value is a new list *new* such that *new*`[`*i*`] = `*list*`[`*i*`] * `*p* respectively *new*`[`*i*`] = `*p*` * `*list*`[`*i*`]`.

list / *p*

The operator / evaluates to the list of quotients of the permutations in *list* with the permutation p. That means that the value is a new list *new* such that *new*`[`*i*`] = `*list*`[`*i*`] /` *p*.

For the precedence of the operators see 2.10.

20.3 IsPerm

`IsPerm(` *obj* `)`

`IsPerm` returns `true` if *obj*, which may be an object of arbitrary type, is a permutation and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsPerm( (1,2) );
true
gap> IsPerm( 1 );
false
```

20.4 LargestMovedPointPerm

`LargestMovedPointPerm(` *perm* `)`

`LargestMoverPointPerm` returns the largest point moved by the permutation *perm*, i.e., the largest positive integer i such that *i*^*perm* `<>` *i*. It will signal an error if *perm* is trivial (see also 20.5).

```
gap> LargestMovedPointPerm( (2,3,1) );
3
gap> LargestMovedPointPerm( (1,2)(1000,1001) );
1001
```

20.5 SmallestMovedPointPerm

`SmallestMovedPointPerm(` *perm* `)`

`SmallestMovedPointPerm` returns the smallest point moved by the permutation *perm*, i.e., the smallest positive integer i such that *i*`^`*perm* `<>` i. It will signal an error if *perm* is trivial (see also 20.4).

```
gap> SmallestMovedPointPerm( (4,7,5) );
4
```

20.6 SignPerm

`SignPerm(` *perm* `)`

`SignPerm` returns the **sign** of the permutation *perm*.

The sign s of a permutation p is defined by $s = \prod_{i<j} (i^p - j^p) / \prod_{i<j} (i - j)$, where n is the largest point moved by p and i, j range over $1...n$.

One can easily show that **sign** is equivalent to the **determinant** of the **permutation matrix** of *perm*. Thus it is obvious that the function **sign** is a homomorphism.

```
gap> SignPerm( (1,2,3)(5,6) );
-1
```

20.7 SmallestGeneratorPerm

`SmallestGeneratorPerm(` *perm* `)`

`SmallestGeneratorPerm` returns the smallest permutation that generates the same cyclic group as the permutation *perm*.

```
gap> SmallestGeneratorPerm( (1,4,3,2) );
(1,2,3,4)
```

Note that `SmallestGeneratorPerm` is very efficient, even when *perm* has huge order.

20.8 ListPerm

`ListPerm(` *perm* `)`

`ListPerm` returns a list *list* that contains the images of the positive integers under the permutation *perm*. That means that *list*`[`*i*`] = ` *i*`^`*perm*, where i lies between 1 and the largest point moved by *perm* (see 20.4).

```
gap> ListPerm( (1,2,3,4) );
[ 2, 3, 4, 1 ]
gap> ListPerm( () );
[  ]
```

`PermList` (see 20.9) performs the inverse operation.

20.9 PermList

`PermList(` *list* `)`

`PermList` returns the permutation *perm* that moves points as describes by the list *list*. That means that `i^perm = list[i]` if *i* lies between 1 and the length of *list*, and `i^perm = i` if *i* is larger than the length of the list *list*. It will signal an error if *list* does not define a permutation, i.e., if *list* is not a list of integers without holes, or if *list* contains an integer twice, or if *list* contains an integer not in the range `[1..Length(list)]`.

```
gap> PermList( [6,2,4,1,5,3] );
(1,6,3,4)
gap> PermList( [] );
()
```

`ListPerm` (see 20.8) performs the inverse operation.

20.10 RestrictedPerm

`RestrictedPerm(` *perm*`,` *list* `)`

`RestrictedPerm` returns the new permutation *new* that operates on the points in the list *list* in the same way as the permutation *perm*, and that fixes those points that are not in *list*. *list* must be a list of positive integers such that for each *i* in *list* the image `i^perm` is also in *list*, i.e., it must be the union of cycles of *perm*.

```
gap> RestrictedPerm( (1,2,3)(4,5), [4,5] );
(4,5)
```

20.11 MappingPermListList

`MappingPermListList(` *list1*`,` *list2* `)`

`MappingPermListList` returns a permutation *perm* such that `list1[i] ^ perm = list2[i]`. *perm* fixes all points larger then the maximum of the entries in *list1* and *list2*. If there are several such permutations, it is not specified which `MappingPermListList` returns. *list1* and *list2* must be lists of positive integers of the same length, and neither may contain an element twice.

```
gap> MappingPermListList( [3,4], [6,9] );
(3,6,4,9,8,7,5)
gap> MappingPermListList( [], [] );
()
```

Chapter 21

Permutation Groups

A permutation group is a group of permutations on a set Ω of positive integers (see chapters 7 and 20).

Our standard example in this chapter will be the symmetric group of degree 4, which is defined by the following GAP3 statements.

```
gap> s4 := Group( (1,2), (1,2,3,4) );
Group( (1,2), (1,2,3,4) )
```

This introduction is followed by a section that describes the function that tests whether an object is a permutation group or not (see section 21.1). The next sections describe the functions that are related to the set of points moved by a permutation group (see 21.2, 21.3, 21.4, and 21.5). The following section describes the concept of stabilizer chains, which are used by most functions for permutation groups (see 21.6). The following sections describe the functions that compute or change a stabilizer chain (see 21.7, 21.9, 21.10, 21.11). The next sections describe the functions that extract information from stabilizer chains (see 21.12, 21.15, 21.13, and 21.14). The next two sections describe the functions that find elements or subgroups of a permutation group with a property (see 21.16 and 21.17).

If the permutation groups become bigger, computations become slower. In many cases it is preferable then, to use random methods for computation. This is explained in section 21.24.

Because each permutation group is a domain all set theoretic functions can be applied to it (see chapter 4 and 21.20). Also because each permutation group is after all a group all group functions can be applied to it (see chapter 7 and 21.21). Finally each permutation group operates naturally on the positive integers, so all operations functions can be applied (see chapter 8 and 21.22). The last section in this chapter describes the representation of permutation groups (see 21.25).

The external functions are in the file `LIBNAME/"permgrp.g"`.

21.1 IsPermGroup

`IsPermGroup(` *obj* `)`

`IsPermGroup` returns `true` if the object *obj*, which may be an object of an arbitrary type, is a permutation group, and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> s4 := Group( (1,2), (1,2,3,4) );; s4.name := "s4";;
gap> IsPermGroup( s4 );
true
gap> f := FactorGroup( s4, Subgroup( s4, [(1,2)(3,4),(1,3)(2,4)] ) );
(s4 / Subgroup( s4, [ (1,2)(3,4), (1,3)(2,4) ] ))
gap> IsPermGroup( f );
false     # see section 7.33
gap> IsPermGroup( [ 1, 2 ] );
false
```

21.2 PermGroupOps.MovedPoints

`PermGroupOps.MovedPoints(` G `)`

`PermGroupOps.MovedPoints` returns the set of moved points of the permutation group G, i.e., points which are moved by at least one element of G (also see 21.5).

```
gap> s4 := Group( (1,3,5,7), (1,3) );;
gap> PermGroupOps.MovedPoints( s4 );
[ 1, 3, 5, 7 ]
gap> PermGroupOps.MovedPoints( Group( () ) );
[  ]
```

21.3 PermGroupOps.SmallestMovedPoint

`PermGroupOps.SmallestMovedPoint(` G `)`

`PermGroupOps.SmallestMovedPoint` returns the smallest positive integer which is moved by the permutation group G (see also 21.4). This function signals an error if G is trivial.

```
gap> s3b := Group( (2,3), (2,3,4) );;
gap> PermGroupOps.SmallestMovedPoint( s3b );
2
```

21.4 PermGroupOps.LargestMovedPoint

`PermGroupOps.LargestMovedPoint(` G `)`

`PermGroupOps.LargestMovedPoint` returns the largest positive integer which is moved by the permutation group G (see also 21.3). This function signals an error if G is trivial.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> PermGroupOps.LargestMovedPoint( s4 );
4
```

21.5 PermGroupOps.NrMovedPoints

`PermGroupOps.NrMovedPoints(` G `)`

`PermGroupOps.NrMovedPoints` returns the number of moved points of the permutation group G, i.e., points which are moved by at least one element of G (also see 21.2).

```
gap> s4 := Group( (1,3,5,7), (1,3) );;
```

```
gap> PermGroupOps.NrMovedPoints( s4 );
4
gap> PermGroupOps.NrMovedPoints( Group( () ) );
0
```

21.6 Stabilizer Chains

Most of the algorithms for permutation groups need a **stabilizer chain** of the group. The concept of stabilizer chains was introduced by Charles Sims in [Sim70].

If $[b_1,\ldots,b_n]$ is a list of points, $G^{(1)} = G$ and $G^{(i+1)} = Stab_{G^{(i)}}(b_i)$ such that $G^{(n+1)} = \{()\}$. The list $[b_1,\ldots,b_n]$ is called a **base** of G, the points b_i are called **basepoints**. A set S of generators for G satisfying the condition $< S \cap G^{(i)} > = G^{(i)}$ for each $1 \leq i \leq n$, is called a **strong generating set** (SGS) of G. More precisely we ought to say that a set S that satisfies the conditions above is a SGS of G **relative** to B. The chain of subgroups of G itself is called the **stabilizer chain** of G relative to B.

Since $[b_1,\ldots,b_n]$, where n is the degree of G and b_i are the moved points of G, certainly is a base for G there exists a base for each permutation group. The number of points in a base is called the **length** of the base. A base B is called **reduced** if no stabilizer in the chain relative to B is trivial, i.e., there exists no i such that $G^{(i)} = G^{(i+1)}$. Note that different reduced bases for one group G may have different length. For example, the Chevalley Group $G_2(4)$ possesses reduced bases of length 5 and 7.

Let $R^{(i)}$ be a right transversal of $G^{(i+1)}$ in $G^{(i)}$, i.e., a set of right coset representatives of the cosets of $G^{(i+1)}$ in $G^{(i)}$. Then each element g of G has a unique representation of the following form $g = r_n \ldots r_1$ with $r_i \in R^{(i)}$. Thus with the knowledge of the transversals $R^{(i)}$ we know each element of G, in principle. This is one reason why stabilizer chains are one of the most useful tools for permutation groups. Furthermore basic group theory tells us that we can identify the cosets of $G^{(i+1)}$ in $G^{(i)}$ with the points in $O^{(i)} := b_i^{G^{(i)}}$. So we could represent a transversal as a list T such that $T[p]$ is a representative of the coset corresponding to the point $p \in O^{(i)}$, i.e., an element of $G^{(i)}$ that takes b_i to p.

For permutation groups of small degree this might be possible, but for permutation groups of large degree it is still not good enough. Our goal then is to store as few different permutations as possible such that we can still reconstruct each representative in $R^{(i)}$, and from them the elements in G. A **factorized inverse transversal** T is a list where $T[p]$ is a **generator** of $G^{(i)}$ such that $p^{T[p]}$ is a point that lies earlier in $O^{(i)}$ than p (note that we consider $O^{(i)}$ as a list not as a set). If we assume inductively that we know an element $r \in G^{(i)}$ that takes b_i to $p^{T[p]}$, then $rT[p]^{-1}$ is an element in $G^{(i)}$ that takes b_i to p.

A stabilizer chain (see 21.7, 21.25) is stored recursively in GAP3. The group record of a permutation group G with a stabilizer chain has the following additional components.

`orbit`
: List of orbitpoints of `orbit[1]` (which is the basepoint) under the action of the generators.

`transversal`
: Factorized inverse transversal as defined above.

`stabilizer`
: Record for the stabilizer of the point `orbit[1]` in the group generated by `generators`.

The components of this record are again `generators`, `orbit`, `transversal`, `identity` and `stabilizer`. The last stabilizer in the stabilizer chain only contains the components `generators`, which is an empty list, and `identity`.

`stabChain`
: A record, that contains all information about the stabilizer chain. Functions acessing the stabilizer chain should do it using this record, as it is planned to remove the above three components from the group record in the future. The components of the `stabChain` record are described in section 21.25.

Note that the values of these components are changed by functions that change, extend, or reduce a base (see 21.7, 21.9, and 21.10).

Note that the records that represent the stabilizers are not group records (see 7.118). Thus you cannot take such a stabilizer and apply group functions to it. The last `stabilizer` in the stabilizer chain is a record whose component `generators` is empty.

Below you find an example for a **stabilizer chain** for the symmetric group of degree 4.

```
rec(
    identity    := (),
    generators  := [ (1,2), (1,2,3,4) ],
    orbit       := [ 1, 2, 4, 3 ],
    transversal := [ (), (1,2), (1,2,3,4), (1,2,3,4) ],
    stabilizer := rec(
        identity    := (),
        generators  := [ (3,4), (2,4) ],
        orbit       := [ 2, 4, 3 ],
        transversal := [ , (), (3,4), (2,4) ],
        stabilizer := rec(
            identity    := (),
            generators  := [ (3,4) ],
            orbit       := [ 3, 4 ],
            transversal := [ ,, (), (3,4) ],
            stabilizer := rec(
                identity   := (),
                generators := [  ]
            )
        )
    )
)
```

21.7 StabChain

`StabChain(` *G* `)`
`StabChain(` *G*`,` *opt* `)`

`StabChain` computes and returns a stabilizer chain for G. The option record *opt* can be given and may contain information that will be used when computing the stabilizer chain. Giving this information might speed up computations. When using random methods (see 21.24), `StabChain` also guarantees, that the computed stabilizer chain confirms to the information given. For example giving the size ensures correctness of the stabilizer chain.

If information of this kind can also be gotten from the parent group, `StabChain` does so.

The following components of the option record are currectly supported:

`size`
: The group size.

`limit`
: An upper limit for the group size.

`base`
: A list of points. If given, `StabChain` computes a reduced base starting with the points in `base`.

`knownBase`
: A list of points, representing a known base.

`random`
: A value to supersede global or parent group setting of `StabChainOptions.random` (see 21.24).

21.8 MakeStabChain

`MakeStabChain(` *G* `)`
`MakeStabChain(` *G*`,` *lst* `)`

`MakeStabChain` computes a **reduced** stabilizer chain for the permutation group *G*.

If no stabilizer chain for *G* is already known and no argument *lst* is given, it computes a reduced stabilizer chain for the lexicographically smallest reduced base of *G*.

If no stabilizer chain for *G* is already known and an argument *lst* is given, it computes a **reduced** stabilizer chain with a base that starts with the points in *lst*. Note that points in *lst* that would lead to trivial stabilizers will be skipped (see 21.9).

Deterministically, the stabilizer chain is computed using the **Schreier-Sims-Algorithm**, which is described in [Leo80]. The time used is in practice proportional to the third power of the degree of the group.

If a stabilizer chain for *G* is already known and no argument *lst* is given, it reduces the known stabilizer chain.

If a stabilizer chain for *G* is already known and an argument *lst* is given, it changes the stabilizer chain such that the result is a **reduced** stabilizer chain with a base that starts with the points in *lst* (see 21.9). Note that points in *lst* that would lead to trivial stabilizers will be skipped.

The algorithm used in this case is called **basechange**, which is described in [But82]. The worst cases for the basechange algorithm are groups of large degree which have a long base.

```
gap> s4 := Group( (1,2), (1,2,3,4) );
Group( (1,2), (1,2,3,4) )
gap> MakeStabChain( s4 );     # compute a stabilizer chain
gap> Base( s4 );
[ 1, 2, 3 ]
gap> MakeStabChain( s4, [4,3,2,1] );     # perform a basechange
gap> Base( s4 );
[ 4, 3, 2 ]
```

`MakeStabChain` mainly works by calling `StabChain` with appropriate parameters.

21.9 ExtendStabChain

`ExtendStabChain(` *G*`,` *lst* `)`

`ExtendStabChain` inserts trivial stabilizers into the known stabilizer chain of the permutation group *G* such that *lst* becomes the base of *G*. The stabilizer chain which belongs to the base *lst* must reduce to the old stabilizer chain (see 21.10).

This function is useful if two different (sub-)groups have to have exactly the same base.

```
gap> s4 := Group( (1,2), (1,2,3,4) );;
gap> MakeStabChain( s4, [3,2,1] );  Base( s4 );
[ 3, 2, 1 ]
gap> h := Subgroup( Parent(s4), [(1,2,3,4), (2,4)] );
Subgroup( Group( (1,2), (1,2,3,4) ), [ (1,2,3,4), (2,4) ] )
gap> Base( h );
[ 1, 2 ]
gap> MakeStabChain( h, Base( s4 ) );  Base( h );
[ 3, 2 ]
gap> ExtendStabChain( h, Base( s4 ) );  Base( h );
[ 3, 2, 1 ]
```

21.10 ReduceStabChain

`ReduceStabChain(` *G* `)`

`ReduceStabChain` removes trivial stabilizers from a known stabilizer chain of the permutation group *G*. The result is a **reduced** stabilizer chain (also see 21.9).

```
gap> s4 := Group( (1,2), (1,2,3,4) );;
gap> Base( s4 );
[ 1, 2, 3 ]
gap> ExtendStabChain( s4, [ 1, 2, 3, 4 ] );  Base( s4 );
[ 1, 2, 3, 4 ]
gap> PermGroupOps.Indices( s4 );
[ 4, 3, 2, 1 ]
gap> ReduceStabChain( s4 );  Base( s4 );
[ 1, 2, 3 ]
```

21.11 MakeStabChainStrongGenerators

`MakeStabChainStrongGenerators(` *G*`,` *base*`,` *stronggens* `)`

`MakeStabChainStrongGenerators` computes a **reduced** stabilizer chain for the permutation group *G* with the base *base* and the strong generating set *stronggens*. *stronggens* must be a strong generating set for *G* relative to the base *base*; note that this is not tested. Since the generators for *G* are not changed the strong generating set of *G* got by `PermGroupOps.StrongGenerators` is not exactly *stronggens* afterwards. This function is mostly used to reconstruct a stabilizer chain for a group *G* and works considerably faster than `MakeStabChain` (see 21.8).

```
gap> G := Group( (1,2), (1,2,3), (4,5) );;
```

```
gap> Base( G );
[ 1, 2, 4 ]
gap> ExtendStabChain( G, [1,2,3,4] );
gap> PermGroupOps.Indices( G );  base := Base( G );
[ 3, 2, 1, 2 ]     # note that the stabilizer chain is not reduced
[ 1, 2, 3, 4 ]
gap> stronggens := PermGroupOps.StrongGenerators( G );
[ (4,5), (2,3), (1,2), (1,2,3) ]
gap> H := Group( (1,2), (1,3), (4,5) );
Group( (1,2), (1,3), (4,5) )     # of course G = H
gap> MakeStabChainStrongGenerators( H, base, stronggens );
gap> PermGroupOps.Indices( H );  Base( H );
[ 3, 2, 2 ]        # note that the stabilizer chain is reduced
[ 1, 2, 4 ]
gap> PermGroupOps.StrongGenerators( H );
[ (4,5), (2,3), (1,2), (1,3) ]
# note that this is different from stronggens
```

21.12 Base for Permutation Groups

`Base(` G `)`

`Base` returns a base for the permutation group G. If a stabilizer chain for G is already known, `Base` returns the base for this stabilizer chain. Otherwise a stabilizer chain for the lexicographically smallest reduced base is computed and its base is returned (see 21.6).

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> Base( s4 );
[ 1, 2, 3 ]
```

21.13 PermGroupOps.Indices

`PermGroupOps.Indices(` G `)`

`PermGroupOps.Indices` returns a list l of indices of the permutation group G with respect to a stabilizer chain of G, i.e., l`[`i`]` is the index of $G^{(i+1)}$ in $G^{(i)}$. Thus the size of G is the product of all indices in l. If a stabilizer chain for G is already known, `PermGroupOps.Indices` returns the indices corresponding to this stabilizer chain. Otherwise a stabilizer chain with the lexicographically smallest reduced base is computed and the indices corresponding to this chain are returned (see 21.6).

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> PermGroupOps.Indices( s4 );
[ 4, 3, 2 ]      # note that for s4 the indices are
                 # actually independent of the base
```

21.14 PermGroupOps.StrongGenerators

`PermGroupOps.StrongGenerators(` G `)`

`PermGroupOps.StrongGenerators` returns a list of strong generators for the permutation group G. If a stabilizer chain for G is already known, `PermGroupOps.StrongGenerators` returns a strong generating set corresponding to this stabilizer chain. Otherwise a stabilizer chain with the lexicographically smallest reduced base is computed and a strong generating set corresponding to this chain is returned (see 21.6).

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> Base( s4 );
[ 1, 2, 3 ]
gap> PermGroupOps.StrongGenerators( s4 );
[ (3,4), (2,3,4), (1,2), (1,2,3,4) ]
```

21.15 ListStabChain

`ListStabChain(` *G* `)`

`ListStabChain` returns a list of stabilizer records of the stabilizer chain of the permutation group G, i.e., the result is a list l such that l`[`i`]` is the i-th stabilizer $G^{(i)}$. The records in that list are identical to the records of the stabilizer chain. Thus changes made in a record l`[`i`]` are simultaneously done in the stabilizer chain (see 46.3).

21.16 PermGroupOps.ElementProperty

`PermGroupOps.ElementProperty(` *G*`,` *prop* `)`
`PermGroupOps.ElementProperty(` *G*`,` *prop*`,` *K* `)`

`PermGroupOps.ElementProperty` returns an element g in the permutation group G such that *prop*`(`g`)` is `true`. *prop* must be a function of one argument that returns either `true` or `false` when applied to an element of G. If G has no such element, `false` is returned.

```
gap> V4 := Group((1,2),(3,4));;
gap> PermGroupOps.ElementProperty( V4, g -> (1,2)^g = (3,4) );
false
```

`PermGroupOps.ElementProperty` first computes a stabilizer chain for G, if necessary. Then it performs a backtrack search through G for an element satisfying *prop*, i.e., enumerates all elements of G as described in section 21.6, and applies *prop* to each until one element g is found for which *prop*`(`g`)` is `true`. This algorithm is described in detail in [But82].

```
gap> S8 := Group( (1,2), (1,2,3,4,5,6,7,8) );;  S8.name := "S8";;
gap> Size( S8 );
40320
gap> V := Subgroup( S8, [(1,2),(1,2,3),(6,7),(6,7,8)] );;
gap> Size( V );
36
gap> U := V ^ (1,2,3,4)(5,6,7,8);;
gap> PermGroupOps.ElementProperty( S8, g -> U ^ g = V );
(1,4,2)(5,6)     # another permutation conjugating U to V
```

This search will of course take quite a while if G is large, especially if no element of G satisfies *prop*, and therefore all elements of G must be tried.

To speed up the computation you may pass a subgroup K of G as optional third argument. This subgroup must preserve *prop* in the sense that either all elements of a left coset g`*`K satisfy *prop* or no element of g`*`K does.

In our example above such a subgroup is the normalizer $N_G(V)$ because $h \in gN_G(V)$ takes U to V if and only if g does. Of course every subgroup of $N_G(V)$ has this property too. Below we use the subgroup V itself. In this example this speeds up the computation by a factor of 4.

```
gap> K := Subgroup( S8, V.generators );;
gap> PermGroupOps.ElementProperty( S8, g -> U ^ g = V, K );
(1,4,2)(5,6)
```

In the following example, we use the same subgroup, but with a larger generating system. This speeds up the computation by another factor of 3. Something like this may happen frequently. The reason is too complicated to be explained here.

```
gap> K2 := Subgroup( S8, Union( V.generators, [(2,3),(7,8)] ) );;
gap> K2 = K;
true
gap> PermGroupOps.ElementProperty( S8, g -> U ^ g = V, K2 );
(1,4,2)(5,6)
```

Passing the full normalizer speeds up the computation in this example by another factor of 2. Beware though that in other examples the computation of the normalizer alone may take longer than calling `PermGroupOps.ElementProperty` with only the subgroup itself as argument.

```
gap> N := Normalizer( S8, V );
Subgroup( S8, [ (1,2), (1,2,3), (6,7), (6,7,8), (2,3), (7,8),
  (1,6)(2,7)(3,8), (4,5) ] )
gap> Size( N );
144
gap> PermGroupOps.ElementProperty( S8, g -> U ^ g = V, N );
(1,4)(5,6)
```

21.17 PermGroupOps.SubgroupProperty

`PermGroupOps.SubgroupProperty(` *G*, *prop* `)`
`PermGroupOps.SubgroupProperty(` *G*, *prop*, *K* `)`

`PermGroupOps.SubgroupProperty` returns the subgroup U of the permutation group G of all elements in G that satisfy *prop*, i.e., the subgroup of all elements g in G such that *prop*`(`g`)` is `true`. *prop* must be a function of one argument that returns either `true` or `false` when applied to an element of G. Of course the elements that satisfy *prop* must form a subgroup of G. `PermGroupOps.SubgroupProperty` builds a stabilizer chain for U.

```
gap> S8 := Group( (1,2), (1,2,3,4,5,6,7,8) );;  S8.name := "S8";;
gap> Size(S8);
40320
gap> V := Subgroup( S8, [(1,2),(1,2,3),(6,7),(6,7,8)] );;
gap> Size(V);
36
gap> PermGroupOps.SubgroupProperty( S8, g -> V ^ g = V );
Subgroup( S8, [ (7,8), (6,7), (4,5), (2,3)(4,5)(6,8,7), (1,2),
  (1,6,3,8)(2,7) ] )
```

```
# the normalizer of V in S8
```

`PermGroupOps.SubgroupProperty` first computes a stabilizer chain for G, if necessary. Then it performs a backtrack search through G for the elements satisfying *prop*, i.e., enumerates all elements of G as described in section 21.6, and applies *prop* to each, adding elements for which *prop*(*g*) is `true` to the subgroup U. Once U has become non-trivial, it is used to eliminate whole cosets of stabilizers in the stabilizer chain of G if they cannot contain elements with the property *prop* that are not already in U. This algorithm is described in detail in [But82].

This search will of course take quite a while if G is large. To speed up the computation you may pass a subgroup K of U as optional third argument.

Passing the subgroup V itself, speeds up the computation in this example by a factor of 2.

```
gap> K := Subgroup( S8, V.generators );;
gap> PermGroupOps.SubgroupProperty( S8, g -> V ^ g = V, K );
Subgroup( S8, [ (1,2), (1,2,3), (6,7), (6,7,8), (2,3), (7,8), (4,5),
  (1,6,3,8)(2,7) ] )
```

21.18 CentralCompositionSeriesPPermGroup

`CentralCompositionSeriesPPermGroup(` *G* `)`

This function calculates a central composition series for the p-group G. The method used is known as Holt's algorithm. If G is not a p-group, an error is signalled.

```
gap> D := Group( (1,2,3,4), (1,3) );; D.name := "d8";;
gap> CentralCompositionSeriesPPermGroup( D );
[ d8, Subgroup( d8, [ (2,4), (1,3) ] ),
  Subgroup( d8, [ (1,3)(2,4) ] ), Subgroup( d8, [  ] ) ]
```

21.19 PermGroupOps.PgGroup

`PermGroupOps.PgGroup(` *G* `)`

This function converts a permutation group G of prime power order p^d into an ag group P such that the presentation corresponds to a p-step central series of G. This central composition series is constructed by calling `CentralCompositionSeriesPPermGroup` (see 21.18). An isomorphism from the ag group to the permutation group is bound to P`.bijection`.

There is no dispatcher to this function, it must be called as `PermGroupOps.PgGroup`.

21.20 Set Functions for Permutation Groups

All set theoretic functions described in chapter 4 are also applicable to permutation groups. This section describes which functions are implemented specially for permutation groups. Functions not mentioned here are handled by the default methods described in the respective sections.

`Random(` *G* `)`

To compute a random element in a permutation group *G* GAP3 computes a stabilizer chain for *G*, takes on each level a random representative and returns the product of those. All elements of *G* are chosen with equal probability by this method.

`Size(` *G* `)`

`Size` calls `StabChain` (see 21.7), if necessary, and returns the product of the indices of the stabilizer chain (see 21.6).

`Elements(` *G* `)`

`Elements` calls `StabChain` (see 21.7), if necessary, and enumerates the elements of *G* as described in 21.6. It returns the set of those elements.

`Intersection(` *G1* `,` *G2* `)`

`Intersection` first computes stabilizer chains for *G1* and *G2* for a common base. If either group already has a stabilizer chain a basechange is performed (see 21.8). `Intersection` enumerates the elements of *G1* and *G2* using a backtrack algorithm, eliminating whole cosets of stabilizers in the stabilizer chains if possible (see 21.17). It builds a stabilizer chain for the intersection.

21.21 Group Functions for Permutation Groups

All group functions for groups described in chapter 7.9 are also applicable to permutation groups. This section describes which functions are implemented specially for permutation groups. Functions not mentioned here are handled by the default methods described in the respective sections.

G `^` *p*
`ConjugateSubgroup(` *G* `,` *p* `)`

Returns the conjugate permutation group of *G* with the permutation *p*. *p* must be an element of the parent group of *G*. If a stabilizer chain for *G* is already known, it is also conjugated.

`Centralizer(` *G* `,` *U* `)`
`Centralizer(` *G* `,` *g* `)`
`Normalizer(` *G* `,` *U* `)`

These functions first compute a stabilizer chain for *G*. If a stabilizer chain is already known a basechange may be performed to obtain a base that is better suited for the problem. These functions then enumerate the elements of *G* with a backtrack algorithm, eliminating whole cosets of stabilizers in the stabilizer chain if possible (see 21.17). They build a stabilizer chain for the resulting subgroup.

`SylowSubgroup(` *G* `,` *p* `)`

If G is not transitive, its p-Sylow subgroup is computed by starting with P:=G, and for each transitive constituent homomorphism *hom* iterating
P := PreImage(SylowSubgroup(Image(*hom*, P), p)).

If G is transitive but not primitive, its p-Sylow subgroup is computed as
SylowSubgroup(PreImage(SylowSubgroup(Image(*hom*,G),p)), p).

If G is primitive, SylowSubgroup takes random elements in G, until it finds a p-element g, whose centralizer in G contains the whole p-Sylow subgroup. Such an element must exist, because a p-group has a nontrivial centre. Then the p-Sylow subgroup of the centralizer is computed and returned. Note that the centralizer must be a proper subgroup of G, because it operates imprimitively on the cycles of g.

Coset(U, g)

Returns the coset U*g. The representative chosen is the lexicographically smallest element of that coset. It is computed with an algorithm that is very similar to the backtrack algorithm.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;  s4.name := "s4";;
gap> u := Subgroup( s4, [(1,2,3)] );;
gap> Coset( u, (1,3,2) );
(Subgroup( s4, [ (1,2,3) ] )*())
gap> Coset( u, (3,2) );
(Subgroup( s4, [ (1,2,3) ] )*(2,3))
```

Cosets(G, U)

Returns the cosets of U in G. Cosets first computes stabilizer chains for G and U with a common base. If either subgroup already has a stabilizer chain, a basechange is performed (see 21.8). A transversal is computed recursively using the fact that if S is a transversal of $U^{(2)} = Stab_U(b_1)$ in $G^{(2)} = Stab_G(b_1)$, and $R^{(1)}$ is a transversal of $G^{(2)}$ in G, then a transversal of U in G is a subset of $S * R^{(1)}$.

```
gap> Cosets( s4, u );
[ (Subgroup( s4, [ (1,2,3) ] )*()),
  (Subgroup( s4, [ (1,2,3) ] )*(3,4)),
  (Subgroup( s4, [ (1,2,3) ] )*(2,3)),
  (Subgroup( s4, [ (1,2,3) ] )*(2,3,4)),
  (Subgroup( s4, [ (1,2,3) ] )*(2,4,3)),
  (Subgroup( s4, [ (1,2,3) ] )*(2,4)),
  (Subgroup( s4, [ (1,2,3) ] )*(1,2,3,4)),
  (Subgroup( s4, [ (1,2,3) ] )*(1,2,4)) ]
```

PermutationCharacter(P)

Computes the character of the natural permutation representation of P, i.e. it does the same as PermutationCharacter(P, $\mathrm{Stab}_P(1)$) but works much faster.

```
gap> G := SymmetricPermGroup(5);;
gap> PermutationCharacter(G);
[ 5, 3, 1, 2, 0, 1, 0 ]
```

`ElementaryAbelianSeries(` G `)`

This function builds an elementary abelian series of G by iterated construction of normal closures. If a partial elementary abelian series reaches up to a subgroup U of G which does not yet contain the generator s of G then the series is extended up to the normal closure N of U and s. If the factor N/U is not elementary abelian, i.e., if some commutator of s with one of its conjugates under G does not lie in U, intermediate subgroups are calculated recursively by extending U with that commutator. If G is solvable this process must come to an end since commutators of arbitrary depth cannot exist in solvable groups.

Hence this method gives an elementary abelian series if G is solvable and gives an infinite recursion if it is not. For permutation groups, however, there is a bound on the derived length that depends only on the degree d of the group. According to Dixon this is $(5\log_3(d))/2$. So if the commutators get deeper than this bound the algorithm stops and sets G`.isSolvable` to `false`, signalling an error. Otherwise G`.isSolvable` is set to `true` and the elementary abelian series is returned as a list of subgroups of G.

```
gap> S := Group( (1,2,3,4), (1,2) );; S.name := "s4";;
gap> ElementaryAbelianSeries( S );
[ Subgroup( s4, [ (1,2), (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,4)(2,3), (1,2)(3,4) ] ), Subgroup( s4, [  ] ) ]
gap> A := Group( (1,2,3), (3,4,5) );;
gap> ElementaryAbelianSeries( A );
Error, <G> must be solvable
```

`IsSolvable(` G `)`

Solvability of a permutation group G is tested by trying to construct an elementary abelian series as described above. After this has been done the flag G`.isSolvable` is set correctly, so its value is returned.

```
gap> S := Group( (1,2,3,4), (1,2) );;
gap> IsSolvable( S );
true
gap> A := Group( (1,2,3), (3,4,5) );;
gap> IsSolvable( A );
false
```

`CompositionSeries(` G `)`

A composition series for the solvable group G is calculated either from a given subnormal series, which must be bound to G`.subnormalSeries`, in which case G`.bssgs` must hold the corresponding base-strong subnormal generating system, or from an elementary abelian series (as computed by `ElementaryAbelianSeries(` G `)` above) by inserting intermediate subgroups (i.e. powers of the polycyclic generators or composition series along bases of the vector spaces in the elementary abelian series). In either case, after execution of this function, G`.bssgs` holds a base-strong pag system corresponding to the composition series calculated.

```
gap> S := Group( (1,2,3,4), (1,2) );; S.name := "s4";;
```

```
gap> CompositionSeries( S );
[ Subgroup( s4, [ (1,2), (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( s4, [ (1,2)(3,4) ] ), Subgroup( s4, [  ] ) ]
```

If G is not solvable then a composition series *cs* is computed with an algorithm by A. Seress and R. Beals. In this case the factor group of each element *cs*`[`*i*`]` in the composition series modulo the next one *cs*`[`*i*`+1]` are represented as primitive permutation groups. One should call *cs*`[`*i*`].operations.FactorGroup(` *cs*`[`*i*`],` *cs*`[`*i*`+1] )` directly to avoid the check in `FactorGroup` that *cs*`[`*i*`+1]` is normal in *cs*`[`*i*`]`. The natural homomorphism of *cs*`[`*i*`]` onto this factor group will be given as a `GroupHomomorphismByImages` (see 7.113).

```
gap> pyl29 := Group( (1,2,3)(4,5,6)(7,8,9), (2,6,4,9,3,8,7,5),
>                      (4,7)(5,8)(6,9), (1,10)(4,7)(5,6)(8,9) );;
gap> pyl29.name := "pyl29";;
gap> cs := CompositionSeries( pyl29 );
[ Subgroup( pyl29, [ (1,9,5)(2,7,6)(3,8,4), (2,7,3,4)(5,8,9,6),
      ( 1, 2,10)( 4, 9, 5)( 6, 8, 7), (2,6,4,9,3,8,7,5),
      (4,7)(5,8)(6,9) ] ),
  Subgroup( pyl29, [ (1,9,5)(2,7,6)(3,8,4), (2,7,3,4)(5,8,9,6),
      ( 1, 2,10)( 4, 9, 5)( 6, 8, 7), (2,6,4,9,3,8,7,5) ] ),
  Subgroup( pyl29, [ (1,9,5)(2,7,6)(3,8,4), (2,7,3,4)(5,8,9,6),
      ( 1, 2,10)( 4, 9, 5)( 6, 8, 7) ] ), Subgroup( pyl29, [  ] ) ]
gap> List( [1..3], i->cs[i].operations.FactorGroup(cs[i],cs[i+1]) );
[ Group( (1,2) ), Group( (1,2) ),
  Group( (1,9,5)(2,7,6)(3,8,4), (2,7,3,4)(5,8,9,6), ( 1, 2,10)
    ( 4, 9, 5)( 6, 8, 7) ) ]
gap> List( last, Size );
[ 2, 2, 360 ]
```

`ExponentsPermSolvablePermGroup(` *G*`,` *perm* `[,` *start* `] )`

`ExponentsPermSolvablePermGroup` returns a list *e*, such that *perm* `=` *G*`.bssgs[1]^`*e*`[1] *` *G*`.bssgs[2]^`*e*`[2] * ... *` *G*`.bssgs[`*n*`]^`*e*`[`*n*`]`, where *G*`.bssgs` must be a prime-step base-strong subnormal generating system as calculated by `ElementaryAbelianSeries` (see 7.39 and above). If the optional third argument *start* is given, the list entries *exps*`[1]`, ..., *exps*`[`*start*`-1]` are left unbound and the element *perm* is decomposed as product of the remaining pag generators *G*`.bssgs[`*start*`], ...,` *G*`.bssgs[`*n*`]`.

```
gap> S := Group( (1,2,3,4), (1,2) );; S.name := "s4";;
gap> ElementaryAbelianSeries( S );;
gap> S.bssgs;
[ (1,2), (1,3,2), (1,4)(2,3), (1,2)(3,4) ]
gap> ExponentsPermSolvablePermGroup( S, (1,2,3) );
[ 0, 2, 0, 0 ]
```

`AgGroup(` *G* `)`

This function converts a solvable permutation group into an ag group. It calculates an elementary abelian series and a prime-step bssgs for *G* (see `ElementaryAbelianSeries` above) and then finds the relators belonging to this prime-step bssgs using the function `ExponentsPermSolvablePermGroup` (see above). An isomorphism from the ag group to the permutation group is bound to `AgGroup(` G `).bijection`.

```
gap> G := WreathProduct( SymmetricGroup( 4 ), CyclicGroup( 3 ) );;
gap> A := AgGroup( G );
Group( g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13 )
gap> (A.1*A.3)^A.bijection;
( 1, 6,10, 2, 5, 9)( 3, 7,11)( 4, 8,12)
```

`DirectProduct(` G`,` H `)`

If G and H are both permutation groups, `DirectProduct` constructs the direct product of G and H as an intransitive permutation group. There are special routines for `Centre`, `Centralizer` and `SylowSubgroup` for such groups that will work faster than the standard permutation group functions. These functions are `DirectProductPermGroupCentre`, `DirectProductPermGroupCentralizer` and `DirectProductPermGroupSylowSubgroup`. You can enforce that these routines will be always used for direct products of permutation groups by issuing the following three commands (They are not performed by standard as the code has not been well-tested).

```
gap> DirectProductPermGroupOps.Centre:=DirectProductPermGroupCentre;;
gap> DirectProductPermGroupOps.Centralizer:=
>    DirectProductPermGroupCentralizer;;
gap> DirectProductPermGroupOps.SylowSubgroup:=
>    DirectProductPermGroupSylowSubgroup;;
```

21.22 Operations of Permutation Groups

All functions that deal with operations of groups are applicable to permutation groups (see 8). This section describes which functions are implemented specially for permutation groups. Functions not mentioned here are handled by the default methods described in the respective sections.

`IsSemiRegular(` G`,` D`,` *opr* `)`

`IsSemiRegular` returns `true` if G operates semiregularly on the domain D and `false` otherwise.

If D is a list of integers and *opr* is `OnPoints`, `IsSemiRegular` uses the lemma that says that such an operation is semiregular if all orbits of G on D have the same length, and if for an arbitrary point p of D and for each generator g of G there is a permutation z_g (not necessarily in G) such that $p^{z_g} = p^g$ and which commutes with all elements of G, and if there is a permutation z (again not necessarily in G) that permutes the orbits of G on D setwise and commutes with all elements of G. This can be tested in time proportional to $on^2 + dn$, where o is the size of a single orbit, n is the number of generators of G, and d is the size of D.

`RepresentativeOperation(` G`,` d`,` e`,` *opr* `)`

`RepresentativeOperation` returns a permutation *perm* in G that maps d to e in respect to the given operation *opr* if such a permutation exists, and `false` otherwise.

If the operation is `OnPoints`, `OnPairs`, `OnTuples`, or `OnSets` and d and e are positive integers or lists of integers, a basechange is performed and the representative is computed from the factorized inverse transversal (see 21.6 and 21.7).

If the operation is `OnPoints`, `OnPairs`, `OnTuples` or `OnSets` and d and e are permutations or lists of permutations, a backtrack search is performed (see 21.16).

`Stabilizer(` G`,` D`,` *opr* `)`

`Stabilizer` returns the stabilizer of D in G using the operation *opr* on the D. If D is a positive integer (respectively a list of positive integers) and the operation *opr* is `OnPoints` (respectively `OnPairs` or `OnTuples`) a basechange of G is performed (see 21.8). If D is a set of positive integers and the operation *opr* is `OnSets` a backtrack algorithm for set-stabilizers of permutation groups is performed.

`Blocks(` G`,` D `[,` *seed* `] [,` *operation* `] )`

Returns a partition of D being a minimal block system of G in respect to the operation *opreration* on the objects of D. If the argument *seed* is given the objects of *seed* are contained in the same block. If D is a list of positive integers an Atkinson algorithm is performed.

Theoretically the algorithm lies in $\mathcal{O}(n^3m)$ but in practice it is mostly in $\mathcal{O}(n^2m)$ with m the number of generators and n the cardinality of D.

21.23 Homomorphisms for Permutation Groups

This section describes the various homomorphisms that are treated specially for permutation groups.

`GroupHomomorphisByImages(` P`,` H`,` *gens*`,` *imgs* `)`

The group homomorphism of a permutation group P into another group H is handled especially by `GroupHomomorphisByImages`. Below we describe how the various mapping functions are implemented for such a group homomorphism *ghom*. The mapping functions not mentioned below are implemented by the default functions described in 7.113.

To work with *ghom*, a stabilizer chain for the source of *ghom* is computed and stored as *ghom*`.orbit`, *ghom*`.transversal`, *ghom*`.stabilizer`. For every stabilizer *stab* in the stabilizer chain there is a list parallel to *stab*`.generators`, which is called *stab*`.genimages`, and contains images of the generators. The stabilizer chain is computed with a random Schreier Sims algorithm, using the size of the source to know when to stop.

`IsMapping(` *ghom* `)`

To test if *ghom* is a (single valued) mapping, all Schreier generatores are computed. Each Schreier generator is then reduced along the stabilizer chain. Because the chain is complete, each one must reduce to the identity. Parallel the images of the strong generators are multiplied. If they also reduce to the identity (in the range), *ghom* is a function, otherwise the remainders form a normal generating set for the subgroup of images of the identity of the source.

`Image(` *ghom*`,` *elm* `)`

The image of an element *elm* can be computed by reducing the element along the stabilizer chain, and at each step multiplying the corresponding images of the strong generators.

`CompositionMapping(` *hom*`,` *ghom* `)`

The composition of an arbitrary group homomorphism *hom* and *ghom* the stabilizer chain of *ghom* is copied. On each level the images of the generators in *stab*`.genimages` are replaced by their images under *hom*.

`OperationHomomorphism(` *P*`, Operation(` *P*`,` *list* `) )`

The operation of a permutation group *P* on a list *list* of integers is handled especially by `OperationHomomorphism`. (Note that *list* must be a union of orbits of *P* for `Operation` to work.) We call the resulting homomorphism a **transitive constituent** homomorphism. Below we describe how the various mapping functions are implemented for a transitive constituent homomorphism *tchom*. The mapping functions not mentioned below are implemented by the default functions described in 8.21.

`Image(` *tchom*`,` *elm* `)`

The image of an element is computed by restricting *elm* to *list* (see 20.10) and conjugating the restricted permutation with *tchom*`.conperm`, which maps it to a permutation that operates on `[1..Length(`*list*`)]` instead of *list*.

`Image(` *tchom*`,` *H* `)`

The image of a subgroup *H* is computed as follows. First a stabilizer chain for *H* is computed. This stabilizer chain is such that the base starts with points in *list*. Then the images of the strong generators of *sub* form a strong generating set of the image.

`PreImages(` *tchom*`,` *H* `)`

The preimage of a subgroup *H* is computed as follows. First a stabilizer chain for the source of *tchom* is computed. This stabilizer chain is such that the base starts with the point in *list*. Then the kernel of *tchom* is a stabilizer in this stabilizer chain. The preimages of the strong generators for *H* together with the strong generators for the kernel form a strong generating set of the preimage subgroup.

`OperationHomomorphism(` *P*`, Operation(` *P*`,` *blocks*`, OnSets ) )`

The operation of a permutation group *P* on a block system *blocks* (see 8.22) is handled especially by `OperationHomomorphism`. We call the resulting homomorphism a **blocks homomorphism**. Below we describe how the various mapping functions are implemented for a blocks homomorphism *bhom*. The mapping functions not mentioned below are implemented by the default functions described in 8.21.

`Image(` *bhom*`,` *elm* `)`

To compute the image of an element *elm* under *bhom*, the record for *bhom* contains a list *bhom*`.reps`, which contains for each point in the union of the blocks the position of this block in *blocks*. Then the image of an element can simply be computed by applying the element to a representative of each block and using *bhom*`.reps` to find in which block the image lies.

`Image(` *bhom*`,` *H* `)`
`PreImage(` *bhom*`,` *elm* `)`
`PreImage(` *bhom*`,` *H* `)`
`Kernel(` *bhom* `)`

The image of a subgroup, the preimage of an element, and the preimage of a subgroup are computed by rather complicated algorithms. For a description of these algorithms see [But85].

21.24 Random Methods for Permutation Groups

When permutation groups become larger, computations become slower. This increase might make it impossible to compute with these groups. The reason is mainly the creation of stabilizer chains (see 21.7): During this process a lot of schreier generators are produced for the next point stabilizer in the chain, and these generators must be processed. In actual examples, it is observed, however, that much fewer generators are needed. This observation can be justified theoretically and the random methods exploit it by using a method of the Schreier-Sims algorithm which gives the correct result with an user-given error probability.

Advantage

Computations become much faster. In fact, large problems may be handled only by using random methods.

Disadvantages

Computations might produce wrong results. However, you can set an error margin, which is guaranteed. The practical performance is even better than our guarantee. You should also keep in mind, that it is impossible, to eliminate system, user or programming errors.

However, there are many situations, when theory offers methods to check correctness of the results. As an example, consider the following situation. You want to compute some maximal subgroups of large sporadic groups. The ATLAS of finite groups then tells you the sizes of the groups as well as the sizes of the subgroups. The error of the random methods is one-sided in the sense that they never create strong generators which are not elements of the group. Hence if the resulting group sizes are correct, you have indeed obtained the correct result. You might also give this information to `StabChain`, and computation will not only be much faster, but also corresponding to the information, i.e. if you give the size, the stabilizer chain is computed correctly.

The stabilizer chain is computed using methods from [BCFS91].

How to use the random methods

GAP3 provides the global variable `StabChainOptions`. This record might contain a component `random`. If it is set to a number i between 1 and 1000 at the beginning, random methods with guaranteed correctness $\frac{i}{10}$ percent are used (though practically the probability for correctness is much higher). This means that at all applicable places random methods will be used automatically by the same function calls. If the component is not set or set to 1000, all computations are deterministic. By standard, this component is not set, so unless you explicitely allow random computations none are used.

If the group acts on not more than a hundered points, the use of random methods has no advantage. For these groups always the deterministic methods are used.

```
gap> g:=SL(4,7);
SL(4,7)
gap> o:=Orbit(g,[1,0,0,0]*Z(7)^0,OnLines);;Length(o);
400
gap> op:=Operation(g,o,OnLines);;
```

We create a large permutation group on 400 points. First we compute deterministic.

```
gap> g:=Group(op.generators,());;
gap> StabChain(g);;time;
164736
gap> Size(g);
2317591180800
```

Now random methods will be used. We allow that the result is guaranteed correct only with 10 percent probability. The group is created anew.

```
gap> StabChainOptions.random:=100;
100
gap> g:=Group(op.generators,());;
gap> StabChain(g);;time;
10350
gap> Size(g);
2317591180800
```

The result is still correct, though it took only less than one tenth of the time (your mileage may vary). If you give the algorithm a chance to check its results, things become even faster.

```
gap> g:=Group(op.generators,());;
gap> StabChain(g,rec(size:=2317591180800));;time;
5054
```

More about random methods

When stabilizer chains are created, while random methods are allowed, it is noted in the respective groups, by setting of a record component G.`stabChainOptions`, which is itself a record, containg the component `random`. This component has the value indicated by `StabChainOptions` at the time the group was created. Values set in this component override the global setting of `StabChainOptions`. Whenever stabilizer chains are created for a group not posessing the `.stabChainOptions.random` entry, it is created anew from the global value `StabChainOptions`.

If a subgroup has no own record `stabChainOptions`, the one of the parent group is used instead.

As errors induced by the random functions might propagate, any (applicable) object created from the group inherits the component `.stabChainOptions` from the group. This applies for example to `Operation`s and Homomorphisms.

21.25 Permutation Group Records

All groups are represented by a record that contains information about the group. A permutation group record contains the following components in addition to those described in section 7.118.

`isPermGroup`
: always `true`.

`isFinite`
: always `true` as permutation groups are always of finite order.

A stabilizer chain (see 21.6) is stored recursively in GAP3. The group record of a permutation group G with a stabilizer chain has the following additional components.

`orbit`
: List of orbitpoints of `orbit[1]` (which is the basepoint) under the action of the generators.

`transversal`
: Factorized inverse transversal as defined in 21.6.

`stabilizer`
: Record for the stabilizer of the point `orbit[1]` in the group generated by `generators`. The components of this record are again `generators`, `orbit`, `transversal` and `stabilizer`. The last stabilizer in the stabilizer chain only contains the component `generators`, which is an empty list.

`stabChainOptions`
: A record, that contains information about creation of the stabilizer chain. For example, whether it has been computed using random methods (see 21.24). Some functions also use this record for passing local information about basechanges.

`stabChain`
: A record, that contains all information about the stabilizer chain. Functions acessing the stabilizer chain should do it using this record, as it is planned to remove the above three components from the group record in the future. The components of the `stabChain` record are described below.

The components of the `stabChain` record for a group G are

`identity`
: Contains G`.identity`.

`generators`
: Contains a copy of the generators of G, created by `ShallowCopy(G.generators)`.

`orbit`
: is the same as G`.orbit`.

`transversal`
: is the same as G`.transversal`.

`stabilizer`
: is the same as G`.stabilizer`.

Note that the values of all these components are changed by functions that change, extend, or reduce a base (see 21.8, 21.9, and 21.10).

Note that the records that represent the stabilizers are not themselves group records (see 7.118). Thus you cannot take such a stabilizer and apply group functions to it. The last `stabilizer` in the stabilizer chain is a record whose component `generators` is empty.

Chapter 22

Words in Abstract Generators

Words in abstract generators are a type of group elements in GAP3. In the following we will abbreviate their full name to **abstract words** or just to **words**.

A word is just a sequence of letters, where each letter is an abstract generator or its inverse. Words are multiplied by concatenating them and removing adjacent pairs of a generator and its inverse. Abstract generators are created by the function `AbstractGenerator` (see 22.1).

Note that words do not belong to a certain group. Any two words can be multiplied. In effect we compute with words in a free group of potentially infinite rank (potentially infinite because we can always create new abstract generators with `AbstractGenerator`).

Words are entered as expressions in abstract generators and are displayed as product of abstract generators (and powers thereof). The trivial word can be entered and is displayed as `IdWord`.

```
gap> a := AbstractGenerator( "a" );
a
gap> b := AbstractGenerator( "b" );
b
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> a^0;
IdWord
```

The first sections in this chapter describe the functions that create abstract generators (see 22.1 and 22.2). The next sections define the operations for words (see 22.3 and 22.4). The next section describes the function that tests whether an object is a word (see 22.5). The next sections describe the functions that compute the number of letters of a word (see 22.6 and 22.7). The next sections describe the functions that extract or find a subword (see 22.8 and 22.9). The final sections describe the functions that modify words (see 22.10, 22.11, and 22.12).

Note that words in abstract generators are different from words in finite polycyclic groups (see 24).

22.1 AbstractGenerator

`AbstractGenerator(` *string* `)`

`AbstractGenerator` returns a new abstract generator. This abstract generator is printed using the string *string* passed as argument to `AbstractGenerator`.

```
gap> a := AbstractGenerator( "a" );
a
gap> a^5;
a^5
```

Note that the string is only used to print the abstract generator and to order abstract generators (see 22.3). It is possible for two different abstract generators to use the same string and still be different.

```
gap> b := AbstractGenerator( "a" );
a
gap> a = b;
false
```

Also when you define abstract generators interactively it is a good idea to use the identifier of the variable as the name of the abstract generator, because then what GAP3 will output for a word is equal to what you can input to obtain this word. The following is an example of what you should probably **not** do.

```
gap> c := AbstractGenerator( "d" );
d
gap> d := AbstractGenerator( "c" );
c
gap> (c*d)^3;
d*c*d*c*d*c
gap> d*c*d*c*d*c;
c*d*c*d*c*d
```

22.2 AbstractGenerators

`AbstractGenerators(` *string*, *n* `)`

`AbstractGenerators` returns a list of n new abstract generators. These new generators are printed using *string*1, *string*2, ..., *string*n.

```
gap> AbstractGenerators( "a", 3 );
[ a1, a2, a3 ]
```

`AbstractGenerators` could be defined as follows (see 22.1).

```
AbstractGenerators := function ( string, n )
    local   gens, i;
    gens := [];
    for i  in [1..n]  do
        Add( gens,
             AbstractGenerator(
                 ConcatenationString( string, String(i) ) ) );
```

```
    od;
    return gens;
end;
```

22.3 Comparisons of Words

w1 = *w2*
w1 <> *w2*

The equality operator = evaluates to `true` if the two words *w1* and *w2* are equal and to `false` otherwise. The inequality operator <> evaluates to `true` if the two words *w1* and *w2* are not equal and to `false` otherwise.

You can compare words with objects of other types, but they are never equal of course.

```
gap> a := AbstractGenerator( "a" );;
gap> b := AbstractGenerator( "b" );;
gap> a = b;
false
gap> (a^2*b)^5*b^-1 = a^2*b*a^2*b*a^2*b*a^2*b*a^2;
true
```

w1 < *w2*
w1 <= *w2*
w1 > *w2*
w1 >= *w2*

The operators <, <=, >, and => evaluate to `true` if the word *w1* is less than, less than or equal to, greater than, and greater than or equal to the word *w2*.

Words are ordered as follows. One word *w1* is considered smaller than another word *w2* it it is shorted, or, if they have the same length, if it is first in the lexicographical ordering implied by the ordering of the abstract generators. The ordering of abstract generators is as follows. The abstract generators are ordered with respect to the strings that were passed to `AbstractGenerator` when creating these abstract generators. Each abstract generator *g* is also smaller than its inverse, but this inverse is smaller than any abstract generator that is larger than *g*.

Words can also be compared with objects of other types. Integers, rationals, cyclotomics, finite field elements, and permutations are smaller than words, everything else is larger.

```
gap> IdWord<a;  a<a^-1;  a^-1<b;  b<b^-1;  b^-1<a^2; a^2<a*b;
true
true
true
true
true
true
```

22.4 Operations for Words

w1 * *w2*

The operator `*` evaluates to the product of the two words *w1* and *w2*. Note that words do not belong to a specific group, thus any two words can be multiplied. Multiplication of words is done by concatenating the words and removing adjacent pairs of an abstract generator and its inverse.

w1 / *w2*

The operator `/` evaluates to the quotient $w1 * w2^{-1}$ of the two words *w1* and *w2*. Inversion of a word is done by reversing the order of its letters and replacing each abstract generator with its inverse.

w1 ^ *w2*

The operator `^` evaluates to the conjugate $w2^{-1} * w1 * w2$ of the word *w1* under the word *w2*.

w1 ^ *i*

The powering operator `^` returns the *i*-th power of the word *w1*, where *i* must be an integer. If *i* is zero, the value is `IdWord`.

list * *w1*
w1 * *list*

In this form the operator `*` returns a new list where each entry is the product of *w1* and the corresponding entry of *list*. Of course multiplication must be defined between *w1* and each entry of *list*.

list / *w1*

In this form the operator `/` returns a new list where each entry is the quotient of *w1* and the corresponding entry of *list*. Of course division must be defined between *w1* and each entry of *list*.

`Comm(` *w1* , *w2* `)`

`Comm` returns the commutator $w1^{-1} * w2^{-1} * w1 * w2$ of two words *w1* and *w2*.

`LeftQuotient(` *w1* , *w2* `)`

`LeftQuotient` returns the left quotient $w1^{-1} * w2$ of two words *w1* and *w2*.

22.5 IsWord

`IsWord(` *obj* `)`

`IsWord` returns `true` if the object *obj*, which may be an object of arbitrary type, is a word and `false` otherwise. Signals an error if *obj* is an unbound variable.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
```

```
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> IsWord( w );
true
gap> a := (1,2,3);;
gap> IsWord( a^2 );
false
```

22.6 LengthWord

`LengthWord(` *w* `)`

`LengthWord` returns the length of the word w, i.e., the number of letters in the word.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> LengthWord( w );
14
gap> LengthWord( a^13 );
13
gap> LengthWord( IdWord );
0
```

22.7 ExponentSumWord

`ExponentSumWord(` *w*, *gen* `)`

`ExponentSumWord` returns the number of times the generator *gen* appears in the word w minus the number of times its inverse appears in w. If *gen* and its inverse do no occur in w, 0 is returned. *gen* may also be the inverse of a generator of course.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> ExponentSumWord( w, a );
10
gap> ExponentSumWord( w, b );
4
gap> ExponentSumWord( (a*b*a^-1)^3, a );
0
gap> ExponentSumWord( (a*b*a^-1)^3, b^-1 );
-3
```

22.8 Subword

`Subword(` *w*, *from*, *to* `)`

`Subword` returns the subword of the word w that begins at position *from* and ends at position *to*. *from* and *to* must be positive integers. Indexing is done with origin 1.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> Subword( w, 5, 8 );
a*b*a^2
```

22.9 PositionWord

`PositionWord(` *w*, *sub*, *from* `)`

`PositionWord` returns the position of the first occurrence of the word *sub* in the word *w* starting at position *from*. If there is no such occurrence, `false` is returned. *from* must be a positive integer. Indexing is done with origin 1.

In other words, `PositionWord(`*w*,*sub*,*from*`)` returns the smallest integer *i* larger than or equal to *from* such that `Subword(` *w*, *i*, *i*`+LengthWord(`*sub*`)-1 ) =` *sub* (see 22.8).

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> PositionWord( w, a^2*b, 2 );
4
gap> PositionWord( w, a*b^2, 2 );
false
```

22.10 SubstitutedWord

`SubstitutedWord(` *w*, *from*, *to*, *by* `)`

`SubstitutedWord` returns a new word where the subword of the word *w* that begins at position *from* and ends at position *to* is replaced by the word *by*. *from* and *to* must be positive integers. Indexing is done with origin 1.

In other words `SubstitutedWord(`*w*,*from*,*to*,*by*`)` is the word `Subword(`*w*`,1,`*from*`-1) *` *by* `* Subword(`*w*,*to*`+1,LengthWord(`*w*`)` (see 22.8).

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> SubstitutedWord(w,5,8,b^-1);
a^2*b*a^3*b*a^2
```

22.11 EliminatedWord

`EliminatedWord(` *word*, *gen*, *by* `)`

`EliminatedWord` returns a new word where each occurrence of the generator *gen* is replaced by the word *by*.

```
gap> a := AbstractGenerator("a");;
```

```
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> EliminatedWord( w, b, b^2 );
a^2*b^2*a^2*b^2*a^2*b^2*a^2*b^2*a^2
```

22.12 MappedWord

`MappedWord(` *w*, *gens*, *imgs* `)`

`MappedWord` returns the new group element that is obtained by replacing each occurrence of a generator *gen* in the list of generators *gens* by the corresponding group element *img* in the list of group elements *imgs*. The lists *gens* and *imgs* must of course have the same length.

```
gap> a := AbstractGenerator("a");;
gap> b := AbstractGenerator("b");;
gap> w := (a^2*b)^5*b^-1;
a^2*b*a^2*b*a^2*b*a^2*b*a^2
gap> MappedWord( w, [a,b], [(1,2,3),(1,2)] );
(1,3,2)
```

If the images in *imgs* are all words, and some of them are equal to the corresponding generators in *gens*, then those may be omitted.

```
gap> MappedWord( w, [a], [a^2] );
a^4*b*a^4*b*a^4*b*a^4*b*a^4
```

Note that the special case that the list *gens* and *imgs* have only length 1 is handled more efficiently by `EliminatedWord` (see 22.11).

Chapter 23

Finitely Presented Groups

A **finitely presented group** is a group generated by a set of **abstract generators** subject to a set of **relations** that these generators satisfy. Each group can be represented as finitely presented group.

A finitely presented group is constructed as follows. First create an appropriate free group (see 23.1). Then create the finitely presented group as a factor of this free group by the relators.

```
gap> F2 := FreeGroup( "a", "b" );
Group( a, b )
gap> A5 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^5 ];
Group( a, b )
gap> Size( A5 );
60
gap> a := A5.1;;  b := A5.2;;
gap> Index( A5, Subgroup( A5, [ a*b ] ) );
12
```

Note that, even though the generators print with the names given to `FreeGroup`, no variables of that name are defined. That means that the generators must be entered as *free-group*`.`*number* and *fp-group*`.`*number*.

Note that the generators of the free group are different from the generators of the finitely presented group (even though they print with the same name). That means that words in the generators of the free group are not elements of the finitely presented group.

Note that the relations are entered as **relators**, i.e., as words in the generators of the free group. To enter an equation use the quotient operator, i.e., for the relation $a^b = ab$ you have to enter `a^b/(a*b)`.

You must **not** change the relators of a finitely presented group at all.

The elements of a finitely presented group are words. There is one fundamental problem with this. Different words can correspond to the same element in a finitely presented group. For example in the group `A5` defined above, `a` and `a^3` are actually the same element. However, `a` is not equal to `a^3` (in the sense that `a = a^3` is `false`). This leads to the following anomaly: `a^3 in A5` is `true`, but `a^3 in Elements(A5)` is `false`. **Some set and group**

functions will not work correctly because of this problem. You should therefore only use the functions mentioned in 23.2 and 23.3.

The first section in this chapter describes the function `FreeGroup` that creates a free group (see 23.1). The next sections describe which set theoretic and group functions are implemented specially for finitely presented groups and how they work (see 23.2 and 23.3). The next section describes the basic function `CosetTableFpGroup` that is used by most other functions for finitely presented groups (see 23.4). The next section describes how you can compute a permutation group that is a homomorphic image of a finitely presented group (see 23.5). The final section describes the function that finds all subgroups of a finitely presented group of small index (see 23.7).

23.1 FreeGroup

`FreeGroup(` *n* `)`
`FreeGroup(` *n*`,` *string* `)`
`FreeGroup(` *name1*`,` *name2*`..` `)`

`FreeGroup` returns the free group on *n* generators. The generators are displayed as *string*`.1`, *string*`.2`, ..., *string*`.`*n*. If *string* is missing it defaults to `"f"`. If *string* is the name of the variable that you use to refer to the group returned by `FreeGroup` you can also enter the generators as *string*`.`*i*.

```
gap> F2 := FreeGroup( 2, "A5" );;
gap> A5 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^5 ];
Group( A5.1, A5.2 )
gap> Size( A5 );
60
gap> F2 := FreeGroup( "a", "b" );;
gap> D8 := F2 / [ F2.1^4, F2.2^2, F2.1^F2.2 / F2.1 ];
Group( a, b )
gap> a := D8.1;;  b := D8.2;;
gap> Index( D8, Subgroup( D8, [ a ] ) );
2
```

23.2 Set Functions for Finitely Presented Groups

Finitely presented groups are domains, thus in principle all set theoretic functions are applicable to them (see chapter 4). However because words that are not equal may denote the same element of a finitely presented group many of them will not work correctly. This sections describes which set theoretic functions are implemented specially for finitely presented groups and how they work. You should **not** use the set theoretic functions that are not mentioned in this section.

The general information that enables GAP3 to work with a finitely presented group G is a **coset table** (see 23.4). Basically a coset table is the permutation representation of the finitely presented group on the cosets of a subgroup (which need not be faithful if the subgroup has a nontrivial core). Most of the functions below use the regular representation of G, i.e., the coset table of G over the trivial subgroup. Such a coset table is computed by a method called **coset enumeration**.

`Size(` *G* `)`

The size is simply the degree of the regular representation of G.

w `in` *G*

A word w lies in a parent group G if all its letters are among the generators of G.

w `in` *H*

To test whether a word w lies in a subgroup H of a finitely presented group G, GAP3 computes the coset table of G over H. Then it tests whether the permutation one gets by replacing each generator of G in w with the corresponding permutation is trivial.

`Elements(` *G* `)`

The elements of a finitely presented group are computed by computing the regular representation of G. Then for each point p GAP3 adds the smallest word w that, when viewed as a permutation, takes 1 to p to the set of elements. Note that this implies that each word in the set returned is the smallest word that denotes an element of G.

`Elements(` *H* `)`

The elements of a subgroup H of a finitely presented group G are computed by computing the elements of G and returning those that lie in H.

`Intersection(` *H1* `,` *H2* `)`

The intersection of two subgroups *H1* and *H2* of a finitely presented group G is computed as follows. First GAP3 computes the coset tables of G over *H1* and *H2*. Then it computes the tensor product of those two permutation representations. The coset table of the intersection is the transitive constituent of 1 in this tensored permutation representation. Finally GAP3 computes a set of Schreier generators for the intersection by performing another coset enumeration using the already complete coset table. The intersection is returned as the subgroup generated by those Schreier generators.

23.3 Group Functions for Finitely Presented Groups

Finitely presented groups are after all groups, thus in principle all group functions are applicable to them (see chapter 7). However because words that are not equal may denote the same element of a finitely presented group many of them will not work correctly. This sections describes which group functions are implemented specially for finitely presented groups and how they work. You should **not** use the group functions that are not mentioned in this section.

The general information that enables GAP3 to work with a finitely presented group G is a **coset table** (see 23.4). Basically a coset table is the permutation representation of the finitely presented group on the cosets of a subgroup (which need not be faithful if the subgroup has a nontrivial core). Most of the functions below use the regular representation of G, i.e., the coset table of G over the trivial subgroup. Such a coset table is computed by a method called **coset enumeration**.

`Order(` G`,` g `)`

The order of an element g is computed by translating the element into the regular permutation representation and computing the order of this permutation (which is the length of the cycle of 1).

`Index(` G`,` H `)`

The index of a subgroup H in a finitely presented group G is simply the degree of the permutation representation of the group G on the cosets of H.

`Normalizer(` G`,` H `)`

The normalizer of a subgroup H of a finitely presented group G is the union of those cosets of H in G that are fixed by all the generators of H when viewed as permutations in the permutation representation of G on the cosets of H. The normalizer is returned as the subgroup generated by the generators of H and representatives of such cosets.

`CommutatorFactorGroup(` G `)`

The commutator factor group of a finitely presented group G is returned as a new finitely presented group. The relations of this group are the relations of G plus the commutator of all the pairs of generators of G.

`AbelianInvariants(` G `)`

The abelian invariants of a abelian finitely presented group (e.g., a commutator factor group of an arbitrary finitely presented group) are computed by building the relation matrix of G and transforming this matrix to diagonal form with `ElementaryDivisorsMat` (see 34.23).

`AbelianInvariantsSubgroupFpGroup(` G`,` H `)`
`AbelianInvariantsSubgroupFpGroup(` G`,` *cosettable* `)`

This function is equivalent to `AbelianInvariantsSubgroupFpGroupRrs` below, but note that there is an alternative function, `AbelianInvariantsSubgroupFpGroupMtc`.

`AbelianInvariantsSubgroupFpGroupRrs(` G`,` H `)`
`AbelianInvariantsSubgroupFpGroupRrs(` G`,` *cosettable* `)`

`AbelianInvariantsSubgroupFpGroupRrs` returns the invariants of the commutator factor group H/H' of a subgroup H of a finitely presented group G. They are computed by first applying an abelianized Reduced Reidemeister-Schreier procedure (see 23.11) to construct a relation matrix of H/H' and then transforming this matrix to diagonal form with `ElementaryDivisorsMat` (see 34.23).

As second argument, you may provide either the subgroup H itself or its coset table in G.

`AbelianInvariantsSubgroupFpGroupMtc(` *G*`,` *H* `)`

`AbelianInvariantsSubgroupFpGroupMtc` returns the invariants of the commutator factor group *H/H'* of a subgroup *H* of a finitely presented group *G*. They are computed by applying an abelianized Modified Todd-Coxeter procedure (see 23.11) to construct a relation matrix of *H/H'* and then transforming this matrix to diagonal form with `ElementaryDivisorsMat` (see 34.23).

`AbelianInvariantsNormalClosureFpGroup(` *G*`,` *H* `)`

This function is equivalent to `AbelianInvariantsNormalClosureFpGroupRrs` below.

`AbelianInvariantsNormalClosureFpGroupRrs(` *G*`,` *H* `)`

`AbelianInvariantsNormalClosureFpGroupRrs` returns the invariants of the commutator factor group *N/N'* of the normal closure *N* a subgroup *H* of a finitely presented group *G*. They are computed by first applying an abelianized Reduced Reidemeister-Schreier procedure (see 23.11) to construct a relation matrix of *N/N'* and then transforming this matrix to diagonal form with `ElementaryDivisorsMat` (see 34.23).

```
gap> # Define the Coxeter group E1.
gap> F5 := FreeGroup( "x1", "x2", "x3", "x4", "x5" );;
gap> E1 := F5 / [ F5.1^2, F5.2^2, F5.3^2, F5.4^2, F5.5^2,
>  ( F5.1 * F5.3 )^2, ( F5.2 * F5.4 )^2, ( F5.1 * F5.2 )^3,
>  ( F5.2 * F5.3 )^3, ( F5.3 * F5.4 )^3, ( F5.4 * F5.1 )^3,
>  ( F5.1 * F5.5 )^3, ( F5.2 * F5.5 )^2, ( F5.3 * F5.5 )^3,
>  ( F5.4 * F5.5 )^2,
>  ( F5.1 * F5.2 * F5.3 * F5.4 * F5.3 * F5.2 )^2 ];;
gap> x1:=E1.1;;  x2:=E1.2;;  x3:=E1.3;;  x4:=E1.4;;  x5:=E1.5;;
gap> # Get normal subgroup generators for B1.
gap> H := Subgroup( E1, [ x5 * x2^-1, x5 * x4^-1 ] );;
gap> # Compute the abelian invariants of B1/B1'.
gap> A := AbelianInvariantsNormalClosureFpGroup( E1, H );
[ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> # Compute a presentation for B1.
gap> P := PresentationNormalClosure( E1, H );
<< presentation with 18 gens and 46 rels of total length 132 >>
gap> SimplifyPresentation( P );
#I  there are 8 generators and 30 relators of total length 148
gap> B1 := FpGroupPresentation( P );
Group( _x1, _x2, _x3, _x4, _x6, _x7, _x8, _x11 )
gap> # Compute normal subgroup generators for B1'.
gap> gens := B1.generators;;
gap> numgens := Length( gens );;
gap> comms := [ ];;
gap> for i in [ 1 .. numgens - 1 ] do
>       for j in [i+1 .. numgens ] do
>           Add( comms, Comm( gens[i], gens[j] ) );
>       od;
```

```
>    od;
gap> # Compute the abelian invariants of B1'/B1".
gap> K := Subgroup( B1, comms );;
gap> A := AbelianInvariantsNormalClosureFpGroup( B1, K );
[ 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

The prededing calculation for B_1 and a similar one for B_0 have been used to prove that $B_1'/B_1'' \cong Z_2^9 \times Z^3$ and $B_0'/B_0'' \cong Z_2^{91} \times Z^{27}$ as stated in Proposition 5 in [FJNT95].

The following functions are not implemented specially for finitely presented groups, but they work nevertheless. However, you probably should not use them for larger finitely presented groups.

`Core(` *G*`,` *U* `)`
`SylowSubgroup(` *G*`,` *p* `)`
`FittingSubgroup(` *G* `)`

23.4 CosetTableFpGroup

`CosetTableFpGroup(` *G*`,` *H* `)`

`CosetTableFpGroup` returns the coset table of the finitely presented group G on the cosets of the subgroup H.

Basically a coset table is the permutation representation of the finitely presented group on the cosets of a subgroup (which need not be faithful if the subgroup has a nontrivial core). Most of the set theoretic and group functions use the regular representation of G, i.e., the coset table of G over the trivial subgroup.

The coset table is returned as a list of lists. For each generator of G and its inverse the table contains a generator list. A generator list is simply a list of integers. If l is the generator list for the generator g and `l[i] = j` then generator g takes the coset i to the coset j by multiplication from the right. Thus the permutation representation of G on the cosets of H is obtained by applying `PermList` to each generator list (see 20.9). The coset table is standardized, i.e., the cosets are sorted with respect to the smallest word that lies in each coset.

```
gap> F2 := FreeGroup( "a", "b" );
Group( a, b )
gap> A5 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^5 ];
Group( a, b )
gap> CosetTableFpGroup( A5,
>             Subgroup( A5, [ A5.1, A5.2*A5.1*A5.2*A5.1*A5.2^-1 ] ) );
[ [ 1, 3, 2, 5, 4 ],
  [ 1, 3, 2, 5, 4 ],    # inverse of above, A5.1 is an involution
  [ 2, 4, 3, 1, 5 ],
  [ 4, 1, 3, 2, 5 ] ]   # inverse of above
gap> List( last, PermList );
[ (2,3)(4,5), (2,3)(4,5), (1,2,4), (1,4,2) ]
```

The coset table is computed by a method called **coset enumeration**. A **Felsch strategy** is used to decide how to define new cosets.

The variable `CosetTableFpGroupDefaultLimit` determines for how many cosets the table has initially room. `CosetTableFpGroup` will automatically extend this table if need arises, but this is an expensive operation. Thus you should set `CosetTableFpGroupDefaultLimit` to the number of cosets that you expect will be needed at most. However you should not set it too high, otherwise too much space will be used by the coset table.

The variable `CosetTableFpGroupDefaultMaxLimit` determines the maximal size of the coset table. If a coset enumeration reaches this limit it signals an error and enters the breakloop. You can either continue or quit the computation from there. Setting the limit to `0` allows arbitrary large coset tables.

23.5 OperationCosetsFpGroup

`OperationCosetsFpGroup(` G, H `)`

`OperationCosetsFpGroup` returns the permutation representation of the finitely presented group G on the cosets of the subgroup H as a permutation group. Note that this permutation representation is faithful if and only if H has a trivial core in G.

```
gap> F2 := FreeGroup( "a", "b" );
Group( a, b )
gap> A5 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^5 ];
Group( a, b )
gap> OperationCosetsFpGroup( A5,
>             Subgroup( A5, [ A5.1, A5.2*A5.1*A5.2*A5.1*A5.2^-1 ] ) );
Group( (2,3)(4,5), (1,2,4) )
gap> Size( last );
60
```

`OperationCosetsFpGroup` simply calls `CosetTableFpGroup`, applies `PermList` to each row of the table, and returns the group generated by those permutations (see 23.4, 20.9).

23.6 IsIdenticalPresentationFpGroup

`IsIdenticalPresentationFpGroup(` G, H `)`

`IsIdenticalPresentationFpGroup` returns `true` if the presentations of the parent groups G and H are identical and `false` otherwise.

Two presentations are considered identical if the have the same number of generators, i.e., G is generated by *g1 ... gn* and H by *h1 ... hn*, and if the set of relators of G stored in G`.relators` is equal to the set of relators of H stored in H`.relators` **after** replacing *hi* by *gi* in these words.

```
gap> F2 := FreeGroup(2);
Group( f.1, f.2 )
gap> g := F2 / [ F2.1^2 / F2.2 ];
Group( f.1, f.2 )
gap> h := F2 / [ F2.1^2 / F2.2 ];
Group( f.1, f.2 )
gap> g = h;
false
gap> IsIdenticalPresentationFpGroup( g, h );
true
```

23.7 LowIndexSubgroupsFpGroup

`LowIndexSubgroupsFpGroup(` *G*, *H*, *index* `)`
`LowIndexSubgroupsFpGroup(` *G*, *H*, *index*, *excluded* `)`

`LowIndexSubgroupsFpGroup` returns a list of representatives of the conjugacy classes of subgroups of the finitely presented group G that contain the subgroup H of H and that have index less than or equal to *index*.

The function provides some intermediate output if `InfoFpGroup2` has been set to `Print` (its default value is `Ignore`).

If the optional argument *excluded* has been specified, then it is expected to be a list of words in the generators of G, and `LowIndexSubgroupsFpGroup` returns only those subgroups of index at most *index* that contain H, but do not contain any conjugate of any of the group elements defined by these words.

```
gap> F2 := FreeGroup( "a", "b" );
Group( a, b )
gap> A5 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^5 ];
Group( a, b )
gap> A5.name := "A5";;
gap> S := LowIndexSubgroupsFpGroup( A5, TrivialSubgroup( A5 ), 12 );
[ A5, Subgroup( A5, [ a, b*a*b^-1 ] ),
  Subgroup( A5, [ a, b*a*b*a^-1*b^-1 ] ),
  Subgroup( A5, [ a, b*a*b*a*b^-1*a^-1*b^-1 ] ),
  Subgroup( A5, [ b*a^-1 ] ) ]
gap> List( S, H -> Index( A5, H ) );
[ 1, 6, 5, 10, 12 ]     # the indices of the subgroups
gap> List( S, H -> Index( A5, Normalizer( A5, H ) ) );
[ 1, 6, 5, 10, 6 ]     # the lengths of the conjugacy classes
```

As an example for an application of the optional parameter *excluded*, we compute all conjugacy classes of torsion free subgroups of index at most 24 in the group $G = \langle x, y, z \mid x^2, y^4, z^3, (xy)^3, (yz)^2, (xz)^3 \rangle$. It is know from theory that each torsion element of this group is conjugate to a power of x, y, z, xy, xz, or yz.

```
gap> G := FreeGroup( "x", "y", "z" );
Group( x, y, z )
gap> x := G.1;; y := G.2;; z := G.3;;
gap> G.relators := [ x^2, y^4, z^3, (x*y)^3, (y*z)^2, (x*z)^3 ];;
gap> torsion := [ x, y, y^2, z, x*y, x*z, y*z ];;
gap> InfoFpGroup2 := Print;;
gap> lis :=
>    LowIndexSubgroupsFpGroup( G, TrivialSubgroup( G ), 24, torsion );;
#I   class 1 of index 24 and length 8
#I   class 2 of index 24 and length 24
#I   class 3 of index 24 and length 24
#I   class 4 of index 24 and length 24
#I   class 5 of index 24 and length 24
gap> InfoFpGroup2 := Ignore;;
gap> lis;
```

```
[ Subgroup( Group( x, y, z ),
    [ x*y*z^-1, z*x*z^-1*y^-1, x*z*x*y^-1*z^-1, y*x*z*y^-1*z^-1 ] ),
  Subgroup( Group( x, y, z ),
    [ x*y*z^-1, z^2*x^-1*y^-1, x*z*y*x^-1*z^-1 ] ),
  Subgroup( Group( x, y, z ),
    [ x*y*z^-1, x*z^2*x^-1*y^-1, y^2*x*y^-1*z^-1*x^-1 ] ),
  Subgroup( Group( x, y, z ), [ x*y*z^-1, y^3*x^-1*z^-1*x^-1,
      y^2*z*x^-1*y^-1 ] ),
  Subgroup( Group( x, y, z ), [ y*x*z^-1, x*y*z*y^-1*z^-1,
      y^2*z*x^-1*z^-1*x^-1 ] ) ]
```

The function `LowIndexSubgroupsFpGroup` finds the requested subgroups by systematically running through a tree of all potential coset tables of G of length at most *index* (where it skips all branches of that tree for which it knows in advance that they cannot provide new classes of such subgroups). The time required to do this depends, of course, on the presentation of G, but in general it will grow exponentially with the value of *index*. So you should be careful with the choice of *index*.

23.8 Presentation Records

In GAP3, **finitely presented groups** are distinguished from **group presentations** which are GAP3 objects of their own and which are stored in **presentation records**. The reason is that very often presentations have to be changed (e.g. simplified) by Tietze transformations, but since in these new generators and relators are introduced, all words in the generators of a finitely presented group would also have to be changed if such a Tietze transformation were applied to the presentation of a finitely presented group. Therefore, in GAP3 the presentation defining a finitely presented group is never changed; changes are only allowed for group presentations which are not considered to define a particular group.

GAP3 offers a bundle of commands to perform Tietze transformations on finite group presentations (see 23.12, 23.13). In order to speed up the respective routines, the relators in such a presentation record are not represented by ordinary (abstract) GAP3 words, but by lists of positive or negative generator numbers which we call **Tietze words**.

The term "**Tietze record**" will sometimes be used as an alias for "**presentation record**". It occurs, in particular, in certain error messages.

The following two commands can be used to create a presentation record from a finitely presented group or, vice versa, to create a finitely presented group from a presentation.

`PresentationFpGroup(` G `)`
`PresentationFpGroup(` G`,` *printlevel* `)`

`PresentationFpGroup` returns a presentation record containing a copy of the presentation of the given finitely presented group G on the same set of generators.

The optional *printlevel* parameter can be used to restrict or to extend the amount of output provided by Tietze transformation commands when being applied to the created presentation record. The default value 1 is designed for interactive use and implies explicit messages to be displayed by most of these commands. A *printlevel* value of 0 will suppress these messages, whereas a *printlevel* value of 2 will enforce some additional output.

`FpGroupPresentation(` *P* `)`

`FpGroupPresentation` returns a finitely presented group defined by the presentation in the given presentation record *P*.

If some presentation record *P*, say, contains a large presentation, then it would be nasty to wait for the end of an unintentionally started printout of all of its components (or, more precisely, of its component *P*`.tietze` which contains the essential lists). Therefore, whenever you use the standard print facilities to display a presentation record, GAP3 will provide just one line of text containing the number of generators, the number of relators, and the total length of all relators. Of course, you may use the `RecFields` and `PrintRec` commands to display all components of *P*.

In addition, you may use the following commands to extract and print different amounts of information from a presentation record.

`TzPrintStatus(` *P* `)`

`TzPrintStatus` prints the current state of a presentation record *P*, i.e., the number of generators, the number of relators, and the total length of all relators.

If you are working interactively, you can get the same information by just typing *P*`;`

`TzPrintGenerators(` *P* `)`
`TzPrintGenerators(` *P*`,` *list* `)`

`TzPrintGenerators` prints the current list of generators of a presentation record *P*, providing for each generator its name, the total number of its occurrences in the relators, and, if that generator is known to be an involution, an appropriate message.

If a list *list* has been specified as second argument, then it is expected to be a list of the position numbers of the generators to be printed. *list* need not be sorted and may contain duplicate elements. The generators are printed in the order in which and as often as their numbers occur in *list*. Position numbers out of range (with respect to the list of generators) will be ignored.

`TzPrintRelators(` *P* `)`
`TzPrintRelators(` *P*`,` *list* `)`

`TzPrintRelators` prints the current list of relators of a presentation record *P*.

If a list *list* has been specified as second argument, then it is expected to be a list of the position numbers of the relators to be printed. *list* need not be sorted and may contain duplicate elements. The relators are printed as Tietze words in the order in which (and as often as) their numbers occur in *list*. Position numbers out of range (with respect to the list of relators) will be ignored.

`TzPrintPresentation(` *P* `)`

`TzPrintPresentation` prints the current lists of generators and relators and the current state of a presentation record *P*. In fact, the command

```
TzPrintPresentation( P )
```

is an abbreviation of the command sequence

```
Print( "generators:\n" ); TzPrintGenerators( P );
Print( "relators:\n" ); TzPrintRelators( P );
TzPrintStatus( P );
```

`TzPrint( P )`
`TzPrint( P, list )`

`TzPrint` provides a kind of **fast print out** for a presentation record *P*.

Remember that in order to speed up the Tietze transformation routines, each relator in a presentation record *P* is internally represented by a list of positive or negative generator numbers, i.e., each factor of the proper GAP3 word is represented by the position number of the corresponding generator with respect to the current list of generators, or by the respective negative number, if the factor is the inverse of a generator which is not known to be an involution. In contrast to the commands `TzPrintRelators` and `TzPrintPresentation` described above, `TzPrint` does not convert these lists back to the corresponding GAP3 words.

`TzPrint` prints the current list of generators, and then for each relator its length and its internal representation as a list of positive or negative generator numbers.

If a list *list* has been specified as second argument, then it is expected to be a list of the position numbers of the relators to be printed. *list* need not be sorted and may contain duplicate elements. The relators are printed in the order in which and as often as their numbers occur in *list*. Position numbers out of range (with respect to the list of relators) will be ignored.

There are four more print commands for presentation records which are convenient in the context of the interactive Tietze transformation commands:

`TzPrintGeneratorImages( P )`

See 23.13.

`TzPrintLengths( P )`

See 23.13.

`TzPrintPairs( P )`
`TzPrintPairs( P, n )`

See 23.13.

`TzPrintOptions( P )`

See 23.13.

Moreover, there are two functions which allow to convert abstract words to Tietze words or Tietze words to abstract words.

`TietzeWordAbstractWord( word, generators )`

Let *generators* be a list of abstract generators and *word* an abstract word in these generators. The function `TietzeWordAbstractWord` returns the corresponding (reduced) Tietze word.

```
gap> F := FreeGroup( "a", "b", "c" );
Group( a, b, c )
gap> tzword := TietzeWordAbstractWord(
>  Comm(F.1,F.2) * (F.3^2 * F.2)^-1, F.generators );
[ -1, -2, 1, -3, -3 ]
```

`AbstractWordTietzeWord(` *word*, *generators* `)`

Let *generators* be a list of abstract generators and *word* a Tietze word in these generators. The function `AbstractWordTietzeWord` returns the corresponding abstract word.

```
gap> AbstractWordTietzeWord( tzword, F.generators );
a^-1*b^-1*a*c^-2
```

`Save(` *file*, *P*, *name* `)`

The function `Save` allows to save a presentation and to recover it in a later GAP3 session.

Let P be a presentation, and let *file* and *name* be strings denoting a file name and a variable name, respectively. The function `Save` generates a new file *file* and writes P and *name* to that file in such a way that a copy of P can be reestablished by just reading the file with the function `Read`. This copy of P will be assigned to a variable called *name*.

Warning: It is not guaranteed that the functions `Save` and `Read` work properly if the presentation record P contains additional, user defined components. For instance, components involving abstract words cannot be read in again as soon as the associated generators are not available any more.

Example.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1^2, F2.2^7, Comm(F2.1,F2.1^F2.2),
>                Comm(F2.1,F2.1^(F2.2^2))*(F2.1^F2.2)^-1 ];
Group( a, b )
gap> a := G.1;; b := G.2;;
gap> P := PresentationFpGroup( G );
<< presentation with 2 gens and 4 rels of total length 30 >>
gap> TzPrintGenerators( P );
#I  1.  a   11 occurrences   involution
#I  2.  b   19 occurrences
gap> TzPrintRelators( P );
#I  1. a^2
#I  2. b^7
#I  3. a*b^-1*a*b*a*b^-1*a*b
#I  4. a*b^-2*a*b^2*a*b^-2*a*b*a*b
gap> TzPrint( P );
#I  generators: [ a, b ]
#I  relators:
#I  1.  2  [ 1, 1 ]
```

```
#I  2.  7  [ 2, 2, 2, 2, 2, 2, 2 ]
#I  3.  8  [ 1, -2, 1, 2, 1, -2, 1, 2 ]
#I  4.  13  [ 1, -2, -2, 1, 2, 2, 1, -2, -2, 1, 2, 1, 2 ]
gap> TzPrintStatus( P );
#I  there are 2 generators and 4 relators of total length 30
gap> Save( "checkpoint", P, "P0" );
gap> Read( "checkpoint" );
#I  presentation record P0 read from file
gap> P0;
<< presentation with 2 gens and 4 rels of total length 30 >>
```

23.9 Changing Presentations

The commands described in this section can be used to change the presentation in a presentation record. Note that, in general, they will change the isomorphism type of the group defined by the presentation. Hence, though they sometimes are called as subroutines by Tietze transformations commands like `TzSubstitute` (see 23.13), they do **not** perform Tietze transformations themselves.

`AddGenerator(` *P* `)`
`AddGenerator(` *P*`,` *generator* `)`

`AddGenerator` adds a new generator to the list of generators.

If you don't specify a second argument, then `AddGenerator` will define a new abstract generator `_x`*i* and save it in a new component *P*`.`*i* of the given presentation record where *i* is the least positive integer which has not yet been used as a generator number. Though this new generator will be printed as `_x`*i*, you will have to use the external variable *P*`.`*i* if you want to access it.

If you specify a second argument, then *generator* must be an abstract generator which does not yet occur in the presentation. `AddGenerator` will add it to the presentation and save it in a new component *P*`.`*i* in the same way as described for `_x`*i* above.

`AddRelator(` *P*`,` *word* `)`

`AddRelator` adds the word *word* to the list of relators. *word* must be a word in the generators of the given presentation.

`RemoveRelator(` *P*`,` *n* `)`

`RemoveRelator` removes the *n*th relator and then resorts the list of relators in the given presentation record *P*.

23.10 Group Presentations

In section 23.8 we have described the funtion `PresentationFpGroup` which supplies a presentation record for a finitely presented group. The following function can be used to compute a presentation record for a concrete (e. g. permutation or matrix) group.

`PresentationViaCosetTable( G )`
`PresentationViaCosetTable( G, F, words )`

`PresentationViaCosetTable` constructs a presentation record for the given group G. The method being used is John Cannon's relations finding algorithm which has been described in [Can73] or in [Neu82].

In its first form, if only the group G has been specified, it applies Cannon's single stage algorithm which, by plain element multiplication, computes a coset table of G with respect to its trivial subgroup and then uses coset enumeration methods to find a defining set of relators for G.

```
gap> G := GeneralLinearGroup( 2, 7 );
GL(2,7)
gap> G.generators;
[ [ [ Z(7), 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ],
  [ [ Z(7)^3, Z(7)^0 ], [ Z(7)^3, 0*Z(7) ] ] ]
gap> Size( G );
2016
gap> P := PresentationViaCosetTable( G );
<< presentation with 2 gens and 5 rels of total length 46 >>
gap> TzPrintRelators( P );
#I  1. f.2^3
#I  2. f.1^6
#I  3. f.1*f.2*f.1*f.2*f.1*f.2*f.1*f.2*f.1*f.2*f.1*f.2
#I  4. f.1*f.2*f.1^-1*f.2*f.1*f.2^-1*f.1^-1*f.2*f.1*f.2*f.1^-1*f.2^-1
#I  5. f.1^2*f.2*f.1*f.2*f.1*f.2^-1*f.1^-1*f.2^-1*f.1^3*f.2^-1
```

The second form allows to call Cannon's two stage algorithm which first applies the single stage algorithm to an appropriate subgroup H of G and then uses the resulting relators of H and a coset table of G with respect to H to find relators of G. In this case the second argument, F, is assumed to be a free group with the same number of generators as G, and *words* is expected to be a list of words in the generators of F which, when being evaluated in the corresponding generators of G, provide subgroup generators for H.

```
gap> M12 := MathieuGroup( 12 );;
gap> M12.generators;
[ ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11), ( 3, 7,11, 8)( 4,10, 5, 6),
  ( 1,12)( 2,11)( 3, 6)( 4, 8)( 5, 9)( 7,10) ]
gap> F := FreeGroup( "a", "b", "c" );
Group( a, b, c )
gap> words := [ F.1, F.2 ];
[ a, b ]
gap> P := PresentationViaCosetTable( M12, F, words );
<< presentation with 3 gens and 10 rels of total length 97 >>
gap> G := FpGroupPresentation( P );
Group( a, b, c )
gap> G.relators;
[ c^2, b^4, a*c*a*c*a*c, a*b^-2*a*b^-2*a*b^-2, a^11,
  a^2*b*a^-2*b^-2*a*b^-1*a^2*b^-1,
  a*b*a^-1*b*a^-1*b^-1*a*b*a^-1*b*a^-1*b^-1,
```

```
      a^2*b*a^2*b^-2*a^-1*b*a^-1*b^-1*a^-1*b^-1,
      a^2*b^-1*a^-1*b^-1*a*c*b*c*a*b*a*b, a^3*b*a^2*b*a^-2*c*a*b*a^-1*c*a
    ]
```

Before it is returned, the resulting presentation is being simplified by appropriate calls of the function `SimplifyPresentation` (see 23.13), but without allowing it to eliminate any generators. This restriction guarantees that we get a bijection between the list of generators of G and the list of generators in the presentation. Hence, if the generators of G are redundant and if you don't care for the bijection, it may be convenient to apply the function `SimplifyPresentation` again.

```
gap> H := Group(
>  [ (2,5,3), (2,7,5), (1,8,4), (1,8,6), (4,8,6), (3,5,7) ], () );;
gap> P := PresentationViaCosetTable( H );
<< presentation with 6 gens and 12 rels of total length 42 >>
gap> SimplifyPresentation( P );
#I  there are 4 generators and 10 relators of total length 36
```

23.11 Subgroup Presentations

`PresentationSubgroupRrs(` *G*`,` *H* `)`
`PresentationSubgroupRrs(` *G*`,` *H*`,` *string* `)`
`PresentationSubgroupRrs(` *G*`,` *cosettable* `)`
`PresentationSubgroupRrs(` *G*`,` *cosettable*`,` *string* `)`

`PresentationSubgroupRrs` returns a presentation record (see 23.8) containing a presentation for the subgroup H of the finitely presented group G. It uses the Reduced Reidemeister-Schreier method to construct this presentation.

As second argument, you may provide either the subgroup H itself or its coset table in G.

The generators in the resulting presentation will be named by *string*`1`, *string*`2`, ..., the default string is `"_x"`.

The Reduced Reidemeister-Schreier algorithm is a modification of the Reidemeister-Schreier algorithm of George Havas [Hav74]. It was proposed by Joachim Neubüser and first implemented in 1986 by Andrea Lucchini and Volkmar Felsch in the SPAS system [Leh89b]. Like George Havas' Reidemeister-Schreier algorithm, it needs only the presentation of G and a coset table of H in G to construct a presentation of H.

Whenever you call the `PresentationSubgroupRrs` command, it checks first whether a coset table of H in G has already been computed and saved in the subgroup record of H by a preceding call of some appropriate command like `CosetTableFpGroup` (see 23.4), `Index` (see 7.51), or `LowIndexSubgroupsFpGroup` (see 23.7). Only if the coset table is not yet available, it is now constructed by `PresentationSubgroupRrs` which calls `CosetTableFpGroup` for this purpose. In this case, of course, a set of generators of H is required, but they will not be used any more in the subsequent steps.

Next, a set of generators of H is determined by reconstructing the coset table and introducing in that process as many Schreier generators of H in G as are needed to do a Felsch strategy coset enumeration without any coincidences. (In general, though containing redundant generators, this set will be much smaller than the set of all Schreier generators. That's why we call the method the **Reduced** Reidemeister-Schreier.)

After having constructed this set of **primary subgroup generators** , say, the coset table is extended to an **augmented coset table** which describes the action of the group generators on coset representatives, i.e., on elements instead of cosets. For this purpose, suitable words in the (primary) subgroup generators have to be associated to the coset table entries. In order to keep the lengths of these words short, additional **secondary subgroup generators** are introduced as abbreviations of subwords. Their number may be large.

Finally, a Reidemeister rewriting process is used to get defining relators for H from the relators of G. As the resulting presentation of H is a presentation on primary **and** secondary generators, in general you will have to simplify it by appropriate Tietze transformations (see 23.13) or by the `DecodeTree` command (see 23.14) before you can use it. Therefore it is returned in the form of a presentation record, P say.

Compared with the Modified Todd-Coxeter method described below, the Reduced Reidemeister-Schreier method (as well as Havas' original Reidemeister-Schreier program) has the advantage that it does not require generators of H to be given if a coset table of H in G is known. This provides a possibility to compute a presentation of the normal closure of a given subgroup (see the `PresentationNormalClosureRrs` command below).

As you may be interested not only to get the resulting presentation, but also to know what the involved subgroup generators are, the function `PresentationSubgroupRrs` will also return a list of the primary generators of H as words in the generators of G. It is provided in form of an additional component P`.primaryGeneratorWords` of the resulting presentation record P.

Note however: As stated in the description of the function `Save` (see 23.8), the function `Read` cannot properly recover a component involving abstract generators different from the current generators when it reads a presentation which has been written to a file by the function `Save`. Therefore the function `Save` will ignore the component P`.primaryGeneratorWords` if you call it to write the presentation P to a file. Hence this component will be lost if you read the presentation back from that file, and it will be left to your own responsibility to remember what the primary generators have been.

A few examples are given in section 23.13.

`PresentationSubgroupMtc(` *G*`,` *H* `)`
`PresentationSubgroupMtc(` *G*`,` *H*`,` *string* `)`
`PresentationSubgroupMtc(` *G*`,` *H*`,` *printlevel* `)`
`PresentationSubgroupMtc(` *G*`,` *H*`,` *string*`,` *printlevel* `)`

`PresentationSubgroupMtc` returns a presentation record (see 23.8) containing a presentation for the subgroup H of the finitely presented group G. It uses a Modified Todd-Coxeter method to construct this presentation.

The generators in the resulting presentation will be named by *string*`1`, *string*`2`, ..., the default string is `"_x"`.

The optional *printlevel* parameter can be used to restrict or to extend the amount of output provided by the `PresentationSubgroupMtc` command. In particular, by specifying the *printlevel* parameter to be 0, you can suppress the output of the `DecodeTree` command which is called by the `PresentationSubgroupMtc` command (see below). The default value of *printlevel* is 1.

The so called Modified Todd-Coxeter method was proposed, in slightly different forms, by Nathan S. Mendelsohn and William O. J. Moser in 1966. Moser's method was proved by Michael J. Beetham and Colin M. Campbell (see [BC76]). Another proof for a special version was given by D. H. McLain (see [McL77]). It was generalized to cover a broad spectrum of different versions (see the survey [Neu82]). Moser's method was implemented by Harvey A. Campbell (see [Cam71]. Later, a Modified Todd-Coxeter program was implemented in St. Andrews by David G. Arrell, Sanjiv Manrai, and Michael F. Worboys (see [AMW82]) and further developed by David G. Arrel and Edmund F. Robertson (see [AR84]) and by Volkmar Felsch in the SPAS system [Leh89b].

The `Modified Todd-Coxeter` method performs an enumeration of coset representatives. It proceeds like an ordinary coset enumeration (see `CosetTableFpGroup` 23.4), but as the product of a coset representative by a group generator or its inverse need not be a coset representative itself, the Modified Todd-Coxeter has to store a kind of correction element for each coset table entry. Hence it builds up a so called **augmented coset table** of H in G consisting of the ordinary coset table and a second table in parallel which contains the associated subgroup elements.

Theoretically, these subgroup elements could be expressed as words in the given generators of H, but in general these words tend to become unmanageable because of their enormous lengths. Therefore, a highly redundant list of subgroup generators is built up starting from the given ("**primary**") generators of H and adding additional ("**secondary**") generators which are defined as abbreviations of suitable words of length two in the preceding generators such that each of the subgroup elements in the augmented coset table can be expressed as a word of length at most one in the resulting (primary **and** secondary) subgroup generators.

Then a rewriting process (which is essentially a kind of Reidemeister rewriting process) is used to get relators for H from the defining relators of G.

The resulting presentation involves all the primary, but not all the secondary generators of H. In fact, it contains only those secondary generators which explicitly occur in the augmented coset table. If we extended this presentation by those secondary generators which are not yet contained in it as additional generators, and by the definitions of all secondary generators as additional relators, we would get a presentation of H, but, in general, we would end up with a large number of generators and relators.

On the other hand, if we avoid this extension, the current presentation will not necessarily define H although we have used the same rewriting process which in the case of the `SubgroupPresentationRrs` command computes a defining set of relators for H from an augmented coset table and defining relators of G. The different behaviour here is caused by the fact that coincidences may have occurred in the Modified Todd-Coxeter coset enumeration.

To overcome this problem without extending the presentation by all secondary generators, the `SubgroupPresentationMtc` command applies the so called **tree decoding** algorithm which provides a more economical approach. The reader is strongly recommended to carefully read section 23.14 where this algorithm is described in more detail. Here we will only mention that this procedure adds many fewer additional generators and relators in a process which in fact eliminates all secondary generators from the presentation and hence finally provides a presentation of H on the primary, i.e., the originally given, generators of H. This is a remarkable advantage of the `SubgroupPresentationMtc` command compared to the `SubgroupPresentationRrs` command. But note that, for some particular subgroup

H, the Reduced Reidemeister-Schreier method might quite well produce a more concise presentation.

The resulting presentation is returned in the form of a presentation record, P say.

As the function `PresentationSubgroupRrs` desribed above (see there for details), the function `PresentationSubgroupMtc` returns a list of the primary subgroup generators of H in form of a component P`.primaryGeneratorWords`. In fact, this list is not very exciting here because it is just a copy of the list H`.generators`, however it is needed to guarantee a certain consistency between the results of the different functions for computing subgroup presentations.

Though the tree decoding routine already involves a lot of Tietze transformations, we recommend that you try to further simplify the resulting presentation by appropriate Tietze transformations (see 23.13).

An example is given in section 23.14.

`PresentationSubgroup(` *G*`,` *H* `)`
`PresentationSubgroup(` *G*`,` *H*`,` *string* `)`
`PresentationSubgroup(` *G*`,` *cosettable* `)`
`PresentationSubgroup(` *G*`,` *cosettable*`,` *string* `)`

`PresentationSubgroup` returns a presentation record (see 23.8) containing a presentation for the subgroup H of the finitely presented group G.

As second argument, you may provide either the subgroup H itself or its coset table in G.

In the case of providing the subgroup H itself as argument, the current GAP3 implementation offers a choice between two different methods for constructing subgroup presentations, namely the Reduced Reidemeister-Schreier and the Modified Todd-Coxeter procedure. You can specify either of them by calling the commands `PresentationSubgroupRrs` or `PresentationSubgroupMtc`, respectively. Further methods may be added in a later GAP3 version. If, in some concrete application, you don't care for the method to be selected, you may use the `PresentationSubgroup` command as a kind of default command. In the present installation, it will call the Reduced Reidemeister-Schreier method, i.e., it is identical with the `PresentationSubgroupRrs` command.

A few examples are given in section 23.13.

`PresentationNormalClosureRrs(` *G*`,` *H* `)`
`PresentationNormalClosureRrs(` *G*`,` *H*`,` *string* `)`

`PresentationNormalClosureRrs` returns a presentation record (see 23.8), P say, containing a presentation for the normal closure of the subgroup H of the finitely presented group G. It uses the Reduced Reidemeister-Schreier method to construct this presentation. This provides a possibility to compute a presentation for a normal subgroup for which only "normal subgroup generators", but not necessarily a full set of generators are known.

The generators in the resulting presentation will be named by *string*`1`, *string*`2`, ..., the default string is `"_x"`.

`PresentationNormalClosureRrs` first establishes an intermediate group record for the factor group of G by the normal closure N, say, of H in G. Then it performs a coset enumeration of the trivial subgroup in that factor group. The resulting coset table can be considered as coset table of N in G, hence a presentation for N can be constructed using the Reduced Reidemeister-Schreier algorithm as described for the `PresentationSubgroupRrs` command.

As the function `PresentationSubgroupRrs` desribed above (see there for details), the function `PresentationNormalClosureRrs` returns a list of the primary subgroup generators of N in form of a component P`.primaryGeneratorWords`.

`PresentationNormalClosure(` G`,` H `)`
`PresentationNormalClosure(` G`,` H`,` *string* `)`

`PresentationNormalClosure` returns a presentation record (see 23.8) containing a presentation for the normal closure of the subgroup H of the finitely presented group G. This provides a possibility to compute a presentation for a normal subgroup for which only "normal subgroup generators", but not necessarily a full set of generators are known.

If, in a later release, GAP3 offers different methods for the construction of normal closure presentations, then `PresentationNormalClosure` will call one of these procedures as a kind of default method. At present, however, the Reduced Reidemeister-Schreier algorithm is the only one implemented so far. Therefore, at present the `PresentationNormalClosure` command is identical with the `PresentationNormalClosureRrs` command described above.

23.12 SimplifiedFpGroup

`SimplifiedFpGroup(` G `)`

`SimplifiedFpGroup` applies Tietze transformations to a copy of the presentation of the given finitely presented group G in order to reduce it with respect to the number of generators, the number of relators, and the relator lengths.

`SimplifiedFpGroup` returns the resulting finitely presented group (which is isomorphic to G).

```
gap> F6 := FreeGroup( 6, "G" );;
gap> G := F6 / [ F6.1^2, F6.2^2, F6.4*F6.6^-1, F6.5^2, F6.6^2,
>         F6.1*F6.2^-1*F6.3, F6.1*F6.5*F6.3^-1, F6.2*F6.4^-1*F6.3,
>         F6.3*F6.4*F6.5^-1, F6.1*F6.6*F6.3^-2, F6.3^4 ];;
gap> H := SimplifiedFpGroup( G );
Group( G.1, G.3 )
gap> H.relators;
[ G.1^2, G.1*G.3^-1*G.1*G.3^-1, G.3^4 ]
```

In fact, the command

```
H := SimplifiedFpGroup( G );
```

is an abbreviation of the command sequence

```
P := PresentationFpGroup( G, 0 );;
SimplifyPresentation( P );
H := FpGroupPresentation( P );
```

which applies a rather simple-minded strategy of Tietze transformations to the intermediate presentation record P (see 23.8). If for some concrete group the resulting presentation is unsatisfying, then you should try a more sophisticated, interactive use of the available Tietze transformation commands (see 23.13).

23.13 Tietze Transformations

The GAP3 commands being described in this section can be used to modify a group presentation in a presentation record by Tietze transformations.

In general, the aim of such modifications will be to **simplify** the given presentation, i.e., to reduce the number of generators and the number of relators without increasing too much the sum of all relator lengths which we will call the **total length** of the presentation. Depending on the concrete presentation under investigation one may end up with a nice, short presentation or with a very huge one.

Unfortunately there is no algorithm which could be applied to find the shortest presentation which can be obtained by Tietze transformations from a given one. Therefore, what GAP3 offers are some lower-level Tietze transformation commands and, in addition, some higher-level commands which apply the lower-level ones in a kind of default strategy which of course cannot be the optimal choice for all presentations.

The design of these commands follows closely the concept of the ANU Tietze transformation program designed by George Havas [Hav69] which has been available from Canberra since 1977 in a stand-alone version implemented by Peter Kenne and James Richardson and later on revised by Edmund F. Robertson (see [HKRR84], [Rob88]).

In this section, we first describe the higher-level commands `SimplifyPresentation`, `TzGo`, and `TzGoGo` (the first two of these commands are identical).

Then we describe the lower-level commands `TzEliminate`, `TzSearch`, `TzSearchEqual`, and `TzFindCyclicJoins`. They are the bricks of which the preceding higher-level commands have been composed. You may use them to try alternative strategies, but if you are satisfied by the performance of `TzGo` and `TzGoGo`, then you don't need them.

Some of the Tietze transformation commands listed so far may eliminate generators and hence change the given presentation to a presentation on a subset of the given set of generators, but they all do **not** introduce new generators. However, sometimes you will need to substitute certain words as new generators in order to improve your presentation. Therefore GAP3 offers the two commands `TzSubstitute` and `TzSubstituteCyclicJoins` which introduce new generators. These commands will be described next.

Then we continue the section with a description of the commands `TzInitGeneratorImages` and `TzPrintGeneratorImages` which can be used to determine and to display the images or preimages of the involved generators under the isomorphism which is defined by the sequence of Tietze transformations which are applied to a presentation.

Subsequently we describe some further print commands, `TzPrintLengths`, `TzPrintPairs`, and `TzPrintOptions`, which are useful if you run the Tietze transformations interactively.

At the end of the section we list the **Tietze options** and give their default values. These are parameters which essentially influence the performance of the commands mentioned above. However, they are not specified as arguments of function calls. Instead, they are associated to the presentation records: Each presentation record keeps its own set of Tietze option values in the form of ordinary record components.

`SimplifyPresentation(` *P* `)`
`TzGo(` *P* `)`

`SimplifyPresentation` performs Tietze transformations on a presentation P. It is perhaps the most convenient of the interactive Tietze transformation commands. It offers a kind of default strategy which, in general, saves you from explicitly calling the lower-level commands it involves.

Roughly speaking, `SimplifyPresentation` consists of a loop over a procedure which involves two phases: In the **search phase** it calls `TzSearch` and `TzSearchEqual` described below which try to reduce the relator lengths by substituting common subwords of relators, in the **elimination phase** it calls the command `TzEliminate` described below (or, more precisely, a subroutine of `TzEliminate` in order to save some administrative overhead) which tries to eliminate generators that can be expressed as words in the remaining generators.

If `SimplifyPresentation` succeeds in reducing the number of generators, the number of relators, or the total length of all relators, then it displays the new status before returning (provided that you did not set the print level to zero). However, it does not provide any output if all these three values have remained unchanged, even if the `TzSearchEqual` command involved has changed the presentation such that another call of `SimplifyPresentation` might provide further progress. Hence, in such a case it makes sense to repeat the call of the command for several times (or to call instead the `TzGoGo` command which we will describe next).

As an example we compute a presentation of a subgroup of index 408 in $PSL(2, 17)$.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1^9, F2.2^2, (F2.1*F2.2)^4, (F2.1^2*F2.2)^3 ];;
gap> a := G.1;;  b := G.2;;
gap> H := Subgroup( G, [ (a*b)^2, (a^-1*b)^2 ] );;
gap> Index( G, H );
408
gap> P := PresentationSubgroup( G, H );
<< presentation with 8 gens and 36 rels of total length 111 >>
gap> P.primaryGeneratorWords;
[ b, a*b*a ]
gap> P.protected := 2;;
gap> P.printLevel := 2;;
gap> SimplifyPresentation( P );
#I  eliminating _x7 = _x5
#I  eliminating _x5 = _x4
#I  eliminating _x18 = _x3
#I  eliminating _x8 = _x3
#I  there are 4 generators and 8 relators of total length 21
#I  there are 4 generators and 7 relators of total length 18
#I  eliminating _x4 = _x3^-1*_x2^-1
#I  eliminating _x3 = _x2*_x1^-1
#I  there are 2 generators and 4 relators of total length 14
#I  there are 2 generators and 4 relators of total length 13
#I  there are 2 generators and 3 relators of total length 9
gap> TzPrintRelators( P );
#I  1. _x1^2
#I  2. _x2^3
#I  3. _x2*_x1*_x2*_x1
```

Note that the number of loops over the two phases as well as the number of subword searches or generator eliminations in each phase are determined by a set of option parameters which may heavily influence the resulting presentation and the computing time (see Tietze options below).

`TzGo` is just another name for the `SimplifyPresentation` command. It has been introduced for the convenience of those GAP3 users who are used to that name from the **go** option of the ANU Tietze transformation stand-alone program or from the **go** command in SPAS.

`TzGoGo(` *P* `)`

`TzGoGo` performs Tietze transformations on a presentation P. It repeatedly calls the `TzGo` command until neither the number of generators nor the number of relators nor the total length of all relators have changed during five consecutive calls of `TzGo`.

This may remarkably save you time and effort if you handle small presentations, however it may lead to annoyingly long and fruitless waiting times in case of large presentations.

`TzEliminate(` *P* `)`
`TzEliminate(` *P*`,` *gen* `)`
`TzEliminate(` *P*`,` *n* `)`

`TzEliminate` tries to eliminate a generator from a presentation P via Tietze transformations.

Any relator which contains some generator just once can be used to substitute that generator by a word in the remaining generators. If such generators and relators exist, then `TzEliminate` chooses a generator for which the product of its number of occurrences and the length of the substituting word is minimal, and then it eliminates this generator from the presentation, provided that the resulting total length of the relators does not exceed the associated Tietze option parameter *P*`.spaceLimit`. The default value of *P*`.spaceLimit` is `infinity`, but you may alter it appropriately (see Tietze options below).

If you specify a generator *gen* as second argument, then `TzEliminate` only tries to eliminate that generator.

If you specify an integer n as second argument, then `TzEliminate` tries to eliminate up to n generators. Note that the calls `TzEliminate(` *P* `)` and `TzEliminate(` *P*`, 1 )` are equivalent.

`TzSearch(` *P* `)`

`TzSearch` performs Tietze transformations on a presentation P. It tries to reduce the relator lengths by substituting common subwords of relators by shorter words.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that $l_1 \leq l_2$ and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length greater than $l_1/2$, and then to substitute each copy of w in r_2 by the inverse complement of w in r_1.

Two of the Tietze option parameters which are listed at the end of this section may strongly influence the performance and the results of the `TzSearch` command. These are the parameters *P*`.saveLimit` and *P*`.searchSimultaneous`. The first of them has the following effect.

When TzSearch has finished its main loop over all relators, then, in general, there are relators which have changed and hence should be handled again in another run through the whole procedure. However, experience shows that it really does not pay to continue this way until no more relators change. Therefore, `TzSearch` starts a new loop only if the loop just finished has reduced the total length of the relators by at least *P*`.saveLimit` per cent.

The default value of *P*`.saveLimit` is 10.

To understand the effect of the parameter *P*`.searchSimultaneous`, we have to look in more detail at how `TzSearch` proceeds.

First, it sorts the list of relators by increasing lengths. Then it performs a loop over this list. In each step of this loop, the current relator is treated as **short relator** r_1, and a subroutine is called which loops over the succeeding relators, treating them as **long relators** r_2 and performing the respective comparisons and substitutions.

As this subroutine performs a very expensive process, it has been implemented as a C routine in the GAP3 kernel. For the given relator r_1 of length l_1, say, it first determines the **minimal match length** l which is $l_1/2+1$, if l_1 is even, or $(l_1+1)/2$, otherwise. Then it builds up a hash list for all subwords of length l occurring in the conjugates of r_1 or r_1^{-1}, and finally it loops over all long relators r_2 and compares the hash values of their subwords of length l against this list. A comparison of subwords which is much more expensive is only done if a hash match has been found.

To improve the efficiency of this process we allow the subroutine to handle several short relators simultaneously provided that they have the same minimal match length. If, for example, it handles n short relators simultaneously, then you save $n-1$ loops over the long relators r_2, but you pay for it by additional fruitless subword comparisons. In general, you will not get the best performance by always choosing the maximal possible number of short relators to be handled simultaneously. In fact, the optimal choice of the number will depend on the concrete presentation under investigation. You can use the parameter *P*`.searchSimultaneous` to prescribe an upper bound for the number of short relators to be handled simultaneously.

The default value of *P*`.searchSimultaneous` is 20.

`TzSearchEqual(` *P* `)`

`TzSearchEqual` performs Tietze transformations on a presentation *P*. It tries to alter relators by substituting common subwords of relators by subwords of equal length.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that l_1 is even, $l_1 \leq l_2$, and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length at least $l_1/2$. Let l be the length of w. Then, if $l > l_1/2$, the pair is handled as in `TzSearch`. Otherwise, if $l = l_1/2$, then `TzSearchEqual` substitutes each copy of w in r_2 by the inverse complement of w in r_1.

The Tietze option parameter *P*`.searchSimultaneous` is used by `TzSearchEqual` in the same way as described for `TzSearch`.

However, `TzSearchEqual` does not use the parameter *P*`.saveLimit`: The loop over the relators is executed exactly once.

`TzFindCyclicJoins(` *P* `)`

`TzFindCyclicJoins` performs Tietze transformations on a presentation P. It searches for pairs of generators which generate the same cyclic subgroup and eliminates one of the two generators of each such pair it finds.

More precisely: `TzFindCyclicJoins` searches for pairs of generators a and b such that (possibly after inverting or conjugating some relators) the set of relators contains the commutator $[a, b]$, a power a^n, and a product of the form $a^s b^t$ with s prime to n. For each such pair, `TzFindCyclicJoins` uses the Euclidian algorithm to express a as a power of b, and then it eliminates a.

`TzSubstitute(` P`,` *word* `)`
`TzSubstitute(` P`,` *word*`,` *string* `)`

There are two forms of the command `TzSubstitute`. This is the first one. It expects P to be a presentation and *word* to be either an abstract word or a Tietze word in the generators of P. It substitutes the given word as a new generator of P. This is done as follows.

First, `TzSubstitute` creates a new abstract generator, g say, and adds it to the presentation P, then it adds a new relator $g^{-1} \cdot word$ to P. If a string *string* has been specified as third argument, the new generator g will be named by *string*, otherwise it will get a default name `_x`i as described with the function `AddGenerator` (see 23.9).

More precisely: If, for instance, `word` is an abstract word, a call

```
TzSubstitute( P, word );
```

is more or less equivalent to

```
AddGenerator( P );
g := P.generators[Length( P.generators )];
AddRelator( P, g^-1 * word );
```

whereas a call

```
TzSubstitute( P, word, string );
```

is more or less equivalent to

```
g := AbstractGenerator( string );
AddGenerator( P, g );
AddRelator( P, g^-1 * word );
```

The essential difference is, that `TzSubstitute`, as a Tietze transformation of P, saves and updates the lists of generator images and preimages if they are being traced under the Tietze transformations applied to P (see the function `TzInitGeneratorImages` below), whereas a call of the function `AddGenerator` (which does not perform Tietze transformations) will delete these lists and hence terminate the tracing.

Example.

```
gap> G := PerfectGroup( 960, 1 );
PerfectGroup(960,1)
gap> P := PresentationFpGroup( G );
<< presentation with 6 gens and 21 rels of total length 84 >>
gap> P.generators;
[ a, b, s, t, u, v ]
```

```
gap> TzGoGo( P );
#I  there are 3 generators and 10 relators of total length 81
#I  there are 3 generators and 10 relators of total length 80
gap> TzPrintGenerators( P );
#I  1.  a   31 occurrences   involution
#I  2.  b   26 occurrences
#I  3.  t   23 occurrences   involution
gap> a := P.generators[1];;
gap> b := P.generators[2];;
gap> TzSubstitute( P, a*b, "ab" );
#I  substituting new generator ab defined by a*b
#I  there are 4 generators and 11 relators of total length 83
gap> TzGo(P);
#I  there are 3 generators and 10 relators of total length 74
gap> TzPrintGenerators( P );
#I  1.  a   23 occurrences   involution
#I  2.  t   23 occurrences   involution
#I  3.  ab   28 occurrences
```

`TzSubstitute(` *P* `)`
`TzSubstitute(` *P*`,` *n* `)`
`TzSubstitute(` *P*`,` *n*`,` *eliminate* `)`

This is the second form of the command `TzSubstitute`. It performs Tietze transformations on the presentation P. Basically, it substitutes a squarefree word of length 2 as a new generator and then eliminates a generator from the extended generator list. We will describe this process in more detail.

The parameters n and *eliminate* are optional. If you specify arguments for them, then n is expected to be a positive integer, and *eliminate* is expected to be 0, 1, or 2. The default values are $n = 1$ and *eliminate* $= 0$.

`TzSubstitute` first determines the n most frequently occurring squarefree relator subwords of length 2 and sorts them by decreasing numbers of occurrences. Let ab be the nth word in that list, and let i be the smallest positive integer which has not yet been used as a generator number. Then `TzSubstitute` defines a new generator P`.`i (see `AddGenerator` for details), adds it to the presentation together with a new relator $P.i^{-1}ab$, and replaces all occurrences of ab in the given relators by P`.`i.

Finally, it eliminates some generator from the extended presentation. The choice of that generator depends on the actual value of the *eliminate* parameter:

If *eliminate* is zero, then the generator to be eliminated is chosen as by the `TzEliminate` command. This means that in this case it may well happen that it is the generator P`.`i just introduced which is now deleted again so that you do not get any remarkable progress in transforming your presentation. On the other hand, this procedure guaranties that the total length of the relators will not be increased by a call of `TzSubstitute` with *eliminate* $= 0$.

Otherwise, if *eliminate* is 1 or 2, then `TzSubstitute` eliminates the respective factor of the substituted word ab, i.e., a for *eliminate* $= 1$ or b for *eliminate* $= 2$. In this case, it may well happen that the total length of the relators increases, but sometimes such an intermediate extension is the only way to finally reduce a given presentation.

In order to decide which arguments might be appropriate for the next call of `TzSubstitute`, often it is helpful to print out a list of the most frequently occurring squarefree relator subwords of length 2. You may use the `TzPrintPairs` command described below to do this.

As an example we handle a subgroup of index 266 in the Janko group J_1.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> J1 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^7,
>    Comm(F2.1,F2.2)^10, Comm(F2.1,F2.2^-1*(F2.1*F2.2)^2)^6 ];;
gap> a := J1.1;;  b := J1.2;;
gap> H := Subgroup ( J1, [ a, b^(a*b*(a*b^-1)^2) ] );;
gap> P := PresentationSubgroup( J1, H );
<< presentation with 23 gens and 82 rels of total length 530 >>
gap> TzGoGo( P );
#I  there are 3 generators and 47 relators of total length 1368
#I  there are 2 generators and 46 relators of total length 3773
#I  there are 2 generators and 46 relators of total length 2570
gap> TzGoGo( P );
#I  there are 2 generators and 46 relators of total length 2568
gap> TzGoGo( P );
gap> # We do not get any more progress without substituting a new
gap> # generator
gap> TzSubstitute( P );
#I  substituting new generator _x28 defined by _x6*_x23^-1
#I  eliminating _x28 = _x6*_x23^-1
gap> # GAP cannot substitute a new generator without extending the
gap> # total length, so we have to explicitly ask for it
gap> TzPrintPairs( P );
#I  1.  504  occurrences of  _x6 * _x23^-1
#I  2.  504  occurrences of  _x6^-1 * _x23
#I  3.  448  occurrences of  _x6 * _x23
#I  4.  448  occurrences of  _x6^-1 * _x23^-1
gap> TzSubstitute( P, 2, 1 );
#I  substituting new generator _x29 defined by _x6^-1*_x23
#I  eliminating _x6 = _x23*_x29^-1
#I  there are 2 generators and 46 relators of total length 2867
gap> TzGoGo( P );
#I  there are 2 generators and 45 relators of total length 2417
#I  there are 2 generators and 45 relators of total length 2122
gap> TzSubstitute( P, 1, 2 );
#I  substituting new generator _x30 defined by _x23*_x29^-1
#I  eliminating _x29 = _x30^-1*_x23
#I  there are 2 generators and 45 relators of total length 2192
gap> TzGoGo( P );
#I  there are 2 generators and 42 relators of total length 1637
#I  there are 2 generators and 40 relators of total length 1286
#I  there are 2 generators and 36 relators of total length 807
#I  there are 2 generators and 32 relators of total length 625
#I  there are 2 generators and 22 relators of total length 369
```

```
#I  there are 2 generators and 18 relators of total length 213
#I  there are 2 generators and 13 relators of total length 141
#I  there are 2 generators and 12 relators of total length 121
#I  there are 2 generators and 10 relators of total length 101
gap> TzPrintPairs( P );
#I  1.  19  occurrences of  _x23 * _x30^-1
#I  2.  19  occurrences of  _x23^-1 * _x30
#I  3.  14  occurrences of  _x23 * _x30
#I  4.  14  occurrences of  _x23^-1 * _x30^-1
gap> # If we save a copy of the current presentation, then later we
gap> # will be able to restart the computation from the current state
gap> P1 := Copy( P );;
gap> # Just for demonstration, let's make an inconvenient choice
gap> TzSubstitute( P, 3, 1 );
#I  substituting new generator _x31 defined by _x23*_x30
#I  eliminating _x23 = _x31*_x30^-1
#I  there are 2 generators and 10 relators of total length 122
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 105
gap> # The presentation is worse than the one we have saved, so let's
gap> # restart from that one again
gap> P := Copy( P1 );
<< presentation with 2 gens and 10 rels of total length 101 >>
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x31 defined by _x23^-1*_x30
#I  eliminating _x23 = _x30*_x31^-1
#I  there are 2 generators and 10 relators of total length 107
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 84
#I  there are 2 generators and 8 relators of total length 75
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x32 defined by _x30^-1*_x31
#I  eliminating _x30 = _x31*_x32^-1
#I  there are 2 generators and 8 relators of total length 71
gap> TzGoGo( P );
#I  there are 2 generators and 7 relators of total length 56
#I  there are 2 generators and 5 relators of total length 36
gap> TzPrintRelators( P );
#I  1. _x32^5
#I  2. _x31^5
#I  3. _x31^-1*_x32^-1*_x31^-1*_x32^-1*_x31^-1*_x32^-1
#I  4. _x31*_x32*_x31^-1*_x32*_x31^-1*_x32*_x31*_x32^-2
#I  5. _x31^-1*_x32^2*_x31*_x32^-1*_x31^2*_x32^-1*_x31*_x32^2
```

As shown in the preceding example, you can use the `Copy` command to save a copy of a presentation record and to restart from it again if you want to try an alternative strategy. However, this copy will be lost as soon as you finish your current GAP3 session. If you use the `Save` command (see 23.8) instead, then you get a permanent copy on a file which you can read in again in a later session.

`TzSubstituteCyclicJoins(` P `)`

`TzSubstituteCyclicJoins` performs Tietze transformations on a presentation P. It tries to find pairs of generators a and b, say, for which among the relators (possibly after inverting or conjugating some of them) there are the commutator $[a, b]$ and powers a^m and b^n with mutually prime exponents m and n. For each such pair, it substitutes the product ab as a new generator, and then it eliminates the generators a and b.

`TzInitGeneratorImages(` P `)`

Any sequence of Tietze transformations applied to a presentation record P, starting from an "old" presentation P_1 and ending up with a "new" presentation P_2, defines an isomorphism, φ say, between the groups defined by P_1 and P_2, respectively. Sometimes it is desirable to know the images of the old generators or the preimages of the new generators under φ. The GAP3 Tietze transformations functions are able to trace these images. This is not automatically done because the involved words may grow to tremendous length, but it will be done if you explicitly request for it by calling the function `TzInitGeneratorImages`.

`TzInitGeneratorImages` initializes three components of P:

P`.oldGenerators`
: This is the list of the old generators. It is initialized by a copy of the current list of generators, P`.generators`.

P`.imagesOldGens`
: This will be the list of the images of the old generators as Tietze words in the new generators. For each generator g_i, the i-th entry of the list is initialized by the Tietze word `[i]`.

P`.preImagesNewGens`
: This will be the list of the preimages of the new generators as Tietze words in the old generators. For each generator g_i, the i-th entry of the list is initialized by the Tietze word `[i]`.

This means, that P_1 is defined to be the current presentation and φ to be the identity on P_1. From now on, the existence of the component P`.imagesOldGens` will cause the Tietze transformations functions to update the lists of images and preimages whenever they are called.

You can reinitialize the tracing of the generator images at any later state by just calling the function `TzInitGeneratorImages` again. For, if the above components do already exist when `TzInitGeneratorImages` is being called, they will first be deleted and then initialized again.

There are a few restrictions concerning the tracing of generator images:

In general, the functions `AddGenerator`, `AddRelator`, and `RemoveRelator` described in section 23.9 do not perform Tietze transformations as they may change the isomorphism type of the presentation. Therefore, if any of them is called for a presentation in which generator images and preimages are being traced, it will delete these lists.

If the function `DecodeTree` is called for a presentation in which generator images and preimages are being traced, it will not continue to trace them. Instead, it will delete the corresponding lists, then decode the tree, and finally reinitialize the tracing for the resulting presentation.

As stated in the description of the function `Save` (see 23.8), the function `Read` cannot properly recover a component involving abstract generators different from the current generators when it reads a presentation which has been written to a file by the function `Save`. Therefore the function `Save` will ignore the component *P*`.oldGenerators` if you call it to write the presentation *P* to a file. Hence this component will be lost if you read the presentation back from that file, and it will be left to your own responsibility to remember what the old generators have been.

`TzPrintGeneratorImages(` *P* `)`

If *P* is a presentation in which generator images and preimages are being traced through all Tietze transformations applied to *P*, `TzPrintGeneratorImages` prints the preimages of the current generators as Tietze words in the old generators and the images of the old generators as Tietze words in the current generators.

```
gap> G := PerfectGroup( 960, 1 );
PerfectGroup(960,1)
gap> P := PresentationFpGroup( G );
<< presentation with 6 gens and 21 rels of total length 84 >>
gap> TzInitGeneratorImages( P );
gap> TzGo( P );
#I  there are 3 generators and 11 relators of total length 96
#I  there are 3 generators and 10 relators of total length 81
gap> TzPrintGeneratorImages( P );
#I  preimages of current generators as Tietze words in the old ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 4 ]
#I  images of old generators as Tietze words in the current ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 1, -2, 1, 3, 1, 2, 1 ]
#I  4. [ 3 ]
#I  5. [ -2, 1, 3, 1, 2 ]
#I  6. [ 1, 3, 1 ]
gap> # Print the old generators as words in the new generators.
gap> gens := P.generators;
[ a, b, t ]
gap> oldgens := P.oldGenerators;
[ a, b, s, t, u, v ]
gap> for i in [ 1 .. Length( oldgens ) ] do
>  Print( oldgens[i], " = ",
>  AbstractWordTietzeWord( P.imagesOldGens[i], gens ), "\n" );
>  od;
```

```
a = a
b = b
s = a*b^-1*a*t*a*b*a
t = t
u = b^-1*a*t*a*b
v = a*t*a
```

`TzPrintLengths(` P `)`

`TzPrintLengths` prints the list of the lengths of all relators of the given presentation P.

`TzPrintPairs(` P `)`
`TzPrintPairs(` P`,` n `)`

`TzPrintPairs` determines in the given presentation P the n most frequently occurring squarefree relator subwords of length 2 and prints them together with their numbers of occurrences. The default value of n is 10. A value $n = 0$ is interpreted as `infinity`.

This list is a useful piece of information in the context of using the `TzSubstitute` command described above.

`TzPrintOptions(` P `)`

Several of the Tietze transformation commands described above are controlled by certain parameters, the **Tietze options**, which often have a tremendous influence on their performance and results. However, in each application of the commands, an appropriate choice of these option parameters will depend on the concrete presentation under investigation. Therefore we have implemented the Tietze options in such a way that they are associated to the presentation records: Each presentation record keeps its own set of Tietze option parameters in the form of ordinary record components. In particular, you may alter the value of any of these Tietze options by just assigning a new value to the respective record component.

`TzPrintOptions` prints the Tietze option components of the specified presentation P.

The Tietze options have the following meaning.

`protected`
: The first P`.protected` generators in a presentation P are protected from being eliminated by the Tietze transformations functions. There are only two exceptions: The option P`.protected` is ignored by the functions `TzEliminate(`P`,`*gen*`)` and `TzSubstitute(`P`,`n`,`*eliminate*`)` because they explicitly specify the generator to be eliminated. The default value of `protected` is 0.

`eliminationsLimit`
: Whenever the elimination phase of the `TzGo` command is entered for a presentation P, then it will eliminate at most P`.eliminationsLimit` generators (except for further ones which have turned out to be trivial). Hence you may use the `eliminationsLimit` parameter as a break criterion for the `TzGo` command. Note, however, that it is ignored by the `TzEliminate` command. The default value of `eliminationsLimit` is 100.

`expandLimit`
Whenever the routine for eliminating more than 1 generators is called for a presentation P by the `TzEliminate` command or the elimination phase of the `TzGo` command, then it saves the given total length of the relators, and subsequently it checks the current total length against its value before each elimination. If the total length has increased to more than P`.expandLimit` per cent of its original value, then the routine returns instead of eliminating another generator. Hence you may use the `expandLimit` parameter as a break criterion for the `TzGo` command. The default value of `expandLimit` is 150.

`generatorsLimit`
Whenever the elimination phase of the `TzGo` command is entered for a presentation P with n generators, then it will eliminate at most $n-P$`.generatorsLimit` generators (except for generators which turn out to be trivial). Hence you may use the `generatorsLimit` parameter as a break criterion for the `TzGo` command. The default value of `generatorsLimit` is 0.

`lengthLimit`
The Tietze transformation commands will never eliminate a generator of a presentation P, if they cannot exclude the possibility that the resulting total length of the relators exceeds the value of P`.lengthLimit`. The default value of `lengthLimit` is `infinity`.

`loopLimit`
Whenever the `TzGo` command is called for a presentation P, then it will loop over at most P`.loopLimit` of its basic steps. Hence you may use the `loopLimit` parameter as a break criterion for the `TzGo` command. The default value of `loopLimit` is `infinity`.

`printLevel`
Whenever Tietze transformation commands are called for a presentation P with P`.printLevel` $= 0$, they will not provide any output except for error messages. If P`.printLevel` $= 1$, they will display some reasonable amount of output which allows you to watch the progress of the computation and to decide about your next commands. In the case P`.printLevel` $= 2$, you will get a much more generous amount of output. Finally, if P`.printLevel` $= 3$, various messages on internal details will be added. The default value of `printLevel` is 1.

`saveLimit`
Whenever the `TzSearch` command has finished its main loop over all relators of a presentation P, then it checks whether during this loop the total length of the relators has been reduced by at least P`.saveLimit` per cent. If this is the case, then `TzSearch` repeats its procedure instead of returning. Hence you may use the `saveLimit` parameter as a break criterion for the `TzSearch` command and, in particular, for the search phase of the `TzGo` command. The default value of `saveLimit` is 10.

`searchSimultaneous`
Whenever the `TzSearch` or the `TzSearchEqual` command is called for a presentation P, then it is allowed to handle up to P`.searchSimultaneously` short relators simultaneously (see for the description of the `TzSearch` command for more details). The choice of this parameter may heavily influence the performance as well as the result of the `TzSearch` and the `TzSearchEqual` commands and hence also of the search phase of the `TzGo` command. The default value of `searchSimultaneous` is 20.

As soon as a presentation record has been defined, you may alter any of its Tietze option parameters at any time by just assigning a new value to the respective component.

To demonstrate the effect of the `eliminationsLimit` parameter, we will give an example in which we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann. First we construct a presentation of the subgroup, and then we apply to it the `TzGoGo` command for different values of the `eliminationsLimit` parameter (including the default value 100). In fact, we also alter the `printLevel` parameter, but this is only done in order to suppress most of the output. In all cases the resulting presentations cannot be improved any more by applying the `TzGoGo` command again, i.e., they are the best results which we can get without substituting new generators.

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> G := F3 / [ F3.1^3, F3.2^3, F3.3^3, (F3.1*F3.2)^5,
>       (F3.1^-1*F3.2)^5, (F3.1*F3.3)^4, (F3.1*F3.3^-1)^4,
>       F3.1*F3.2^-1*F3.1*F3.2*F3.3^-1*F3.1*F3.3*F3.1*F3.3^-1,
>       (F3.2*F3.3)^3, (F3.2^-1*F3.3)^4 ];;
gap> a := G.1;;  b := G.2;;  c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> P := PresentationSubgroup( G, H );
<< presentation with 224 gens and 593 rels of total length 2769 >>
gap> for i in [ 28, 29, 30, 94, 100 ] do
>       Pi := Copy( P );
>       Pi.eliminationsLimit := i;
>       Print( "#I  eliminationsLimit set to ", i, "\n" );
>       Pi.printLevel := 0;
>       TzGoGo( Pi );
>       TzPrintStatus( Pi );
>    od;
#I  eliminationsLimit set to 28
#I  there are 2 generators and 95 relators of total length 10817
#I  eliminationsLimit set to 29
#I  there are 2 generators and 5 relators of total length 35
#I  eliminationsLimit set to 30
#I  there are 3 generators and 98 relators of total length 2928
#I  eliminationsLimit set to 94
#I  there are 4 generators and 78 relators of total length 1667
#I  eliminationsLimit set to 100
#I  there are 3 generators and 90 relators of total length 3289
```

Similarly, we demonstrate the influence of the `saveLimit` parameter by just continuing the preceding example for some different values of the `saveLimit` parameter (including its default value 10), but without changing the `eliminationsLimit` parameter which keeps its default value 100.

```
gap> for i in [ 9, 10, 11, 12, 15 ] do
>       Pi := Copy( P );
>       Pi.saveLimit := i;
>       Print( "#I  saveLimit set to ", i, "\n" );
>       Pi.printLevel := 0;
>       TzGoGo( Pi );
```

```
>       TzPrintStatus( Pi );
>    od;
#I  saveLimit set to 9
#I  there are 3 generators and 97 relators of total length 5545
#I  saveLimit set to 10
#I  there are 3 generators and 90 relators of total length 3289
#I  saveLimit set to 11
#I  there are 3 generators and 103 relators of total length 3936
#I  saveLimit set to 12
#I  there are 2 generators and 4 relators of total length 21
#I  saveLimit set to 15
#I  there are 3 generators and 143 relators of total length 18326
```

23.14 DecodeTree

`DecodeTree(` P `)`

`DecodeTree` eliminates the secondary generators from a presentation P constructed by the Modified Todd-Coxeter (see `PresentationSubgroupMtc`) or the Reduced Reidemeister-Schreier procedure (see `PresentationSubgroupRrs`, `PresentationNormalClosureRrs`). It is called automatically by the `PresentationSubgroupMtc` command where it reduces P to a presentation on the given subgroup generators.

In order to explain the effect of this command we need to insert a few remarks on the subgroup presentation commands described in section 23.11. All these commands have the common property that in the process of constructing a presentation for a given subgroup H of a finitely presented group G they first build up a highly redundant list of generators of H which consists of an (in general small) list of "primary" generators, followed by an (in general large) list of "secondary" generators, and then construct a presentation P_0, say, **on a sublist of these generators** by rewriting the defining relators of G. This sublist contains all primary, but, at least in general, by far not all secondary generators.

The role of the primary generators depends on the concrete choice of the subgroup presentation command. If the Modified Todd-Coxeter method is used, they are just the given generators of H, whereas in the case of the Reduced Reidemeister-Schreier algorithm they are constructed by the program.

Each of the secondary generators is defined by a word of length two in the preceding generators and their inverses. By historical reasons, the list of these definitions is called the **subgroup generators tree** though in fact it is not a tree but rather a kind of bush.

Now we have to distinguish two cases. If P_0 has been constructed by the Reduced Reidemeister-Schreier routines, it is a presentation of H. However, if the Modified Todd-Coxeter routines have been used instead, then the relators in P_0 are valid relators of H, but they do not necessarily define H. We handle these cases in turn, starting with the latter one.

Also in the case of the Modified Todd-Coxeter method, we could easily extend P_0 to a presentation of H by adding to it all the secondary generators which are not yet contained in it and all the definitions from the generators tree as additional generators and relators. Then we could recursively eliminate all secondary generators by Tietze transformations using the new relators. However, this procedure turns out to be too inefficient to be of interest.

Instead, we use the so called **tree decoding** procedure which has been developed in St. Andrews by David G. Arrell, Sanjiv Manrai, Edmund F. Robertson, and Michael F. Worboys (see [AMW82], [AR84]). It proceeds as follows.

Starting from $P = P_0$, it runs through a number of steps in each of which it eliminates the current "last" generator (with respect to the list of all primary and secondary generators). If the last generator g, say, is a primary generator, then the procedure finishes. Otherwise it checks whether there is a relator in the current presentation which can be used to substitute g by a Tietze transformation. If so, this is done. Otherwise, and only then, the tree definition of g is added to P as a new relator, and the generators involved are added as new generators if they have not yet been contained in P. Subsequently, g is eliminated.

Note that the extension of P by one or two new generators is **not** a Tietze transformation. In general, it will change the isomorphism type of the group defined by P. However, it is a remarkable property of this procedure, that at the end, i.e., as soon as all secondary generators have been eliminated, it provides a presentation $P = P_1$, say, which defines a group isomorphic to H. In fact, it is this presentation which is returned by the `DecodeTree` command and hence by the `PresentationSubgroupMtc` command.

If, in the other case, the presentation P_0 has been constructed by the Reduced Reidemeister-Schreier algorithm, then P_0 itself is a presentation of H, and the corresponding subgroup presentation command (`PresentationSubgroupRrs` or `PresentationNormalClosureRrs`) just returns P_0.

As mentioned in section 23.11, we recommend further simplifying this presentation before using it. The standard way to do this is to start from P_0 and to apply suitable Tietze transformations, e.g., by calling the `TzGo` or `TzGoGo` commands. This is probably the most efficient approach, but you will end up with a presentation on some unpredictable set of generators. As an alternative, GAP3 offers you the `DecodeTree` command which you can use to eliminate all secondary generators (provided that there are no space or time problems). For this purpose, the subgroup presentation commands do not only return the resulting presentation, but also the tree (together with some associated lists) as a kind of side result in a component P`.tree` of the resulting presentation record P.

Note, however, that the tree decoding routines will not work correctly any more on a presentation from which generators have already been eliminated by Tietze transformations. Therefore, to prevent you from getting wrong results by calling the `DecodeTree` command in such a situation, GAP3 will automatically remove the subgroup generators tree from a presentation record as soon as one of the generators is substituted by a Tietze transformation.

Nevertheless, a certain misuse of the command is still possible, and we want to explicitly warn you from this. The reason is that the Tietze option parameters described in section 23.13 apply to the `DecodeTree` command as well. Hence, in case of inadequate values of these parameters, it may happen that the `DecodeTree` routine stops before all the secondary generators have vanished. In this case GAP3 will display an appropriate warning. Then you should change the respective parameters and continue the process by calling the `DecodeTree` command again. Otherwise, if you would apply Tietze transformations, it might happen because of the convention described above that the tree is removed and that you end up with a wrong presentation.

After a successful run of the `DecodeTree` command it is convenient to further simplify the resulting presentation by suitable Tietze transformations.

As an example of an explicit call of the DecodeTree command we compute two presentations of a subgroup of order 384 in a group of order 6912. In both cases we use the Reduced Reidemeister-Schreier algorithm, but in the first run we just apply the Tietze transformations offered by the TzGoGo command with its default parameters, whereas in the second run we call the DecodeTree command before.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1*F2.2^2*F2.1^-1*F2.2^-1*F2.1^3*F2.2^-1,
>                F2.2*F2.1^2*F2.2^-1*F2.1^-1*F2.2^3*F2.1^-1 ];;
gap> a := G.1;;  b := G.2;;
gap> H := Subgroup( G, [ Comm(a^-1,b^-1), Comm(a^-1,b), Comm(a,b) ] );;
gap> #
gap> # We use the Reduced Reidemeister Schreier method and default
gap> # Tietze transformations to get a presentation for H.
gap> P := PresentationSubgroupRrs( G, H );
<< presentation with 18 gens and 35 rels of total length 169 >>
gap> TzGoGo( P );
#I  there are 3 generators and 20 relators of total length 488
#I  there are 3 generators and 20 relators of total length 466
gap> # We end up with 20 relators of total length 466.
gap> #
gap> # Now we repeat the procedure, but we call the tree decoding
gap> # algorithm before doing the Tietze transformations.
gap> P := PresentationSubgroupRrs( G, H );
<< presentation with 18 gens and 35 rels of total length 169 >>
gap> DecodeTree( P );
#I  there are 9 generators and 26 relators of total length 185
#I  there are 6 generators and 23 relators of total length 213
#I  there are 3 generators and 20 relators of total length 252
#I  there are 3 generators and 20 relators of total length 244
gap> TzGoGo( P );
#I  there are 3 generators and 19 relators of total length 168
#I  there are 3 generators and 17 relators of total length 138
#I  there are 3 generators and 15 relators of total length 114
#I  there are 3 generators and 13 relators of total length 96
#I  there are 3 generators and 12 relators of total length 84
gap> # This time we end up with a shorter presentation.
```

As an example of an implicit call of the command via the PresentationSubgroupMtc command we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann.

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> a := F3.1;;  b := F3.2;;  c := F3.3;;
gap> G := F3 / [ a^3, b^3, c^3, (a*b)^5, (a^-1*b)^5, (a*c)^4,
>     (a*c^-1)^4, a*b^-1*a*b*c^-1*a*c*a*c^-1, (b*c)^3, (b^-1*c)^4 ];;
gap> a := G.1;;  b := G.2;;  c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> InfoFpGroup1 := Print;;
gap> P := PresentationSubgroupMtc( G, H );;
```

```
#I  index = 240  total = 4737  max = 4507
#I  MTC defined 2 primary and 4446 secondary subgroup generators
#I  there are 246 generators and 617 relators of total length 2893
#I  calling DecodeTree
#I  there are 115 generators and 382 relators of total length 1837
#I  there are 69 generators and 298 relators of total length 1785
#I  there are 44 generators and 238 relators of total length 1767
#I  there are 35 generators and 201 relators of total length 2030
#I  there are 26 generators and 177 relators of total length 2084
#I  there are 23 generators and 167 relators of total length 2665
#I  there are 20 generators and 158 relators of total length 2848
#I  there are 20 generators and 148 relators of total length 3609
#I  there are 21 generators and 148 relators of total length 5170
#I  there are 24 generators and 148 relators of total length 7545
#I  there are 27 generators and 146 relators of total length 11477
#I  there are 32 generators and 146 relators of total length 18567
#I  there are 36 generators and 146 relators of total length 25440
#I  there are 39 generators and 146 relators of total length 38070
#I  there are 43 generators and 146 relators of total length 54000
#I  there are 41 generators and 143 relators of total length 64970
#I  there are 8 generators and 129 relators of total length 20031
#I  there are 7 generators and 125 relators of total length 27614
#I  there are 4 generators and 113 relators of total length 36647
#I  there are 3 generators and 108 relators of total length 44128
#I  there are 2 generators and 103 relators of total length 35394
#I  there are 2 generators and 102 relators of total length 34380
gap> TzGoGo( P );
#I  there are 2 generators and 101 relators of total length 19076
#I  there are 2 generators and 84 relators of total length 6552
#I  there are 2 generators and 38 relators of total length 1344
#I  there are 2 generators and 9 relators of total length 94
#I  there are 2 generators and 8 relators of total length 86
gap> TzPrintGenerators( P );
#I  1.  _x1   43 occurrences
#I  2.  _x2   43 occurrences
```

Chapter 24

Words in Finite Polycyclic Groups

Ag words are the GAP3 datatype for elements of finite polycyclic groups. Unlike permutations, which are all considered to be elements of one large symmetric group, each ag word belongs to a specified group. Only ag words of the same finite polycyclic group can be multiplied.

The following sections describe ag words and their parent groups (see 24.1), how ag words are compared (see 24.2), functions for ag words and some low level functions for ag words (starting at 24.3 and 24.9).

For operations and functions defined for group elements in general see 7.2, 7.3.

24.1 More about Ag Words

Let G be a group and $G = G_0 > G_1 > ... > G_n = 1$ be a subnormal series of $G \neq 1$ with finite cyclic factors, i.e., $G_i \lhd G_{i-1}$ for all $i = 1, ..., n$ and $G_{i-1} = \langle G_i, g_i \rangle$. Then G will be called an **ag group** with **AG generating sequence** or, for short, **AG system** $(g_1, ..., g_n)$. Let o_i be the order of G_{i-1}/G_i. If all $o_1, ..., o_n$ are primes the system $(g_1, ..., g_n)$ is called a **PAG system** . With respect to a given AG system the group G has a so called **power-commutator presentation**

$$\begin{array}{rcll} {g_i}^{o_i} & = & w_{ii}(g_{i+1}, ..., g_n) & \text{for } 1 \leq i \leq n, \\ [g_i, g_j] & = & w_{ij}(g_{j+1}, ..., g_n) & \text{for } 1 \leq j < i \leq n \end{array}$$

and a so called **power-conjugate presentation**

$$\begin{array}{rcll} {g_i}^{o_i} & = & w_{ii}(g_{i+1}, ..., g_n) & \text{for } 1 \leq i \leq n, \\ g_i^{g_j} & = & w'_{ij}(g_{j+1}, ..., g_n) & \text{for } 1 \leq j < i \leq n. \end{array}$$

For both kinds of presentations we shall use the term **AG presentation**. Each element g of G can be expressed uniquely in the form

$$g = g_1^{\nu_1} * ... * g_n^{\nu_n} \quad \text{for } 0 \leq \nu_i < o_i.$$

We call the composition series $G_0 > G_1 > ... > G_n$ the **AG series** of G and define $\nu_i(g) := \nu_i$. If $\nu_i = 0$ for $i = 1, ..., k-1$ and $\nu_k \neq 0$, we call ν_k the **leading exponent** and k the **depth** of g and denote them by $\nu_k =: \lambda(g)$ and $k =: \delta(g)$. We call o_k the **relative order** of g.

Each element g of G is called **ag word** and we say that G is the parent group of g. A parent group is constructed in GAP3 using `AgGroup` (see 25.25) or `AgGroupFpGroup` (see 25.27).

Our standard example in the following sections is the symmetric group of degree 4, defined by the following sequence of GAP3 statements. You should enter them before running any example. For details on `AbstractGenerators` see 22.1.

```
gap> a  := AbstractGenerator( "a" );;  # (1,2)
gap> b  := AbstractGenerator( "b" );;  # (1,2,3)
gap> c  := AbstractGenerator( "c" );;  # (1,3)(2,4)
gap> d  := AbstractGenerator( "d" );;  # (1,2)(3,4)
gap> s4 := AgGroupFpGroup( rec(
>        generators := [ a, b, c, d ],
>        relators   := [ a^2, b^3, c^2, d^2, Comm( b, a ) / b,
>                        Comm( c, a ) / d, Comm( d, a ),
>                        Comm( c, b ) / ( c*d ), Comm( d, b ) / c,
>                        Comm( d, c ) ] ) );
Group( a, b, c, d )
gap> s4.name := "s4";;
gap> a := s4.generators[1];; b := s4.generators[2];;
gap> c := s4.generators[3];; d := s4.generators[4];;
```

24.2 Ag Word Comparisons

g `<` h
g `<=` h
g `>=` h
g `>` h

The operators `<`, `>`, `<=` and `>=` return `true` if g is strictly less, strictly greater, not greater, not less, respectively, than h. Otherwise they return `false`.

If g and h have a common parent group they are compared with respect to the AG series of this group. If two ag words have different depths, the one with the higher depth is less than the other one. If two ag words have the same depth but different leading exponents, the one with the smaller leading exponent is less than the other one. Otherwise the leading generator is removed in both ag words and the remaining ag words are compared.

If g and h do not have a common parent group, then the composition lengths of the parent groups are compared.

You can compare ag words with objects of other types. Field elements, unkowns, permutations and abstract words are smaller than ag words. Objects of other types, i.e., functions, lists and records are larger.

```
gap> 123/47 < a;
true
```

```
gap> (1,2,3,4) < a;
true
gap> [1,2,3,4] < a;
false
gap> true < a;
false
gap> rec() < a;
false
gap> c < a;
true
gap> a*b < a*b^2;
true
```

24.3 CentralWeight

`CentralWeight( g )`

`CentralWeight` returns the central weight of an ag word g, with respect to the central series used in the combinatorial collector, as integer.

This presumes that g belongs to a parent group for which the combinatorial collector is used. See 25.33 for details.

If g is the identity, 0 is returned.

Note that `CentralWeight` allows records that mimic ag words as arguments.

```
gap> d8 := AgGroup( Subgroup( s4, [ a, c, d ] ) );
Group( g1, g2, g3 )
gap> ChangeCollector( d8, "combinatorial" );
gap> List( d8.generators, CentralWeight );
[ 1, 1, 2 ]
```

24.4 CompositionLength

`CompositionLength( g )`

Let G be the parent group of the ag word g. Then `CompositionLength` returns the length of the AG series of G as integer.

Note that `CompositionLength` allows records that mimic ag words as arguments.

```
gap> CompositionLength( c );
5
```

24.5 Depth

`Depth( g )`

`Depth` returns the depth of an ag word g with respect to the AG series of its parent group as integer.

Let G be the parent group of g and $G = G_0 > ... > G_n = \{1\}$ the AG series of G. Let δ be the maximal positive integer such that g is an element of $G_{\delta-1}$. Then δ is the **depth** of g.

Note that `Depth` allows record that mimic ag words as arguments.

```
gap> Depth( a );
1
gap> Depth( d );
4
gap> Depth( a^0 );
5
```

24.6 IsAgWord

`IsAgWord(` *obj* `)`

`IsAgWord` returns `true` if *obj*, which can be an arbitrary object, is an ag word and `false` otherwise.

```
gap> IsAgWord( 5 );
false
gap> IsAgWord( a );
true
```

24.7 LeadingExponent

`LeadingExponent(` *g* `)`

`LeadingExponent` returns the leading exponent of an ag word g as integer.

Let G be the parent group of g and $(g_1, ..., g_n)$ the AG system of G and let o_i be the relative order of g_i. Then the element g can be expressed uniquely in the form $g_1^{\nu_1} * ... * g_n^{\nu_n}$ for integers ν_i such that $0 \leq \nu_i < o_i$. The **leading exponent** of g is the first nonzero ν_i.

If g is the identity 0 is returned.

Although `ExponentAgWord(` *g*`, Depth(` *g* `) )` returns the leading exponent of g, too, this function is faster and is able to handle the identity.

Note that `LeadingExponent` allows records that mimic ag words as arguments.

```
gap> LeadingExponent( a * b^2 * c^2 * d );
1
gap> LeadingExponent( b^2 * c^2 * d );
2
```

24.8 RelativeOrder

`RelativeOrder(` *g* `)`

`RelativeOrder` returns the relative order of an ag word g as integer.

Let G be the parent group of g and $G = G_0 > ... > G_n = \{1\}$ the AG series of G. Let δ be the maximal positive integer such that g is an element of $G_{\delta-1}$. The **relative order** of g is the index of $G_{\delta+1}$ in G_δ, that is the order of the factor group $G_\delta/G_{\delta+1}$.

If g is the identity 1 is returned.

Note that `RelativeOrder` allows records that mimic agwords as arguments.

```
gap> RelativeOrder( a );
2
gap> RelativeOrder( b );
3
gap> RelativeOrder( b^2 * c * d );
3
```

24.9 CanonicalAgWord

`CanonicalAgWord( U, g )`

Let U be an ag group with parent group G, let g be an element of G. Let $(u_1, ..., u_m)$ be an induced generating system of U and $(g_1, ..., g_n)$ be a canonical generating system of G. Then `CanonicalAgWord` returns a word $x = g * u = g_{i_1}^{e_1} * ... * g_{i_k}^{e_k}$ such that $u \in U$ and no i_j is equal to the depth of any generator u_l.

```
gap> v4 := MergedCgs( s4, [ a*b^2, c*d ] );
Subgroup( s4, [ a*b^2, c*d ] )
gap> CanonicalAgWord( v4, a*c );
b^2*d
gap> CanonicalAgWord( v4, a*b*c*d );
b
gap> (a*b*c*d) * (a*b^2);
b*c*d
gap> last * (c*d);
b
```

24.10 DifferenceAgWord

`DifferenceAgWord( u, v )`

`DifferenceAgWord` returns an ag word s representing the difference of the exponent vectors of u and v.

Let G be the parent group of u and v. Let $(g_1, ..., g_n)$ be the AG system of G and o_i be the relative order or g_i. Then u can be expressed uniquely as $g_1^{u_1} * ... * g_n^{u_n}$ for integers u_i between 0 and $o_i - 1$ and v can be expressed uniquely as $g_1^{v_1} * ... * g_n^{v_n}$ for integers v_i between 0 and $o_i - 1$. The function `DifferenceAgWord` returns an ag word $s = g_1^{s_1} * ... * g_n^{s_n}$ with integer s_i such that $0 \leq s_i < o_i$ and $s_i \equiv u_i - v_i \bmod o_i$.

```
gap> DifferenceAgWord( a * b, a );
b
gap> DifferenceAgWord( a, b );
a*b^2
gap> z27 := CyclicGroup( AgWords, 27 );
Group( c27_1, c27_2, c27_3 )
gap> x := z27.1 * z27.2;
c27_1*c27_2
gap> x * x;
c27_1^2*c27_2^2
gap> DifferenceAgWord( x, x );
IdAgWord
```

24.11 ReducedAgWord

`ReducedAgWord( b, x )`

Let b and x be ag words of the same depth, then `ReducedAgWord` returns an ag word a such that a is an element of the coset Ub, where U is the cyclic group generated by x, and a has a higher depth than b and x.

Note that the relative order of b and x must be a prime.

Let p be the relative order of b and x. Let β and ξ be the leading exponent of b and x respectively. Then there exits an integer i such that $\xi * i = \beta$ modulo p. We can set $a = x^{-i}b$.

Typically this function is used when b and x occur in a generating set of a subgroup W. Then b can be replaced by a in the generating set of W, but a and x have different depth.

```
gap> ReducedAgWord( a*b^2*c, a );
b^2*c
gap> ReducedAgWord( last, b );
c
```

24.12 SiftedAgWord

`SiftedAgWord( U, g )`

`SiftedAgWord` tries to sift an ag word g, which must be an element of the parent group of an ag group U, through an induced generating system of U. `SiftedAgWord` returns the remainder of this shifting process.

The identity is returned if and only if g is an element of U.

Let $u_1, ..., u_m$ be an induced generating system of U. If there exists an u_i such that u_i and g have the same depth, then g is reduced with u_i using `ReducedAgWord` (see 24.11). The process is repeated until no u_i can be found or the g is reduced to the identity.

`SiftedAgWord` allows factor group arguments. See 25.57 for details.

Note that `SiftedAgGroup` adds a record component `U.shiftInfo` to the ag group record of U. This entry is used by subsequent calls with the same ag group in order to speed up computation. If you ever change the component `U.igs` by hand, not using `Normalize`, you must unbind `U.shiftInfo`, otherwise all following results of `SiftedAgWord` will be corrupted.

```
gap> s3 := Subgroup( s4, [ a, b ] );
Subgroup( s4, [ a, b ] )
gap> SiftedAgWord( s3, a * b^2 * c );
c
```

24.13 SumAgWord

`SumAgWord( u, v )`

`SumAgWord` returns an ag word s representing the sum of the exponent vectors of u and v.

Let G be the parent group of u and v. Let $(g_1, ..., g_n)$ be the AG system of G and o_i be the relative order or g_i. Then u can be expressed uniquely as $g_1^{u_1} * ... * g_n^{u_n}$ for integers u_i

between 0 and $o_i - 1$ and v can be expressed uniquely as $g_1^{v_1} * ... * g_n^{v_n}$ for integers v_i between 0 and $o_i - 1$. Then `SumAgWord` returns an ag word $s = g_1^{s_1} * ... * g_n^{s_n}$ with integer s_i such that $0 \leq s_i < o_i$ and $s_i \equiv u_i + v_i \bmod o_i$.

```
gap> SumAgWord( b, a );
a*b
gap> SumAgWord( a*b, a );
b
gap> RelativeOrderAgWord( a );
2
gap> z27 := CyclicGroup( AgWords, 27 );
Group( c27_1, c27_2, c27_3 )
gap> x := z27.1 * z27.2;
c27_1*c27_2
gap> y := x ^ 2;
c27_1^2*c27_2^2
gap> x * y;
c27_2*c27_3
gap> SumAgWord( x, y );
IdAgWord
```

24.14 ExponentAgWord

`ExponentAgWord(` *g*, *k* `)`

`ExponentAgWord` returns the exponent of the k.th generator in an ag word g as integer, where k refers to the numbering of generators of the parent group of g.

Let G be the parent group of g and $(g_1, ..., g_n)$ the AG system of G and let o_i be the relative order of g_i. Then the element g can be expressed uniquely in the form $g_1^{\nu_1} * ... * g_n^{\nu_n}$ for integers ν_i between 0 and $o_i - 1$. The **exponent** of the k.th generator is ν_k.

See also 24.15 and 25.73.

```
gap> ExponentAgWord( a * b^2 * c^2 * d, 2 );
2
gap> ExponentAgWord( a * b^2 * c^2 * d, 4 );
1
gap> ExponentAgWord( a * b^2 * c^2 * d, 3 );
0
gap> a * b^2 * c^2 * d;
a*b^2*d
```

24.15 ExponentsAgWord

`ExponentsAgWord(` *g* `)`
`ExponentsAgWord(` *g*, *s*, *e* `)`
`ExponentsAgWord(` *g*, *s*, *e*, *root* `)`

In its first form `ExponentsAgWord` returns the exponent vector of an ag word g, with respect to the AG system of the supergroup of g, as list of integers. In the second form `ExponentsAgWord` returns the sublist of the exponent vector of g starting at position s and ending at position e as list of integers. In the third form the vector is returned as list of finite field elements over the same finite field as *root*.

Let G be the parent group of g and $(g_1, ..., g_n)$ the AG system of G and let o_i be the relative order of g_i. Then the element g can be expressed uniquely in the form $g_1^{\nu_1} * ... * g_n^{\nu_n}$ for integers ν_i between 0 and $o_i - 1$. The exponent vector of g is the list `[`ν_1`, ..., `ν_n`]`.

Note that you must use `Exponents` if you want to get the exponent list of g with respect not to the parent group of g but to a given subgroup, which contains g. See 25.73 for details.

```
gap> ExponentsAgWord( a * b^2 * c^2 * d );
[ 1, 2, 0, 1 ]
gap> a * b^2 * c^2 * d;
a*b^2*d
```

Chapter 25

Finite Polycyclic Groups

Ag groups (see 24) are a subcategory of finitely generated groups (see 7).

The following sections describe how subgroups of ag groups are represented (see 25.1), additional operators and record components of ag groups (see 25.3 and 25.4) and functions which work only with ag groups (see 25.24 and 25.61). Some additional information about generating systems of subgroups and factor groups are given in 25.48 and 25.57.

25.85 describes how to compute the groups of one coboundaries and one cocycles for given ag groups. 25.88 gives informations how to obtain complements and conjugacy classes of complements for given ag groups.

25.1 More about Ag Groups

Let G be a finite polycyclic group with PAG system $(g_1, ..., g_n)$ as described in 24. Let U be a subgroup of G. A generating system $(u_1, ..., u_r)$ of U is called the **canonical generating system**, CGS for short, of U with respect to $(g_1, ..., g_n)$ if and only if

(i)	$(u_1, ..., u_r)$ is a PAG system for U,
(ii)	$\delta(u_i) > \delta(u_j)$ for $i > j$,
(iii)	$\lambda(u_i) = 1$ for $i = 1, ..., r$,
(iv)	$\nu_{\delta(u_i)}(u_j) = 0$ for $i \neq j$.

If a generating system $(u_1, ..., u_r)$ fulfills only conditions (i) and (ii) this system is called an **induced generating system**, IGS for short, of U. With respect to the PAG system of G a CGS but not an IGS of U is unique.

If a power-commutator or power-conjugate presentation of G is known, a finite polycyclic group with collector can be initialized in GAP3 using `AgGroupFpGroup` (see 25.27). `AgGroup` (see 25.25) converts other types of finite solvable groups, for instance solvable permutation groups, into an ag group. The collector can be changed by `ChangeCollector` (see 25.33). The elements of these group are called **ag words**.

A canonical generating system of a subgroup U of G is returned by `Cgs` (see 25.50) if a generating set of ag words for U is known. See 25.48 for details.

We call G a **parent**, that is a ag group with collector and U a **subgroup**, that is a group which is obtained as subgroup of a parent group. An **ag group** is either a parent group with PAG system or a subgroup of such a parent group.

Although parent groups need only an AG system, only `AgGroupFpGroup` (see 25.27) and `RefinedAgSeries` (see 25.32) work correctly with a parent group represented by an AG system which is not a PAG system, because subgroups are identified by canonical generating systems with respect to the PAG system of the parent group. Inconsistent power-conjugate or power-commutator presentations are not allowed (see 25.28). Some functions support factor group arguments. See 25.57 and 25.60 for details.

Our standard example in the following sections is the symmetric group of degree 4, defined by the following sequence of GAP3 statements. You should enter them before running any example. For details on `AbstractGenerators` see 22.1.

```
gap> a  := AbstractGenerator( "a" );;  # (1,2)
gap> b  := AbstractGenerator( "b" );;  # (1,2,3)
gap> c  := AbstractGenerator( "c" );;  # (1,3)(2,4)
gap> d  := AbstractGenerator( "d" );;  # (1,2)(3,4)
gap> s4 := AgGroupFpGroup( rec(
>       generators := [ a, b, c, d ],
>       relators   := [ a^2, b^3, c^2, d^2, Comm( b, a ) / b,
>                       Comm( c, a ) / d, Comm( d, a ),
>                       Comm( c, b ) / ( c*d ), Comm( d, b ) / c,
>                       Comm( d, c ) ] ) );;
gap> s4.name := "s4";;
gap> a := s4.generators[1];; b := s4.generators[2];;
gap> c := s4.generators[3];; d := s4.generators[4];;
```

25.2 Construction of Ag Groups

The most fundamental way to construct a new finite polycyclic group is `AgGroupFpGroup` (see 25.27) together with `RefinedAgSeries` (see 25.32), if a presentation for an AG system of a finite polycyclic group is known.

But usually new finite polycyclic groups are constructed from already existing finite polycyclic groups. The direct product of known ag groups can be formed by `DirectProduct` (see 7.99); also, if for instance a permutation representation P of a finite polycyclic group G is known, `WreathProduct` (see 7.104) returns the P-wreath product of G with a second ag group. If a homomorphism of a finite polycyclic group G into the automorphism group of another finite polycyclic group H is known, `SemidirectProduct` returns the semi direct product of G with H.

Fundamental finite polycyclic groups, such as elementary abelian, arbitrary finite abelian groups, and cyclic groups, are constructed by the appropriate functions (see 38.1).

25.3 Ag Group Operations

In addition to the operators described in 7.117 the following operator can be used for ag groups.

G `mod` *H*

`mod` returns a record representing an factor group argument, which can be used as argument for some functions (see 25.73). See 25.57 and 25.60 for details.

25.4 Ag Group Records

In addition to the record components described in 7.118 the following components may be present in the group record of an ag group *G*.

`isAgGroup`
: is always `true`.

`isConsistent`
: is `true` if *G* has a consistent presentation (see 25.28).

`compositionSeries`
: contains a composition series of *G* (see 7.38).

`cgs`
: contains a canonical generating system for *G*. If *G* is a parent group, it is always present. See 25.48 for details.

`igs`
: contains an induced generating system for *G*. See 25.48 for details.

`elementaryAbelianFactors`
: see 7.39.

`sylowSystem`
: contains a Sylow system (see 25.67).

25.5 Set Functions for Ag Groups

As already mentioned in the introduction of the chapter, ag groups are domains. Thus all set theoretic functions, for example `Intersection` and `Size`, can be applied to ag groups. This and the following sections give further comments on the definition and implementations of those functions for ag groups. All set theoretic functions not mentioned here not treated special for ag groups.

`Elements(` *G* `)`

The elements of a group *G* are constructed using a canonical generating system. See 25.6.

g `in` *G*

Membership is tested using `SiftedAgWord` (see 24.12), if *g* lies in the parent group of *G*. Otherwise `false` is returned.

`IsSubset(` *G*`,` *H* `)`

If *G* and *H* are groups then `IsSubset` tests if the generators of *H* are elements of *G*. Otherwise `DomainOps.IsSubset` is used.

`Intersection(` G`,` H `)`

The intersection of ag groups G and H is computed using Glasby's algorithm. See 25.7.

`Size(` G `)`

The size of G is computed using a canonical generating system of G. See 25.8.

25.6 Elements for Ag Groups

`AgGroupOps.Elements(` G `)`

Let G be an ag group with canonical generating system $(g_1, ..., g_n)$ where the relative order of g_i is o_i. Then $\{g_1^{e_1}...g_n^{e_n}; 0 \leq e_i < o_i\}$ is the set of elements of G.

25.7 Intersection for Ag Groups

`AgGroupOps.Intersection(` U`,` V `)`

If either V or U is not an ag group then `GroupOps.Intersection` is used in order to compute the intersection of U and V. If U and V have different parent groups then the empty list is returned.

Let U and V be two ag group with common parent group G. If one subgroup if known to be normal in G the `NormalIntersection` (see 7.26) is used in order to compute the intersection.

If the size of U or V is smaller than `GS_SIZE` then the intersection is computed using `GroupOps.Intersection`. By default `GS_SIZE` is 20.

If an elementary abelian ag series of G is known, Glasby's generalized covering algorithm is used (see [GS90]). Otherwise a warning is given and `GroupOps.Intersection` is used, but this may be too slow.

```
gap> d8_1 := Subgroup( s4, [ a, c, d ] );
Subgroup( s4, [ a, c, d ] )
gap> d8_2 := Subgroup( s4, [ a*b, c, d ] );
Subgroup( s4, [ a*b, c, d ] )
gap> Intersection( d8_1, d8_2 );
Subgroup( s4, [ c, d ] )
gap> Intersection( d8_1^b, d8_2^b );
Subgroup( s4, [ c*d, d ] )
```

25.8 Size for Ag Groups

`AgGroupOps.Size(` G `)`

Let G be an ag group with induced generating system $(g_1, ..., g_n)$ where the relative order of g_i is o_i. Then the size of G is $o_1 * ... * o_n$.

`AgGroupOps.Size` allows a factor argument (see 25.60) for G. It uses `Index` (see 7.51) in such a case.

25.9 Group Functions for Ag Groups

As ag groups are groups, all group functions, for example `IsAbelian` and `Normalizer`, can be applied to ag groups. This and the following sections give further comments on the definition and implementations of those functions for ag groups. All group functions not mentioned here are not treated in a special way.

`Group(` *U* `)`

See 25.11.

`CompositionSeries(` *G* `)`

Let $(g_1, ..., g_n)$ be an induced generating system of G with respect to the parent group of G. Then for $i \in \{1, ..., n\}$ the i.th composition subgroup S_i of the AG system is generated by $(g_i, ..., g_n)$. The $n + 1$.th composition subgroup S_{n+1} is the trivial subgroup of G. The AG series of G is the series $\{S_1, ..., S_{n+1}\}$.

`Centralizer(` *U* `)`

The centralizer of an ag group U in its parent group is computed using linear methods while stepping down an elementary abelian series of its parent group.

`Centralizer(` *U*`,` *H* `)`

This function call computes the centralizer of H in U using linear methods. H and U must have a common parent.

`Centralizer(` *U*`,` *g* `)`

The centralizer of a single element g in an ag group U may be computed whenever g lies in the parent group of U. In that case the same algorithm as for the centralizer of subgroups is used.

`ConjugateSubgroup(` *U*`,` *g* `)`

If g is an element of U then U is returned. Otherwise the remainder of the shifting of g through U is used to conjugate an induced generating system of U. In that case the information bound to *U*`.isNilpotent`, *U*`.isAbelian`, *U*`.isElementaryAbelian` and *U*`.isCyclic`, if known, is copied to the conjugate subgroup.

`Core(` *S*`,` *U* `)`

`AgGroupOps.Core` computes successively the core of U stepping up a composition series of S. See [Thi87].

`CommutatorSubgroup(` *G*`,` *H* `)`

See 25.12 for details.

`ElementaryAbelianSeries(` *G* `)`

`AgGroupOps.ElementaryAbelianSeries` returns a series of normal subgroups of G with elementary abelian factors.

```
gap> ElementaryAbelianSeries( s4 );
[ s4, Subgroup( s4, [ b, c, d ] ), Subgroup( s4, [ c, d ] ),
  Subgroup( s4, [  ] ) ]
gap> d8 := Subgroup( s4, [ a*b^2, c, d ] );
Subgroup( s4, [ a*b^2, c, d ] )
gap> ElementaryAbelianSeries( d8 );
[ Subgroup( s4, [ a*b^2, c, d ] ), Subgroup( s4, [ c, d ] ),
  Subgroup( s4, [  ] ) ]
```

If G is no parent group then `AgGroupOps.ElementaryAbelianSeries` will compute a elementary abelian series for the parent group and intersect this series with G. If G is a parent group then `IsElementaryAbelianAgSeries` (see 25.29) is used in order to check if such a series exists. Otherwise an elementary abelian is computed refining the derived series (see [LNS84, Gla87]).

`ElementaryAbelianSeries(` *L* `)`

L must be a list of ag groups $S_1 = H, ..., S_m = \{1\}$ with a common parent group such that S_i is a subgroup of S_{i-1} and S_i is normal in G for all $i \in \{2, ..., m\}$. Then the function returns a series of normal subgroups of G with elementary abelian factors refining the series L.

`NormalIntersection(` *V*`,` *W* `)`

If V is an element of the AG series of G, then `AgGroupOps.NormalIntersection` uses the depth of V in order to compute the intersection. Otherwise it uses the Zassenhaus sum-intersection algorithm (see [GS90]).

`Normalizer(` *G*`,` *U* `)`

See 25.13.

`SylowSubgroup(` *G*`,` *p* `)`

`AgGroupOps.SylowSubgroup` uses `HallSubgroup` (see 25.63) in order to compute the sylow subgroup of G.

`DerivedSeries(` *G* `)`

`AgGroupOps.DerivedSeries` uses `DerivedSubgroup` (see 7.22) in order to compute the derived series of G. It checks if G is normal in its parent group H. If it is normal all the derived subgroups are also normal in H. G is always the first element of this list and the trivial group always the last one since G is soluble.

`LowerCentralSeries(` *G* `)`

`AgGroupOps.LowerCentralSeries` uses `CommutatorSubgroup` (see 7.19) in order to compute the lower central series of G. It checks if G is normal in its parent group H. If it is normal all subgroups of the lower central series are also normal in H.

`Random(` U `)`

Let $(u_1, ..., u_r)$ be a induced generating system of U. Let $e_1, ..., e_r$ be the relative order of $u_1, ..., u_r$. Then for r random integers ν_i between 0 and $e_i - 1$ the word $u_1^{\nu_1} * ... * u_r^{\nu_r}$ is returned.

`IsCyclic(` G `)`

See 25.14.

`IsFinite(` G `)`

As G is a finite solvable group `AgGroupOps.IsFinite` returns `true`.

`IsNilpotent(` U `)`

`AgGroupOps.IsNilpotent` uses Glasby's nilpotency test for ag groups (see [Gla87]).

`IsNormal(` G`,` U `)`

See 25.15.

`IsPerfect(` G `)`

As G is a finite solvable group it is perfect if and only if G is trivial.

`IsSubgroup(` G`,` U `)`

See 25.16.

`ConjugacyClasses(` H `)`

The conjugacy classes of elements are computed using linear methods. The algorithm depends on the ag series of the parent group of H being a refinement of an elementary abelian series. Thus if this is not true or if H is not a member of the elementary abelian series, an isomorphic group, in which the computation can be done, is created.

The algorithm that is used steps down an elementary abelian series of the parent group of H, basically using affine operation to construct the conjugacy classes of H step by step from its factorgroups.

`Orbit(` U`,` pt`,` op `)`

`AgGroupOps.Orbit` returns the orbit of pt under U using the operation op. The function calls `AgOrbitStabilizer` in order to compute the orbit, so please refer to 25.78 for details.

`Stabilizer(` U`,` pt`,` op `)`

See 25.17.

`AsGroup(` D `)`

See 25.10.

`FpGroup(` U `)`

See 25.23.

`RightCoset(` U`,` g `)`

See 25.22.

`AbelianGroup(` D`,` L `)`

Let L be the list $[o_1, ..., o_n]$ of nonnegative integers $o_i > 1$. Then `AgWordsOps.AbelianGroup` returns the direct product of the cyclic groups of order o_i using the domain description D. The generators of these cyclic groups are named beginning with "a", "b", "c", ... followed by a number if o_i is a composite integer.

`CyclicGroup(` D`,` n `)`

See 25.18.

`ElementaryAbelianGroup(` D`,` n `)`

See 25.19.

`DirectProduct(` L `)`

See 25.20.

`WreathProduct(` G`,` H`,` α `)`

See 25.21.

25.10 AsGroup for Ag Groups

`AgGroupOps.AsGroup(` G `)`

`AgGroupOps.AsGroup` returns a copy H of G. It does not change the parent status. If G is a subgroup so is H.

`AgWordsOps.AsGroup(` L `)`

Let L be a list of ag words. Then `AgWordsOps.AsGroup` uses `MergedCgs` (see 25.55) in order to compute a canonical generating system for the subgroup generated by L in the parent group of the elements of L.

25.11 Group for Ag Groups

`AgGroupOps.Group(` *G* `)`

`AgGroupOps.Group` returns an isomorphic group H such that H is a parent group and H`.bijection` is bond to an isomorphism between H and G.

`AgWordsOps.Group(` *D*`,` *gens*`,` *id* `)`

Constructs the group G generated by *gens* with identity *id*. If these generators do not generate a parent group, a new parent group H is construct. In that case new generators are used and H`.bijection` is bound to isomorphism between H and G.

25.12 CommutatorSubgroup for Ag Groups

`AgGroupOps.CommutatorSubgroup(` *G*`,` *H* `)`

Let $g_1, ..., g_n$ be an canonical generating system for G and $h_1, ..., h_m$ be an canonical generating system for H. The normal closure of the subgroup S generated by $Comm(g_i, h_j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ under G and H is the commutator subgroup of G and H.

But if G or H is known to be normal in the common parent of G amd H then the subgroup S is returned because if G normalizes H or vice versa then S is already the commutator subgroup (see [Gla87]).

If $G = H$ the commutator subgroup is generated by $Comm(g_i, g_j)$ for $1 \leq i < j \leq n$ (see [LNS84]). Note that `AgGroupOps.CommutatorSubgroup` checks G`.derivedSubgroup` in that case.

25.13 Normalizer for Ag Groups

`AgGroupOps.Normalizer(` *S*`,` *U* `)`

Note that the AG series of G should be the refinement of an elementary abelian series, see 25.29. Otherwise the calculation of the normalizer is done using an orbit algorithm, which is generally too slow or space extensive. You can construct a new polycyclic presentation for G such that AG series is a refinement of an elementary abelian series with `ElementaryAbelianSeries` (see 7.39) and `IsomorphismAgGroup`.

For details on the implementation see [GS90, CNW90].

25.14 IsCyclic for Ag Groups

`AgGroupOps.IsCyclic(` *G* `)`

`AgGroupOps.IsCyclic` returns `false` if G is no abelian group. Otherwise G is finite of order $p_1^{e_1}...p_n^{e_n}$ where the p_i are distinct primes then G is cyclic if and only if each G^{p_i} has index p_i in G.

`AgGroupOps.IsCyclic` computes the groups G_i^p using the fact that the map $x \mapsto x^{p_i}$ is a homomorphism of G, so that the p_i.th powers of an induced generating system of G are a homomorphic image of an igs (see [Cel92]).

25.15 IsNormal for Ag Groups

`AgGroupOps.IsNormal(` *G*, *U* `)`

Let G be a parent group. Then `AgGroupOps.IsNormal` checks if the conjugate of each generator of U under each induced generator of G which has a depth not contained in U is an element of U. Otherwise `AgGroupOps.IsNormal` checks if the conjugate of each generator of U under each generator of G is an element of U.

25.16 IsSubgroup for Ag Groups

`AgGroupOps.IsSubgroup(` *G*, *U* `)`

If G is a parent group of U, then `AgGroupOps.IsSubgroup` returns `true`. If the CGS of U is longer than that of G, U cannot be a subgroup of G. Otherwise `AgGroupOps.IsSubgroup` shifts each generator of U through G (see 24.12) in order to check if U is a subgroup of G.

25.17 Stabilizer for Ag Groups

`AgGroupOps.Stabilizer(` *U*, *pt*, *op* `)`

Let U be an ag group acting on a set Ω by *op*. Let *pt* be an element of Ω. Then `AgGroupOps.Stabilizer` returns the stabilizer of *pt* in U.

op must be a function taking two arguments such that $op(p, u)$ is the image of a point $p \in \Omega$ under the action of an element u of U. If conjugation should be used *op* must be `OnPoints`.

The stabilizer is computed by stepping up the composition series of U. The whole orbit pt^U is not stored during the computation (see [LNS84]). Of course this saving of space is bought at the cost of time. If you need a faster function, which may use more memory, you can use `AgOrbitStabilizer` (see 25.78) instead.

25.18 CyclicGroup for Ag Groups

`AgWordsOps.CyclicGroup(` *D*, *n* `)`
`AgWordsOps.CyclicGroup(` *D*, *n*, *str* `)`

Let n be a nonnegative integer. `AgWordsOps.CyclicGroup` returns the cyclic group of order n.

Let n be a composite number with r prime factors. If no *str* is given, the names of the r generators are $cn\_1, ..., cn\_r$. Otherwise, the names of the r generators are $str1, ..., strr$, where *str* must be a string of letters, digits and the special symbol "_".

If the order n is a prime, the name of the generator is either cn or *str*.

```
gap> CyclicGroup( AgWords, 31 );
Group( c31 )
gap> AgWordsOps.CyclicGroup( AgWords, 5 * 5, "e" );
Group( e1, e2 )
```

25.19 ElementaryAbelianGroup for Ag Groups

`AgWordsOps.ElementaryAbelianGroup(` *D* `,` *n* `)`
`AgWordsOps.ElementaryAbelianGroup(` *D* `,` *n* `,` *str* `)`

`AgWordsOps.ElementaryAbelianGroup` returns the elementary abelian group of order n, which must be a prime power.

Let n be a prime power p^r. If no *str* is given the names of the r generators are $mn\_1, ..., mn\_r$. Otherwise the names of the r generators are $str1, ..., strr$, where *str* must be a string of letters, digits and the special symbol "_".

If the order n is a prime, the name of the generator is either mn or *str*.

```
gap> ElementaryAbelianGroup( AgWords, 31 );
Group( m31 )
gap> ElementaryAbelianGroup( AgWords, 31^2 );
Group( m961_1, m961_2 )
gap> AgWordsOps.ElementaryAbelianGroup( AgWords, 31^2, "e" );
Group( e1, e2 )
```

25.20 DirectProduct for Ag Groups

`AgGroupOps.DirectProduct(` *L* `)`

L must be list of groups or pairs of group and name as described below. If not all groups are ag groups `GroupOps.DirectProduct` (see 7.99) is used in order to construct the direct product.

Let L be a list of ag groups $L = [U_1, ..., U_n]$. `AgGroupOps.DirectProduct` returns the direct product of all U_i as new ag group with collector.

If L is a pair `[` U_i `,` S `]` instead of a group U_i the generators of the direct product corresponding to U_i are named Sj for integers j starting with 1 up to the number of induced generators for U_i. If the group is cyclic of prime order the name is just S.

`AgGroupOps.DirectProduct` computes for each U_i its natural power-commutator presentation for an induced generating system of U_i.

Note that the arguments need no common parent group.

```
gap> z3 := CyclicGroup( AgWords, 3 );;
gap> g := AgGroupOps.DirectProduct( [ [z3, "a"], [z3, "b"] ] );
Group( a, b )
```

25.21 WreathProduct for Ag Groups

`AgGroupOps.WreathProduct(` *G* `,` *H* `,` α `)`

If H and G are not both ag group `GroupOps.WreathProduct` (see 7.104) is used.

Let H and G be two ag group with possible different parent group and let α be a homomorphism H into a permutation group of degree d.

Let $(g_1, ..., g_r)$ be an IGS of G, $(h_1, ..., h_n)$ an IGS of H. The wreath product has a PAG system $(b_1, ..., b_n, a_{11}, ..., a_{1r}, a_{d1}, ..., a_{dr})$ such that $b_1, ..., b_n$ generate a subgroup isomorph to H and $a_{i1},...,a_{ir}$ generate a subgroup isomorph to G for each i in $\{1,...,r\}$. The names of $b_1,...,b_n$ are $h1,...,hn$, the names of $a_{i1},...,a_{ir}$ are $ni\_1,...,ni\_r$.

`AgGroupOps.WreathProduct` uses the natural power-commutator presentations of H and G for induced generating system of H and G (see [Thi87]).

```
gap> s3 := Subgroup( s4, [ a, b ] );
Subgroup( s4, [ a, b ] )
gap> c2 := Subgroup( s4, [ a ] );
Subgroup( s4, [ a ] )
gap> r := RightCosets( s3, c2 );;
gap> S3 := Operation( s3, r, OnRight );
Group( (2,3), (1,2,3) )
gap> f := GroupHomomorphismByImages(s3,S3,[a,b],[(2,3),(1,2,3)]);
GroupHomomorphismByImages( Subgroup( s4, [ a, b ] ), Group( (2,3),
(1,2,3) ), [ a, b ], [ (2,3), (1,2,3) ] )
gap> WreathProduct( c2, s3, f );
Group( h1, h2, n1_1, n2_1, n3_1 )
```

25.22 RightCoset for Ag Groups

`AgGroupOps.Coset(` *G* `)`

A coset $C = G*x$ is represented as record with the following components.

`representative`
: contains the representative x.

`group`
: contains the group G.

`isDomain`
: is `true`.

`isRightCoset`
: is `true`.

`isFinite`
: is `true`.

`operations`
: contains the operations record `RightCosetAgGroupOps`.

`RightCosetAgGroupOps.<(` *C1* `,` *C2* `)`

If *C1* and *C2* do not have a common group or if one argument is no coset then the functions uses `DomainOps.<` in order to compare *C1* and *C2*. Note that this will compute the set of elements of *C1* and *C2*.

If *C1* and *C2* have a common group then `AgGroupCosetOps.<` will use `SiftedAgWord` (see 24.12) and `ConjugateSubgroup` (see 7.20) in order to compare *C1* and *C2*. It does not compute the set of elements of *C1* and *C2*.

25.23 FpGroup for Ag Groups

`AgGroupOps.FpGroup(` *U* `)`
`AgGroupOps.FpGroup(` *U*, *str* `)`

`AgGroupOps.FpGroup` returns a finite presentation of an ag group U.

If no *str* is given, the abstract group generators have the same names as the generators of the ag group U. Otherwise they have names of the form *stri* for integers i from 1 to the number of induced generators.

`AgGroupOps.FpGroup` computes the natural power-commutator presentation of an induced generating system of the finite polycyclic group U.

25.24 Ag Group Functions

The following functions either construct new parent ag group (see 25.25 and 25.27), test properties of parent ag groups (see 25.28 and 25.29) or change the collector (see 25.33) but they do not compute subgroups. These functions are either described in general in chapter 7 or in 25.61 for specialized functions.

25.25 AgGroup

`AgGroup(` *D* `)`

`AgGroup` converts a finite polycyclic group D into an ag group G. G`.bijection` is bound to isomorphism between G and D.

```
gap> S4p := Group( (1,2,3,4), (1,2) );
Group( (1,2,3,4), (1,2) )
gap> S4p.name := "S4_PERM";;
gap> S4a := AgGroup( S4p );
Group( g1, g2, g3, g4 )
gap> S4a.name := "S4_AG";;
gap> L := CompositionSeries( S4a );
[ S4_AG, Subgroup( S4_AG, [ g2, g3, g4 ] ),
  Subgroup( S4_AG, [ g3, g4 ] ), Subgroup( S4_AG, [ g4 ] ),
  Subgroup( S4_AG, [  ] ) ]
gap> List( L, x -> Image( S4a.bijection, x ) );
[ Subgroup( S4_PERM, [ (1,2), (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( S4_PERM, [ (1,3,2), (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( S4_PERM, [ (1,4)(2,3), (1,2)(3,4) ] ),
  Subgroup( S4_PERM, [ (1,2)(3,4) ] ), Subgroup( S4_PERM, [  ] ) ]
```

Note that the function will not work for finitely presented groups, see 25.27 for details.

25.26 IsAgGroup

`IsAgGroup(` *obj* `)`

`IsAgGroup` returns `true` if *obj*, which can be an arbitrary object, is an ag group and `false` otherwise.

```
gap> IsAgGroup( s4 );
true
gap> IsAgGroup( a );
false
```

25.27 AgGroupFpGroup

`AgGroupFpGroup(` F `)`

`AgGroupFpGroup` returns an ag group isomorphic to a finitely presented finite polycyclic group F.

A finitely presented finite polycyclic group F must be a record with components `generators` and `relators`, such that `generators` is a list of abstract generators and `relators` a list of words in these generators describing a power-commutator or power-conjugate presentation.

Let $g_1, ..., g_n$ be the generators of F. Then the words of `relators` must be the power relators $g_k^{e_k} * w_{kk}^{-1}$ and commutator relator $Comm(g_i, g_j) * w_{ij}^{-1}$ or conjugate relators $g_i^{g_j} * w_{ij}^{-1}$ for all $1 \leq k \leq n$ and $1 \leq j < i \leq n$, such that w_{kk} are words in $g_{k+1}, ..., g_n$ and w_{ij} are words in $g_{j+1}, ..., g_n$. It is possible to omit some of the commutator or conjugate relators. Pairs of generators without commutator or conjugate relator are assumed to commute.

The relative order e_i of g_i need not to be primes, but as all functions for ag groups need a PAG system, not only an AG system, you must refine the AG series using `RefinedAgSeries` (see 25.32) in case some e_i are composite numbers.

Note that it is not checked if the AG presentation is consistent. You can use `IsConsistent` (see 25.28) to test the consistency of a presentation. Inconsistent presentations may cause other ag group functions to return incorrect results.

Initially a collector from the left following the algorithm described in [LGS90] is used in order to collect elements of the ag group. This could be changed using `ChangeCollector` (see 25.33).

Note that `AgGroup` will not work with finitely presented groups, you must use the function `AgGroupFpGroup` instead. As no checks are done you can construct an ag group with inconsistent presentation using `AgGroupFpGroup`.

25.28 IsConsistent

`IsConsistent(` G `)`
`IsConsistent(` G`,` *all* `)`

`IsConsistent` returns `true` if the finite polycyclic presentation of a parent group G is consistent and `false` otherwise.

If *all* is `true` then G`.inconsistencies` contains a list for pairs $[w_1, w_2]$ such that the words w_1 and w_2 are equal in G but have different normal forms.

Note that `IsConsistent` check and sets G`.isConsistent`.

```
gap> InfoAgGroup2 := Print;;
gap> x := AbstractGenerator( "x" );;
gap> y := AbstractGenerator( "y" );;
gap> z := AbstractGenerator( "z" );;
```

```
gap> G := AgGroupFpGroup( rec(
>       generators := [ x, y, z ],
>       relators   := [ x^2 / y, y^2 / z, z^2,
>                       Comm( y, x ) / ( y * z ),
>                       Comm( z, x ) / ( y * z )] ) );
Group( x, y, z )
gap> IsConsistent( G );
#I  IsConsistent: y * ( y * x ) <> ( y * y ) * x
false
gap> IsConsistent( G, true );
#I  IsConsistent: y * ( y * x ) <> ( y * y ) * x
#I  IsConsistent: z * ( z * x ) <> ( z * z ) * x
#I  IsConsistent: y * ( x * x ) <> ( y * x ) * x
#I  IsConsistent: z * ( x * x ) <> ( z * x ) * x
#I  IsConsistent: x * ( x * x ) <> ( x * x ) * x
false
gap> G.inconsistencies;
[ [ x, x*y ], [ x*z, x ], [ z, y ], [ y*z, y ], [ x*y, x ] ]
gap> InfoAgGroup2 := Ignore;;
```

25.29 IsElementaryAbelianAgSeries

`IsElementaryAbelianAgSeries(` *G* `)`

Let *G* be a parent group. `IsElementaryAbelianAgSeries` returns `true` if and only if the AG series of *G* is the refinement of an elementary abelian series of *G*.

The function sets *G*`.elementaryAbelianSeries` *G* in case of a `true` result. This component is described in 7.39.

```
gap> IsElementaryAbelianAgSeries( s4 );
true
gap> ElementaryAbelianSeries( s4 );
[ s4, Subgroup( s4, [ b, c, d ] ), Subgroup( s4, [ c, d ] ),
  Subgroup( s4, [  ] ) ]
gap> CompositionSeries( s4 );
[ s4, Subgroup( s4, [ b, c, d ] ), Subgroup( s4, [ c, d ] ),
  Subgroup( s4, [ d ] ), Subgroup( s4, [  ] ) ]
```

25.30 MatGroupAgGroup

`MatGroupAgGroup(` *U*`,` *M* `)`

Let *U* and *M* be two ag groups with a common parent group and let *M* be a elementary abelian group normalized by *U*. Then `MatGroupAgGroup` returns the matrix representation of *U* acting on *M*.

See also 25.79.

```
gap> v4 := AgSubgroup( s4, [ c, d ], true );
Subgroup( s4, [ c, d ] )
gap> a4 := AgSubgroup( s4, [ b, c, d ], true );
```

```
Subgroup( s4, [ b, c, d ] )
gap> MatGroupAgGroup( s4, v4 );
Group( [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ] )
```

25.31 PermGroupAgGroup

`PermGroupAgGroup(` G, U `)`

Let U be a subgroup of a ag group G. Then `PermGroupAgGroup` returns the permutation representation of G acting on the cosets of U.

```
gap> v4 := AgSubgroup( s4, [ s4.1, s4.4 ], true );
Subgroup( s4, [ a, d ] )
gap> PermGroupAgGroup( s4, v4 );
Group( (3,5)(4,6), (1,3,5)(2,4,6), (1,2)(3,4), (3,4)(5,6) )
```

25.32 RefinedAgSeries

`RefinedAgSeries(` G `)`

`RefinedAgSeries` returns a new parent group isomorphic to a parent group G with a PAG series, if the ag group G has only an AG series.

Note that in the case that G has a PAG series, G is returned without any further action.

The names of the new generators are constructed as follows. Let $(g_1, ..., g_n)$ be the AG system of the ag group G and $n(g_i)$ the name of g_i. If the relative order of g_i is a prime, then $n(g_i)$ is the name of a new generator. If the relative order is a composite number with r prime factors, then there exist r new generators with names $n(g_i)\_1, ..., n(g_i)\_r$.

```
gap> c12 := AbstractGenerator( "c12" );;
gap> F := rec( generators := [ c12 ],
>              relators   := [ c12 ^ ( 2 * 2 * 3 ) ] );
rec(
  generators := [ c12 ],
  relators := [ c12^12 ] )
gap> G := AgGroupFpGroup( F );
#W  AgGroupFpGroup: composite index, use 'RefinedAgSeries'
Group( c12 )
gap> RefinedAgSeries( G );
Group( c121, c122, c123 )
```

25.33 ChangeCollector

`ChangeCollector(` G, *name* `)`
`ChangeCollector(` G, *name*, n `)`

`ChangeCollector` changes the collector of a parent group G and all its subgroups. *name* is the name of the new collector. The following collectors are supported.

“single” initializes a collector from the left following the algorithm described in [LGS90].

"triple" initializes a collector from the left collecting with triples $g_i \hat{} g_j^r$ for $j < i$ and $r = 1, ..., n$ (see [Bis89]).

"quadruple" initializes a collector from the left collecting with quadruples $g_i^s \hat{} g_j^r$ for $j < i$, $r = 1, ..., n$ and $s = 1, ..., n$. Note that r and s have the same upper bound (see [Bis89]).

"combinatorial" initializes a combinatorial collector from the left for a p-group G. In that case the commutator or conjugate relations of the G must be of the form $g_i^{g_j} = w_{ij}$ or $Comm(g_i, g_j) = w_{ij}$ for $1 \leq j < i \leq n$, such that w_{ij} are words in $g_{i+1}, ..., g_n$ fulfilling the central weight condition (see [HN80, VL84]). If these conditions are not fulfilled, the collector could not be initialized, a warning will be printed and collection will be done with the old collector.

For collectors which collect with tuples a maximal bound of those tuples is n, set to 5 by default.

25.34 The Prime Quotient Algorithm

The following sections describe the np-quotient functions. `PQuotient` allows to compute quotient of prime power order of finitely presented groups. For further references see [HN80] and [VL84].

There is a C standalone version of the p-quotient algorithm, the ANU p-Quotient Program, which can be called from `GAP3`. For further information see chapter 58.

25.35 PQuotient

`PQuotient(` *G*`,` *p*`,` *cl* `)`
`PrimeQuotient(` *G*`,` *p*`,` *cl* `)`

`PQuotient` computes quotients of prime power order of finitely presented groups. G must be a group given by generators and relations. `PQuotient` expects G to be a record with the record fields `generators` and `relators`. The record field `generators` must be a list of abstract generators created by the function `AbstractGenerator` (see 22.1). The record field `relators` must be a list of words in the generators which are the relators of the group. p must be a prime. *cl* has to be an integer, which specifies that the quotient of prime power order computed by `PQuotient` is the largest p-quotient of G of class at most *cl*. `PQuotient` returns a record `Q`, the **PQp record**, which has, among others, the following record fields describing the p-quotient Q.

`generators`
: A list of abstract generators which generate Q.

`pcp`
: The internal power-commutator presentation for Q.

`dimensions`
: A list, where `dimensions[i]` is the dimension of the i-th factor in the lower exponent-p central series calculated by the p-quotient algorithm.

`prime`
: The integer p, which is a prime.

`definedby`
: A list which contains the definition of the k-th generator in the k-th place. There are

three different types of entries, namely lists, positive and negative integers.

`[ j, i ]`
the generator is defined to be the commutator of the j-th and the i-th element in `generators`.

`i`
the generator is defined as the p-th power of the i-th element in `generators`.

`-i`
the generator is defined as an image of the i-th generator in the finite presentation for G, consequently it must be a generator of weight 1.

`epimorphism`
A list containing an image in Q of each generator of G. The image is either an integer `i` if it is the i-th element of `generators` of Q or an abstract word `w` if it is the abstract word `w` in the generators of Q.

An example of the computation of the largest quotient of class 4 of the group given by the finite presentation $\{x, y \mid x^{25}/(x \cdot y)^5, [x, y]^5, (x^y)^{25}\}$.

```
# Define the group
gap> x := AbstractGenerator("x");;
gap> y := AbstractGenerator("y");;
gap> G := rec( generators := [x,y],
>              relators := [ x^25/(x*y)^5, Comm(x,y)^5, (x^y)^25] );
rec(
  generators := [ x, y ],
  relators :=
   [ x^25*y^-1*x^-1*y^-1*x^-1*y^-1*x^-1*y^-1*x^-1*y^-1*x^-1,
      x^-1*y^-1*x*y*x^-1*y^-1*x*y*x^-1*y^-1*x*y*x^-1*y^-1*x*y*x^-1*y^-\
1*x*y, y^-1*x^25*y ] )

# Call pQuotient
gap> P := PQuotient( G, 5, 4 );
#I  PQuotient: class 1 : 2
#I  PQuotient: Runtime : 0
#I  PQuotient: class 2 : 2
#I  PQuotient: Runtime : 27
#I  PQuotient: class 3 : 2
#I  PQuotient: Runtime : 1437
#I  PQuotient: class 4 : 3
#I  PQuotient: Runtime : 1515
PQp( rec(
   generators  := [ g1, g2, a3, a4, a6, a7, a11, a12, a14 ],
   definedby   := [ -1, -2, [ 2, 1 ], 1, [ 3, 1 ], [ 3, 2 ],
  [ 5, 1 ], [ 5, 2 ], [ 6, 2 ] ],
   prime       := 5,
   dimensions  := [ 2, 2, 2, 3 ],
   epimorphism := [ 1, 2 ],
   powerRelators := [ g1^5/(a4), g2^5/(a4^4), a3^5, a4^5, a6^5, a7^
5, a11^5, a12^5, a14^5 ],
   commutatorRelators := [ Comm(g2,g1)/(a3), Comm(a3,g1)/(a6), Comm(a3\
```

```
,g2)/(a7), Comm(a6,g1)/(a11), Comm(a6,g2)/(a12), Comm(a7,g1)/(a12), Co\
mm(a7,g2)/(a14) ],
   definingCommutators := [ [ 2, 1 ], [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
  [ 5, 2 ], [ 6, 1 ], [ 6, 2 ] ] ) )
```

The p-quotient algorithm returns a PQp record for the exponent-5 class 4 quotient. Note that instead of printing the PQp record `P` an equivalent representation is printed which can be read in to GAP3. See 25.37 for details.

The quotient defined by `P` has nine generators, `g1, g2, a3, a4, a6, a7,a11, a12, a14`, stored in the list `P.generators`. From `powerRelators` we can read off that `g1^5 =: a4` and `g2^5 = a4^4` and all other generators have trivial 5-th powers. From the list `commutatorRelators` we can read off the non-trivial commutator relations `Comm(g2,g1) =: a3`, `Comm(a3,g1) =: a6`, `Comm(a3,g2) =: a7`, `Comm(a6,g1) =: a11`,`Comm(a6,g2) =: a12`, `Comm(a7,g1) = a12` and `Comm(a7,g2) =: a14`. In this list `=:` denotes that the generator on the right hand side is defined as the left hand side. This information is given by the list `definedby`. The list `dimensions` shows that `P` is a class-4 quotient of order $5^2 \cdot 5^2 \cdot 5^2 \cdot 5^3 = 5^9$. The epimorphism of `G` onto the quotient `P` is given by the map `x` $\mapsto$ `g1` and `y` $\mapsto$ `g2`.

25.36 Save

`Save(` *file*, Q, N `)`

`Save` saves the PQp record Q to the file *file* in such a way that the file can be read by GAP3. The name of the record in the file will be N. This differs from printing Q to a file in that the required abstract generators are also created in *file*.

```
gap> x := AbstractGenerator("x");;
gap> y := AbstractGenerator("y");;
gap> G := rec( generators := [x,y],
>              relators := [ x^25/(x*y)^5, Comm(x,y)^5, (x^y)^25] );;
gap> P := PQuotient( G, 5, 4 );;
#I  PQuotient: class 1 : 2
#I  PQuotient: Runtime : 0
#I  PQuotient: class 2 : 2
#I  PQuotient: Runtime : 27
#I  PQuotient: class 3 : 2
#I  PQuotient: Runtime : 78
#I  PQuotient: class 4 : 3
#I  PQuotient: Runtime : 156
gap> Save( "Quo54", P, "Q" );
gap> # The Unix command 'cat' in the next statement should be
gap> # replaced appropriately if you are working under a different
gap> # operating system.
gap> Exec( "cat Quo54" );
g1 := AbstractGenerator("g1");
g2 := AbstractGenerator("g2");
a3 := AbstractGenerator("a3");
a4 := AbstractGenerator("a4");
a6 := AbstractGenerator("a6");
```

```
a7 := AbstractGenerator("a7");
a11 := AbstractGenerator("a11");
a12 := AbstractGenerator("a12");
a14 := AbstractGenerator("a14");
Q := PQp( rec(
   generators  := [ g1, g2, a3, a4, a6, a7, a11, a12, a14 ],
   definedby   := [ -1, -2, [ 2, 1 ], 1, [ 3, 1 ], [ 3, 2 ],
  [ 5, 1 ], [ 5, 2 ], [ 6, 2 ] ],
   prime       := 5,
   dimensions  := [ 2, 2, 2, 3 ],
   epimorphism := [ 1, 2 ],
   powerRelators := [ g1^5/(a4), g2^5/(a4^4), a3^5, a4^5, a6^5, a7^
5, a11^5, a12^5, a14^5 ],
   commutatorRelators := [ Comm(g2,g1)/(a3), Comm(a3,g1)/(a6), Comm(a3\
,g2)/(a7), Comm(a6,g1)/(a11), Comm(a6,g2)/(a12), Comm(a7,g1)/(a12), Co\
mm(a7,g2)/(a14) ],
   definingCommutators := [ [ 2, 1 ], [ 3, 1 ], [ 3, 2 ], [ 5, 1 ],
  [ 5, 2 ], [ 6, 1 ], [ 6, 2 ] ] ) );
```

25.37 PQp

`PQp(` *r* `)`

`PQp` takes as argument a record r containing all information necessary to restore a PQp record Q. A PQp record Q is printed as function call to `PQp` with an argument describing Q. This is necessary because the internal power-commutator representation cannot be printed. Therefore all information about Q is encoded in a record r and Q is printed as `PQp( <r> )`.

25.38 InitPQp

`InitPQp(` *n*, *p* `)`

`InitPQp` creates a PQp record for an elementary abelian group of rank n and of order p^n for a prime p.

25.39 FirstClassPQp

`FirstClassPQp(` *G*, *p* `)`

`FirstClassPQp` returns a PQp record for the exponent-p class 1 quotient of G.

25.40 NextClassPQp

`NextClassPQp(` *G*, *P* `)`

Let P be the PQp record for the exponent-p class c quotient of G. `NextClassPQp` returns a PQp record for the class $c+1$ quotient of G, if such a quotient exists, and P otherwise. In latter case there exists a maximal p-quotient of G which has class c and this is indicated by a comment if `InfoPQ1` is set the `Print`.

25.41 Weight

`Weight( P, w )`

Let P be a PQp record and w a word in the generators of P. The function `Weight` returns the weight of w with respect to the lower exponent-p central series defined by P.

25.42 Factorization for PQp

`Factorization( P, w )`

Let P be a PQp record and w a word in the generators of P. The function `Factorization` returns a word in the weight 1 generators of P expressing w.

25.43 The Solvable Quotient Algorithm

The following sections describe the solvable quotient functions (or sq functions for short). `SolvableQuotient` allows to compute finite solvable quotients of finitely presented groups.

The solvable quotient algorithm tries to find solvable quotients of a given finitely presented group G. First it computes the commutator factor group Q, which must be finite. It then chooses a prime p and repeats the following three steps:(1) compute all irreducible modules of Q over $GF(p)$, (2) for each module M compute (up to equivalence) all extensions of Q by M, (3) for each extension E check whether E is isomorphic to a factor group of G. As soon as a non-trivial extension of Q is found which is still isomorphic to a factor group of G the process is repeated.

25.44 SolvableQuotient

`SolvableQuotient( G, primes )`

Let G be a finitely presented group and *primes* a list of primes. `SolvableQuotient` tries to compute the largest finite solvable quotient Q of G, such that the prime decomposition of the order the derived subgroup Q' of Q only involves primes occuring in the list *primes*. The quotient Q is returned as finitely presented group. You can use `AgGroupFpGroup` (see 25.27) to convert the finitely presented group into a polycyclic one.

Note that the commutator factor group of G must be finite.

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>      f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>      f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>      f.2^-1*f.4^-1*f.2*f.4 ];
Group( a, b, c, d )
gap> InfoSQ1 := Ignore;;
gap> g := SolvableQuotient( f4, [3] );
Group( e1, e2, m3, m4 )
gap> Size(AgGroupFpGroup(g));
36
gap> g := SolvableQuotient( f4, [2] );
Group( e1, e2 )
```

```
gap> Size(AgGroupFpGroup(g));
4
gap> g := SolvableQuotient( f4, [2,3] );
Group( e1, e2, m3, m4, m5, m6, m7, m8, m9 )
gap> Size(AgGroupFpGroup(g));
1152
```

Note that the order in which the primes occur in *primes* is important. If *primes* is the list $[2, 3]$ then in each step `SolvableQuotient` first tries a module over GF(2) and only if this fails a module over GF(3). Whereas if *primes* is the list $[3, 2]$ the function first tries to find a downward extension by a module over GF(3) before considering modules over GF(2).

`SolvableQuotient(` G, n `)`

Let G be a finitely presented group. `SolvableQuotient` attempts to compute a finite solvable quotient of G of order n.

Note that n must be divisible by the order of the commutator factor group of G, otherwise the function terminates with an error message telling you the order of the commutator factor group.

Note that a warning is printed if there does not exist a solvable quotient of order n. In this case the largest solvable quotient whose order divides n is returned.

Providing the order n or a multiple of the order makes the algorithm run much faster than providing only the primes which should be tested, because it can restrict the dimensions of modules it has to investigate. Thus if the order or a small enough multiple of it is known, `SolvableQuotient` should be called in this way to obtain a power conjugate presentation for the group.

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>       f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>       f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>       f.2^-1*f.4^-1*f.2*f.4 ];;
gap> g := SolvableQuotient( f4, 12 );
Group( e1, e2, m3 )
gap> Size(AgGroupFpGroup(g));
12
gap> g := SolvableQuotient( f4, 24 );
#W  largest quotient has order 2^2*3
Group( e1, e2, m3 )
gap> g := SolvableQuotient( f4, 2 );
Error, commutator factor group is of size 2^2
```

`SolvableQuotient(` G, l `)`

If something is already known about the structure of the finite soluble quotient to be constructed then `SolvableQuotient` can be aided in its construction.

l must be a list of lists each of which is a list of integers occurring in pairs p, n.

`SolvableQuotient` first constructs the commutator factor group of G, it then tries to extend this group by modules over $GF(p)$ of dimension at most n where p is a prime occurring in the first list of l. If n is zero no bound on the dimension of the module is imposed. For example, if l is $[[2,0,3,4],[5,0,2,0]]$ then `SolvableQuotient` will try to extend the commutator factor group by a module over GF(2). If no such module exists all modules over GF(3) of dimension at most 4 are tested. If neither a GF(2) nor a GF(3) module extend `SolvableQuotient` terminates. Otherwise the algorithm tries to extend this new factor group with a GF(5) and then a GF(2) module.

Note that it is possible to influence the construction even more precisely by using the functions `InitSQ`, `ModulesSQ`, and `NextModuleSQ`. These functions allow you to interactively select the modules. See 25.45, 25.46, and 25.47 for details.

Note that the ordering inside the lists of l is important. If l is the list $[[2,0,3,0]]$ then `SolvableQuotient` will first try a module over GF(2) and attempt to construct an extension by a module over GF(3) only if the GF(2) extension fails, whereas in the case that l is the list $[[3,0,2,0]]$ the function first attempts to extend with modules over GF(3) and then with modules over GF(2).

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>        f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>        f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>        f.2^-1*f.4^-1*f.2*f.4 ];;
gap> g := SolvableQuotient( f4, [[5,0],[2,0,3,0]] );
Group( e1, e2 )
gap> Size(AgGroupFpGroup(g));
4
gap> g := SolvableQuotient( f4, [[3,0],[2,0]] );
Group( e1, e2, m3, m4, m5, m6, m7, m8, m9 )
gap> Size(AgGroupFpGroup(g));
1152
```

25.45 InitSQ

`InitSQ(` G `)`

Let G be a finitely presented group. `InitSQ` computes an SQ record for the commutator factor group of G. This record can be used to investigate finite solvable quotients of G .

Note that the commutator factor group of G must be finite otherwise an error message is printed.

See also 25.46 and 25.47.

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>        f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>        f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>        f.2^-1*f.4^-1*f.2*f.4 ];;
gap> s := InitSQ(f4);
<< solvable quotient of size 2^2 >>
```

25.46 ModulesSQ

`ModulesSQ(` S, F `)`
`ModulesSQ(` S, F, d `)`

Let S be an SQ record describing a finite solvable quotient Q of a finitely presented group G. `ModulesSQ` computes all irreducible representations of Q over the prime field F of dimension at most d. If d is zero or missing no restriction on the dimension is enforced.

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>       f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>       f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>       f.2^-1*f.4^-1*f.2*f.4 ];;
gap> s := InitSQ(f4);
<< solvable quotient of size 2^2 >>
gap> ModulesSQ( s, GF(2) );;
```

25.47 NextModuleSQ

`NextModuleSQ(` s, M `)`

Let S be an SQ record describing a finite solvable quotient Q of a finitely presented group G. `NextModuleSQ` tries to extend Q by the module M such that the extension is still a quotient of G

```
gap> f := FreeGroup( "a", "b", "c", "d" );;
gap> f4 := f / [ f.1^2, f.2^2, f.3^2, f.4^2, f.1*f.2*f.1*f.2*f.1*f.2,
>       f.2*f.3*f.2*f.3*f.2*f.3*f.2*f.3, f.3*f.4*f.3*f.4*f.3*f.4,
>       f.1^-1*f.3^-1*f.1*f.3, f.1^-1*f.4^-1*f.1*f.4,
>       f.2^-1*f.4^-1*f.2*f.4 ];;
gap> s := InitSQ(f4);
<< solvable quotient of size 2^2 >>
gap> m := ModulesSQ( s, GF(3) );;
gap> NextModuleSQ( s, m[1] );
<< solvable quotient of size 2^2 >>
gap> NextModuleSQ( s, m[2] );
<< solvable quotient of size 2^2*3 >>
gap> NextModuleSQ( s, m[3] );
<< solvable quotient of size 2^2 >>
gap> NextModuleSQ( s, m[4] );
<< solvable quotient of size 2^2*3 >>
```

25.48 Generating Systems of Ag Groups

For an ag group G there exists three different types of generating systems. The generating system in G`.generators` is a list of ag words generating the group G with the only condition that none of the ag words is the identity of G. If an induced generating system for G is known it is bound to G`.igs`, while an canonical generating system is bound to G`.cgs`. But as every canonical generating system is also an induced one, G`.cgs` and G`.igs` may contain the same system.

The functions `Cgs`, `Igs`, `Normalize`, `Normalized` and `IsNormalized` change or manipulate these systems. The following overview shows when to use this functions. For details see 25.50, 25.51, 25.53, 25.54 and 25.52.

`Igs` returns an induced generating system for G. If neither G`.igs` nor G`.cgs` are present, it uses `MergedIgs` (see 25.56) in order to construct an induced generating system from G`.generators`. In that case the induced generating system is bound to G`.igs`. If G`.cgs` but not G`.igs` is present, this is returned, as every canonical generating system is also an induced one. If G`.igs` is present this is returned.

`Cgs` returns a canonical generating system for G. If neither G`.igs` nor G`.cgs` are present, it uses `MergedCgs` (see 25.55) in order to construct a canonical generating system from G`.generators`. In that case the canonical generating system is bound to G`.cgs`. If G`.igs` but not G`.cgs` is present, this is normalized and bound to G`.cgs`, but G`.igs` is left unchanged. If G`.cgs` is present this is returned.

`Normalize` computes a canonical generating system, binds it to G`.cgs` and unbinds an induced generating bound to G`.igs`. `Normalized` normalizes a copy without changing the original ag group. This function should be preferred.

`IsNormalized` checks if an induced generating system is a canonical one and, if being canonical, binds it to G`.cgs` and unbinds G`.igs`. If G`.igs` is unbound `IsNormalized` computes a canonical generating system, binds it to G`.cgs` and returns `true`.

Most functions need an induced or canonical generating system, all function descriptions state clearly what is used, if this is relevant, see 25.73 for example.

25.49 AgSubgroup

`AgSubgroup(` U, *gens*, *flag* `)`

Let U be an ag group with ag group G, *gens* be an induct or canonical generating system for a subgroup S of U and *flag* a boolean. Then `AgSubgroup` returns the record of an ag group representing this finite polycyclic group S as subgroup of G.

If *flag* is `true`, *gens* must be a canonical generating with respect to G. If *flag* is `false` *gens* must be a an induced generating with respect to G.

gens will be bound to S`.generators`. If *flag* is `true`, it is also bound to S`.cgs`, if it is `false`, *gens* is also bound to S`.igs`. Note that `AgSubgroup` does not copy *gens*.

Note that it is not check whether *gens* are an induced or canonical system. If *gens* fails to be one, all following computations with this group are most probably wrong.

```
gap> v4 := AgSubgroup( s4, [ c, d ], true );
Subgroup( s4, [ c, d ] )
```

25.50 Cgs

`Cgs(` U `)`

`Cgs` returns a canonical generating system of U with respect to the parent group of U as list of ag words (see 25.1).

If U`.cgs` is bound, this is returned without any further action. If U`.igs` is bound, a copy of this component is normalized, bound to U`.cgs` and returned. If neither U`.igs` nor U`.cgs` are bound, a canonical generating system for U is computed using `MergedCgs` (see 25.55) and bound to U`.cgs`.

25.51 Igs

`Igs(` *U* `)`

`Igs` returns an induced generating system of *U* with respect to the parent group of *U* as list of ag words (see 25.1).

If *U*`.igs` is bound, this is returned without any further action. If *U*`.cgs` but not *U*`.igs` is bound, this is returned. If neither *U*`.igs` nor *U*`.cgs` are bound, an induced generating system for *U* is computed using `MergedIgs` (see 25.56) and bound to *U*`.igs`.

25.52 IsNormalized

`IsNormalized(` *U* `)`

`IsNormalized` returns `true` if no induced generating system but an canonical generating system for *U* is known.

If *U*`.cgs` but not *U*`.igs` is bound, `true` is returned. If neither *U*`.cgs` nor *U*`.igs` are bound, a canonical generating system is computed, bound to *U*`.cgs` and `true` is retuned. If *U*`.igs` is present, it is check, if *U*`.igs` is a canonical generating. If so, the canonical generating system is bound to *U*`.cgs` and *U*`.igs` is unbound.

25.53 Normalize

`Normalize(` *U* `)`

`Normalize` converts an induced generating system of an ag group *U* into a canonical one.

If *U*`.cgs` and not *U*`.igs` is bound, *U* is returned without any further action. If *U* contains both components *U*`.cgs` and *U*`.igs`, *U*`.igs` is unbound. If only *U*`.igs` but not *U*`.cgs` is bound the generators in *U*`.igs` are converted into a canonical generating and bound to *U*`.cgs`, while *U*`.igs` is unbound. If neither *U*`.igs` nor *U*`.cgs` are bound a canonical generating system is computed using `Cgs` (see 25.50).

25.54 Normalized

`Normalized(` *U* `)`

`Normalized` returns a normalized copy of an ag group *U*. For details see 25.53.

Note that this function does not alter the record of *U* and always returns a copy of *U*, even if *U* is already normalized.

25.55 MergedCgs

`MergedCgs(` *U*`,` *objs* `)`

Let *U* be an ag group with parent group *G* and *objs* be a list of elements and subgroups of *U*. Then `MergedCgs` returns the subgroup *S* of *G* generated by the elements and subgroups in the list *objs*. The subgroup *S* contains a canonical generating system bound to *S*`.cgs`.

As *objs* contains only elements and subgroups of *U*, the subgroup *S* is not only a subgroup of *G* but also of *U*. Its parent group is nevertheless *G* and `MergedCgs` computes a canonical generating system of *S* with respect to *G*.

If subgroups of S are known at least the largest should be an element of *objs*, because `MergedCgs` is much faster in such cases.

Note that this function may return a wrong subgroup, if the elements of *objs* do not belong to U. See also 25.48 for differences between canonical and induced generating systems.

```
gap> d8 := MergedCgs( s4, [ a*c, c ] );
Subgroup( s4, [ a, c, d ] )
gap> MergedCgs( s4, [ a*b*c*d, d8 ] );
s4
gap> v4 := MergedCgs( d8, [ c*d, c ] );
Subgroup( s4, [ c, d ] )
```

25.56 MergedIgs

`MergedIgs(` U`,` S`,` *gens*`,` *normalize* `)`

Let U and S be ag groups with a common parent group G such that S is a subgroup of U. Let *gens* be a list of elements of U. Then `MergedIgs` returns the subgroup K of G generated by S and *gens*.

As *gens* contains only elements of U, the subgroup K is not only a subgroup of G but also of U. Its parent group is nevertheless G and `MergedIgs` computes a induced generating system of S with respect to G.

If *normalize* is `true`, a canonical generating system for K is computed and bound to K`.cgs`. If *normalize* is `false` only an induced generating system is computed and bound to K`.igs` or K`.cgs`. If no subgroup S is known, `rec()` can be given instead.

Note that U must be an ag group which contains S and *gens*.

25.57 Factor Groups of Ag Groups

It is possible to deal with factor groups of ag groups in three different ways. If an ag group G and a normal subgroup N of G is given, you can construct a new polycyclic presentation for $F = G/N$ using `FactorGroup`. You can apply all functions for ag groups to that new parent group F and even switch between G and F using the homomorphisms returned by `NaturalHomomorphism`. See 7.33 for more information on that kind of factor groups.

But if you are only interested in an easy way to test a property or an easy way to calculate a subgroup of a factor group, the first approach might be too slow, as it involves the construction of a new polycyclic presentation for the factor group together with the creation of a new collector for that factor group. In that case you can use `CollectorlessFactorGroup` in order to construct a new ag group without initializing a new collector but using records faking ag words instead. But now multiplication is still done in G and the words must be canonicalized with respect to N, so that multiplication in this group is rather slow. However if you for instance want to check if a chief factor, which is not part of the AG series, is central this may be faster then constructing a new collector. But generally `FactorGroup` should be used.

The third possibility works only for `Exponents` (see 25.73) and `SiftedAgWord` (see 24.12). If you want to compute the action of G on a factor M/N then you can pass M/N as factor group argument using M `mod` N or `FactorArg` (see 25.60).

25.58 FactorGroup for AgGroups

`AgGroupOps.FactorGroup(` *U*, *N* `)`

Let N and U be ag groups with a common parent group, such that N is a normal subgroup of U. Let H be the factor group U/N. `FactorGroup` returns the finite polycyclic group H as new parent group.

If the ag group U is not a parent group or if N is not an element of the AG series of U (see 25.70), then `FactorGroup` constructs a new polycyclic presentation and collector for the factor group using both `FpGroup` (see 25.23) and `AgGroupFpGroup` (see 25.27). Otherwise `FactorGroup` copies the old collector of U and cuts of the tails which lie in N.

Note that N must be a normal subgroup of U. You should keep in mind, that although the new generators and the old ones may have the same names, they cannot be multiplitied as they are elements of different groups. The only way to transfer information back and forth is to use the homomorphisms returned by `NaturalHomomorphism` (see 7.33).

```
gap> c2 := Subgroup( s4, [ d ] );
Subgroup( s4, [ d ] )
gap> d8 := Subgroup( s4, [ a, c, d ] );
Subgroup( s4, [ a, c, d ] )
gap> v4 := FactorGroup( d8, c2 );
Group( g1, g2 )
gap> v4.2 ^ v4.1;
g2
gap> d8 := Subgroup( s4, [ a, c, d ] );
Subgroup( s4, [ a, c, d ] )
gap> d8.2^d8.1;
c*d
```

25.59 CollectorlessFactorGroup

`CollectorlessFactorgroup(` *G*, *N* `)`

`CollectorlessFactorgroup` constructs the factorgroup $F = G/N$ without initializing a new collector. The elements of F are records faking ag words.

Each element f of F contains the following components.

`representative`
: a canonical representative d in G for f.

`isFactorGroupElement` contains `true`.

`info`
: a record containing information about the factor group.

`operations`
: the operations record `FactorGroupAgWordsOps`.

25.60 FactorArg

`FactorArg(` *U*, *N* `)`

Let N be a normal subgroup of an ag group U. Then `FactorArg` returns a record with the following components with can be used as argument for `Exponents`.

`isFactorArg`
: is `true`.

`factorNum`
: contains U.

`factorDen`
: contains N.

`identity`
: contains the identity of U.

`generators`
: contains a list of those induced generators u_i of U of depth d_i such that no induced generator in N has depth d_i.

`operations`
: contains the operations record `FactorArgOps`.

Note that `FactorArg` is bound to `AgGroupOps.mod`.

```
gap> d8 := Subgroup( s4, [ a, c, d ] );
Subgroup( s4, [ a, c, d ] )
gap> c2 := Subgroup( s4, [ d ] );
Subgroup( s4, [ d ] )
gap> M := d8 mod c2;;
gap> d8.1 * d8.2 * d8.3;
a*c*d
gap> Exponents( M, last );
[ 1, 1 ]
gap> d8 := AgSubgroup( s4, [ a*c, c, d ], false );
Subgroup( s4, [ a*c, c, d ] )
gap> M := d8 mod c2;;
gap> Exponents( M, a*c*d );
[ 1, 0 ]
```

25.61 Subgroups and Properties of Ag Groups

The subgroup functions compute subgroups or series of subgroups from given ag groups, e.g. `PRump` (see 25.64) or `ElementaryAbelianSeries` (see 7.39). They return group records described in 7.118 and 25.4 for the computed subgroups.

All the following functions only work for ag groups. Subgroup functions which work for various types of groups are described in 7.14. Properties and property tests which work for various types of groups are described in 7.45.

25.62 CompositionSubgroup

`CompositionSubgroup(` G, i `)`

`CompositionSubgroup` returns the i.th subgroup of the AG series of an ag group G.

Let $(g_1, ..., g_n)$ be an induced generating system of G with respect to the parent group of G. Then the i.th composition subgroup S of the AG series is generated by $(g_i, ..., g_n)$.

```
gap> CompositionSubgroup( s4, 2 );
Subgroup( s4, [ b, c, d ] )
gap> CompositionSubgroup( s4, 4 );
Subgroup( s4, [ d ] )
gap> CompositionSubgroup( s4, 5 );
Subgroup( s4, [  ] )
```

25.63 HallSubgroup

`HallSubgroup(` G, n `)`
`HallSubgroup(` G, L `)`

Let G be an ag group. Then `HallSubgroup` returns a π-Hall-subgroup of G for the set π of all prime divisors of the integer n or the join π of all prime divisors of the integers of L.

The Hall-subgroup is constructed using Glasby's algorithm (see [Gla87]), which descends along an elementary abelian series for G and constructs complements in the coprime case (see 25.91). If no such series is known for G the function uses `ElementaryAbelianSeries` (see 7.39) in order to construct such a series for G.

```
gap> HallSubgroup( s4, 2 );
Subgroup( s4, [ a, c, d ] )
gap> HallSubgroup( s4, [ 3 ] );
Subgroup( s4, [ b ] )
gap> z5 := CyclicGroup( AgWords, 5 );
Group( c5 )
gap> DirectProduct( s4, z5 );
Group( a1, a2, a3, a4, b )
gap> HallSubgroup( last, [ 5, 3 ] );
Subgroup( Group( a1, a2, a3, a4, b ), [ a2, b ] )
```

25.64 PRump

`PRump(` G, p `)`

`PRump` returns the p-rump of an ag group G for a prime p.

The **p-rump** of a group G is the normal closure under G of the subgroup generated by the commutators and p.th powers of the generators of G.

```
gap> PRump( s4, 2 );
Subgroup( s4, [ b, c, d ] )
gap> PRump( s4, 3 );
s4
```

25.65 RefinedSubnormalSeries

`RefinedSubnormalSeries(` L `)`

Let L be a list of ag groups $G_1, ..., G_n$, such that G_{i+1} is a normal subgroup of G_i. Then the function computes a composition series $H_1 = G_1, ..., H_m = G_n$ which refines the given subnormal series L and has cyclic factors of prime order (see also 7.43).

```
gap> v4 := Subgroup( s4, [ c, d ] );
Subgroup( s4, [ c, d ] )
gap> T := TrivialSubgroup( s4 );
Subgroup( s4, [  ] )
gap> RefinedSubnormalSeries( [ s4, v4, T ] );
[ s4, Subgroup( s4, [ b, c, d ] ), Subgroup( s4, [ c, d ] ),
  Subgroup( s4, [ d ] ), Subgroup( s4, [  ] ) ]
```

25.66 SylowComplements

`SylowComplements(` U `)`

`SylowComplements` returns a Sylow complement system of U. This system S is represented as a record with at least the components S`.primes` and S`.sylowComplements`, additionally there may be a component S`.sylowSubgroups` (see 25.67).

`primes`
: A list of all prime divisors of the group order of U.

`sylowComplements`
: contains a list of Sylow complements for all primes in S`.primes`, so that if the i.th element of S`.primes` is p, then the i.th element of `sylowComplements` is a Sylow-p-complement of U.

`sylowSubgroups`
: contains a list of Sylow subgroups for all primes in S`.primes`, such that if the i.th element of S`.primes` is p, then the i.th element of S`.sylowSubgroups` is a Sylow-p-subgroup of U.

`SylowComplements` uses `HallSubgroup` (see 25.63) in order to compute the various Sylow complements of U, if no Sylow system is known for U. If a Sylow system $\{S_1, ..., S_n\}$ is known, `SylowComplements` computes the various Hall subgroups H_i using the fact that H_i is the product of all S_j except S_i.

`SylowComplements` sets and checks U`.sylowSystem`.

```
gap> SylowComplements( s4 );
rec(
  primes := [ 2, 3 ],
  sylowComplements :=
   [ Subgroup( s4, [ b ] ), Subgroup( s4, [ a, c, d ] ) ] )
```

25.67 SylowSystem

`SylowSystem(` U `)`

`SylowSystem` returns a Sylow system $\{S_1, ..., S_n\}$ of an ag group U. The system S is represented as a record with at least the components S`.primes` and S`.sylowSubgroups`, additionally there may be a component S`.sylowComplements`, see 25.66 for information about this addtional component.

`primes`
: A list of all prime divisors of the group order of U.

`sylowComplements`
: contains a list of Sylow complements for all primes in S`.primes`, so that if the i.th element of S`.primes` is p, then the i.th element of `sylowComplements` is a Sylow-p-complement of U.

`sylowSubgroups`
: contains a list of Sylow subgroups for all primes in S`.primes`, such that if the i.th element of S`.primes` is p, then the i.th element of S`.sylowSubgroups` is a Sylow-p-subgroup of U.

A **Sylow system** of a group U is a system of Sylow subgroups S_i for each prime divisor of the group order of U such that $S_i * S_j = S_j * S_i$ is fulfilled for each pair i, j.

`SylowSystem` uses `SylowComplements` (see 25.67) in order to compute the various Sylow complements H_i of U. Then the Sylow system is constructed using the fact that the intersection S_i of all Sylow complements H_j except H_i is a Sylow subgroup and that all these subgroups S_i form a Sylow system of U. See [Gla87].

`SylowSystem` sets and checks S`.sylowSystem`.

```
gap> z5 := CyclicGroup( AgWords, 5 );
Group( c5 )
gap> D := DirectProduct( z5, s4 );
Group( a, b1, b2, b3, b4 )
gap> D.name := "z5Xs4";;
gap> SylowSystem( D );
rec(
  primes := [ 2, 3, 5 ],
  sylowComplements :=
   [ Subgroup( z5Xs4, [ a, b2 ] ), Subgroup( z5Xs4, [ a, b1, b3, b4
         ] ), Subgroup( z5Xs4, [ b1, b2, b3, b4 ] ) ],
  sylowSubgroups :=
   [ Subgroup( z5Xs4, [ b1, b3, b4 ] ), Subgroup( z5Xs4, [ b2 ] ),
      Subgroup( z5Xs4, [ a ] ) ] )
```

25.68 SystemNormalizer

`SystemNormalizer(` G `)`

`SystemNormalizer` returns the system normalizer of a Sylow system of the group G.

The **system normalizer** of a Sylow system is the intersection of all normalizers of subgroups in the Sylow system in G.

```
gap> SystemNormalizer( s4 );
Subgroup( s4, [ a ] )
gap> SystemNormalizer( D );
Subgroup( z5Xs4, [ a, b1 ] )
```

25.69 MinimalGeneratingSet

`MinimalGeneratingSet(` *G* `)`

Let G be an ag group. Then `MinimalGeneratingSet` returns a subset L of G of minimal cardinality with the property that L generates G.

```
gap> l := MinimalGeneratingSet(s4);
[ b, a*c*d ]
gap> s4 = Subgroup( s4, l );
true
```

25.70 IsElementAgSeries

`IsElementAgSeries(` *U* `)`

`IsElementAgSeries` returns `true` if the ag group U is part of the AG series of the parent group G of U and `false` otherwise.

25.71 IsPNilpotent

`IsPNilpotent(` *U*, *p* `)`

`IsPNilpotent` returns `true`, if the ag group U is p-nilpotent for the prime p, and `false` otherwise.

`IsPNilpotent` uses Glasby's p-nilpotency test (see [Gla87]).

```
gap> IsPNilpotent( s4, 2 );
false
gap> s3 := Subgroup( s4, [ a, b ] );
Subgroup( s4, [ a, b ] )
gap> IsPNilpotent( s3, 2 );
true
gap> IsPNilpotent( s3, 3 );
false
```

25.72 NumberConjugacyClasses

`NumberConjugacyClasses(` *H* `)`

This functions computes the number of conjugacy classes of elements of a group H.

The function uses an algorithm that steps down an elementary abelian series of the parent group of H and computes the number of conjugacy classes using the same method as `AgGroupOps.ConjugacyClasses` does, up to the last factor group. In the last step the Cauchy-Frobenius-Burnside lemma is used.

This algorithm is especially designed to supply at least the number of conjugacy classes of H, whenever `ConjugacyClasses` fails because of storage reasons. So one would rather use this function if the last normal subgroup of the elementary abelian series is too big to be dealt with `ConjugacyClasses`.

`NumberConjugacyClasses(` *U*, *H* `)`

This version of the call to `NumberConjugacyClasses` computes the number of conjugacy classes of H under the operation of U. Thus for the operation to be well defined, U must be a subgroup of the normalizer of H in their common parent group.

```
gap> a4 := DerivedSubgroup(s4);;
gap> NumberConjugacyClasses( s4 );
5
gap> NumberConjugacyClasses( a4, s4 );
6
gap> NumberConjugacyClasses( a4 );
4
gap> NumberConjugacyClasses( s4, a4 );
3
```

25.73 Exponents

`Exponents(` *U*`,` *u* `)`
`Exponents(` *U*`,` *u*`,` *F* `)`

`Exponents` returns the exponent vector of an ag word u with respect to an induced generating system of U as list of integers if no field F is given. Otherwise the product of the exponent vector and F`.one` is returned. Note that u must be an element of U.

Let $(u_1, ..., u_r)$ be an induced generating system of U. Then u can be uniquely written as $u_1^{\nu_1} * ... * u_r^{\nu_r}$ for integer ν_i. The **exponent vector** of u is $[\nu_1, ..., \nu_r]$.

`Exponents` allows factor group arguments. See 25.57 for details.

Note that `Exponents` adds a record component U`.shiftInfo`. This entry is used by subsequent calls with the same ag group in order to speed up computation. If you ever change the component U`.igs` by hand, not using `Normalize`, you must unbind the component U`.shiftInfo`, otherwise all following results of `Exponents` will be corrupted. In case U is a parent group you can use `ExponentsAgWord` (see 24.15), which is slightly faster but requires a parent group U.

Note that you you may get a weird error message if u is no element of U. So it is strictly required that u is an element of U.

Note that `Exponents` uses `ExponentsAgWord` but not `ExponentAgWord`, so for records that mimic agwords `Exponents` may be used in `ExponentAgWord`.

```
gap> v4 := AgSubgroup( s4, [ c, d ], true );
Subgroup( s4, [ c, d ] )
gap> Exponents( v4, c * d );
[ 1, 1 ]
gap> Exponents( s4 mod v4, a * b^2 * c * d );
[ 1, 2 ]
```

25.74 FactorsAgGroup

`FactorsAgGroup(` *U* `)`

`FactorsAgGroup` returns the factorization of the group order of an ag group U as list of positive integers.

Note that it is faster to use `FactorsAgGroup` than to use `Factors` and `Size`.

```
gap> v4 := Subgroup( s4, [ c, d ] );;
gap> FactorsAgGroup( s4 );
[ 2, 2, 2, 3 ]
gap> Factors( Size( s4 ) );
[ 2, 2, 2, 3 ]
```

25.75 MaximalElement

`MaximalElement(` U `)`

`MaximalElement` returns the ag word in U with maximal exponent vector.

Let G be the parent group of U with canonical generating system $(g_1, ..., g_n)$ and let $(u_1, ..., u_m)$ be the canonical generating system of U, d_i is the depth of u_i with respect to G. Then an ag word $u = g_1^{\nu_1} * ... * g_n^{\nu_n} \in U$ is returned such that $\sum_{i=1}^{m} \nu_{d_i}$ is maximal.

25.76 Orbitalgorithms of Ag Groups

The functions `Orbit` (see 8.16) and `Stabilizer` (see 8.24 and 25.17) compute the orbit and stabilizer of an ag group acting on a domain.

`AgOrbitStabilizer` (see 25.78) computes the orbit and stabilizer in case that a compatible homomorphism into a group H exists, such that the action of H on the domain is more efficient than the operation of the ag group; for example, if an ag group acts linearly on a vector space, the operation can by described using matrices.

The functions `AffineOperation` (see 25.77) and `LinearOperation` (see 25.79) compute matrix groups describing the affine or linear action of an ag group on a vector space.

25.77 AffineOperation

`AffineOperation(` U `,` V `,` φ `,` τ `)`

Let U be an ag group with an induced generating system $u_1, ..., u_m$ and let V be a vector space with base $(o_1, ..., o_n)$. Further U should act affinely on V. So if v is an element of V and u is an element of U, then $v^u = v_u + x_u$, such that the function which maps v to v_u is linear and x_u is an element of V. These actions are given by the functions φ and τ as follows. $\varphi(v, u)$ must return the representation of v_u with respect to the base $(o_1, ..., o_n)$ as sequence of finite field elements. $\tau(u)$ must return the representation of x_u in the base $(o_1, ..., o_n)$ as sequence of finite field elements. If these conditions are fulfilled, `AffineOperation` returns a matrix group M describing this action.

Note that M`.images` contains a list of matrices m_i, such that m_i describes the action of u_i and m_i is of the form

$$\begin{pmatrix} L_{u_i} & 0 \\ x_{u_i} & 1 \end{pmatrix},$$

where L_u is the matrix which describes the linear operation $v \in V \mapsto v_u$.

```
gap> v4 := AgSubgroup( s4, [ c, d ], true );
```

```
Subgroup( s4, [ c, d ] )
gap> v4.field := GF( 2 );
GF(2)
gap> phi := function( v, g )
>      return Exponents( v4, v^g, v4.field );
>    end;
function ( v, g ) ... end
gap> tau := g -> Exponents( v4, v4.identity, v4.field );
function ( g ) ... end
gap> V := rec( base := [ c, d ], isDomain := true );
rec(
  base := [ c, d ],
  isDomain := true )
gap> AffineOperation( s4, V, phi, tau );
Group( [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), Z(2)^0 ] ],
[ [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0, 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), Z(2)^0 ] ] )
```

25.78 AgOrbitStabilizer

`AgOrbitStabilizer(` U, *gens*, ω `)`
`AgOrbitStabilizer(` U, *gens*, ω, f `)`

Let U be an ag group acting on a set Ω. Let ω be an element of Ω. Then `AgOrbitStabilizer` returns the point-stabilizer of ω in the group U and the orbit of ω under this group. The stabilizer and orbit are returned as record R with components R`.stabilizer` and R`.orbit`. R`.stabilizer` is the point-stabilizer of ω. R`.orbit` is the list of the elements of ω^U.

Let $(u_1, ..., u_n)$ be an induced generating system of U and *gens* be a list $h_1, ..., h_n$ of generators of a group H, such that the map $u_i \mapsto h_i$ extends to an homomorphism α from U to H, which is compatible with the action of G and H on Ω, such that $g \in Stab_U(\omega)$ if and only if $g^\alpha \in Stab_H(\omega)$. If f is missing `OnRight` is assumed, a typical application of this function being the linear action of U on an vector space. If f is `OnPoints` then `^` is used as operation of H on Ω. Otherwise f must be a function, which takes two arguments, the first one must be a point p of Ω and the second an element h of H and which returns p^h.

```
gap> AgOrbitStabilizer( s4, [a,b,c,d], d, OnPoints );
rec(
  stabilizer := Subgroup( s4, [ a, c, d ] ),
  orbit := [ d, c*d, c ] )
```

25.79 LinearOperation

`LinearOperation(` U, V, φ `)`

Let U be an ag group with an induced generating system $u_1, ..., u_m$ and V a vector space with base $(o_1, ..., o_n)$. U must act linearly on V. Let v be an element of V, u be an element of U. The action of U on V should be given as follows. If $v^u = a_1 * o_1 + ... + a_n * o_n$, then the function $\varphi(v, u)$ must return $(a_1, ..., a_n)$ as list of finite field elements. If these condition are fulfilled, `LinearOperation` returns a matrix group M describing this action.

Note that M.images is bound to a list of matrices m_i, such that m_i describes the action of u_i.

```
gap> v4 := AgSubgroup( s4, [ c, d ], true );
Subgroup( s4, [ c, d ] )
gap> v4.field := GF( 2 );
GF(2)
gap> V := rec( base := [ c, d ], isDomain := true );
rec(
  base := [ c, d ],
  isDomain := true )
gap> phi := function( v, g )
>      return Exponents( v4, v^g, v4.field );
>    end;
function ( v, g ) ... end
gap> LinearOperation( s4, V, phi );
Group( [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ] )
```

25.80 Intersections of Ag Groups

There are two kind of intersection algorithms. Whenever the product of two subgroups is a subgroup, a generalized Zassenhaus algorithm can be used in order to compute the intersection and sum (see [GS90]). In case one subgroup is a normalized by the other, an element of the sum can easyly be decomposed. The functions `IntersectionSumAgGroup` (see 25.82), `NormalIntersection` (see 7.26), `SumFactorizationFunctionAgGroup` (see 25.84) and `SumAgGroup` (see 25.83) should be used in such cases.

These functions are faster than the general function `Intersection` (see 4.12 and 25.7), which can compute the intersection of two subgroups even if their product is no subgroup.

25.81 ExtendedIntersectionSumAgGroup

`ExtendedIntersectionSumAgGroup(` V`,` W `)`

Let V and W be ag groups with a common parent group, such that $W \leq N(V)$. Then $V * W$ is a subgroup and `ExtendedIntersectionSumAgGroup` returns the intersection and the sum of V and W. The information about these groups is returned as a record with the components `intersection`, `sum` and the additional information `leftSide` and `rightSide`.

`intersection`
: is bound to the intersection $W \cap V$.

`sum`
: is bound to the sum $V * W$.

`leftSide`
: is lists of ag words, see below.

`rightSide`
: is lists of agwords, see below.

The function uses the Zassenhaus sum-intersection algorithm. Let V be generated by $v_1, ..., v_a$, W be generated by $w_1, ..., w_b$. Then the matrix

$$\begin{pmatrix} v_1 & 1 \\ \vdots & \vdots \\ v_a & 1 \\ w_1 & w_1 \\ \vdots & \vdots \\ w_b & w_b \end{pmatrix}$$

is echelonized by using the sifting algorithm to produce the following matrix

$$\begin{pmatrix} l_1 & k_1 \\ \vdots & \vdots \\ l_c & k_c \\ 1 & k_{c+1} \\ \vdots & \vdots \\ 1 & k_{a+b} \end{pmatrix}.$$

Then $l_1, ..., l_c$ is a generating sequence for the sum, while the sequence $k_{c+1}, ..., k_{a+b}$ is is a generating sequence for the intersection. `leftSide` is bound to a list, such that the i.th list element is l_j, if there exists a j, such that l_j has depth i, and `IdAgWord` otherwise. `rightSide` is bound to a list, such that the i.th list element is k_j, if there exists a j less than $c + 1$, such that k_j has depth i, and `IdAgWord` otherwise. See also 25.84.

Note that this functions returns an incorrect result if $W \not\leq N(V)$.

```
gap> v4_1 := AgSubgroup( s4, [ a*b, c ], true );
Subgroup( s4, [ a*b, c ] )
gap> v4_2 := AgSubgroup( s4, [ c, d ], true );
Subgroup( s4, [ c, d ] )
gap> ExtendedIntersectionSumAgGroup( v4_1, v4_2 );
rec(
  leftSide := [ a*b, IdAgWord, c, d ],
  rightSide := [ IdAgWord, IdAgWord, c, d ],
  sum := Subgroup( s4, [ a*b, c, d ] ),
  intersection := Subgroup( s4, [ c ] ) )
```

25.82 IntersectionSumAgGroup

`IntersectionSumAgGroup(` V`,` W `)`

Let V and W be ag groups with a common parent group, such that $W \leq N(V)$. Then $V * W$ is a subgroup and `IntersectionSumAgGroup` returns the intersection and the sum of V and W as record R with components R`.intersection` and R`.sum`.

The function uses the Zassenhaus sum-intersection algorithm. See also 7.26 and 25.83. For more information about the Zassenhaus algorithm see 25.81 and 25.84.

Note that this functions returns an incorrect result if $W \not\leq N(V)$.

```
gap> d8_1 := AgSubgroup( s4, [ a, c, d ], true );
Subgroup( s4, [ a, c, d ] )
gap> d8_2 := AgSubgroup( s4, [ a*b, c, d ], true );
Subgroup( s4, [ a*b, c, d ] )
gap> IntersectionSumAgGroup( d8_1, d8_2 );
rec(
  sum := Group( a*b, b^2, c, d ),
  intersection := Subgroup( s4, [ c, d ] ) )
```

25.83 SumAgGroup

`SumAgGroup(` V, W `)`

Let V and W be ag groups with a common parent group, such that $W \leq N(V)$. Then $V * W$ is a subgroup and `SumAgGroup` returns $V * W$.

The function uses the Zassenhaus sum-intersection algorithm (see [GS90]).

Note that this functions returns an incorrect result if $W \not\leq N(V)$.

```
gap> d8_1 := Subgroup( s4, [ a, c, d ] );
Subgroup( s4, [ a, c, d ] )
gap> d8_2 := Subgroup( s4, [ a*b, c, d ] );
Subgroup( s4, [ a*b, c, d ] )
gap> SumAgGroup( d8_1, d8_2 );
Group( a*b, b^2, c, d )
```

25.84 SumFactorizationFunctionAgGroup

`SumFactorizationFunctionAgGroup(` U, N `)`

Let U and N be ag group with a common parent group such that U normalizes N. Then the function returns a record R with the following components.

`intersection`
: is bound to the intersection $U \cap N$.

`sum`
: is bound to the sum $U * N$.

`factorization`
: is bound to function, which takes an element g of $U * N$ and returns the factorization of g in an element u of U and n of N, such that $g = u{*}n$. This factorization is returned as record r with components r`.u` and r`.n`, where r`.u` is bound to the ag word u, r`.n` to the ag word n.

Note that N must be a normal subgroup of $U * N$, it is not sufficient that $U * N = N * U$.

```
gap> v4 := AgSubgroup( s4, [ a*b, c ], true );
Subgroup( s4, [ a*b, c ] )
gap> a4 := AgSubgroup( s4, [ b, c, d ], true );
Subgroup( s4, [ b, c, d ] )
gap> sd := SumFactorizationFunctionAgGroup;
function ( U, N ) ... end
```

```
gap> sd := SumFactorizationFunctionAgGroup( v4, a4 );
rec(
  sum := Group( a*b, b, c, d ),
  intersection := Subgroup( s4, [ c ] ),
  factorization := function ( un ) ... end )
gap> sd.factorization( a*b*c*d );
rec(
  u := a*b*c,
  n := d )
gap> sd.factorization( a*b^2*c*d );
rec(
  u := a*b*c,
  n := b*c )
```

25.85 One Cohomology Group

Let G be a finite group, M a normal p-elementary abelian subgroup of G. Then the group of one coboundaries $B^1(G/M, M)$ is defined as

$$B^1(G/M, M) = \{\gamma : G/M \to M\ ;\ \exists m \in M \forall g \in G : \gamma(gM) = (m^{-1})^g \cdot m\}$$

and is a Z_p-vector space. The group of cocycles $Z^1(G/M, M)$ is defined as

$$Z^1(G/M, M) = \{\gamma : G/M \to M\ ;\ \forall g_1, g_2 \in G : \gamma(g_1M \cdot g_2M) = \gamma(g_1M)^{g_2} \cdot \gamma(g_2M)\}$$

and is also a Z_p-vector space.

Let α be the isomorphism of M into a row vector space $\mathcal{W}$ and $(g_1, ..., g_l)$ representatives for a generating set of G/M. Then there exists a monomorphism β of $Z^1(G/M, M)$ in the l-fold direct sum of $\mathcal{W}$, such that $\beta(\gamma) = (\alpha(\gamma(g_1M)), ..., \alpha(\gamma(g_lM)))$ for every $\gamma \in Z^1(G/M, M)$.

`OneCoboundaries` (see 25.86) and `OneCocycles` (see 25.87) compute the group of one coboundaries and one cocyles given a ag group G and a elementary abelian normal subgroup M. If `Info1Coh1`, `Info1Coh2` and `Info1Coh3` are set to `Print` information about the computation is given.

25.86 OneCoboundaries

`OneCoboundaries(` G`,` M `)`

Let M be a normal p-elementary abelian subgroup of G. Then `OneCoboundaries` computes the vector space $\mathcal{V} = \beta(B^1(G/M, M))$, which is isomorphic to the group of one coboundaries $B^1(G, M)$ as described in 25.85. The functions returns a record C with the following components.

`oneCoboundaries`
: contains the vector space $\mathcal{V}$.

`generators`
: contains representatives $(g_1, ..., g_l)$ for the canonical generating system of G/M

`cocycleToList`
: contains a functions which takes an element v of $\mathcal{V}$ as argument and returns a list $[n_1, ..., n_l]$, where n_i is an element of M, such that $n_i = (\beta^{-1}(v))(g_i M)$.

`listToCocycles`
: is the inverse of `cocycleToList`.

`OneCoboundaries(` G, α, M `)`

In that form `OneCoboundaries` computes the one coboundaries in the semidirect product of G and M where G acts on M using α (see 7.101).

```
gap> s4xc2 := DirectProduct( s4, CyclicGroup( AgWords, 2 ) );
Group( a1, a2, a3, a4, b )
gap> m := CompositionSubgroup( s4xc2, 3 );
Subgroup( Group( a1, a2, a3, a4, b ), [ a3, a4, b ] )
gap> oc := OneCoboundaries( s4xc2, m );
rec(
  oneCoboundaries := RowSpace( GF(2),
    [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ] ),
  generators := [ a1, a2 ],
  cocycleToList := function ( c ) ... end,
  listToCocycle := function ( L ) ... end )
gap> v := Base( oc.oneCoboundaries );
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> oc.cocycleToList( v[1] );
[ a4, a4 ]
gap> oc.cocycleToList( v[2] );
[ IdAgWord, a3 ]
gap> oc.cocycleToList( v[1]+v[2] );
[ a4, a3*a4 ]
```

25.87 OneCocycles

`OneCocycles(` G, M `)`

Let M be a normal p-elementary abelian subgroup of G. Then `OneCocycles` computes the vector space $\mathcal{V} = \beta(Z^1(G/M, M))$, which is isomorphic to the group of one cocyles $Z^1(G, M)$ as described in 25.85. The function returns a record C with the following components.

`oneCoboundaries`
: contains the vector space isomorphic to $B^1(G, M)$.

`oneCocycles`
: contains the vector space $\mathcal{V}$.

`generators`
: contains representatives $(g_1, ..., g_l)$ for the canonical generating system of G/M

`isSplitExtension`
: If G splits over M, C`.isSplitExtension` is `true`, otherwise it is `false`. In case of

a split extension three more components C.`complement`, C.`cocycleToComplement` and C.`complementToCycles` are returned.

`complement`
: contains a subgroup of G which is a complement of M.

`cocycleToList`
: contains a functions which takes an element v of $\mathcal{V}$ as argument and returns a list $[n_1, ..., n_l]$, where n_i is an element of M, such that $n_i = (\beta^{-1}(v))(g_i M)$.

`listToCocycles`
: is the inverse of `cocycleToList`.

`cocycleToComplement`
: contains a function which takes an element of $\mathcal{V}$ as argument and returns a complement of M.

`complementToCocycle`
: is its inverse. This is possible, because in a split extension there is a one to one correspondence between the elements of $\mathcal{V}$ and the complements of M.

`OneCocycles(` G, α, M `)`

In that form `OneCocycles` computes the one cocycles in the semidirect product of G and M where G acts on M using α (see 7.101). In that case C only contains C.`oneCoboundaries`, C.`oneCocycles`, C.`generators`, C.`cocycleToList` and C.`listToCocycle`.

```
gap> s4xc2 := DirectProduct( s4, CyclicGroup( AgWords, 2 ) );;
gap> s4xc2.name := "s4xc2";;
gap> m := CompositionSubgroup( s4xc2, 3 );
Subgroup( s4xc2, [ a3, a4, b ] )
gap> oc := OneCocycles( s4xc2, m );;
gap> oc.oneCocycles;
RowSpace( GF(2), [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ] )
gap> v := Base( oc.oneCocycles );
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> oc.cocycleToList( v[1] );
[ a4, a4 ]
gap> oc.cocycleToList( v[2] );
[ b, IdAgWord ]
gap> oc.cocycleToList( v[2] );
[ b, IdAgWord ]
gap> oc.cocycleToList( v[3] );
[ IdAgWord, a3 ]
gap> Igs( oc.complement );
[ a1, a2 ]
gap> Igs( oc.cocycleToComplement( v[1]+v[2]+v[3] ) );
[ a1*a4*b, a2*a3*a4 ]
gap> z4 := CyclicGroup( AgWords, 4 );
```

```
Group( c4_1, c4_2 )
gap> m := CompositionSubgroup( z4, 2 );
Subgroup( Group( c4_1, c4_2 ), [ c4_2 ] )
gap> OneCocycles( z4, m );
rec(
  oneCoboundaries := RowSpace( GF(2), [ [ 0*Z(2) ] ] ),
  oneCocycles := RowSpace( GF(2), [ [ Z(2)^0 ] ] ),
  generators := [ c4_1 ],
  isSplitExtension := false )
```

25.88 Complements

`Complement` (see 25.89) tries to find one complement to a given normal subgroup, while `Complementclasses` (see 25.90) finds all complements and returns representatives for the conjugacy classes of complements in a given ag group.

If `InfoAgCo1` and `InfoAgCo2` are set to `Print` information about the computation is given.

25.89 Complement

`Complement(` U, N `)`

Let N and U be ag group such that N is a normal subgroup of U. `Complement` returns a complement of N in U if the U splits over N. Otherwise `false` is returned.

`Complement` descends along an elementary abelian series of U containing N. See [CNW90] for details.

```
gap> v4 := Subgroup( s4, [ c, d ] );
Subgroup( s4, [ c, d ] )
gap> Complement( s4, v4 );
Subgroup( s4, [ a, b ] )
gap> z4 := CyclicGroup( AgWords, 4 );
Group( c4_1, c4_2 )
gap> z2 := Subgroup( z4, [ z4.2 ] );
Subgroup( Group( c4_1, c4_2 ), [ c4_2 ] )
gap> Complement( z4, z2 );
false
gap> m9 := ElementaryAbelianGroup( AgWords, 9 );
Group( m9_1, m9_2 )
gap> m3 := Subgroup( m9, [ m9.2 ] );
Subgroup( Group( m9_1, m9_2 ), [ m9_2 ] )
gap> Complement( m9, m3 );
Subgroup( Group( m9_1, m9_2 ), [ m9_1 ] )
```

25.90 Complementclasses

`Complementclasses(` U, N `)`

Let U and N be ag groups such that N is a normal subgroup of U. `Complementclasses` returns a list of representatives for the conjugacy classes of complements of N in U.

Note that the empty list is returned if U does not split over N.

`Complementclasses` descends along an elementary abelian series of U containing N. See [CNW90] for details.

```
gap> v4 := Subgroup( s4, [ c, d ] );
Subgroup( s4, [ c, d ] )
gap> Complementclasses( s4, v4 );
[ Subgroup( s4, [ a, b ] ) ]
gap> z4 := CyclicGroup( AgWords, 4 );
Group( c4_1, c4_2 )
gap> z2 := Subgroup( z4, [ z4.2 ] );
Subgroup( Group( c4_1, c4_2 ), [ c4_2 ] )
gap> Complementclasses( z4, z2 );
[  ]
gap> m9 := ElementaryAbelianGroup( AgWords, 9 );
Group( m9_1, m9_2 )
gap> m3 := Subgroup( m9, [ m9.2 ] );
Subgroup( Group( m9_1, m9_2 ), [ m9_2 ] )
gap> Complementclasses( m9, m3 );
[ Subgroup( Group( m9_1, m9_2 ), [ m9_1 ] ),
  Subgroup( Group( m9_1, m9_2 ), [ m9_1*m9_2 ] ),
  Subgroup( Group( m9_1, m9_2 ), [ m9_1*m9_2^2 ] ) ]
```

25.91 CoprimeComplement

`CoprimeComplement(` U, N `)`

`CoprimeComplement` returns a complement of a normal p-elementary abelian Hall subgroup N of U.

Note that, as N is a normal Hall-subgroup of U, the theorem of Schur guarantees the existence of a complement.

```
gap> s4xc25 := DirectProduct( s4, CyclicGroup( AgWords, 25 ) );
Group( a1, a2, a3, a4, b1, b2 )
gap> s4xc25.name := "s4xc25";;
gap> a4xc25 := Subgroup( s4xc25,
>                  Sublist( s4xc25.generators, [2..5] ) );
Subgroup( s4xc25, [ a2, a3, a4, b1 ] )
gap> N := Subgroup( s4xc25, [ s4xc25.3, s4xc25.4 ] );
Subgroup( s4xc25, [ a3, a4 ] )
gap> CoprimeComplement( a4xc25, N );
Subgroup( s4xc25, [ a2, b1, b2 ] )
```

25.92 ComplementConjugatingAgWord

`ComplementConjugatingAgWord(` N, U, V `)`
`ComplementConjugatingAgWord(` N, U, V, K `)`

Let N, U, V and K be ag groups with a common parent group G, such that N is p-elementary abelian and normal in G, $U * N = V * N$, $U \cap N = V \cap N = \{1\}$, K is a normal subgroup of UN contained in $U \cap V$ and U is conjugate to V under an element n of N. Then this function returns an element n of N such that $U^n = V$ as ag word. If K is not given, the trivial subgroup is assumed.

In a typical application N is a normal p-elementary abelian subgroup and U, V and K are subgroups such that U/K is a q-group with $q \neq p$.

Note that this function does not check any of the above conditions. So the result may either be `false` or an ag word with does not conjugate U into V, if U and V are not conjugate.

```
gap> c3a := Subgroup( s4, [ b ] );
Subgroup( s4, [ b ] )
gap> c3b := Subgroup( s4, [ b*c ] );
Subgroup( s4, [ b*c ] )
gap> v4 := Subgroup( s4, [ c, d ] );
Subgroup( s4, [ c, d ] )
gap> ComplementConjugatingAgWord( v4, c3a, c3b );
d
gap> c3a ^ d;
Subgroup( s4, [ b*c ] )
```

25.93 HallConjugatingWordAgGroup

`HallConjugatingAgWord(` S, H, K `)`

Let H, K and S be ag group with a common parent group such that H and K are Hall-subgroups of S, then `HallConjugatingAgWord` returns an element g of S as ag word, such that $H^g = K$.

```
gap> d8 := HallSubgroup( s4, 2 );
Subgroup( s4, [ a, c, d ] )
gap> d8 ^ b;
Subgroup( s4, [ a*b^2, c*d, d ] )
gap> HallConjugatingAgWord( s4, d8, d8 ^ b );
b
gap> HallConjugatingAgWord( s4, d8 ^ b, d8 );
b^2
```

25.94 Example, normal closure

We will now show you how to write a GAP3 function, which computes the normal closure of an ag group. Such a function already exists in the library (see 7.25), but this should be an example on how to put functions together. You should at least be familiar with the basic definitions and functions for ag groups, so please refer to 24, 25 and 25.1 for the definitions of finite polycyclic groups and its subgroups, see 25.48 for information about calculating induced or canonical generating system for subgroups.

Let U and S be subgroups of a group G. Then the **normal closure** N of U under S is the smallest subgroup in G, which contains U and is invariant under conjugation with elements of S. It is clear that N is invariant under conjugating with generators of S if and only if it is invariant under conjugating with all elements of S.

So in order to compute the normal closure of U, we can start with $N := U$, conjugate N with a generator s of S and set N to the subgroup generated by N and N^s. Then we take the next generator of S. The whole process is repeated until N is stable. A GAP3 function doing this looks like

```
NormalClosure := function( S, U )

    local   G,  # the common supergroup of S and U
            N,  # closure computed so far
            M,  # next closure under generators of S
            s;  # one generator of S

    G := Parent( S, U );
    M := U;
    repeat
        N := M;
        for s  in Igs( S )  do
            M := MergedCgs( G, [ M ^ s, M ] );
        od;
    until M = N;
    return N;

end;
```

Let $S = G$ be the wreath product of the symmetric group on four points with itself using the natural permutation representation. Let U be a randomly chosen subgroup of order 12. The above functions needs, say, 100 time units to compute the normal closure of U under S, which is a subgroup N of index 2 in G.

```
gap> prms := [ (1,2), (1,2,3), (1,3)(2,4), (1,2)(3,4) ];
[ (1,2), (1,2,3), (1,3)(2,4), (1,2)(3,4) ]
gap> f := GroupHomomorphismByImages( s4, Group( prms, () ),
>              s4.generators, prms );;
gap> G := WreathProduct( s4, s4, f );
Group( h1, h2, h3, h4, n1_1, n1_2, n1_3, n1_4, n2_1, n2_2, n2_3,
n2_4, n3_1, n3_2, n3_3, n3_4, n4_1, n4_2, n4_3, n4_4 )
gap> G.name := "G";;
gap> u := Random( G );
h1*h3*h4*n1_1*n1_3*n1_4*n2_1*n2_2*n2_3*n2_4*n3_2*n3_3*n4_1*n4_3*n4_4
gap> U := MergedCgs( G, [ u ] );
Subgroup( G,
[ h1*h3*n1_2^2*n1_3*n1_4*n2_1*n2_3*n3_1*n3_2*n3_4*n4_1*n4_3,
  h4*n1_4*n2_1*n2_4*n3_1*n3_2*n4_2^2*n4_4,
  n1_1*n2_1*n3_1*n3_2^2*n3_3*n3_4*n4_1*n4_4 ] )
gap> Size( U );
8
```

Now we can ask to speed up things. The first observation is that computing a canonical generating system is usablely a more time consuming task than computing a conjugate subgroup. So we form a canonical generating system after we have computed all conjugate subgroups, although now an additional repeat-until loop could be necessary.

```
NormalClosure := function( S, U )
    local  G,  N,  M,  s,  gens;

    G := Parent( S, U );
    M := U;
    repeat
        N := M;
        gens := [ M ];
        for s  in Igs( S )  do
            Add( gens, M ^ s );
        od;
        M := MergedCgs( G, gens );
    until M = N;
    return N;

end;
```

If we now test this new normal closure function with the above groups, we see that the running time has decreased to 48 time units. The canonical generating system algorithm is faster if it knows a large subgroup of the group which should be generated but it does not gain speed if it knows several of them. A canonical generating system for the conjugated subgroup `M^s` is computed, although we only need generators for this subgroup. So we can rewrite our algorithm.

```
NormalClosure := function( S, U )

    local  G,      # the common supergroup of S and U
           N,      # closure computed so far
           M,      # next closure under generators of S
           gensS,  # generators of S
           gens;   # generators of next step

    G := Parent( S, U );
    M := U;
    gens := Igs( S );
    repeat
        N := M;
        gens := Concatenation( [ M ], Concatenation( List( S, s ->
                    List( Igs( M ), m -> m ^ s ) ) ) );
        M := MergedCgs( G, gens );
    until M = N;
    return N;

end;
```

Now a canonical generating system is generated only once per repeat-until loop. This reduces the running time to 33 time units. Let $m \in M$ and $s \in S$. Then $\langle M, m^s \rangle = \langle M, m^{-1}m^s \rangle$. So we can substitute `m^s` by `Comm( m,  s )`. If m is invariant under s the new generator would be 1 instead of m. With this modification the running times drops to 23 time units.

As next step we can try to compute induced generating systems instead of canonical ones.

In that case we cannot compare aggroups by =, but as N is a subgroup M it is sufficient to compare the composition lengths.

```
NormalClosure := function( S, U )

    local   G,       # the common supergroup of S and U
            N,       # closure computed so far
            M,       # next closure under generators of S
            gensS,   # generators of S
            gens;    # generators of next step

    G := Parent( S, U );
    M := U;
    gens := Igs( S );
    repeat
        N := M;
        gens := Concatenation( List( S, s -> List( Igs( M ),
                    m -> Comm( m, s ) ) ) );
        M := MergedIgs( G, N, gens, false );
    until Length( Igs( M ) ) = Length( Igs( M ) );
    Normalize( N );
    return N;

end;
```

But if we try the example above the running time has increased to 31. As the normal closure has index 2 in G the agwords involved in a canonical generating system are of length one or two. But agwords of induced generating system may have much large length. So we have avoided some collections but made the collection process itself much more complicated. Nevertheless in examples with subgroups of greater index the last function is slightly faster.

Chapter 26

Special Ag Groups

Special ag groups are a subcategory of ag groups (see 25).

Let G be an ag group with PAG system $(g_1, \ldots, g_n)$. Then $(g_1, \ldots, g_n)$ is a **special ag system** if it is an ag system with some additional properties, which are described below.

In general a finite polycyclic group has several different ag systems and at least one of this is a special ag system, but in GAP3 an ag group is defined by a fixed ag system and according to this an ag group is called a special ag group if its ag system is a special ag system.

Special ag systems give more information about their corresponding group than arbitrary ag systems do (see 26.1) and furthermore there are many algorithms, which are much more efficient for a special ag group than for an arbitrary one. (See 26.7)

The following sections describe the special ag system (see 26.1), their construction in GAP3 (see 26.2 and 26.3) and their additional record entries (see 26.4). Then follow two sections with functions which do only work for special ag groups (see 26.5 and 26.6).

26.1 More about Special Ag Groups

Now the properties of a special ag system are described. First of all the **Leedham-Green series** will be introduced.

Let $G = G_1 > G_2 > \ldots > G_m > G_{m+1} = \{1\}$ be the **lower nilpotent series** of G, i.e., G_i is the smallest normal subgroup of G_{i-1} such that G_{i-1}/G_i is nilpotent.

To refine this series the **lower elementary abelian series** of a nilpotent group N will be constructed. Let $N = P_1 \cdot \ldots \cdot P_l$ be the direct product of its Sylow-subgroups such that P_h is a p_h-group and $p_1 < p_2 < \ldots < p_l$ holds. Let $\lambda_j(P_h)$ be the j-th term of the p_h-central series of P_h and let k_h be the length of this series (see 7.42). Define N_{j,p_h} as the subgroup of N with

$$N_{j,p_h} = \lambda_{j+1}(P_1) \cdots \lambda_{j+1}(P_{h-1}) \cdot \lambda_j(P_h) \cdots \lambda_j(P_l).$$

With $k = \max\{k_1, \ldots, k_l\}$ the series

$$N = N_{1,p_1} \geq N_{1,p_2} \geq \ldots \geq N_{1,p_l} \geq N_{2,p_1} \geq \ldots \geq N_{k,p_l} = \{1\}$$

is obtained. Since the p-central series may have different lengths for different primes, some subgroups might be equal. The lower elementary abelian series is obtained, if for all pairs of equal subgroups the one with the lexicographically greater index is removed. This series is a characteristic central series with maximal elementary abelian factors.

To get the Leedham-Green series of G, each factor of the lower nilpotent series of G is refined by its lower elementary abelian series. The subgroups of the Leedham-Green series are denoted by $G_{i,j,p_{i,h}}$ such that $G_{i,j,p_{i,h}}/G_{i+1} = (G_i/G_{i+1})_{j,p_{i,h}}$ for each prime $p_{i,h}$ dividing the order of G_i/G_{i+1}. The Leedham-Green series is a characteristic series with elementary abelian factors.

A PAG system corresponds naturally to a composition series of its group. The first additional property of a special ag system is that the corresponding composition series refines the Leedham-Green series.

Secondly, all the elements of a special ag system are of prime-power order, and furthermore, if a set of primes $\pi = \{q_1, \ldots, q_r\}$ is given, all elements of a special ag system which are of q_h-power order for some q_h in π generate a Hall-π-subgroup of G. In fact they form a canonical generating sequence of the Hall-π-subgroup. These Hall subgroups are called **public subgroups**, since a subset of the PAG system is an induced generating set for the subgroup. Note that the set of all public Sylow subgroups forms a Sylow system of G.

The last property of the special ag systems is the existence of public **local head complements**. For a nilpotent group N, the group

$$\lambda_2(N) = \lambda_2(P_1) \cdots \lambda_2(P_l)$$

is the Frattini subgroup of N. The **local heads** of the group G are the factors

$$(G_i/G_{i+1})/\lambda_2(G_i/G_{i+1}) = G_i/G_{i,2,p_{i,1}}$$

for each i. A local head complement is a subgroup K_i of G such that $K_i/G_{i,2,p_{i,1}}$ is a complement of $G_i/G_{i,2,p_{i1}}$. Now a special ag system has a public local head complement for each local head. This complement is generated by the elements of the special ag system which do not lie in $G_i \backslash G_{i,2,p_{i,1}}$. Note that all complements of a local head are conjugate. The factors

$$\lambda_2(G_i/G_{i+1}) = G_{i,2,p_{i,1}}/G_{i+1}$$

are called the **tails** of the group G.

To handle the special ag system the **weights** are introduced. Let $(g_1, \ldots, g_n)$ be a special ag system. The triple (w_1, w_2, w_3) is called the weight of the generator g_i if g_i lies in G_{w_1,w_2,w_3} but not lower down in the Leedham-Green series. That means w_1 corresponds to the subgroup in the lower nilpotent series and w_2 to the subgroup in the elementary-abelian series of this factor, and w_3 is the prime dividing the order of g_i. Then $weight(g_i) = (w_1, w_2, w_3)$ and $weight_j(g_i) = w_j$ for $j = 1, 2, 3$ is set. With this definition $\{g_i | weight_3(g_i) \in \pi\}$ is a Hall-π-subgroup of G and $\{g_i | weight(g_i) \neq (j, 1, p)$ for some $p\}$ is a local head complement.

Now some advantages of a special ag system are summarized.

1. You have a characteristic series with elementary abelian factors of G explicitly given in the ag system. This series is refined by the composition series corresponding to the ag system.

2. You can see whether G is nilpotent or even a p-group, and if it is, you have a central series explicitly given by the Leedham-Green series. Analogously you can see whether the group is even elementary abelian.

3. You can easily calculate Hall-π-subgroups of G. Furthermore the set of public Sylow subgroups forms a Sylow system.

4. You get a smaller generating set of the group by taking only the elements which correspond to local heads of the group.

5. The collection with a special ag system may be faster than the collection with an arbitrary ag system, since in the calculation of the public subgroups of G the commutators of the ag generators are shortened.

6. Many algorithms are faster for special ag groups than for arbitrary ag groups.

26.2 Construction of Special Ag Groups

`SpecialAgGroup(` *G* `)`

The function `SpecialAgGroup` takes an ag group G as input and calculates a special ag group H, which is isomorphic to G.

To obtain the additional information of a special ag system see 26.4.

26.3 Restricted Special Ag Groups

If one is only interested in some of the information of special ag systems then it is possible to suppress the calculation of one or all types of the public subgroups by calling the function `SpecialAgGroup(` *G*`,` *flag* `)`, where *flag* is "noHall", "noHead"or "noPublic". With this options the algorithm takes less time. It calculates an ag group H, which is isomorphic to G. But be careful, because the output H is not handled as a special ag group by GAP3 but as an arbitrary ag group. Exspecially none of the functions listet in 26.7 use the algorithms for special ag groups.

`SpecialAgGroup(` *G*`, "noPublic")`

calculates an ag group H, which is isomorphic to G and whose ag system is corresponding to the Leedham-Green series.

`SpecialAgGroup(` *G*`, "noHall")`

calculates an ag group H, which is isomorphic to G and whose ag system is corresponding to the Leedham-Green series and has public local head complements.

`SpecialAgGroup(` *G*`, "noHead")`

calculates an ag group H, which is isomorphic to G and whose ag system is corresponding to the Leedham-Green series and has public Hall subgroups.

To obtain the additional information of a special ag system see 26.4.

26.4 Special Ag Group Records

In addition to the record components of ag groups (see 25) the following components are present in the group record of a special ag group H.

`weights`
: This is a list of weights such that the i-th entry gives the weight of the element h_i, i.e., the triple (w_1, w_2, w_3) when h_i lies in G_{w_1,w_2,w_3} but not lower down in the Leedham-Green series (see 26.1).

The entries `layers`, `first`, `head` and `tail` only depend on the `weights`. These entries are useful in many of the programs using the special ag system.

`layers`
: This is a list of integers. Assume that the subgroups of the Leedham-Green series are numbered beginning at G and ending at the trivial group. Then the i-th entry gives the number of the subgroup in the Leedham-Green series to which h_i corresponds as described in `weights`.

`first`
: This is a list of integers, and `first`$[j] = i$ if h_i is the first element of the j-th layer. Additionally the last entry of the list `first` is always $n + 1$.

`head`
: This is a list of integers, and `head`$[j] = i$ if h_i is the first element of the j-th local head. Additionally the last entry of the list `head` is always $n + 1$ (see 26.1).

`tail`
: This is a list of integers, and `tail`$[j] = i$ if h_{i-1} is the last element of the j-th local head. In other words h_i is either the first element of the tail of the j-th layer in the lower nilpotent series, or in case this tail is trivial, then h_i is the first element of the $j + 1$-st layer in the lower nilpotent series. If the tail of the smallest nontrivial subgroup of the lower nilpotent series is trivial, then the last entry of the list `tail` is $n + 1$ (see 26.1).

`bijection`
: This is the isomorphism from H to G given through the images of the generators of H.

The next four entries indicate if any *flag* and which one is used in the calculation of the special ag system (see 26.2 and 26.3).

`isHallSystem`
: This entry is a Boolean. It is true if public Hall subgroups have been calculated, and false otherwise.

`isHeadSystem`
: This entry is a Boolean. It is true if public local head complements have been calculated, and false otherwise.

`isSagGroup`
: This entry is a Boolean. It is true if public Hall subgroups and public local head complements have been calculated, and false otherwise.

Note that in GAP3 an ag group is called a special ag group if and only if the record entry `isSagGroup` is true.

```
# construct a wreath product of a4 with s3 where s3 operates on 3 points.
gap> s3 := SymmetricGroup( AgWords, 3 );;
gap> a4 := AlternatingGroup( AgWords, 4 );;
gap> a4wrs3 := WreathProduct(a4, s3, s3.bijection);
Group( h1, h2, n1_1, n1_2, n1_3, n2_1, n2_2, n2_3, n3_1, n3_2, n3_3 )

# now calculate the special ag group
gap> S := SpecialAgGroup( a4wrs3 );
Group( h1, n3_1, h2, n2_1, n1_1, n1_2, n1_3, n2_2, n2_3, n3_2, n3_3 )
gap> S.weights;
[ [ 1, 1, 2 ], [ 1, 1, 3 ], [ 2, 1, 3 ], [ 2, 1, 3 ], [ 2, 2, 3 ],
  [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ],
  [ 3, 1, 2 ] ]
gap> S.layers;
[ 1, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5 ]
gap> S.first;
[ 1, 2, 3, 5, 6, 12 ]
gap> S.head;
[ 1, 3, 6, 12 ]
gap> S.tail;
[ 3, 5, 12 ]
gap> S.bijection;
GroupHomomorphismByImages( Group( h1, n3_1, h2, n2_1, n1_1, n1_2,
n1_3, n2_2, n2_3, n3_2, n3_3 ), Group( h1, h2, n1_1, n1_2, n1_3,
n2_1, n2_2, n2_3, n3_1, n3_2, n3_3 ),
[ h1, n3_1, h2, n2_1, n1_1, n1_2, n1_3, n2_2, n2_3, n3_2, n3_3 ],
[ h1, n3_1, h2, n2_1*n3_1^2, n1_1*n2_1*n3_1, n1_2, n1_3, n2_2, n2_3,
  n3_2, n3_3 ] )
gap> S.isHallSystem;
true
gap> S.isHeadSystem;
true
gap> S.isSagGroup;
true
```

In the next sections the functions which only apply to special ag groups are described.

26.5 MatGroupSagGroup

`MatGroupSagGroup(` *H*, *i* `)`

`MatGroupSagGroup` calculates the matrix representation of *H* on the *i*-th layer of the Leedham-Green series of *H* (see 26.1).

See also `MatGroupAgGroup`.

```
gap> S := SpecialAgGroup( a4wrs3 );;
gap> S.weights;
[ [ 1, 1, 2 ], [ 1, 1, 3 ], [ 2, 1, 3 ], [ 2, 1, 3 ], [ 2, 2, 3 ],
  [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ], [ 3, 1, 2 ],
  [ 3, 1, 2 ] ]
```

```
gap> MatGroupSagGroup(S,3);
Group( [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
[ [ Z(3)^0, Z(3)^0 ], [ 0*Z(3), Z(3)^0 ] ] )
```

26.6 DualMatGroupSagGroup

`DualMatGroupSagGroup(` *H*, *i* `)`

`DualMatGroupSagGroup` calculates the dual matrix representation of H on the i-th layer of the Leedham-Green series of H (see 26.1).

Let V be an FH-module for a field F. Then the dual module to V is defined by $V^* := \{f : V \to F | f \text{ is linear } \}$. This module is also an FH-module and the dual matrix representation is the representation on the dual module.

```
gap> S := SpecialAgGroup( a4wrs3 );;
gap> DualMatGroupSagGroup(S,3);
Group( [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
[ [ Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0 ] ] )
```

26.7 Ag Group Functions for Special Ag Groups

Since special ag groups are ag groups all functions for ag groups are applicable to special ag groups. However certain of these functions use special implementations to treat special ag groups, i.e. there exists functions like SagGroupOps.FunctionName, which are called by the corresponding general function in case a special ag group given. If you call one of these general functions with an arbitrary ag group, the general function will not calculate the special ag group but use the function for ag groups. For the special implementations to treat special ag groups note the following.

`Centre(` *H* `)`
`MinimalGeneratingSet(` *H* `)`
`Intersection(` *U*, *L*`)`
`EulerianFunction(` *H* `)` `MaximalSubgroups(` *H* `)`
`ConjugacyClassesMaximalSubgroups(` *H* `)`
`PrefrattiniSubgroup(` *H* `)`
`FrattiniSubgroup(` *H* `)`
`IsNilpotent(` *H* `)`
These functions are often faster and often use less space for special ag groups.

`ElementaryAbelianSeries(` *H* `)`
This function returns the Leedham-Green series (see 26.1).

`IsElementaryAbelianSeries(` *H* `)`
Returns true.

`HallSubgroup(` *H*, *primes* `)`
`SylowSubgroup(` *H*, *p* `)`

`SylowSystem(` *H* `)`
These functions return the corresponding public subgroups (see 26.1).

`Subgroup(` *H* `,` *gens* `)`
`AgSubgroup(` *H* `,` *gens* `,` *bool* `)`
These functions return an ag group which is not special, except if the group itself is returned.

All domain functions not mentioned here use no special treatments for special ag groups.

Note also that there exists a package to compute formation theoretic subgroups of special ag groups. This may be used to compute the system normalizer of the public Sylow system, which is the F-normalizer for the formation of nilpotent groups F. It is also possible to compute F-normalizers as well as F-covering subgroups and F-residuals of special ag groups for a number of saturated formations F which are given within the package or for self-defined saturated formations F.

Chapter 27

Lists

Lists are the most important way to collect objects and treat them together. A **list** is a collection of elements. A list also implies a partial mapping from the integers to the elements. I.e., there is a first element of a list, a second, a third, and so on.

List constants are written by writing down the elements in order between square brackets `[`, `]`, and separating them with commas `,`. An **empty list**, i.e., a list with no elements, is written as `[]`.

```
gap> [ 1, 2, 3 ];
[ 1, 2, 3 ]     # a list with three elements
gap> [ [], [ 1 ], [ 1, 2 ] ];
[ [  ], [ 1 ], [ 1, 2 ] ]     # a list may contain other lists
```

Usually a list has no holes, i.e., contain an element at every position. However, it is absolutely legal to have lists with holes. They are created by leaving the entry between the commas empty. Lists with holes are sometimes convenient when the list represents a mapping from a finite, but not consecutive, subset of the positive integers. We say that a list that has no holes is **dense**.

```
gap> l := [ , 4, 9,, 25,, 49,,,, 121 ];;
gap> l[3];
9
gap> l[4];
Error, List Element: <list>[4] must have a value
```

It is most common that a list contains only elements of one type. This is not a must though. It is absolutely possible to have lists whose elements are of different types. We say that a list whose elements are all of the same type is **homogeneous**.

```
gap> l := [ 1, E(2), Z(3), (1,2,3), [1,2,3], "What a mess" ];;
gap> l[1];  l[3];  l[5][2];
1
Z(3)
2
```

The first sections describe the functions that test if an object is a list and convert an object to a list (see 27.1 and 27.2).

The next section describes how one can access elements of a list (see 27.4 and 27.5).

The next sections describe how one can change lists (see 27.6, 27.7, 27.8, 27.9, 27.11).

The next sections describe the operations applicable to lists (see 27.12 and 27.13).

The next sections describe how one can find elements in a list (see 27.14, 27.15, 27.16, 27.19).

The next sections describe the functions that construct new lists, e.g., sublists (see 27.22, 27.23, 27.24, 27.25, 27.26).

The next sections describe the functions deal with the subset of elements of a list that have a certain property (see 27.27, 27.28, 27.30, 27.32, 27.33, 27.34).

The next sections describe the functions that sort lists (see 27.35, 27.36, 27.38, 27.41).

The next sections describe the functions to compute the product, sum, maximum, and minimum of the elements in a list (see 27.42, 27.43, 27.45, 27.46, 27.47).

The final section describes the function that takes a random element from a list (see 27.48).

Lists are also used to represent sets, subsets, vectors, and ranges (see 28, 29, 32, and 31).

27.1 IsList

`IsList(` *obj* `)`

`IsList` returns `true` if the argument *obj*, which can be an arbitrary object, is a list and `false` otherwise. Will signal an error if *obj* is an unbound variable.

```
gap> IsList( [ 1, 3, 5, 7 ] );
true
gap> IsList( 1 );
false
```

27.2 List

`List(` *obj* `)`
`List(` *list*`,` *func* `)`

In its first form `List` returns the argument *obj*, which must be a list, a permutation, a string or a word, converted into a list. If *obj* is a list, it is simply returned. If *obj* is a permutation, `List` returns a list where the *i*-th element is the image of *i* under the permutation *obj*. If *obj* is a word, `List` returns a list where the *i*-th element is the *i*-th generator of the word, as a word of length 1.

```
gap> List( [1,2,3] );
[ 1, 2, 3 ]
gap> List( (1,2)(3,4,5) );
[ 2, 1, 4, 5, 3 ]
```

In its second form `List` returns a new list, where each element is the result of applying the function *func*, which must take exactly one argument and handle the elements of *list*, to the corresponding element of the list *list*. The list *list* must not contain holes.

```
gap> List( [1,2,3], x->x^2 );
[ 1, 4, 9 ]
gap> List( [1..10], IsPrime );
```

```
[ false, true, true, false, true, false, true, false, false, false ]
```

Note that this function is called `map` in Lisp and many other similar programming languages. This name violates the GAP3 rule that verbs are used for functions that change their arguments. According to this rule `map` would change *list*, replacing every element with the result of the application *func* to this argument.

27.3 ApplyFunc

`ApplyFunc(` *func*, *arglist* `)`

`ApplyFunc` invokes the function *func* as if it had been called with the elements of *arglist* as its arguments and returns the value, if any, returned by that invocation.

```
gap> foo := function(arg1, arg2)
> Print("first ",arg1," second ",arg2,"\n"); end;
function ( arg1, arg2 ) ... end
gap> foo(1,2);
first 1 second 2
gap> ApplyFunc(foo,[1,2]);
first 1 second 2
gap> ApplyFunc(Position,[[1,2,3],2]);
2
```

27.4 List Elements

list`[` *pos* `]`

The above construct evaluates to the *pos*-th element of the list *list*. *pos* must be a positive integer. List indexing is done with origin 1, i.e., the first element of the list is the element at position 1.

```
gap> l := [ 2, 3, 5, 7, 11, 13 ];;
gap> l[1];
2
gap> l[2];
3
gap> l[6];
13
```

If *list* does not evaluate to a list, or *pos* does not evaluate to a positive integer, or *list*`[`*pos*`]` is unbound an error is signalled. As usual you can leave the break loop (see 3.2) with `quit;`. On the other hand you can return a result to be used in place of the list element by `return` *expr*`;`.

list`{` *poss* `}`

The above construct evaluates to a new list *new* whose first element is *list*`[`*poss*`[1]]`, whose second element is *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. If for any i, *list*`[` *poss*`[`i`]` `]` is unbound, an error is signalled.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[4..6]};
```

```
[ 7, 11, 13 ]
gap> l{[1,7,1,8]};
[ 2, 17, 2, 19 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the left operand (see 27.9).

It is possible to nest such sublist extractions, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]};
[ [ 3, 2 ], [ 6, 5 ], [ 9, 8 ] ]
gap> l := m{[1,2,3]};; l{[3,2]};
[ [ 7, 8, 9 ], [ 4, 5, 6 ] ]
```

Note the difference between the two examples. The latter extracts elements 1, 2, and 3 from *m* and then extracts the elements 3 and 2 from **this list**. The former extracts elements 1, 2, and 3 from *m* and then extracts the elements 3 and 2 from **each of those element lists**.

To be precise. With each selector `[`*pos*`]` or {*poss*} we associate a **level** that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

```
        l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1      1     1      2
```

Then a selector *list*`[`*pos*`]` of level *level* is computed as `ListElement(`*list*`,`*pos*`,`*level*`)`, where `ListElement` is defined as follows

```
ListElement := function ( list, pos, level )
    if level = 0  then
        return list[pos];
    else
        return List( list, elm -> ListElement(elm,pos,level-1) );
    fi;
end;
```

and a selector *list*{*poss*} of level *level* is computed as `ListElements(`*list*`,`*poss*`,`*level*`)`, where `ListElements` is defined as follows

```
ListElements := function ( list, poss, level )
    if level = 0  then
        return list{poss};
    else
        return List( list, elm -> ListElements(elm,poss,level-1) );
    fi;
end;
```

27.5 Length

`Length(` *list* `)`

`Length` returns the length of the list *list*. The **length** is defined as 0 for the empty list, and as the largest positive integer *index* such that *list*`[`*index*`]` has an assigned value for nonempty lists. Note that the length of a list may change if new elements are added to it or assigned to previously unassigned positions.

```
gap> Length( [] );
0
gap> Length( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
8
gap> Length( [ 1, 2,,, 5 ] );
5
```

For lists that contain no holes `Length`, `Number` (see 27.27), and `Size` (see 4.10) return the same value. For lists with holes `Length` returns the largest index of a bound entry, `Number` returns the number of bound entries, and `Size` signals an error.

27.6 List Assignment

list`[` *pos* `] :=` *object*`;`

The list assignment assigns the object *object*, which can be of any type, to the list entry at the position *pos*, which must be a positive integer, in the list *list*. That means that accessing the *pos*-th element of the list *list* will return *object* after this assignment.

```
gap> l := [ 1, 2, 3 ];;
gap> l[1] := 3;;  l;                # assign a new object
[ 3, 2, 3 ]
gap> l[2] := [ 4, 5, 6 ];;  l;      # object may be of any type
[ 3, [ 4, 5, 6 ], 3 ]
gap> l[ l[1] ] := 10;;  l;          # index may be an expression
[ 3, [ 4, 5, 6 ], 10 ]
```

If the index *pos* is larger than the length of the list *list* (see 27.5), the list is automatically enlarged to make room for the new element. Note that it is possible to generate lists with holes that way.

```
gap> l[4] := "another entry";;  l; # list is enlarged
[ 3, [ 4, 5, 6 ], 10, "another entry" ]
gap> l[ 10 ] := 1;;  l;            # now list has a hole
[ 3, [ 4, 5, 6 ], 10, "another entry",,,,,, 1 ]
```

The function `Add` (see 27.7) should be used if you want to add an element to the end of the list.

Note that assigning to a list changes the list. The ability to change an object is only available for lists and records (see 27.9).

If *list* does not evaluate to a list, *pos* does not evaluate to a positive integer or *object* is a call to a function which does not return a value, for example `Print` (see 3.14), an error is signalled As usual you can leave the break loop (see 3.2) with `quit;`. On the other hand you can continue the assignment by returning a list, an index or an object using `return` *expr*`;`.

list`{` *poss* `} :=` *objects*`;`

The list assignment assigns the object *objects*`[1]`, which can be of any type, to the list *list* at the position *poss*`[1]`, the object *objects*`[2]` to *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. *objects* must be a dense list and must have the same length as *poss*.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[1..4]} := [10..13];;  l;
[ 10, 11, 12, 13, 11, 13, 17, 19 ]
gap> l{[1,7,1,10]} := [ 1, 2, 3, 4 ];; l;
[ 3, 11, 12, 13, 11, 13, 2, 19,, 4 ]
```

It is possible to nest such sublist assignments, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]} := [ [11,12], [13,14], [15,16] ];;  m;
[ [ 1, 12, 11 ], [ 4, 14, 13 ], [ 7, 16, 15 ], [ 10, 11, 12 ] ]
```

The exact behaviour is defined in the same way as for list extractions (see 27.4). Namely with each selector `[`*pos*`]` or {*poss*} we associate a **level** that is defined as the number of selectors of the form {*poss*} to its left in the same expression. For example

```
        l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1      1     1      2
```

Then a list assignment *list*`[`*pos*`] := `*vals*`;` of level *level* is computed as `ListAssignment(` *list*`,` *pos*`,` *vals*`,` *level* `)`, where `ListAssignment` is defined as follows

```
ListAssignment := function ( list, pos, vals, level )
    local  i;
    if level = 0  then
        list[pos] := vals;
    else
        for i  in [1..Length(list)]  do
            ListAssignment( list[i], pos, vals[i], level-1 );
        od;
    fi;
end;
```

and a list assignment *list*{*poss*} `:=` *vals* of level *level* is computed as `ListAssignments(` *list*`,` *poss*`,` *vals*`,` *level* `)`, where `ListAssignments` is defined as follows

```
ListAssignments := function ( list, poss, vals, level )
    local  i;
    if level = 0  then
        list{poss} := vals;
    else
        for i  in [1..Length(list)]  do
            ListAssignments( list[i], poss, vals[i], level-1 );
        od;
    fi;
end;
```

27.7 Add

`Add(` *list*`,` *elm* `)`

`Add` adds the element *elm* to the end of the list *list*, i.e., it is equivalent to the assignment *list*`[ Length(`*list*`) + 1 ] :=` *elm*. The list is automatically enlarged to make room for the new element. `Add` returns nothing, it is called only for its side effect.

Note that adding to a list changes the list. The ability to change an object is only available for lists and records (see 27.9).

To add more than one element to a list use `Append` (see 27.8).

```
gap> l := [ 2, 3, 5 ];;  Add( l, 7 );  l;
[ 2, 3, 5, 7 ]
```

27.8 Append

`Append(` *list1* `,` *list2* `)`

`Append` adds (see 27.7) the elements of the list *list2* to the end of the list *list1*. *list2* may contain holes, in which case the corresponding entries in *list1* will be left unbound. `Append` returns nothing, it is called only for its side effect.

```
gap> l := [ 2, 3, 5 ];; Append( l, [ 7, 11, 13 ] );  l;
[ 2, 3, 5, 7, 11, 13 ]
gap> Append( l, [ 17,, 23 ] ); l;
[ 2, 3, 5, 7, 11, 13, 17,, 23 ]
```

Note that appending to a list changes the list. The ability to change an object is only available for lists and records (see 27.9).

Note that `Append` changes the first argument, while `Concatenation` (see 27.22) creates a new list and leaves its arguments unchanged. As usual the name of the function that work destructively is a verb, but the name of the function that creates a new object is a substantive.

27.9 Identical Lists

With the list assignment (see 27.6, 27.7, 27.8) it is possible to change a list. The ability to change an object is only available for lists and records. This section describes the semantic consequences of this fact.

You may think that in the following example the second assignment changes the integer, and that therefore the above sentence, which claimed that only lists and records can be changed is wrong

```
i := 3;
i :=  i + 1;
```

But in this example not the **integer** `3` is changed by adding one to it. Instead the **variable** `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the following example

```
l := [ 1,  2 ];
l := [ 1, 2,  3 ];
```

The second assignment does not change the first list, instead it assigns a new list to the variable `l`. On the other hand, in the following example the list is changed by the second assignment.

```
l := [ 1,  2 ];
l[3] := 3;
```

To understand the difference first think of a variable as a name for an object. The important point is that a list can have several names at the same time. An assignment *var* `:=` *list*`;` means in this interpretation that *var* is a name for the object *list*. At the end of the following example `l2` still has the value `[ 1, 2 ]` as this list has not been changed and nothing else has been assigned to it.

```
l1 := [ 1, 2 ];
l2 := l1;
l1 := [ 1, 2, 3 ];
```

But after the following example the list for which `l2` is a name has been changed and thus the value of `l2` is now `[ 1, 2, 3 ]`.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[3] := 3;
```

We shall say that two lists are **identical** if changing one of them by a list assignment also changes the other one. This is slightly incorrect, because if **two** lists are identical, there are actually only two names for **one** list. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical lists must be equal, because there is only one list with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two lists are identical.

If you enter a list literal than the list denoted by this literal is a new list that is not identical to any other list. Thus in the following example `l1` and `l2` are not identical, though they are equal of course.

```
l1 := [ 1, 2 ];
l2 := [ 1, 2 ];
```

Also in the following example, no lists in the list `l` are identical.

```
l := [];
for i  in [1..10]  do l[i] := [ 1, 2 ];  od;
```

If you assign a list to a variable no new list is created. Thus the list value of the variable on the left hand side and the list on the right hand side of the assignment are identical. So in the following example `l1` and `l2` are identical lists.

```
l1 := [ 1, 2 ];
l2 := l1;
```

If you pass a list as argument, the old list and the argument of the function are identical. Also if you return a list from a function, the old list and the value of the function call are identical. So in the following example `l1` and `l2` are identical list

```
l1 := [ 1, 2 ];
f := function ( l )  return l;  end;
l2 := f( l1 );
```

The functions `Copy` and `ShallowCopy` (see 46.11 and 46.12) accept a list and return a new list that is equal to the old list but that is **not** identical to the old list. The difference between `Copy` and `ShallowCopy` is that in the case of `ShallowCopy` the corresponding elements of the new and the old lists will be identical, whereas in the case of `Copy` they will only be equal. So in the following example `l1` and `l2` are not identical lists.

```
l1 := [ 1, 2 ];
l2 := Copy( l1 );
```

If you change a list it keeps its identity. Thus if two lists are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two lists that are not identical will never become identical if you change one of them. So in the following example both `l1` and `l2` are changed, and are still identical.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[1] := 2;
```

27.10 IsIdentical

`IsIdentical(` *l*`,` *r* `)`

`IsIdentical` returns `true` if the objects *l* and *r* are identical. Unchangeable objects are considered identical if the are equal. Changeable objects, i.e., lists and records, are identical if changing one of them by an assignment also changes the other one, as described in 27.9.

```
gap> IsIdentical( 1, 1 );
true
gap> IsIdentical( 1, () );
false
gap> l := [ 'h', 'a', 'l', 'l', 'o' ];;
gap> l = "hallo";
true
gap> IsIdentical( l, "hallo" );
false
```

27.11 Enlarging Lists

The previous section (see 27.6) told you (among other things), that it is possible to assign beyond the logical end of a list, automatically enlarging the list. This section tells you how this is done.

It would be extremly wasteful to make all lists large enough so that there is room for all assignments, because some lists may have more than 100000 elements, while most lists have less than 10 elements.

On the other hand suppose every assignment beyond the end of a list would be done by allocating new space for the list and copying all entries to the new space. Then creating a list of 1000 elements by assigning them in order, would take half a million copy operations and also create a lot of garbage that the garbage collector would have to reclaim.

So the following strategy is used. If a list is created it is created with exactly the correct size. If a list is enlarged, because of an assignment beyond the end of the list, it is enlarged by at least *length*`/8 + 4` entries. Therefore the next assignments beyond the end of the list do not need to enlarge the list. For example creating a list of 1000 elements by assigning them in order, would now take only 32 enlargements.

The result of this is of course that the **physical length**, which is also called the size, of a list may be different from the **logical length**, which is usually called simply the length of the list. Aside from the implications for the performance you need not be aware of the physical length. In fact all you can ever observe, for example by calling `Length` is the logical length.

Suppose that `Length` would have to take the physical length and then test how many entries at the end of a list are unassigned, to compute the logical length of the list. That would take too much time. In order to make `Length`, and other functions that need to know the logical length, more efficient, the length of a list is stored along with the list.

A note aside. In the previous version 2.4 of GAP3 a list was indeed enlarged every time an assignment beyond the end of the list was performed. To deal with the above inefficiency the following hacks where used. Instead of creating lists in order they were usually created in reverse order. In situations where this was not possible a dummy assignment to the last position was performed, for example

```
l := [];
l[1000] := "dummy";
l[1] := first_value();
for i  from 2  to 1000  do l[i] := next_value(l[i-1]);  od;
```

27.12 Comparisons of Lists

list1 `=` *list2*
list1 `<>` *list2*

The equality operator `=` evaluates to `true` if the two lists *list1* and *list2* are equal and `false` otherwise. The inequality operator `<>` evaluates to `true` if the two lists are not equal and `false` otherwise. Two lists *list1* and *list2* are equal if and only if for every index i, either both entries *list1*`[`i`]` and *list2*`[`i`]` are unbound, or both are bound and are equal, i.e., *list1*`[`i`]` `=` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3 ] = [ 1, 2, 3 ];
true
gap> [ , 2, 3 ] = [ 1, 2, ];
false
gap> [ 1, 2, 3 ] = [ 3, 2, 1 ];
false
```

list1 `<` *list2*, *list1* `<=` *list2* *list1* `>` *list2*, *list1* `>=` *list2*

The operators `<`, `<=`, `>` and `>=` evaluate to `true` if the list *list1* is less than, less than or equal to, greater than, or greater than or equal to the list *list2* and to `false` otherwise. Lists are ordered lexicographically, with unbound entries comparing very small. That means the following. Let i be the smallest positive integer i, such that neither both entries *list1*`[`i`]` and *list2*`[`i`]` are unbound, nor both are bound and equal. Then *list1* is less than *list2* if either *list1*`[`i`]` is unbound (and *list2*`[`i`]` is not) or both are bound and *list1*`[`i`]` `<` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3, 4 ] < [ 1, 2, 4, 8 ];
true    # list1[3] < list2[3]
gap> [ 1, 2, 3 ] < [ 1, 2, 3, 4 ];
```

```
true     # list1[4] is unbound and therefore very small
gap> [ 1, , 3, 4 ] < [ 1, 2, 3 ];
true     # list1[2] is unbound and therefore very small
```

You can also compare objects of other types, for example integers or permutations with lists. Of course those objects are never equal to a list. Records (see 46) are greater than lists, objects of every other type are smaller than lists.

```
gap> 123 < [ 1, 2, 3 ];
true
gap> [ 1, 2, 3 ] < rec( a := 123 );
true
```

27.13 Operations for Lists

list `*` *obj*
obj `*` *list*

The operator `*` evaluates to the product of list *list* by an object *obj*. The product is a new list that at each position contains the product of the corresponding element of *list* by *obj*. *list* may contain holes, in which case the result will contain holes at the same positions.

The elements of *list* and *obj* must be objects of the following types; integers (see 10), rationals (see 12), cyclotomics (see 13), elements of a finite field (see 18), permutations (see 20), matrices (see 34), words in abstract generators (see 22), or words in solvable groups (see 24).

```
gap> [ 1, 2, 3 ] * 2;
[ 2, 4, 6 ]
gap> 2 * [ 2, 3,, 5,, 7 ];
[ 4, 6,, 10,, 14 ]
gap> [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ] * (1,4);
[ (1,4), (1,4)(2,3), (1,2,4), (1,2,3,4), (1,3,2,4), (1,3,4) ]
```

Many more operators are available for vectors and matrices, which are also represented by lists (see 32.1, 34.1).

27.14 In

elm `in` *list*

The `in` operator evaluates to `true` if the object *elm* is an element of the list *list* and to `false` otherwise. *elm* is an element of *list* if there is a positive integer *index* such that *list*`[`*index*`]=`*elm* is `true`. *elm* may be an object of an arbitrary type and *list* may be a list containing elements of any type.

It is much faster to test for membership for sets, because for sets, which are always sorted (see 28), `in` can use a binary search, instead of the linear search used for ordinary lists. So if you have a list for which you want to perform a large number of membership tests you may consider converting it to a set with the function `Set` (see 28.2).

```
gap> 1 in [ 2, 2, 1, 3 ];
true
gap> 1 in [ 4, -1, 0, 3 ];
```

```
false
gap> s := Set([2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]);;
gap> 17 in s;
false        # uses binary search and only 4 comparisons
gap> 1 in [ "This", "is", "a", "list", "of", "strings" ];
false
gap> [1,2] in [ [0,6], [0,4], [1,3], [1,5], [1,2], [3,4] ];
true
```

`Position` (see 27.15) and `PositionSorted` (see 27.16) allow you to find the position of an element in a list.

27.15 Position

`Position(` *list*`,` *elm* `)`
`Position(` *list*`,` *elm*`,` *after* `)`

`Position` returns the position of the element *elm*, which may be an object of any type, in the list *list*. If the element is not in the list the result is `false`. If the element appears several times, the first position is returned.

The three argument form begins the search at position *after*+1, and returns the position of the next occurence of *elm*. If there are no more, it returns `false`.

It is much faster to search for an element in a set, because for sets, which are always sorted (see 28), `Position` can use a binary search, instead of the linear search used for ordinary lists. So if you have a list for which you want to perform a large number of searches you may consider converting it to a set with the function `Set` (see 28.2).

```
gap> Position( [ 2, 2, 1, 3 ],  1 );
3
gap> Position( [ 2, 1, 1, 3 ], 1 );
2
gap> Position( [ 2, 1, 1, 3 ], 1, 2 );
3
gap> Position( [ 2, 1, 1, 3 ], 1, 3 );
false
gap> Position( [ 4, -1, 0, 3 ],  1 );
false
gap> s := Set([2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]);;
gap> Position( s,  17 );
false        # uses binary search and only 4 comparisons
gap> Position( [ "This", "is", "a", "list", "of", "strings" ],  1 );
false
gap> Position( [ [0,6], [0,4], [1,3], [1,5], [1,2], [3,4] ],  [1,2] );
5
```

The `in` operator (see 27.14) can be used if you are only interested to know whether the element is in the list or not. `PositionSorted` (see 27.16) can be used if the list is sorted. `PositionProperty` (see 27.19) allows you to find the position of an element that satisfies a certain property in a list.

27.16 PositionSorted

`PositionSorted(` *list*`,` *elm* `)`
`PositionSorted(` *list*`,` *elm*`,` *func* `)`

In the first form `PositionSorted` returns the position of the element *elm*, which may be an object of any type, with respect to the sorted list *list*.

In the second form `PositionSorted` returns the position of the element *elm*, which may be an object of any type with respect to the list *list*, which must be sorted with respect to *func*. *func* must be a function of two arguments that returns `true` if the first argument is less than the second argument and `false` otherwise.

`PositionSorted` returns *pos* such that *list*`[`*pos*`-1] <`*elm* and *elm* `<=` *list*`[`*pos*`]`. That means, if *elm* appears once in *list*, its position is returned. If *elm* appears several times in *list*, the position of the first occurrence is returned. If *elm* is not an element of *list*, the index where *elm* must be inserted to keep the list sorted is returned.

```
gap> PositionSorted( [1,4,5,5,6,7], 0 );
1
gap> PositionSorted( [1,4,5,5,6,7], 2 );
2
gap> PositionSorted( [1,4,5,5,6,7], 4 );
2
gap> PositionSorted( [1,4,5,5,6,7], 5 );
3
gap> PositionSorted( [1,4,5,5,6,7], 8 );
7
```

`Position` (see 27.15) is another function that returns the position of an element in a list. `Position` accepts unsorted lists, uses linear instead of binary search and returns `false` if *elm* is not in *list*.

27.17 PositionSet

`PositionSet(` *list*`,` *elm* `)`
`PositionSet(` *list*`,` *elm*`,` *func* `)`

In the first form `PositionSet` returns the position of the element *elm*, which may be an object of any type, with respect to the sorted list *list*.

In the second form `PositionSet` returns the position of the element *elm*, which may be an object of any type with respect to the list *list*, which must be sorted with respect to *func*. *func* must be a function of two arguments that returns `true` if the first argument is less than the second argument and `false` otherwise.

`PositionSet` returns *pos* such that *list*`[`*pos*`-1] <`*elm* and *elm* `=` *list*`[`*pos*`]`. That means, if *elm* appears once in *list*, its position is returned. If *elm* appears several times in *list*, the position of the first occurrence is returned. If *elm* is not an element of *list*, then `false` is returned.

```
gap> PositionSet( [1,4,5,5,6,7], 0 );
false
gap> PositionSet( [1,4,5,5,6,7], 2 );
```

```
false
gap> PositionSet( [1,4,5,5,6,7], 4 );
2
gap> PositionSet( [1,4,5,5,6,7], 5 );
3
gap> PositionSet( [1,4,5,5,6,7], 8 );
false
```

`PositionSet` is very similar to `PositionSorted` (see 27.16) but returns `false` when *elm* is not an element of *list*.

27.18 Positions

`Positions(` *list*, *elm* `)`

Returns the list of indices in *list* where *elm* occurs, where *elm* may be an object of any type.

```
gap> Positions([2,1,3,1],1);
[ 2, 4 ]
gap> Positions([2,1,3,1],4);
[  ]
gap> Positions([2,1,3,1],2);
[ 1 ]
```

27.19 PositionProperty

`PositionProperty(` *list*, *func* `)`

`PositionProperty` returns the position of the first element in the list *list* for which the unary function *func* returns `true`. *list* must not contain holes. If *func* returns `false` for all elements of *list* `false` is returned. *func* must return `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> PositionProperty( [10^7..10^8], IsPrime );
20
gap> PositionProperty( [10^5..10^6],
>                       n -> not IsPrime(n) and IsPrimePowerInt(n) );
490
```

`First` (see 27.34) allows you to extract the first element of a list that satisfies a certain property.

27.20 PositionsProperty

`PositionsProperty(` *list*, *func* `)`

`PositionsProperty` returns the list of indices `i` in *list* for which `func(list[i])` returns true. Here *list* should be a list without holes and *func* be a unary function.

```
gap> PositionsProperty([1..9],IsPrime);
[ 2, 3, 5, 7 ]
gap> PositionsProperty([1..9],x->x>5);
[ 6, 7, 8, 9 ]
```

27.21 PositionSublist

`PositionSublist(`*l*`,`*sub*`)`

Returns the position of the first occurrence of the list *sub* as a sublist of consecutive elements in *l*, or `false` if there is no such occurrence.

```
gap> PositionSublist("abcde","cd");
3
gap> PositionSublist([1,0,0,1,0,1],[1,0,1]);
4
```

27.22 Concatenation

`Concatenation(` *list1* `,` *list2* `..` `)`
`Concatenation(` *list* `)`

In the first form `Concatenation` returns the concatenation of the lists *list1*, *list2*, etc. The **concatenation** is the list that begins with the elements of *list1*, followed by the elements of *list2* and so on. Each list may also contain holes, in which case the concatenation also contains holes at the corresponding positions.

```
gap> Concatenation( [ 1, 2, 3 ], [ 4, 5 ] );
[ 1, 2, 3, 4, 5 ]
gap> Concatenation( [2,3,,5,,7], [11,,13,,,,17,,19] );
[ 2, 3,, 5,, 7, 11,, 13,,,, 17,, 19 ]
```

In the second form *list* must be a list of lists *list1*, *list2*, etc, and `Concatenation` returns the concatenation of those lists.

```
gap> Concatenation( [ [1,2,3], [2,3,4], [3,4,5] ] );
[ 1, 2, 3, 2, 3, 4, 3, 4, 5 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument lists (see 27.9).

Note that `Concatenation` creates a new list and leaves it arguments unchanged, while `Append` (see 27.8) changes its first argument. As usual the name of the function that works destructively is a verb, but the name of the function that creates a new object is a substantive.

`Set(Concatenation(`*set1*`,`*set2*`..))` (see 28.2) is a way to compute the union of sets, however, `Union` (see 4.13) is more efficient.

27.23 Flat

`Flat(` *list* `)`

`Flat` returns the list of all elements that are contained in the list *list* or its sublists. That is, `Flat` first makes a new empty list *new*. Then it loops over the elements *elm* of *list*. If *elm* is not a list it is added to *new*, otherwise `Flat` appends `Flat(` *elm* `)` to *new*.

```
gap> Flat( [ 1, [ 2, 3 ], [ [ 1, 2 ], 3 ] ] );
[ 1, 2, 3, 1, 2, 3 ]
gap> Flat( [ ] );
[ ]
```

27.24 Reversed

`Reversed(` *list* `)`

`Reversed` returns a new list that contains the elements of the list *list*, which must not contain holes, in reverse order. The argument list is unchanged.

```
gap> Reversed( [ 1, 4, 5, 5, 6, 7 ] );
[ 7, 6, 5, 5, 4, 1 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 27.9).

27.25 Sublist

`Sublist(` *list*`,` *inds* `)`

`Sublist` returns a new list in which the i-th element is the element *list*`[` *inds*`[` *i* `] ]`, of the list *list*. *inds* must be a list of positive integers without holes, it need, however, not be sorted and may contains duplicate elements.

```
gap> Sublist( [ 2, 3, 5, 7, 11, 13, 17, 19 ], [4..6] );
[ 7, 11, 13 ]
gap> Sublist( [ 2, 3, 5, 7, 11, 13, 17, 19 ], [1,7,1,8] );
[ 2, 17, 2, 19 ]
gap> Sublist( [ 1, , 2, , , 3 ], [ 1..4 ] );
[ 1,, 2 ]
```

`Filtered` (see 27.30) allows you to extract elements from a list according to a predicate.

`Sublist` has been made obsolete by the introduction of the construct *list*$\{$ *inds* $\}$ (see 27.4), excepted that in the last case an error is signaled if *list*`[` *inds*`[` *i* `] ]` is unbound for some `i`.

27.26 Cartesian

`Cartesian(` *list1*`,` *list2*`..` `)`
`Cartesian(` *list* `)`

In the first form `Cartesian` returns the cartesian product of the lists *list1*, *list2*, etc.

In the second form *list* must be a list of lists *list1*, *list2*, etc., and `Cartesian` returns the cartesian product of those lists.

The **cartesian product** is a list *cart* of lists *tup*, such that the first element of *tup* is an element of *list1*, the second element of *tup* is an element of *list2*, and so on. The total number of elements in *cart* is the product of the lengths of the argument lists. In particular *cart* is empty if and only if at least one of the argument lists is empty. Also *cart* contains duplicates if and only if no argument list is empty and at least one contains duplicates.

The last index runs fastest. That means that the first element *tup1* of *cart* contains the first element from *list1*, from *list2* and so on. The second element *tup2* of *cart* contains the first element from *list1*, the first from *list2*, an so on, but the last element of *tup2* is the second element of the last argument list. This implies that *cart* is a set if and only if all argument lists are sets.

```
gap> Cartesian( [1,2], [3,4], [5,6] );
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cartesian( [1,2,2], [1,1,2] );
[ [ 1, 1 ], [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 1 ], [ 2, 2 ],
  [ 2, 1 ], [ 2, 1 ], [ 2, 2 ] ]
```

The function `Tuples` (see 47.9) computes the k-fold cartesian product of a list.

27.27 Number

`Number(` *list* `)`
`Number(` *list*, *func* `)`

In the first form `Number` returns the number of bound entries in the list *list*.

For lists that contain no holes `Number`, `Length` (see 27.5), and `Size` (see 4.10) return the same value. For lists with holes `Number` returns the number of bound entries, `Length` returns the largest index of a bound entry, and `Size` signals an error.

`Number` returns the number of elements of the list *list* for which the unary function *func* returns `true`. If an element for which *func* returns `true` appears several times in *list* it will also be counted several times. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> Number( [ 2, 3, 5, 7 ] );
4
gap> Number( [, 2, 3,, 5,, 7,,,, 11 ] );
5
gap> Number( [1..20], IsPrime );
8
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
4
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>             n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
2
```

`Filtered` (see 27.30) allows you to extract the elements of a list that have a certain property.

27.28 Collected

`Collected(` *list* `)`

`Collected` returns a new list *new* that contains for each different element *elm* of *list* a list of two elements, the first element is *elm* itself, and the second element is the number of times *elm* appears in *list*. The order of those pairs in *new* corresponds to the ordering of the elements *elm*, so that the result is sorted.

```
gap> Factors( Factorial( 10 ) );
[ 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7 ]
gap> Collected( last );
[ [ 2, 8 ], [ 3, 4 ], [ 5, 2 ], [ 7, 1 ] ]
gap> Collected( last );
[ [ [ 2, 8 ], 1 ], [ [ 3, 4 ], 1 ], [ [ 5, 2 ], 1 ], [ [ 7, 1 ], 1 ] ]
```

27.29 CollectBy

`CollectBy(`*list*`,` *f*`)`

list should be a list and f a unary function, or a list of the same length as *list*. Let $v_1, \ldots, v_n$ be the distinct values (sorted) that the function f takes on the elements of *list* (resp. the distinct entries of the list f). The function `CollectBy` returns a list whose i-th item is the sublist of the elements of *list* where f takes the value v_i (resp. where the corresponding element of f is equal to v_i).

```
gap> CollectBy([1..15],x->x mod 4);
 [ [ 4, 8, 12 ], [ 1, 5, 9, 13 ], [ 2, 6, 10, 14 ], [ 3, 7, 11, 15 ] ]
```

27.30 Filtered

`Filtered(` *list*`,` *func* `)`

`Filtered` returns a new list that contains those elements of the list *list* for which the unary function *func* returns `true`. The order of the elements in the result is the same as the order of the corresponding elements of *list*. If an element, for which *func* returns `true` appears several times in *list* it will also appear the same number of times in the result. *list* may contain holes, they are ignored by `Filtered`. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> Filtered( [1..20], IsPrime );
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
[ 3, 4, 4, 7 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>              n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
[ 3, 7 ]
```

The result is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 27.9).

`Sublist` (see 27.25) allows you to extract elements of a list according to indices given in another list.

27.31 Zip

`Zip(`*a1*`,...,`*an*`,`*f*`)`

The first arguments *a1*,...,*an* should be lists of the same length, and the last argument a function taking n arguments. This functions zips with the function f the lists *a1*,..,*an*, that is it returns a list whose i-th entry is `f(a1[i],a2[i],..,an[i])`.

```
gap> Zip([1..9],[1..9],function(x,y)return x*y;end);
[ 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

27.32 ForAll

`ForAll(` *list*`,` *func* `)`

`ForAll` returns `true` if the unary function *func* returns `true` for all elements of the list *list* and `false` otherwise. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled.

```
gap> ForAll( [1..20], IsPrime );
false
gap> ForAll( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAll( [2..14], n -> IsPrimePowerInt(n) or n mod 2 = 0 );
true
```

`ForAny` (see 27.33) allows you to test if any element of a list satisfies a certain property.

27.33 ForAny

`ForAny(` *list*, *func* `)`

`ForAny` returns `true` if the unary function *func* returns `true` for at least one element of the list *list* and `false` otherwise. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise `ForAny` signals an error.

```
gap> ForAny( [1..20], IsPrime );
true
gap> ForAny( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAny( [2..14],
>    n -> IsPrimePowerInt(n) and n mod 5 = 0 and not IsPrime(n) );
false
```

`ForAll` (see 27.32) allows you to test if all elements of a list satisfies a certain propertie.

27.34 First

`First(` *list*, *func* `)`

`First` returns the first element of the list *list* for which the unary function *func* returns `true`. *list* may contain holes. *func* must return either `true` or `false` for every element of *list*, otherwise an error is signalled. If *func* returns `false` for every element of *list* an error is signalled.

```
gap> First( [10^7..10^8], IsPrime );
10000019
gap> First( [10^5..10^6],
>           n -> not IsPrime(n) and IsPrimePowerInt(n) );
100489
```

`PositionProperty` (see 27.19) allows you to find the position of the first element in a list that satisfies a certain property.

27.35 Sort

`Sort(` *list* `)`
`Sort(` *list*, *func* `)`

`Sort` sorts the list *list* in increasing order. In the first form `Sort` uses the operator `<` to compare the elements. In the second form `Sort` uses the function *func* to compare elements. This function must be a function taking two arguments that returns `true` if the first is strictly smaller than the second and `false` otherwise.

`Sort` does not return anything, since it changes the argument *list*. Use `ShallowCopy` (see 46.12) if you want to keep *list*. Use `Reversed` (see 27.24) if you want to get a new list sorted in decreasing order.

It is possible to sort lists that contain multiple elements which compare equal. In the first form, it is guaranteed that those elements keep their relative order, but not in the seccond i.e., `Sort` is stable in the first form but not in te second.

```
gap> list := [ 5, 4, 6, 1, 7, 5 ];;  Sort( list );  list;
[ 1, 4, 5, 5, 6, 7 ]
gap> list := [ [0,6], [1,2], [1,3], [1,5], [0,4], [3,4] ];;
gap> Sort( list, function(v,w) return v*v < w*w; end ); list;
[ [ 1, 2 ], [ 1, 3 ], [ 0, 4 ], [ 3, 4 ], [ 1, 5 ], [ 0, 6 ] ]
# sorted according to the Euclidian distance from [0,0]
gap> list := [ [0,6], [1,3], [3,4], [1,5], [1,2], [0,4], ];;
gap> Sort( list, function(v,w) return v[1] < w[1]; end ); list;
[ [ 0, 6 ], [ 0, 4 ], [ 1, 3 ], [ 1, 5 ], [ 1, 2 ], [ 3, 4 ] ]
# note the random order of the elements with equal first component
```

`SortParallel` (see 27.36) allows you to sort a list and apply the exchanges that are necessary to another list in parallel. `Sortex` (see 27.38) sorts a list and returns the sorting permutation.

27.36 SortParallel

`SortParallel(` *list1* `,` *list2* `)`
`SortParallel(` *list1* `,` *list2* `,` *func* `)`

`SortParallel` sorts the list *list1* in increasing order just as `Sort` (see 27.35) does. In parallel it applies the same exchanges that are necessary to sort *list1* to the list *list2*, which must of course have at least as many elements as *list1* does.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 2, 3, 5, 7, 8, 9 ];;
gap> SortParallel( list1, list2 );
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> list2;
[ 7, 3, 2, 9, 5, 8 ]     # [ 7, 3, 9, 2, 5, 8 ] is also possible
```

`Sortex` (see 27.38) sorts a list and returns the sorting permutation.

27.37 SortBy

`SortBy(`*list*`,` *func*`)`

list should be a list and *func* a unary function. The function `SortBy` sorts the list *list* according to the value that the function *func* takes on each element of the list.

```
gap> l:=[1..15];
```

```
[ 1 .. 15 ]
gap> SortBy(l,x->x mod 4);
gap> l;
[ 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15 ]
```

27.38 Sortex

`Sortex(` *list* `)`

`Sortex` sorts the list *list* and returns the permutation that must be applied to *list* to obtain the sorted list.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := Copy( list1 );;
gap> perm := Sortex( list1 );
(1,3,5,6,4)
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

`Permuted` (see 27.41) allows you to rearrange a list according to a given permutation.

27.39 SortingPerm

`SortingPerm(` *list* `)`

`SortingPerm` returns the permutation that must be applied to *list* to sort it into ascending order.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := Copy( list1 );;
gap> perm := SortingPerm( list1 );
(1,3,5,6,4)
gap> list1;
[ 5, 4, 6, 1, 7, 5 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

`Sortex(` *list* `)` (see 27.38) returns the same permutation as `SortingPerm(` *list* `)`, and also applies it to *list* (in place).

27.40 PermListList

`PermListList(` *list1* `,` *list2* `)`

`PermListList` returns a permutation that may be applied to *list1* to obtain *list2*, if there is one. Otherwise it returns `false`.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 4, 1, 7, 5, 5, 6 ];;
gap> perm := PermListList(list1, list2);
(1,2,4)(3,5,6)
gap> Permuted( list2, perm );
[ 5, 4, 6, 1, 7, 5 ]
```

27.41 Permuted

`Permuted(` *list*`,` *perm* `)`

`Permuted` returns a new list *new* that contains the elements of the list *list* permuted according to the permutation *perm*. That is *new*`[`*i*`^`*perm*`] =` *list*`[`*i*`]`.

```
gap> Permuted( [ 5, 4, 6, 1, 7, 5 ], (1,3,5,6,4) );
[ 1, 4, 5, 5, 6, 7 ]
```

`Sortex` (see 27.38) allows you to compute the permutation that must be applied to a list to get the sorted list.

27.42 Product

`Product(` *list* `)`
`Product(` *list*`,` *func* `)`

In the first form `Product` returns the product of the elements of the list *list*, which must have no holes. If *list* is empty, the integer 1 is returned.

In the second form `Product` applies the function *func* to each element of the list *list*, which must have no holes, and multiplies the results. If the *list* is empty, the integer 1 is returned.

```
gap> Product( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
9699690
gap> Product( [1..10], x->x^2 );
13168189440000
gap> Product( [ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) ] );
(1,4)(2,3)
```

`Sum` (see 27.43) computes the sum of the elements of a list.

27.43 Sum

`Sum(` *list* `)`
`Sum(` *list*`,` *func* `)`

In the first form `Sum` returns the sum of the elements of the list *list*, which must have no holes. If *list* is empty 0 is returned.

In the second form `Sum` applies the function *func* to each element of the list *list*, which must have no holes, and sums the results. If the *list* is empty 0 is returned.

```
gap> Sum( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
77
gap> Sum( [1..10], x->x^2 );
385
gap> Sum( [ [1,2], [3,4], [5,6] ] );
[ 9, 12 ]
```

`Product` (see 27.42) computes the product of the elements of a list.

27.44 ValuePol

`ValuePol(` *list*, *x* `)`

list represents the coefficients of a polynomial. The function `ValuePol` returns the value of that polynomial at x, using Horner's scheme. It thus represents the most efficient way to evaluate the value of a polynomial.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> ValuePol([1..5],q);
5*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
```

27.45 Maximum

`Maximum(` *obj1*, *obj2*.. `)`
`Maximum(` *list* `)`

`Maximum` returns the maximum of its arguments, i.e., that argument obj_i for which $obj_k <= obj_i$ for all k. In its second form `Maximum` takes a list *list* and returns the maximum of the elements of this list.

Typically the arguments or elements of the list respectively will be integers, but actually they can be objects of an arbitrary type. This works because any two objects can be compared using the `<` operator.

```
gap> Maximum( -123, 700, 123, 0, -1000 );
700
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
700
gap> Maximum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 2, -11 ]          # lists are compared elementwise
```

27.46 Minimum

`Minimum(` *obj1*, *obj2*.. `)`
`Minimum(` *list* `)`

`Minimum` returns the minimum of its arguments, i.e., that argument obj_i for which $obj_i <= obj_k$ for all k. In its second form `Minimum` takes a list *list* and returns the minimum of the elements of this list.

Typically the arguments or elements of the list respectively will be integers, but actually they can be objects of an arbitrary type. This works because any two objects can be compared using the `<` operator.

```
gap> Minimum( -123, 700, 123, 0, -1000 );
-1000
gap> Minimum( [ -123, 700, 123, 0, -1000 ] );
-1000
gap> Minimum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 0, 15 ]          # lists are compared elementwise
```

27.47 Iterated

`Iterated(` *list*`,` *f* `)`

`Iterated` returns the result of the iterated application of the function *f*, which must take two arguments, to the elements of *list*. More precisely `Iterated` returns the result of the following application, *f*`(..`*f*`(` *f*`(` *list*`[1],` *list*`[2] ),` *list*`[3] ),..,`*list*`[`*n*`] )`.

```
gap> Iterated( [ 126, 66, 105 ], Gcd );
3
```

27.48 RandomList

`RandomList(` *list* `)`

`RandomList` returns a random element of the list *list*. The results are equally distributed, i.e., all elements are equally likely to be selected.

```
gap> RandomList( [1..200] );
192
gap> RandomList( [1..200] );
152
gap> RandomList( [ [ 1, 2 ], 3, [ 4, 5 ], 6 ] );
[ 4, 5 ]
```

`RandomSeed(` *n* `)`

`RandomSeed` seeds the pseudo random number generator `RandomList`. Thus to reproduce a computation exactly you can call `RandomSeed` each time before you start the computation. When GAP3 is started the pseudo random number generator is seeded with 1.

```
gap> RandomSeed(1);  RandomList([1..100]);  RandomList([1..100]);
96
76
gap> RandomSeed(1);  RandomList([1..100]);  RandomList([1..100]);
96
76
```

`RandomList` is called by all random functions for domains (see 4.16).

Chapter 28

Sets

A very important mathematical concept, maybe the most important of all, are sets. Mathematically a **set** is an abstract object such that each object is either an element of the set or it is not. So a set is a collection like a list, and in fact GAP3 uses lists to represent sets. Note that this of course implies that GAP3 only deals with finite sets.

Unlike a list a set must not contain an element several times. It simply makes no sense to say that an object is twice an element of a set, because an object is either an element of a set, or it is not. Therefore the list that is used to represent a set has no duplicates, that is, no two elements of such a list are equal.

Also unlike a list a set does not impose any ordering on the elements. Again it simply makes no sense to say that an object is the first or second etc. element of a set, because, again, an object is either an element of a set, or it is not. Since ordering is not defined for a set we can put the elements in any order into the list used to represent the set. We put the elements sorted into the list, because this ordering is very practical. For example if we convert a list into a set we have to remove duplicates, which is very easy to do after we have sorted the list, since then equal elements will be next to each other.

In short sets are represented by sorted lists without holes and duplicates in GAP3. Such a list is in this document called a proper set. Note that we guarantee this representation, so you may make use of the fact that a set is represented by a sorted list in your functions.

In some contexts (for example see 47), we also want to talk about multisets. A **multiset** is like a set, except that an element may appear several times in a multiset. Such multisets are represented by sorted lists with holes that may have duplicates.

The first section in this chapter describes the functions to test if an object is a set and to convert objects to sets (see 28.1 and 28.2).

The next section describes the function that tests if two sets are equal (see 28.3).

The next sections describe the destructive functions that compute the standard set operations for sets (see 28.4, 28.5, 28.6, 28.7, and 28.8).

The last section tells you more about sets and their internal representation (see 28.10).

All set theoretic functions, especially `Intersection` and `Union`, also accept sets as arguments. Thus all functions described in chapter 4 are applicable to sets (see 28.9).

Since sets are just a special case of lists, all the operations and functions for lists, especially the membership test (see 27.14), can be used for sets just as well (see 27).

28.1 IsSet

`IsSet(` *obj* `)`

`IsSet` returns `true` if the object *obj* is a set and `false` otherwise. An object is a set if it is a sorted lists without holes or duplicates. Will cause an error if evaluation of *obj* is an unbound variable.

```
gap> IsSet( [] );
true
gap> IsSet( [ 2, 3, 5, 7, 11 ] );
true
gap> IsSet( [, 2, 3,, 5,, 7,,,, 11 ] );
false        # this list contains holes
gap> IsSet( [ 11, 7, 5, 3, 2 ] );
false        # this list is not sorted
gap> IsSet( [ 2, 2, 3, 5, 5, 7, 11, 11 ] );
false        # this list contains duplicates
gap> IsSet( 235711 );
false        # this argument is not even a list
```

28.2 Set

`Set(` *list* `)`

`Set` returns a new proper set, which is represented as a sorted list without holes or duplicates, containing the elements of the list *list*.

`Set` returns a new list even if the list *list* is already a proper set, in this case it is equivalent to `ShallowCopy` (see 46.12). Thus the result is a new list that is not identical to any other list. The elements of the result are however identical to elements of *list*. If *list* contains equal elements, it is not specified to which of those the element of the result is identical (see 27.9).

```
gap> Set( [3,2,11,7,2,,5] );
[ 2, 3, 5, 7, 11 ]
gap> Set( [] );
[  ]
```

28.3 IsEqualSet

`IsEqualSet(` *list1* `,` *list2* `)`

`IsEqualSet` returns `true` if the two lists *list1* and *list2* are equal **when viewed as sets**, and `false` otherwise. *list1* and *list2* are equal if every element of *list1* is also an element of *list2* and if every element of *list2* is also an element of *list1*.

If both lists are proper sets then they are of course equal if and only if they are also equal as lists. Thus `IsEqualSet(` *list1* `,` *list2* `)` is equivalent to `Set(` *list1* `) = Set(` *list2* `)` (see 28.2), but the former is more efficient.

```
gap> IsEqualSet( [2,3,5,7,11], [11,7,5,3,2] );
true
gap> IsEqualSet( [2,3,5,7,11], [2,3,5,7,11,13] );
false
```

28.4 AddSet

`AddSet(` *set*, *elm* `)`

`AddSet` adds *elm*, which may be an elment of an arbitrary type, to the set *set*, which must be a proper set, otherwise an error will be signalled. If *elm* is already an element of the set *set*, the set is not changed. Otherwise *elm* is inserted at the correct position such that *set* is again a set afterwards.

```
gap> s := [2,3,7,11];;
gap> AddSet( s, 5 );  s;
[ 2, 3, 5, 7, 11 ]
gap> AddSet( s, 13 );  s;
[ 2, 3, 5, 7, 11, 13 ]
gap> AddSet( s, 3 );  s;
[ 2, 3, 5, 7, 11, 13 ]
```

`RemoveSet` (see 28.5) is the counterpart of `AddSet`.

28.5 RemoveSet

`RemoveSet(` *set*, *elm* `)`

`RemoveSet` removes the element *elm*, which may be an object of arbitrary type, from the set *set*, which must be a set, otherwise an error will be signalled. If *elm* is not an element of *set* nothing happens. If *elm* is an element it is removed and all the following elements in the list are moved one position forward.

```
gap> s := [ 2, 3, 4, 5, 6, 7 ];;
gap> RemoveSet( s, 6 );
gap> s;
[ 2, 3, 4, 5, 7 ]
gap> RemoveSet( s, 10 );
gap> s;
[ 2, 3, 4, 5, 7 ]
```

`AddSet` (see 28.4) is the counterpart of `RemoveSet`.

28.6 UniteSet

`UniteSet(` *set1*, *set2* `)`

`UniteSet` unites the set *set1* with the set *set2*. This is equivalent to adding all the elements in *set2* to *set1* (see 28.4). *set1* must be a proper set, otherwise an error is signalled. *set2* may also be list that is not a proper set, in which case `UniteSet` silently applies `Set` to it first (see 28.2). `UniteSet` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 5, 7, 11 ];;
```

```
gap> UniteSet( set, [ 4, 8, 9 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> UniteSet( set, [ 16, 9, 25, 13, 16 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 25 ]
```

The function `UnionSet` (see 28.9) is the nondestructive counterpart to the destructive procedure `UniteSet`.

28.7 IntersectSet

`IntersectSet(` *set1* `,` *set2* `)`

`IntersectSet` intersects the set *set1* with the set *set2*. This is equivalent to removing all the elements that are not in *set2* from *set1* (see 28.5). *set1* must be a set, otherwise an error is signalled. *set2* may be a list that is not a proper set, in which case `IntersectSet` silently applies `Set` to it first (see 28.2). `IntersectSet` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16 ];;
gap> IntersectSet( set, [ 3, 5, 7, 9, 11, 13, 15, 17 ] );  set;
[ 3, 5, 7, 9, 11, 13 ]
gap> IntersectSet( set, [ 9, 4, 6, 8 ] );  set;
[ 9 ]
```

The function `IntersectionSet` (see 28.9) is the nondestructive counterpart to the destructive procedure `IntersectSet`.

28.8 SubtractSet

`SubtractSet(` *set1* `,` *set2* `)`

`SubtractSet` subtracts the set *set2* from the set *set1*. This is equivalent to removing all the elements in *set2* from *set1* (see 28.5). *set1* must be a proper set, otherwise an error is signalled. *set2* may be a list that is not a proper set, in which case `SubtractSet` applies `Set` to it first (see 28.2). `SubtractSet` returns nothing, it is only called to change *set1*.

```
gap> set := [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ];;
gap> SubtractSet( set, [ 6, 10 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> SubtractSet( set, [ 9, 4, 6, 8 ] );  set;
[ 2, 3, 5, 7, 11 ]
```

The function `Difference` (see 4.14) is the nondestructive counterpart to destructive the procedure `SubtractSet`.

28.9 Set Functions for Sets

As was already mentioned in the introduction to this chapter all domain functions also accept sets as arguments. Thus all functions described in the chapter 4 are applicable to sets. This section describes those functions where it might be helpful to know the implementation of those functions for sets.

`IsSubset(` *set1* `,` *set2* `)`

This is implemented by `IsSubsetSet`, which you can call directly to save a little bit of time. Either argument to `IsSubsetSet` may also be a list that is not a proper set, in which case `IsSubset` silently applies `Set` (see 28.2) to it first.

`Union(` *set1* `,` *set2* `)`

This is implemented by `UnionSet`, which you can call directly to save a little bit of time. Note that `UnionSet` only accepts two sets, unlike `Union`, which accepts several sets or a list of sets. The result of `UnionSet` is a new set, represented as a sorted list without holes or duplicates. Each argument to `UnionSet` may also be a list that is not a proper set, in which case `UnionSet` silently applies `Set` (see 28.2) to this argument. `UnionSet` is implemented in terms of its destructive counterpart `UniteSet` (see 28.6).

`Intersection(` *set1* `,` *set2* `)`

This is implemented by `IntersectionSet`, which you can call directly to save a little bit of time. Note that `IntersectionSet` only accepts two sets, unlike `Intersection`, which accepts several sets or a list of sets. The result of `IntersectionSet` is a new set, represented as a sorted list without holes or duplicates. Each argument to `IntersectionSet` may also be a list that is not a proper set, in which case `IntersectionSet` silently applies `Set` (see 28.2) to this argument. `IntersectionSet` is implemented in terms of its destructive counterpart `IntersectSet` (see 28.7).

The result of `IntersectionSet` and `UnionSet` is always a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of *set1*. If *set1* is not a proper list it is not specified to which of a number of equal elements in *set1* the element in the result is identical (see 27.9).

28.10 More about Sets

In the previous section we defined a proper set as a sorted list without holes or duplicates. This representation is not only nice to use, it is also a good internal representation supporting efficient algorithms. For example the `in` operator can use binary instead of a linear search since a set is sorted. For another example `Union` only has to merge the sets.

However, all those set functions also allow lists that are not proper sets, silently making a copy of it and converting this copy to a set. Suppose all the functions would have to test their arguments every time, comparing each element with its successor, to see if they are proper sets. This would chew up most of the performance advantage again. For example suppose `in` would have to run over the whole list, to see if it is a proper set, so it could use the binary search. That would be ridiculous.

To avoid this a list that is a proper set may, but need not, have an internal flag set that tells those functions that this list is indeed a proper set. Those functions do not have to check this argument then, and can use the more efficient algorithms. This section tells you when a proper set obtains this flag, so you can write your functions in such a way that you make best use of the algorithms.

The results of `Set`, `Difference`, `Intersection` and `Union` are known to be sets by construction, and thus have the flag set upon creation.

If an argument to `IsSet`, `IsEqualSet`, `IsSubset`, `Set`, `Difference`, `Intersection` or `Union` is a proper set, that does not yet have the flag set, those functions will notice that and set the flag for this set. Note that `in` will use linear search if the right operand does not have the flag set, will therefore not detect if it is a proper set and will, unlike the functions above, never set the flag.

If you change a proper set, that does have this flag set, by assignment, `Add` or `Append` the set will generally lose it flag, even if the change is such that the resulting list is still a proper set. However if the set has more than 100 elements and the value assigned or added is not a list and not a record and the resulting list is still a proper set than it will keep the flag. Note that changing a list that is not a proper set will never set the flag, even if the resulting list is a proper set. Such a set will obtain the flag only if it is passed to a set function.

Suppose you have built a proper set in such a way that it does not have the flag set, and that you now want to perform lots of membership tests. Then you should call `IsSet` with that set as an argument. If it is indeed a proper set `IsSet` will set the flag, and the subsequent `in` operations will use the more efficient binary search. You can think of the call to `IsSet` as a hint to GAP3 that this list is a proper set.

There is no way you can set the flag for an ordinary list without going through the checking in `IsSet`. The internal functions depend so much on the fact that a list with this flag set is indeed sorted and without holes and duplicates that the risk would be too high to allow setting the flag without such a check.

Chapter 29

Boolean Lists

This chapter describes boolean lists. A **boolean list** is a list that has no holes and contains only boolean values, i.e., `true` and `false`. In function names we call boolean lists **blist** for brevity.

Boolean lists can be used in various ways, but maybe the most important application is their use for the description of **subsets** of finite sets. Suppose *set* is a finite set, represented as a list. Then a subset *sub* of *set* is represented by a boolean list *blist* of the same length as *set* such that *blist*`[`*i*`]` is `true` if *set*`[`*i*`]` is in *sub* and `false` otherwise.

This package contains functions to switch between the representations of subsets of a finite set either as sets or as boolean lists (see 29.1, 29.2), to test if a list is a boolean list (see 29.3), and to count the number of `true` entries in a boolean list (see 29.4).

Next there are functions for the standard set operations for the subsets represented by boolean lists (see 29.5, 29.6, 29.7, and 29.8). There are also the corresponding destructive procedures that change their first argument (see 29.9, 29.10, and 29.11). Note that there is no function to add or delete a single element to a subset represented by a boolean list, because this can be achieved by assigning `true` or `false` to the corresponding position in the boolean list (see 27.6).

Since boolean lists are just a special case of lists, all the operations and functions for lists, can be used for boolean lists just as well (see 27). For example `Position` (see 27.15) can be used to find the `true` entries in a boolean list, allowing you to loop over the elements of the subset represented by the boolean list.

There is also a section about internal details (see 29.12).

29.1 BlistList

`BlistList(` *list*`,` *sub* `)`

`BlistList` returns a new boolean list that describes the list *sub* as a sublist of the list *list*, which must have no holes. That is `BlistList` returns a boolean list *blist* of the same length as *list* such that *blist*`[`*i*`]` is `true` if *list*`[`*i*`]` is in *sub* and `false` otherwise.

list need not be a proper set (see 28), even though in this case `BlistList` is most efficient. In particular *list* may contain duplicates. *sub* need not be a proper sublist of *list*, i.e., *sub* may contain elements that are not in *list*. Those elements of course have no influence on the result of `BlistList`.

```
gap> BlistList( [1..10], [2,3,5,7] );
[ false, true, true, false, true, false, true, false, false, false ]
gap> BlistList( [1,2,3,4,5,2,8,6,4,10], [4,8,9,16] );
[ false, false, false, true, false, false, true, false, true, false ]
```

`ListBlist` (see 29.2) is the inverse function to `BlistList`.

29.2 ListBlist

`ListBlist(` *list*, *blist* `)`

`ListBlist` returns the sublist *sub* of the list *list*, which must have no holes, represented by the boolean list *blist*, which must have the same length as *list*. *sub* contains the element *list*`[`*i*`]` if *blist*`[`*i*`]` is `true` and does not contain the element if *blist*`[`*i*`]` is `false`. The order of the elements in *sub* is the same as the order of the corresponding elements in *list*.

```
gap> ListBlist([1..8],[false,true,true,true,true,false,true,true]);
[ 2, 3, 4, 5, 7, 8 ]
gap> ListBlist( [1,2,3,4,5,2,8,6,4,10],
> [false,false,false,true,false,false,true,false,true,false] );
[ 4, 8, 4 ]
```

`BlistList` (see 29.1) is the inverse function to `ListBlist`.

29.3 IsBlist

`IsBlist(` *obj* `)`

`IsBlist` returns `true` if *obj*, which may be an object of arbitrary type, is a boolean list and `false` otherwise. A boolean list is a list that has no holes and contains only `true` and `false`.

```
gap> IsBlist( [ true, true, false, false ] );
true
gap> IsBlist( [] );
true
gap> IsBlist( [false,,true] );
false    # has holes
gap> IsBlist( [1,1,0,0] );
false    # contains not only boolean values
gap> IsBlist( 17 );
false    # is not even a list
```

29.4 SizeBlist

`SizeBlist(` *blist* `)`

`SizeBlist` returns the number of entries of the boolean list *blist* that are `true`. This is the size of the subset represented by the boolean list *blist*.

```
gap> SizeBlist( [ true, true, false, false ] );
2
```

29.5 IsSubsetBlist

`IsSubsetBlist(` *blist1* `,` *blist2* `)`

`IsSubsetBlist` returns `true` if the boolean list *blist2* is a subset of the boolean list *list1*, which must have equal length, and `false` otherwise. *blist2* is a subset if *blist1* if *blist1*`[`*i*`] =` *blist1*`[`*i*`] or` *blist2*`[`*i*`]` for all *i*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IsSubsetBlist( blist1, blist2 );
false
gap> blist2 := [ true, false, false, false ];;
gap> IsSubsetBlist( blist1, blist2 );
true
```

29.6 UnionBlist

`UnionBlist(` *blist1* `,` *blist2* `..` `)`
`UnionBlist(` *list* `)`

In the first form `UnionBlist` returns the union of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The **union** is a new boolean list such that *union*`[`*i*`] =` *blist1*`[`*i*`] or` *blist2*`[`*i*`] or ...`

In the second form *list* must be a list of boolean lists *blist1*, *blist2*, etc., which must have equal length, and `Union` returns the union of those boolean list.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UnionBlist( blist1, blist2 );
[ true, true, true, false ]
```

Note that `UnionBlist` is implemented in terms of the procedure `UniteBlist` (see 29.9).

29.7 IntersectionBlist

`IntersectionBlist(` *blist1* `,` *blist2* `..` `)`
`IntersectionBlist(` *list* `)`

In the first form `IntersectionBlist` returns the intersection of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The **intersection** is a new boolean list such that *inter*`[`*i*`] =` *blist1*`[`*i*`] and` *blist2*`[`*i*`] and ...`

In the second form *list* must be a list of boolean lists *blist1*, *blist2*, etc., which must have equal length, and `IntersectionBlist` returns the intersection of those boolean lists.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IntersectionBlist( blist1, blist2 );
[ true, false, false, false ]
```

Note that `IntersectionBlist` is implemented in terms of the procedure `IntersectBlist` (see 29.10).

29.8 DifferenceBlist

`DifferenceBlist(` *blist1* `,` *blist2* `)`

`DifferenceBlist` returns the asymmetric set difference of the two boolean lists *blist1* and *blist2*, which must have equal length. The **asymmetric set difference** is a new boolean list such that *union*`[`*i*`] =` *blist1*`[`*i*`] and not` *blist2*`[`*i*`]`.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> DifferenceBlist( blist1, blist2 );
[ false, true, false, false ]
```

Note that `DifferenceBlist` is implemented in terms of the procedure `SubtractBlist` (see 29.11).

29.9 UniteBlist

`UniteBlist(` *blist1* `,` *blist2* `)`

`UniteBlist` unites the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*`[`*i*`] :=` *blist1*`[`*i*`] or` *blist2*`[`*i*`]` for all i. `UniteBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UniteBlist( blist1, blist2 );
gap> blist1;
[ true, true, true, false ]
```

The function `UnionBlist` (see 29.6) is the nondestructive counterpart to the procedure `UniteBlist`.

29.10 IntersectBlist

`IntersectBlist(` *blist1* `,` *blist2* `)`

`IntersectBlist` intersects the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*`[`*i*`]:=` *blist1*`[`*i*`] and` *blist2*`[`*i*`]` for all i. `IntersectBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IntersectBlist( blist1, blist2 );
gap> blist1;
[ true, false, false, false ]
```

The function `IntersectionBlist` (see 29.7) is the nondestructive counterpart to the procedure `IntersectBlist`.

29.11 SubtractBlist

`SubtractBlist(` *blist1* `,` *blist2* `)`

`SubtractBlist` subtracts the boolean list *blist2* from the boolean list *blist1*, which must have equal length. This is equivalent to assigning *blist1*`[`*i*`] :=` *blist1*`[`*i*`] and not` *blist2*`[`*i*`]` for all i. `SubtractBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> SubtractBlist( blist1, blist2 );
gap> blist1;
[ false, true, false, false ]
```

The function `DifferenceBlist` (see 29.8) is the nondestructive counterpart to the procedure `SubtractBlist`.

29.12 More about Boolean Lists

In the previous section (see 29) we defined a boolean list as a list that has no holes and contains only `true` and `false`. There is a special internal representation for boolean lists that needs only 1 bit for every entry. This bit is set if the entry is `true` and reset if the entry is `false`. This representation is of course much more compact than the ordinary representation of lists, which needs 32 bits per entry.

Not every boolean list is represented in this compact representation. It would be too much work to test every time a list is changed, whether this list has become a boolean list. This section tells you under which circumstances a boolean list is represented in the compact representation, so you can write your functions in such a way that you make best use of the compact representation.

The results of `BlistList`, `UnionBlist`, `IntersectionBlist` and `DifferenceBlist` are known to be boolean lists by construction, and thus are represented in the compact representation upon creation.

If an argument of `IsBlist`, `IsSubsetBlist`, `ListBlist`, `UnionBlist`, `IntersectionBlist`, `DifferenceBlist`, `UniteBlist`, `IntersectBlist` and `SubtractBlist` is a list represented in the ordinary representation, it is tested to see if it is in fact a boolean list. If it is not, `IsBlist` returns `false` and the other functions signal an error. If it is, the representation of the list is changed to the compact representation.

If you change a boolean list that is represented in the compact representation by assignment (see 27.6) or `Add` (see 27.7) in such a way that the list remains a boolean list it will remain represented in the compact representation. Note that changing a list that is not represented in the compact representation, whether it is a boolean list or not, in such a way that the resulting list becomes a boolean list, will never change the representation of the list.

Chapter 30

Strings and Characters

A **character** is simply an object in GAP3 that represents an arbitrary character from the character set of the operating system. Character literals can be entered in GAP3 by enclosing the character in **singlequotes** `'`.

```
gap> 'a';
'a'
gap> '*';
'*'
```

A **string** is simply a dense list of characters. Strings are used mainly in filenames and error messages. A string literal can either be entered simply as the list of characters or by writing the characters between **doublequotes** `"`. GAP3 will always output strings in the latter format.

```
gap> s1 := ['H','a','l','l','o',' ','w','o','r','l','d','.'];
"Hallo world."
gap> s2 := "Hallo world.";
"Hallo world."
gap> s1 = s2;
true
gap> s3 := "";
""    # the empty string
gap> s3 = [];
true
```

Note that a string is just a special case of a list. So everything that is possible for lists (see 27) is also possible for strings. Thus you can access the characters in such a string (see 27.4), test for membership (see 27.14), etc. You can even assign to such a string (see 27.6). Of course unless you assign a character in such a way that the list stays dense, the resulting list will no longer be a string.

```
gap> Length( s2 );
12
gap> s2[2];
'a'
```

```
gap> 'e' in s2;
false
gap> s2[2] := 'e';;  s2;
"Hello world."
```

If a string is displayed as result of an evaluation (see 3.1), it is displayed with enclosing doublequotes. However, if a string is displayed by `Print`, `PrintTo`, or `AppendTo` (see 3.14, 3.15, 3.16) the enclosing doublequotes are dropped.

```
gap> s2;
"Hello world."
gap> Print( s2 );
Hello world.gap>
```

There are a number of **special character sequences** that can be used between the single quote of a character literal or between the doublequotes of a string literal to specify characters, which may otherwise be inaccessible. They consist of two characters. The first is a backslash \. The second may be any character. The meaning is given in the following list

`n` **newline character**. This is the character that, at least on UNIX systems, separates lines in a text file. Printing of this character in a string has the effect of moving the cursor down one line and back to the beginning of the line.

`"` **doublequote character**. Inside a string a doublequote must be escaped by the backslash, because it is otherwise interpreted as end of the string.

`'` **singlequote character**. Inside a character a singlequote must escaped by the backslash, because it is otherwise interpreted as end of the character.

\ **backslash character**. Inside a string a backslash must be escaped by another backslash, because it is otherwise interpreted as first character of an escape sequence.

`b` **backspace character**. Printing this character should have the effect of moving the cursor back one character. Whether it works or not is system dependent and should not be relied upon.

`r` **carriage return character**. Printing this character should have the effect of moving the cursor back to the beginning of the same line. Whether this works or not is again system dependent.

`c` **flush character**. This character is not printed. Its purpose is to flush the output queue. Usually GAP3 waits until it sees a *newline* before it prints a string. If you want to display a string that does not include this character use *c*.

other For any other character the backslash is simply ignored.

Again, if the line is displayed as result of an evaluation, those escape sequences are displayed in the same way that they are input. They are displayed in their special way only by `Print`, `PrintTo`, or `AppendTo`.

```
gap> "This is one line.\nThis is another line.\n";
"This is one line.\nThis is another line.\n"
gap> Print( last );
This is one line.
This is another line.
```

It is not allowed to enclose a *newline* inside the string. You can use the special character sequence *n* to write strings that include *newline* characters. If, however, a string is too long to fit on a single line it is possible to **continue** it over several lines. In this case the last character of each line, except the last must be a backslash. Both backslash and *newline* are thrown away. Note that the same continuation mechanism is available for identifiers and integers.

```
gap> "This is a very long string that does not fit on a line \
gap> and is therefore continued on the next line.";
"This is a very long string that does not fit on a line and is therefo\
re continued on the next line."
# note that the output is also continued, but at a different place
```

This chapter contains sections describing the function that creates the printable representation of a string (see 30.1), the functions that create new strings (see 30.2, 30.3), the functions that tests if an object is a string (see 30.5), the string comparisons (see 30.4), and the function that returns the length of a string (see 27.5).

30.1 String

`String(` *obj* `)`
`String(` *obj*, *length* `)`

`String` returns a representation of the *obj*, which may be an object of arbitrary type, as a string. This string should approximate as closely as possible the character sequence you see if you print *obj*.

If *length* is given it must be an integer. The absolute value gives the minimal length of the result. If the string representation of *obj* takes less than that many characters it is filled with blanks. If *length* is positive it is filled on the left, if *length* is negative it is filled on the right.

```
gap> String( 123 );
"123"
gap> String( [1,2,3] );
"[ 1, 2, 3 ]"
gap> String( 123, 10 );
"       123"
gap> String( 123, -10 );
"123       "
gap> String( 123, 2 );
"123"
```

30.2 ConcatenationString

`ConcatenationString(` *string1*, *string2* `)`

`ConcatenationString` returns the concatenation of the two strings *string1* and *string2*. This is a new string that starts with the characters of *string1* and ends with the characters of *string2*.

```
gap> ConcatenationString( "Hello ", "world.\n" );
"Hello world.\n"
```

Because strings are now lists, `Concatenation` (see 27.22) does exactly the right thing, and the function `ConcatenationString` is obsolete.

30.3 SubString

`SubString(` *string*`,` *from*`,` *to* `)`

`SubString` returns the substring of the string *string* that begins at position *from* and continues to position *to*. The characters at these two positions are included. Indexing is done with origin 1, i.e., the first character is at position 1. *from* and *to* must be integers and are both silently forced into the range `1..Length(`*string*`)` (see 27.5). If *to* is less than *from* the substring is empty.

```
gap> SubString( "Hello world.\n", 1, 5 );
"Hello"
gap> SubString( "Hello world.\n", 5, 1 );
""
```

Because strings are now lists, substrings can also be extracted with *string*`{[`*from*`..`*to*`]}` (see 27.4). `SubString` forces *from* and *to* into the range `1..Length(`*string*`)`, which the above does not, but apart from that `SubString` is obsolete.

30.4 Comparisons of Strings

string1 `=` *string2*, *string1* `<>` *string2*

The equality operator `=` evaluates to `true` if the two strings *string1* and *string2* are equal and `false` otherwise. The inequality operator `<>` returns `true` if the two strings *string1* and *string2* are not equal and `false` otherwise.

```
gap> "Hello world.\n" = "Hello world.\n";
true
gap> "Hello World.\n" = "Hello world.\n";
false     # string comparison is case sensitive
gap> "Hello world." = "Hello world.\n";
false     # the first string has no newline
gap> "Goodbye world.\n" = "Hello world.\n";
false
gap> [ 'a', 'b' ] = "ab";
true
```

string1 `<` *string2*, *string1* `<=` *string2*, *string1* `>` *string2*, *string1* `=>` *string2*

The operators `<`, `<=`, `>`, and `=>` evaluate to `true` if the string *string1* is less than, less than or equal to, greater than, greater than or equal to the string *string2* respectively. The ordering of strings is lexicographically according to the order implied by the underlying, system dependent, character set.

You can also compare objects of other types, for example integers or permutations with strings. As strings are dense character lists they compare with other objects as lists do, i.e., they are never equal to those objects, records (see 46) are greater than strings, and objects of every other type are smaller than strings.

```
gap> "Hello world.\n" < "Hello world.\n";
false     # the strings are equal
gap> "Hello World.\n" < "Hello world.\n";
true     # in ASCII uppercase letters come before lowercase letters
```

```
gap> "Hello world." < "Hello world.\n";
true     # prefixes are always smaller
gap> "Goodbye world.\n" < "Hello world.\n";
true     # G comes before H, in ASCII at least
```

30.5 IsString

`IsString( obj )`

`IsString` returns `true` if the object *obj*, which may be an object of arbitrary type, is a string and `false` otherwise. Will cause an error if *obj* is an unbound variable.

```
gap> IsString( "Hello world.\n" );
true
gap> IsString( "123" );
true
gap> IsString( 123 );
false
gap> IsString( [ '1', '2', '3' ] );
true
gap> IsString( [ '1', '2', , '4' ] );
false     # strings must be dense
gap> IsString( [ '1', '2', 3 ] );
false     # strings must only contain characters
```

30.6 Join

`Join( list [, delimiter] )`

The function `Join` is similar to the Perl function of the same name. It first applies the function `String` to all elements of the *list*, then joins the resulting strings, separated by the given *delimiter* (if omitted, `","` is used as a delimiter)

```
gap> Join([1..4]);
"1,2,3,4"
gap> Join([1..4],"foo");
"1foo2foo3foo4"
```

30.7 SPrint

`SPrint(s1,...,sn)`

`SPrint(s1,...,sn)` is a synonym for `Join([s1,...,sn],"")`. That is, it first applies the function `String` to all arguments, then joins the resulting strings. If *s1*,...,*sn* have string methods, the effect of `Print(SPrint(s1,...,sn))` is the same as directly `Print(s1,...,sn)`.

```
gap> SPrint(1,"a",[3,4]);
"1a[ 3, 4 ]"
```

30.8 PrintToString

`PrintToString(s,s1,...,sn)`

`PrintToString(`*s*`,`*s1*`,...,`*sn*`)` appends to string *s* the string `SPrint(`*s1*`,...,`*sn*`)`.

```
gap> s:="a";
"a"
gap> PrintToString(s,[1,2]);
gap> s;
"a[ 1, 2 ]"
```

30.9 Split

`Split(` *s* `[,` *delimiter*`] )`

This function is similar to the Perl function of the same name. It splits the string *s* at each occurrence of the *delimiter* (a character). If *delimiter* is omitted, `','` is used as a delimiter.

```
gap> Split("14,2,2,1,");
 [ "14", "2", "2", "1", "" ]
```

30.10 StringDate

`StringDate(`*days*`)`

`StringDate([`*day*`,` *month*`,` *year*`])`

This function converts to a readable string a date, which can be a number of days since 1-Jan-1970 or a list `[`*day*`,` *month*`,` *year*`]`.

```
gap> StringDate([11,3,1998]);
"11-Mar-1998"
gap> StringDate(2^14);
"10-Nov-2014"
```

30.11 StringTime

`StringTime(`*msec*`)`

`StringTime([`*hour*`,` *min*`,` *sec*`,` *msec*`])`

This function converts to a readable string atime, which can be a number of milliseconds or a list `[`*hour*`,` *min*`,` *sec*`,` *msec*`]`.

```
gap> StringTime([1,10,5,13]);
" 1:10:05.013"
gap> StringTime(2^22);
" 1:09:54.304"
```

30.12 StringPP

`StringPP(`*int*`)`

returns a string representing the prime factor decomposition of the integer int.

```
   gap> StringPP(40320);
"2^7*3^2*5*7"
```

Chapter 31

Ranges

A **range** is a dense list of integers, such that the difference between consecutive elements is a nonzero constant. Ranges can be abbreviated with the syntactic construct `[` *first*`,` *second* `..` *last* `]` or, if the difference between consecutive elements is 1, as `[` *first* `..` *last* `]`.

If *first* `>` *last*, `[`*first*`,`*second*`..`*last*`]` is the empty list, which by definition is also a range. If *first* = *last*, `[`*first*`,`*second*`..`*last*`]` is a singleton list, which is a range too. Note that *last* `-` *first* must be divisible by the increment *second* `-` *first*, otherwise an error is signalled.

Note that a range is just a special case of a list. So everything that is possible for lists (see 27) is also possible for ranges. Thus you can access elements in such a range (see 27.4), test for membership (see 27.14), etc. You can even assign to such a range (see 27.6). Of course, unless you assign *last* `+` *second*`-`*first* to the entry *range*`[Length(`*range*`)+1]`, the resulting list will no longer be a range.

Most often ranges are used in connection with the `for`-loop (see 2.17). Here the construct `for` *var* `in [`*first*`..`*last*`] do` *statements* `od` replaces the `for` *var* `from` *first* `to` *last* `do` *statements* `od`, which is more usual in other programming languages.

Note that a range is at the same time also a set (see 28), because it contains no holes or duplicates and is sorted, and also a vector (see 32), because it contains no holes and all elements are integers.

```
gap> r := [10..20];
[ 10 .. 20 ]
gap> Length( r );
11
gap> r[3];
12
gap> 17 in r;
true
gap> r[12] := 25;; r;
[ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25 ]
gap> r := [1,3..17];
[ 1, 3 .. 17 ]
```

```
gap> Length( r );
9
gap> r[4];
7
gap> r := [0,-1..-9];
[ 0, -1 .. -9 ]
gap> r[5];
-4
gap> r := [ 1, 4 .. 32 ];
Error, Range: <high>-<low> must be divisible by <inc>
gap> s := [];; for i  in [10..20]  do Add( s, i^2 );  od;  s;
[ 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400 ]
```

The first section in this chapter describes the function that tests if a list is a range (see 31.1).

The other section tells you more about the internal representation of ranges (see 31.2).

31.1 IsRange

`IsRange(` *obj* `)`

`IsRange` returns `true` if *obj*, which may be an object of any type, is a range and `false` otherwise. A range is a list without holes such that the elements are integers with a constant increment. Will cause an error if *obj* is an unassigned variable.

```
gap> IsRange( [1,2,3] );
true     # this list is a range
gap> IsRange( [7,5,3,1] );
true     # this list is a range
gap> IsRange( [1,2,4,5] );
false     # this list is a set and a vector, but not a range
gap> IsRange( [1,,3,,5,,7] );
false     # this list contains holes
gap> IsRange( 1 );
false     # is not even a list
gap> IsRange( [] );
true     # the empty list is a range by definition
gap> IsRange( [1] );
true     # singleton lists are a range by definition too
```

31.2 More about Ranges

For some lists the kernel knows that they are in fact ranges. Those lists are represented internally in a compact way instead of the ordinary way. This is important since this representation needs only 12 bytes for the entire list while the ordinary representation needs $4length$ bytes.

Note that a list that is represented in the ordinary way might still be a range. It is just that GAP3 does not know this. This section tells you under which circumstances a range is represented in the compact way, so you can write your program in such a way that you make best use of this compact representation for ranges.

Lists created by the syntactic construct `[` *first*`,` *second* `..` *last* `]` are of course known to be ranges and are represented in the compact way.

If you call `IsRange` for a list represented the ordinary way that is indeed a range, `IsRange` will note this, change the representation from the ordinary to the compact representation, and then return `true`;

If you change a range that is represented in the compact way, by assignment, `Add` or `Append`, the range will be converted to the ordinary representation, even if the change is such that the resulting list is still a proper range.

Suppose you have built a proper range in such a way that it is represented in the ordinary way and that you now want to convert it to the compact representation to save space. Then you should call `IsRange` with that list as an argument. If it is indeed a proper range, `IsRange` will convert it to the compact representation. You can think of the call to `IsRange` as a hint to GAP3 that this list is a proper range.

Chapter 32

Vectors

A important concept in algebra is the vector space over a field F. A **vector space** V is a set of **vectors**, for which an addition $u+v$ and a multiplication by **scalars**, i.e., elements from F, sv must be defined. A **base** of V is a list of vectors, such that every vector in V can be uniquely written as linear combination of the base vectors. If the base if finite, its size is called the **dimension** of V. Using a base it can be shown that V is isomorphic to the set n-tuples of elements with the componentwise addition and multiplication.

This comment suggests the representation that is actually used in GAP3. A GAP3 vector is a list without holes whose elements all come from a common field. We call the length of the list the dimension of the vector. This is a little bit lax, because the dimension is a property of the vector space, not of the vector, but should seldom cause confusion.

The first possibility for this field are the rationals (see 12). We call a list without holes whose elements are all rationals a rational vector, which is a bit lax too, but should again cause no confusion. For example `[ 1/2, 0, -1/3, 2 ]` is a rational vector of dimension 4.

The second possibility are cyclotomics (see 13). Note that the rationals are the prime field of cyclotomic fields and therefore rational vectors are just a special case of cyclotomic vectors. An example of a cyclotomic vector is `[ E(3)+E(3)^2, 1, E(15) ]`.

Third the common field may be a finite field (see 18). Note that it is not enough that all elements are finite field elements of the same characteristic, the common finite field containing all elements must be representable in GAP3, i.e., must have at most 2^{16} elements. An example of such a vector over the finite field $GF(3^4)$ with 81 elements is `[ Z(3^4)^3, Z(3^2)^5, Z(3^4)^11 ]`.

Finally a list all of whose elements are records is also considered a vector. In that case the records should all have an `operations` record with the necessary functions `+`, `-`, `*`, `^`. This allows for vectors over library and/or user defined fields (or rings) such as a polynomial ring (see 19).

The first section in this chapter describes the operations applicable to vectors (see 32.1).

The next section describes the function that tests if an object is a vector (see 32.2).

The next section describes the function that returns a canonical multiple of a vector (see 32.3).

The last section tells you more about the internal representation of vectors (see 32.4).

Because vectors are just a special case of lists, all the operations and functions for lists are applicable to vectors also (see chapter 27). This especially includes accessing elements of a vector (see 27.4), changing elements of a vector (see 27.6), and comparing vectors (see 27.12).

Vectorspaces are a special category of domains and are described by vectorspace records (see chapter 9).

Vectors play an important role for matrices (see chapter 34), which are implemented as lists of vectors.

32.1 Operations for Vectors

vec1 `+` *vec2*

In this form the addition operator `+` evaluates to the sum of the two vectors *vec1* and *vec2*, which must have the same dimension and lie in a common field. The sum is a new vector where each entry is the sum of the corresponding entries of the vectors. As an exception it is also possible to add an integer vector to a finite field vector, in which case the integers are interpreted as *scalar* `*` *GF*`.one`.

scalar `+` *vec*
vec `+` *scalar*

In this form `+` evaluates to the sum of the scalar *scalar* and the vector *vec*, which must lie in a common field. The sum is a new vector where each entry is the sum of the scalar and the corresponding entry of the vector. As an exception it is also possible to add an integer scalar to a finite field vector, in which case the integer is interpreted as *scalar* `*` *GF*`.one`.

```
gap> [ 1, 2, 3 ] + [ 1/2, 1/3, 1/4 ];
[ 3/2, 7/3, 13/4 ]
gap> [ 1/2, 3/2, 1/2 ] + 1/2;
[ 1, 2, 1 ]
```

vec1 `-` *vec2*
scalar `-` *vec*
vec `-` *scalar*

The difference operator `-` returns the componentwise difference of its two operands and is defined subject to the same restrictions as `+`.

```
gap> [ 1, 2, 3 ] - [ 1/2, 1/3, 1/4 ];
[ 1/2, 5/3, 11/4 ]
gap> [ 1/2, 3/2, 1/2 ] - 1/2;
[ 0, 1, 0 ]
```

vec1 `*` *vec2*

In this form the multiplication operator `*` evaluates to the product of the two vectors *vec1* and *vec2*, which must have the same dimension and lie in a common field. The product is the sum of the products of the corresponding entries of the vectors. As an exception it is also possible to multiply an integer vector to a finite field vector, in which case the integers are interpreted as *scalar* `*` *GF*`.one`.

scalar * *vec*
vec * *scalar*

In this form * evaluates to the product of the scalar *scalar* and the vector *vec*, which must lie in a common field. The product is a new vector where each entry is the product of the scalar and the corresponding entry of the vector. As an exception it is also possible to multiply an integer scalar to a finite field vector, in which case the integer is interpreted as *scalar* **GF*`.one`.

```
gap> [ 1, 2, 3 ] * [ 1/2, 1/3, 1/4 ];
23/12
gap> [ 1/2, 3/2, 1/2 ] * 2;
[ 1, 3, 1 ]
```

Further operations with vectors as operands are defined by the matrix operations (see 34.1).

32.2 IsVector

`IsVector(` *obj* `)`

`IsVector` returns `true` if *obj*, which may be an object of arbitrary type, is a vector and `false` else. A vector is a list without holes, whose elements all come from a common field.

```
gap> IsVector( [ 0, -3, -2, 0, 6 ] );
true
gap> IsVector( [ Z(3^4)^3, Z(3^2)^5, Z(3^4)^13 ] );
true
gap> IsVector( [ 0, Z(2^3)^3, Z(2^3) ] );
false     # integers are not finite field elements
gap> IsVector( [ , 2, 3,, 5,, 7 ] );
false     # list that have holes are not vectors
gap> IsVector( 0 );
false     # not even a list
```

32.3 NormedVector

`NormedVector(` *vec* `)`

`NormedVector` returns the scalar multiple of *vec* such that the first nonzero entry of *vec* is the one from the field over which the vector is defined. If *vec* contains only zeroes a copy of it is returned.

```
gap> NormedVector( [ 0, -3, -2, 0, 6 ] );
[ 0, 1, 2/3, 0, -2 ]
gap> NormedVector( [ 0, 0 ] );
[ 0, 0 ]
gap> NormedVector( [ Z(3^4)^3, Z(3^2)^5, Z(3^4)^13 ] );
[ Z(3)^0, Z(3^4)^47, Z(3^2) ]
```

32.4 More about Vectors

In the first section of this chapter we defined a vector as a list without holes whose elements all come from a common field. This representation is quite nice to use. However, suppose that GAP3 would have to check that a list is a vector every time this vector appears as operand in a addition or multiplication. This would be quite wasteful.

To avoid this a list that is a vector may, but need not, have an internal flag set that tells the operations that this list is indeed a vector. Then this operations do not have to check this operand and can perform the operation right away. This section tells you when a vector obtains this flag, so you can write your functions in such a way that you make best use of this feature.

The results of vector operations, i.e., binary operations that involve vectors, are known by construction to be vectors, and thus have the flag set upon creation.

If the operand of one of the binary operation is a list that does not yet have the flag set, those operations will check that this operand is indeed a vector and set the flag if it is. If it is not a vector and not a matrix an error is signalled.

If the argument to `IsVector` is a list that does not yet have this flag set, `IsVector` will test if all elements come from a common field. If they do, `IsVector` will set the flag. Thus on the one hand `IsVector` is a test whether the argument is a vector. On the other hand `IsVector` can be used as a hint to GAP3 that a certain list is indeed a vector.

If you change a vector, that does have this flag set, by assignment, `Add`, or `Append`, the vectors will loose its flag, even if the change is such that the resulting list is still a vector. However if the vector is a vector over a finite field and you assign an element from the same finite field the vector will keep its flag. Note that changing a list that is not a vector will never set the flag, even if the resulting list is a vector. Such a vector will obtain the flag only if it appears as operand in a binary operation, or is passed to `IsVector`.

Vectors over finite fields have one additional feature. If they are known to be vectors, not only do they have the flag set, but also are they represented differently. This representation is much more compact. Instead of storing every element separately and storing for every element separately in which field it lies, the field is only stored once. This representation takes up to 10 times less memory.

Chapter 33

Row Spaces

This chapter consists essentially of four parts, according to the four different types of data structures that are described, after the usual brief discussion of the objects (see 33.1, 33.2, 33.3, 33.4, 33.5).

The first part introduces row spaces, and their operations and functions (see 33.6, 33.7, 33.8, 33.9, 33.10, 33.11, 33.12, 33.13).

The second part introduces bases for row spaces, and their operations and functions (see 33.14, 33.15, 33.16, 33.17, 33.18, 33.19, 33.20, 33.21).

The third part introduces row space cosets, and their operations and functions (see 33.22, 33.23, 33.24).

The fourth part introduces quotient spaces of row spaces, and their operations and functions (see 33.25, 33.26).

The obligatory last sections describe the details of the implementation of the data structures (see 33.27, 33.28, 33.29, 33.30).

Note: The current implementation of row spaces provides no homomorphisms of row spaces (linear maps), and also quotient spaces of quotient spaces are not supported.

33.1 More about Row Spaces

A **row space** is a vector space (see chapter 9), whose elements are row vectors, that is, lists of elements in a common field.

Note that for a row space V over the field F necessarily the characteristic of F is the same as the characteristic of the vectors in V. Furthermore at the moment the field F must contain the field spanned by all the elements in vectors of V, since in many computations vectors are normed, that is, divided by their first nonzero entry.

The implementation of functions for these spaces and their elements uses the well-known linear algebra methods, such as Gaussian elimination, and many functions delegate the work to functions for matrices, e.g., a basis of a row space can be computed by performing Gaussian elimination to the matrix formed by the list of generators. Thus in a sense, a row space in GAP3 is nothing but a GAP3 object that knows about the interpretation of a matrix as a generating set, and that knows about the functions that do the work.

Row spaces are constructed using 33.6 `RowSpace`, full row spaces can also be constructed by F `^` n, for a field F and a positive integer n.

The **zero element** of a row space V in GAP3 is not necessarily stored in the row space record. If necessary, it can be computed using `Zero(` V `)`.

The `generators` component may contain zero vectors, so no function should expect a generator to be nonzero.

For the usual concept of substructures and parent structures see 33.5.

See 33.7 and 33.8 for an overview of applicable operators and functions, and 33.27 for details of the implementation.

33.2 Row Space Bases

Many computations with row spaces require the computation of a **basis** (which will always mean a vector space basis in GAP3), such as the computation of the dimension, or efficient membership test for the row space.

Most of these computations do not rely on special properties of the chosen basis. The computation of coefficients lists, however, is basis dependent. A natural way to distinguish these two situations is the following.

For basis independent questions the row space is allowed to compute a suitable basis, and may store bases. For example the dimension of the space V can be computed this way using `Dimension(` V `)`. In such situations the component V`.basis` is used. The value of this component depends on how it was constructed, so no function that accesses this component should assume special properties of this basis.

On the other hand, the computation of coefficients of a vector v with respect to a basis B of V depends on this basis, so you have to call `Coefficients(` B`,` v `)`, and **not** `Coefficients(` V`,` v `)`.

It should be mentioned that there are two types of row space bases. A basis of the first type is **semi-echelonized** (see 33.18 for the definition and examples), its structure allows to perform efficient calculations of membership test and coefficients.

A basis of the second type is **arbitrary**, that is, it has no special properties. There are two ways to construct such a (user-defined) basis that is **not** necessarily semi-echelonized. The first is to call `RowSpace` with the optional argument `"basis"`; this means that the generators are known to be linearly independent (see 33.6). The second way is to call `Basis` with two arguments (see 33.16). The computation of coefficients with respect to an arbitrary basis is performed by computing a semi-echelonized basis, delegating the task to this basis, and then performing the base change.

The functions that are available for row space bases are `Coefficients` (see 33.14) and `SiftedVector` (see 33.15).

The several available row space bases are described in 33.16, 33.17, and 33.18. For details of the implementation see 33.28.

33.3 Row Space Cosets

Let V be a vector space, and U a subspace of V. The set $v+U=\{v+u; u\in U\}$ is called a **coset** of U in V.

In GAP3, cosets are of course domains that can be formed using the `'+'` operator, see 33.22 and 33.23 for an overview of applicable operators and functions, and 33.29 for details of the implementation.

A coset $C = v + U$ is described by any representative v and the space U. Equal cosets may have different representatives. A canonical representative of the coset C can be computed using `CanonicalRepresentative(` C `)`, it does only depend on C, especially not on the basis of U.

Row spaces cosets can be regarded as elements of quotient spaces (see 33.4).

33.4 Quotient Spaces

Let V be a vector space, and U a subspace of V. The set $\{v + U; v \in V\}$ is again a vector space, the **quotient space** (or factor space) of V modulo U.

By definition of row spaces, a quotient space is not a row space. (One reason to describe quotient spaces here is that for general vector spaces at the moment no factor structures are supported.)

Quotient spaces in GAP3 are formed from two spaces using the `/` operator. See the sections 33.25 and 33.26 for an overview of applicable operators and functions, and 33.30 for details of the implementation.

Bases for Quotient Spaces of Row Spaces

A basis B of a quotient V/U for row spaces V and U is best described by bases of V and U. If B is a basis without special properties then it will delegate the work to a semi-echelonized basis. The concept of **semi-echelonized bases** makes sense also for quotient spaces of row spaces since for any semi-echelonized basis of U the set S of pivot columns is a subset of the set of pivot columns of a semi-echelonized basis of V. So the cosets $v + U$ for basis vectors v with pivot column not in S form a semi-echelonized basis of V/U. The **canonical basis** of V/U is the semi-echelonized basis derived in that way from the canonical basis of V (see 33.17).

See 33.26 for details about the bases.

33.5 Subspaces and Parent Spaces

The concept described in this section is essentially the same as the concept of parent groups and subgroups (see 7.6).

(The section should be moved to chapter 9, but for general vector spaces the concept does not yet apply.)

Every row space U is either constructed as **subspace** of an existing space V, for example using 33.10 `Subspace`, or it is not.

In the latter case the space is called a **parent space**, in the former case V is called the **parent** of U.

One can only form sums of subspaces of the same parent space, form quotient spaces only for spaces with same parent, and cosets $v + U$ only for representatives v in the parent of U.

`Parent(` V `)`
: returns the parent space of the row space V,

`IsParent(` V `)`
: returns `true` if the row space V is a parent space, and `false` otherwise.

See 33.11, 33.12 for conversion functions.

33.6 RowSpace

`RowSpace(` F`,` *generators* `)`

returns the row space that is generated by the vectors *generators* over the field F. The elements in *generators* must be GAP3 vectors.

`RowSpace(` F`,` *generators*`,` *zero* `)`

Whenever the list *generators* is empty, this call of `RowSpace` has to be used, with *zero* the zero vector of the space.

`RowSpace(` F`,` *generators*`, "basis" )`

also returns the F-space generated by *generators*. When the space is constructed in this way, the vectors *generators* are assumed to form a basis, and this is used for example when `Dimension` is called for the space.

It is **not** checked that the vectors are really linearly independent.

`RowSpace(` F`,` *dimension* `)`
F `^` n

return the full row space of dimension n over the field F. The elements of this row space are all the vectors of length n with entries in F.

```
gap> v1:= RowSpace( GF(2), [ [ 1, 1 ], [ 0, 1 ] ] * Z(2) );
RowSpace( GF(2), [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> v2:= RowSpace( GF(2), [], [ 0, 0 ] * Z(2) );
RowSpace( GF(2), [ [ 0*Z(2), 0*Z(2) ] ] )
gap> v3:= RowSpace( GF(2), [ [ 1, 1 ], [ 0, 1 ] ] * Z(2), "basis" );
RowSpace( GF(2), [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> v4:= RowSpace( GF(2), 2 );
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> v5:= GF(2) ^ 2 ;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> v3 = v4;
true
```

Note that the list of generators may contain zero vectors.

33.7 Operations for Row Spaces

Comparisons of Row Spaces

V = W

returns `true` if the two row spaces V, W are equal as sets, and `false` otherwise.

V < W

returns `true` if the row space V is smaller than the row space W, and `false` otherwise. The first criteria of this ordering are the comparison of the fields and the dimensions, row spaces over the same field and of same dimension are compared by comparison of the reversed canonical bases (see 33.17).

Arithmetic Operations for Row Spaces

V + W

returns the sum of the row spaces V and W, that is, the row space generated by V and W. This is computed using the Zassenhaus algorithm.

V / U

returns the quotient space of V modulo its subspace U (see 33.4).

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> s:= Subspace( v, [ [ 1, 1 ] * Z(2) ] );
Subspace( v, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> t:= Subspace( v, [ [ 0, 1 ] * Z(2) ] );
Subspace( v, [ [ 0*Z(2), Z(2)^0 ] ] )
gap> s = t;
false
gap> s < t;
false
gap> t < s;
true
gap> u:= s+t;
Subspace( v, [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> u = v;
true
gap> f:= u / s;
Subspace( v, [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ) /
[ [ Z(2)^0, Z(2)^0 ] ]
```

33.8 Functions for Row Spaces

The following functions are overlaid in the operations record of row spaces.

The **set theoretic functions**

`Closure`, `Elements`, `Intersection`, `Random`, `Size`.

`Intersection(` V, W `)`

returns the intersection of the two row spaces V and W that is computed using the Zassenhaus algorithm.

The **vector space specific functions**

`Base(` V `)`

returns the list of vectors of the canonical basis of the row space V (see 33.17).

`Cosets(` *V* `,` *U* `)`
returns the list of cosets of the subspace *U* in *V*, as does `Elements(` *V* `/` *U* `)`.

`Dimension(` *V* `)`
returns the dimension of the row space. For this, a basis of the space is computed if not yet known.

`Zero(` *V* `)`
returns the zero element of the row space *V* (see 33.1).

33.9 IsRowSpace

`IsRowSpace(` *obj* `)`

returns `true` if *obj*, which can be an object of arbitrary type, is a row space and `false` otherwise.

```
gap> v:= GF(2) ^ 2;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> IsRowSpace( v );
true
gap> IsRowSpace( v / [ v.generators[1] ] );
false
```

33.10 Subspace

`Subspace(` *V* `,` *gens* `)`

returns the subspace of the row space *V* that is generated by the vectors in the list *gens*.

```
gap> v:= GF(3)^2; v.name:= "v";;
RowSpace( GF(3), [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] )
gap> s:= Subspace( v, [ [ 1, -1 ] *Z(3)^0 ] );
Subspace( v, [ [ Z(3)^0, Z(3) ] ] )
```

33.11 AsSubspace

`AsSubspace(` *V* `,`*U* `)`

returns the row space *U*, viewed as a subspace of the rows space *V*. For that, *V* must be a parent space.

```
gap> v:= GF(2)^2; v.name:="v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> u:= RowSpace( GF(2), [ [ 1, 1 ] * Z(2) ] );
RowSpace( GF(2), [ [ Z(2)^0, Z(2)^0 ] ] )
gap> w:= AsSubspace( v, u );
Subspace( v, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> w = u;
true
```

33.12 AsSpace

`AsSpace(` *U* `)`

returns the subspace U as a parent space.

```
gap> v:= GF(2)^2; v.name:="v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> u:= Subspace( v, [ [ 1, 1 ] * Z(2) ] );
Subspace( v, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> w:= AsSpace( u );
RowSpace( GF(2), [ [ Z(2)^0, Z(2)^0 ] ] )
gap> w = u;
true
```

33.13 NormedVectors

`NormedVectors(` *V* `)`

returns the set of those vectors in the row space V for that the first nonzero entry is the identity of the underlying field.

```
gap> v:= GF(3)^2;
RowSpace( GF(3), [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] )
gap> NormedVectors( v );
[ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0 ],
  [ Z(3)^0, Z(3) ] ]
```

33.14 Coefficients for Row Space Bases

`Coefficients(` *B*`,` *v* `)`

returns the coefficients vector of the vector v with respect to the basis B (see 33.2) of the vector space V, if v is an element of V. Otherwise `false` is returned.

```
gap> v:= GF(3)^2; v.name:= "v";;
RowSpace( GF(3), [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] )
gap> b:= Basis( v );
Basis( v, [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] )
gap> Coefficients( b, [ Z(3), Z(3) ] );
[ Z(3), Z(3) ]
gap> Coefficients( b, [ Z(3), Z(3)^2 ] );
[ Z(3), Z(3)^0 ]
```

33.15 SiftedVector

`SiftedVector(` *B*`,` *v* `)`

returns the residuum of the vector v with respect to the basis B of the vector space V. The exact meaning of this depends on the special properties of B.

But in general this residuum is obtained on subtracting appropriate multiples of basis vectors, and v is contained in V if and only if `SiftedVector(` *B*`,` *v* `)` is the zero vector of V.

```
gap> v:= GF(3)^2; v.name:= "v";;
RowSpace( GF(3), [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ] )
gap> s:= Subspace( v, [ [ 1, -1 ] *Z(3)^0 ] ); s.name:= "s";;
Subspace( v, [ [ Z(3)^0, Z(3) ] ] )
gap> b:= Basis(s);
SemiEchelonBasis( s, [ [ Z(3)^0, Z(3) ] ] )
gap> SiftedVector( b, [ Z(3), 0*Z(3) ] );
[ 0*Z(3), Z(3) ]
```

33.16 Basis

`Basis(` *V* `)`
`Basis(` *V* `,` *vectors* `)`

`Basis(` *V* `)` returns a basis of the row space *V*. If the component *V*`.canonicalBasis` or *V*`.semiEchelonBasis` was bound before the first call to `Basis` for *V* then one of these bases is returned. Otherwise a semi-echelonized basis (see 33.2) is computed. The basis is stored in *V*`.basis`.

`Basis(` *V* `,` *vectors* `)` returns the basis of *V* that consists of the vectors in the list *vectors*. In the case that *V*`.basis` was not bound before the call the basis is stored in this component.

Note that it is not necessarily checked whether *vectors* is really linearly independent.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> b:= Basis( v, [ [ 1, 1 ], [ 1, 0 ] ] * Z(2) );
Basis( v, [ [ Z(2)^0, Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] )
gap> Coefficients( b, [ 0, 1 ] * Z(2) );
[ Z(2)^0, Z(2)^0 ]
gap> IsSemiEchelonBasis( b );
false
```

33.17 CanonicalBasis

`CanonicalBasis(` *V* `)`

returns the canonical basis of the row space *V*. This is a special semi-echelonized basis (see 33.18), with the additional properties that for $j > i$ the position of the pivot of row j is bigger than that of the pivot of row i, and that the pivot columns contain exactly one nonzero entry.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> cb:= CanonicalBasis( v );
CanonicalBasis( v )
gap> cb.vectors;
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
```

The canonical basis is obtained on applying a full Gaussian elimination to the generators of *V*, using 34.19 `BaseMat`. If the component *V*`.semiEchelonBasis` is bound then this basis is used to compute the canonical basis, otherwise `TriangulizeMat` is called.

33.18 SemiEchelonBasis

`SemiEchelonBasis(` *V* `)`
`SemiEchelonBasis(` *V*`,` *vectors* `)`

returns a semi-echelonized basis of the row space *V*. A basis is called **semi-echelonized** if the first non-zero element in every row is one, and all entries exactly below these elements are zero.

If a second argument *vectors* is given, these vectors are taken as basis vectors. Note that if the rows of *vectors* do not form a semi-echelonized basis then an error is signalled.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> SemiEchelonBasis( v );
SemiEchelonBasis( v, [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> b:= Basis( v, [ [ 1, 1 ], [ 0, 1 ] ] * Z(2) );
Basis( v, [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> IsSemiEchelonBasis( b );
true
gap> b;
SemiEchelonBasis( v, [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> Coefficients( b, [ 0, 1 ] * Z(2) );
[ 0*Z(2), Z(2)^0 ]
gap> Coefficients( b, [ 1, 0 ] * Z(2) );
[ Z(2)^0, Z(2)^0 ]
```

33.19 IsSemiEchelonBasis

`IsSemiEchelonBasis(` *B* `)`

returns `true` if *B* is a semi-echelonized basis (see 33.18), and `false` otherwise. If *B* is semi-echelonized, and this was not yet stored before, after the call the operations record of *B* will be `SemiEchelonBasisRowSpaceOps`.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> b1:= Basis( v, [ [ 0, 1 ], [ 1, 0 ] ] * Z(2) );
Basis( v, [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] )
gap> IsSemiEchelonBasis( b1 );
true
gap> b1;
SemiEchelonBasis( v, [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] )
gap> b2:= Basis( v, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );
Basis( v, [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ] )
gap> IsSemiEchelonBasis( b2 );
false
gap> b2;
Basis( v, [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ] )
```

33.20 NumberVector

`NumberVector( B, v )`

Let $v = \sum_{i=1}^{n} \lambda_i b_i$ where $B = (b_1, b_2, \ldots, b_n)$ is a basis of the vector space V over the finite field F with $|F| = q$, and the λ_i are elements of F. Let $\overline{\lambda}$ be the integer corresponding to λ as defined by 39.29 `FFList`.

Then `NumberVector( B, v )` returns $\sum_{i=1}^{n} \overline{\lambda_i} q^{i-1}$.

```
gap> v:= GF(3)^3;; v.name:= "v";;
gap> b:= CanonicalBasis( v );;
gap> l:= List( [0 .. 6 ], x -> ElementRowSpace( b, x ) );
[ [ 0*Z(3), 0*Z(3), 0*Z(3) ], [ Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ Z(3)^0, Z(3)^0, 0*Z(3) ], [ Z(3), Z(3)^0, 0*Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ]
```

33.21 ElementRowSpace

`ElementRowSpace( B, n )`

returns the n-th element of the row space with basis B, with respect to the ordering defined in 33.20 `NumberVector`.

```
gap> v:= GF(3)^3;; v.name:= "v";;
gap> b:= CanonicalBasis( v );;
gap> l:= List( [0 .. 6 ], x -> ElementRowSpace( b, x ) );;
gap> List( l, x -> NumberVector( b, x ) );
[ 0, 1, 2, 3, 4, 5, 6 ]
```

33.22 Operations for Row Space Cosets

Comparison of Row Space Cosets

`C1 = C2`
: returns `true` if the two row space cosets *C1*, *C2* are equal, and `false` otherwise. **Note** that equal cosets need not have equal representatives (see 33.3).

`C1 < C2`
: returns `true` if the row space coset *C1* is smaller than the row space coset *C2*, and `false` otherwise. This ordering is defined by comparison of canonical representatives.

Arithmetic Operations for Row Space Cosets

`C1 + C2`
: If *C1* and *C2* are row space cosets that belong to the same quotient space, the result is the row space coset that is the sum resp. the difference of these vectors. Otherwise an error is signalled.

`s * C`
: returns the row space coset that is the product of the scalar *s* and the row space coset

C, where s must be an element of the ground field of the vector space that defines C.

Membership Test for Row Space Cosets

v `in` C

returns `true` if the vector `v` is an element of the row space coset C, and false otherwise.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> u:= Subspace( v, [ [ 1, 1 ] * Z(2) ] ); u.name:="u";;
Subspace( v, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> f:= v / u;
v / [ [ Z(2)^0, Z(2)^0 ] ]
gap> elms:= Elements( f );
[ ([ 0*Z(2), 0*Z(2) ]+u), ([ 0*Z(2), Z(2)^0 ]+u) ]
gap> 2 * elms[2];
([ 0*Z(2), 0*Z(2) ]+u)
gap> elms[2] + elms[1];
([ 0*Z(2), Z(2)^0 ]+u)
gap> [ 1, 0 ] * Z(2) in elms[2];
true
gap> elms[1] = elms[2];
false
```

33.23 Functions for Row Space Cosets

Since row space cosets are domains, all set theoretic functions are applicable to them.

`Representative` returns the value of the `representative` component. **Note** that equal cosets may have different representatives. Canonical representatives can be computed using `CanonicalRepresentative`.

`CanonicalRepresentative(` C `)`

returns the canonical representative of the row space coset C, which is defined as the result of `SiftedVector(` B`,` v `)` where C `=` v `+` U, and B is the canonical basis of U.

33.24 IsSpaceCoset

`IsSpaceCoset(` *obj* `)`

returns `true` if *obj*, which may be an arbitrary object, is a row space coset, and `false` otherwise.

```
gap> v:= GF(2)^2; v.name:= "v";;
RowSpace( GF(2), [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
gap> u:= Subspace( v, [ [ 1, 1 ] * Z(2) ] );
Subspace( v, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> f:= v / u;
v / [ [ Z(2)^0, Z(2)^0 ] ]
```

```
gap> IsSpaceCoset( u );
false
gap> IsSpaceCoset( Random( f ) );
true
```

33.25 Operations for Quotient Spaces

W1 `=` *W2*
: returns `true` if for the two quotient spaces *W1* `=` *V1* `/` *U1* and *W2* `=` *V2* `/` *U2* the equalities *V1* `=` *V2* and *U1* `=` *U2* hold, and `false` otherwise.

W1 `<` *W2*
: returns `true` if for the two quotient spaces *W1* `=` *V1* `/` *U1* and *W2* `=` *V2* `/` *U2* either *U1* `<` *U2* or *U1* `=` *U2* and *V1* `<` *V2* hold, and `false` otherwise.

33.26 Functions for Quotient Spaces

Computations in quotient spaces usually delegate the work to computations in numerator and denominator.

The following functions are overlaid in the operations record for quotient spaces.

The **set theoretic** functions

`Closure`, `Elements`, `IsSubset`, `Intersection`,

and the **vector space** functions

`Base(` *V* `)`
: returns the vectors of the canonical basis of V,

`Generators(` *V* `)`
: returns a list of cosets that generate V,

`CanonicalBasis(` *V* `)`
: returns the canonical basis of $V = W/U$, this is derived from the canonical basis of W.

`SemiEchelonBasis(` *V* `)`
: `SemiEchelonBasis(` *V* `,` *vectors* `)`
: return a semi-echelonized basis of the quotient space V. *vectors* can be a list of elements of V, or of representatives.

`Basis(` *V* `)`
: `Basis(` *V* `,` *vectors* `)`
: return a basis of the quotient space V. *vectors* can be a list of elements of V, or of representatives.

33.27 Row Space Records

In addition to the record components described in 9.3 the following components must be present in a row space record.

`isRowSpace`
: is always `true`,

`operations`
: the record `RowSpaceOps`.

Depending on the calculations in that the row space was involved, it may have lots of optional components, such as `basis`, `dimension`, `size`.

33.28 Row Space Basis Records

A vector space basis is a record with at least the following components.

`isBasis`
: always `true`,

`vectors`
: the list of basis vectors,

`structure`
: the underlying vector space,

`operations`
: a record that contains the functions for the basis, at least `Coefficients`, `Print`, and `SiftedVector`. Of course these functions depend on the special properties of the basis, so different basis types have different operations record.

Depending on the type of the basis, the basis record additionally contains some components that are assumed and used by the functions in the `operations` record.

For arbitrary bases these are `semiEchelonBasis` and `basechange`, for semi-echelonized bases these are the lists `heads` and `ishead`. Furthermore, the booleans `isSemiEchelonBasis` and `isCanonicalBasis` may be present.

The operations records for the supported bases are

`BasisRowSpaceOps`
: for arbitrary bases,

`CanonicalBasisRowSpaceOps`
: for the canonical basis of a space,

`SemiEchelonBasisRowSpaceOps`
: for semi-echelonized bases.

33.29 Row Space Coset Records

A row space coset $v + U$ is a record with at least the following components.

`isDomain`
: always `true`,

`isRowSpaceCoset`
: always `true`,

`isSpaceCoset`
: always `true`,

`factorDen`
: the row space U if the coset is an element of V/U for a space V,

`representative`
: one element of the coset, **note** that equal cosets need not have equal representatives (see 33.3),

`operations`
: the record `SpaceCosetRowSpaceOps`.

33.30 Quotient Space Records

A quotient space V/U is a record with at least the following components.

`isDomain`
: always `true`,

`isRowSpace`
: always `true`,

`isFactorSpace`
: always `true`,

`field`
: the coefficients field,

`factorNum`
: the row space V (the numerator),

`factorDen`
: the row space U (the denominator),

`operations`
: the record `FactorRowSpaceOps`.

Chapter 34

Matrices

Matrices are an important tool in algebra. A matrix nicely represents a homomorphism between two vector spaces with respect to a choice of bases for the vector spaces. Also matrices represent systems of linear equations.

In GAP3 matrices are represented by list of vectors (see 32). The vectors must all have the same length, and their elements must lie in a common field. The field may be the field of rationals (see 12), a cyclotomic field (see 13), a finite field (see 18), or a library and/or user defined field (or ring) such as a polynomial ring (see 19).

The first section in this chapter describes the operations applicable to matrices (see 34.1). The next sections describes the function that tests whether an object is a matrix (see 34.2). The next sections describe the functions that create certain matrices (see 34.3, 34.4, 34.5, and 34.6). The next sections describe functions that compute certain characteristic values of matrices (see 34.7, 34.14, 34.15, 34.16, and 34.17). The next sections describe the functions that are related to the interpretation of a matrix as a system of linear equations (see 34.18, 34.19, 34.20, and 34.21). The last two sections describe the functions that diagonalize an integer matrix (see 34.22 and 34.23).

Because matrices are just a special case of lists, all operations and functions for lists are applicable to matrices also (see chapter 27). This especially includes accessing elements of a matrix (see 27.4), changing elements of a matrix (see 27.6), and comparing matrices (see 27.12).

34.1 Operations for Matrices

mat + *scalar*
scalar + *mat*

This forms evaluates to the sum of the matrix *mat* and the scalar *scalar*. The elements of *mat* and *scalar* must lie in a common field. The sum is a new matrix where each entry is the sum of the corresponding entry of *mat* and *scalar*.

mat1 + *mat2*

This form evaluates to the sum of the two matrices *mat1* and *mat2*, which must have the same dimensions and whose elements must lie in a common field. The sum is a new matrix where each entry is the sum of the corresponding entries of *mat1* and *mat2*.

mat - *scalar*
scalar - *mat*
mat1 - *mat2*

The definition for the - operator are similar to the above definitions for the + operator, except that - subtracts of course.

mat * *scalar*
scalar * *mat*

This forms evaluate to the product of the matrix *mat* and the scalar *scalar*. The elements of *mat* and *scalar* must lie in a common field. The product is a new matrix where each entry is the product of the corresponding entries of *mat* and *scalar*.

vec * *mat*

This form evaluates to the product of the vector *vec* and the matrix *mat*. The length of *vec* and the number of rows of *mat* must be equal. The elements of *vec* and *mat* must lie in a common field. If *vec* is a vector of length n and *mat* is a matrix with n rows and m columns, the product is a new vector of length m. The element at position i is the sum of *vec*[*l*] * *mat*[*l*][*i*] with l running from 1 to n.

mat * *vec*

This form evaluates to the product of the matrix *mat* and the vector *vec*. The number of columns of *mat* and the length of *vec* must be equal. The elements of *mat* and *vec* must lie in a common field. If *mat* is a matrix with m rows and n columns and *vec* is a vector of length n, the product is a new vector of length m. The element at position i is the sum of *mat*[*i*][*l*] * *vec*[*l*] with l running from 1 to n.

mat1 * *mat2*

This form evaluates to the product of the two matrices *mat1* and *mat2*. The number of columns of *mat1* and the number of rows of *mat2* must be equal. The elements of *mat1* and *mat2* must lie in a common field. If *mat1* is a matrix with m rows and n columns and *mat2* is a matrix with n rows and o columns, the result is a new matrix with m rows and o columns. The element in row i at position k of the product is the sum of *mat1*[*i*][*l*] * *mat2*[*l*][*k*] with l running from 1 to n.

mat1 / *mat2*
scalar / *mat*
mat / *scalar*
vec / *mat*

In general *left* / *right* is defined as *left* * *right*^-1. Thus in the above forms the right operand must always be invertable.

mat ^ *int*

This form evaluates to the *int*-th power of the matrix *mat*. *mat* must be a square matrix, *int* must be an integer. If *int* is negative, *mat* must be invertible. If *int* is 0, the result is the identity matrix, even if *mat* is not invertible.

mat1 ^ *mat2*

This form evaluates to the conjugation of the matrix *mat1* by the matrix *mat2*, i.e., to *mat2*^-1 * *mat1* * *mat2*. *mat2* must be invertible and *mat1* must be such that these product can be computed.

vec ^ *mat*

This is in every respect equivalent to *vec* * *mat*. This operations reflects the fact that matrices operate on the vector space by multiplication from the right.

scalar + *matlist*
matlist + *scalar*
scalar - *matlist*
matlist - *scalar*
scalar * *matlist*
matlist * *scalar*
matlist / *scalar*

A scalar *scalar* may also be added, subtracted, multiplied with, or divide into a whole list of matrices *matlist*. The result is a new list of matrices where each matrix is the result of performing the operation with the corresponding matrix in *matlist*.

mat * *matlist*
matlist * *mat*

A matrix *mat* may also be multiplied with a whole list of matrices *matlist*. The result is a new list of matrices, where each matrix is the product of *mat* and the corresponding matrix in *matlist*.

matlist / *mat*

This form evaluates to *matlist* * *mat*^-1. *mat* must of course be invertable.

vec * *matlist*

This form evaluates to the product of the vector *vec* and the list of matrices *mat*. The length *l* of *vec* and *matlist* must be equal. All matrices in *matlist* must have the same dimensions. The elements of *vec* and the elements of the matrices in *matlist* must lie in a common field. The product is the sum of *vec*[*i*] * *matlist*[*i*] with *i* running from 1 to *l*.

`Comm(` *mat1* `,` *mat2* `)`

`Comm` returns the commutator of the matrices *mat1* and *mat2*, i.e., *mat1*^-1 * *mat2*^-1 * *mat1* * *mat2*. *mat1* and *mat2* must be invertable and such that these product can be computed.

There is one exception to the rule that the operands or their elements must lie in common field. It is allowed that one operand is a finite field element, a finite field vector, a finite field matrix, or a list of finite field matrices, and the other operand is an integer, an integer vector, an integer matrix, or a list of integer matrices. In this case the integers are interpreted as *int* * *GF*`.one`, where *GF* is the finite field (see 18.3).

For all the above operations the result is new, i.e., not identical to any other list (see 27.9). This is the case even if the result is equal to one of the operands, e.g., if you add zero to a matrix.

34.2 IsMat

`IsMat(` *obj* `)`

`IsMat` return `true` if *obj*, which can be an object of arbitrary type, is a matrix and `false` otherwise. Will cause an error if *obj* is an unbound variable.

```
gap> IsMat( [ [ 1, 0 ], [ 0, 1 ] ] );
true    # a matrix is a list of vectors
gap> IsMat( [ [ 1, 2, 3, 4, 5 ] ] );
true
gap> IsMat( [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] );
true
gap> IsMat( [ [ Z(2)^0, 0 ], [ 0, Z(2)^0 ] ] );
false   # Z(2)^0 and 0 do not lie in a common field
gap> IsMat( [ 1, 0 ] );
false   # a vector is not a matrix
gap> IsMat( 1 );
false   # neither is a scalar
```

34.3 IdentityMat

`IdentityMat(` *n* `)`
`IdentityMat(` *n*, *F* `)`

`IdentityMat` returns the identity matrix with n rows and n columns over the field F. If no field is given, `IdentityMat` returns the identity matrix over the field of rationals. Each call to `IdentityMat` returns a new matrix, so it is safe to modify the result.

```
gap> IdentityMat( 3 );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> PrintArray( last );
[ [  1,  0,  0 ],
  [  0,  1,  0 ],
  [  0,  0,  1 ] ]
gap> PrintArray( IdentityMat( 3, GF(2) ) );
[ [  Z(2)^0,  0*Z(2),  0*Z(2) ],
  [  0*Z(2),  Z(2)^0,  0*Z(2) ],
  [  0*Z(2),  0*Z(2),  Z(2)^0 ] ]
```

34.4 NullMat

`NullMat(` *m* `)`
`NullMat(` *m*, *n* `)`
`NullMat(` *m*, *n*, *F* `)`

`NullMat` returns the null matrix with m rows and n columns over the field F; if n is omitted, it is assumed equal to m. If no field is given, `NullMat` returns the null matrix over the field of rationals. Each call to `NullMat` returns a new matrix, so it is safe to modify the result.

```
gap> PrintArray( NullMat( 2, 3 ) );
[ [  0,  0,  0 ],
  [  0,  0,  0 ] ]
gap> PrintArray( NullMat( 2, 2, GF(2) ) );
[ [  0*Z(2),  0*Z(2) ],
  [  0*Z(2),  0*Z(2) ] ]
```

34.5 TransposedMat

`TransposedMat(` *mat* `)`

`TransposedMat` returns the transposed of the matrix *mat*. The transposed matrix is a new matrix *trn*, such that *trn*`[`*i*`][`*k*`]` is *mat*`[`*k*`][`*i*`]`.

```
gap> TransposedMat( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ 1, 3 ], [ 2, 4 ] ]
gap> TransposedMat( [ [ 1..5 ] ] );
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
```

34.6 KroneckerProduct

`KroneckerProduct(` *mat1* `, ...,` *matn* `)`

`KroneckerProduct` returns the Kronecker product of the matrices *mat1*, ..., *matn*. If *mat1* is a *m* by *n* matrix and *mat2* is a *o* by *p* matrix, the Kronecker product of *mat1* by *mat2* is a *m*`*`*o* by *n*`*`*p* matrix, such that the entry in row `(`*i1*`-1)*`*o*`+`*i2* at position `(`*k1*`-1)*`*p*`+`*k2* is *mat1*`[`*i1*`][`*k1*`] *` *mat2*`[`*i2*`][`*k2*`]`.

```
gap> mat1 := [ [ 0, -1, 1 ], [ -2, 0, -2 ] ];;
gap> mat2 := [ [ 1, 1 ], [ 0, 1 ] ];;
gap> PrintArray( KroneckerProduct( mat1, mat2 ) );
[ [   0,   0,  -1,  -1,   1,   1 ],
  [   0,   0,   0,  -1,   0,   1 ],
  [  -2,  -2,   0,   0,  -2,  -2 ],
  [   0,  -2,   0,   0,   0,  -2 ] ]
```

34.7 DimensionsMat

`DimensionsMat(` *mat* `)`

`DimensionsMat` returns the dimensions of the matrix *mat* as a list of two integers. The first entry is the number of rows of *mat*, the second entry is the number of columns.

```
gap> DimensionsMat( [ [ 1, 2, 3 ], [ 4, 5, 6 ] ] );
[ 2, 3 ]
gap> DimensionsMat( [ [ 1 .. 5 ] ] );
[ 1, 5 ]
```

34.8 IsDiagonalMat

`IsDiagonalMat(` *mat* `)`

mat must be a matrix. This function returns `true` if *mat* is square and all entries `mat[i][j]` with `i<>j` are equal to `0*mat[i][j]` and `false` otherwise.

```
gap> mat := [ [ 1, 2 ], [ 3, 1 ] ];;
gap> IsDiagonalMat( mat );
false
```

34.9 IsLowerTriangularMat

`IsLowerTriangularMat( `*mat*` )`

mat must be a matrix. This function returns `true` if all entries `mat[i][j]` with `j>i` are equal to `0*mat[i][j]` and `false` otherwise.

```
gap> a := [ [ 1, 2 ], [ 3, 1 ] ];;
gap> IsLowerTriangularMat( a );
false
gap> a[1][2] := 0;;
gap> IsLowerTriangularMat( a );
true
```

34.10 IsUpperTriangularMat

`IsUpperTriangularMat( `*mat*` )`

mat must be a matrix. This function returns `true` if all entries `mat[i][j]` with `j <i` are equal to `0*mat[i][j]` and `false` otherwise.

```
gap> a := [ [ 1, 2 ], [ 3, 1 ] ];;
gap> IsUpperTriangularMat( a );
false
gap> a[2][1] := 0;;
gap> IsUpperTriangularMat( a );
true
```

34.11 DiagonalOfMat

`DiagonalOfMat( `*mat*` )`

This function returns the list of diagonal entries of the matrix *mat*, that is the list of *mat*`[i][i]`.

```
gap> mat := [ [ 1, 2 ], [ 3, 1 ] ];;
gap> DiagonalOfMat( mat );
[ 1, 1 ]
```

34.12 DiagonalMat

`DiagonalMat( `*mat1*` , ... , `*matn*` )`

returns the block diagonal direct sum of the matrices *mat1*, ..., *matn*. Blocks of size 1×1 may be given as scalars.

```
gap> C1 := [ [  2, -1,  0,  0 ],
>            [ -1,  2, -1,  0 ],
>            [  0, -1,  2, -1 ],
>            [  0,  0, -1,  2 ] ];;
gap> C2 := [ [  2,  0, -1 ],
>            [  0,  2, -1 ],
>            [ -1, -1,  2 ] ];;
```

```
gap> PrintArray( DiagonalMat( C1, 3, C2 ) );
[ [   2,  -1,   0,   0,   0,   0,   0,   0 ],
  [  -1,   2,  -1,   0,   0,   0,   0,   0 ],
  [   0,  -1,   2,  -1,   0,   0,   0,   0 ],
  [   0,   0,  -1,   2,   0,   0,   0,   0 ],
  [   0,   0,   0,   0,   3,   0,   0,   0 ],
  [   0,   0,   0,   0,   0,   2,   0,  -1 ],
  [   0,   0,   0,   0,   0,   0,   2,  -1 ],
  [   0,   0,   0,   0,   0,  -1,  -1,   2 ] ]
```

To make a diagonal matrix with a specified diagonal d use `ApplyFunc(DiagonalMat, d )`.

`PermutationMat( perm, dim[, F] )` (function) returns a matrix in dimension dim over the field given by F (i.e. the smallest field containing the element F or F itself if it is a field) that represents the permutation perm acting by permuting the basis vectors as it permutes points.

34.13 PermutationMat

`PermutationMat(` *perm*`,` *dim* `[,`*F*`])`

returns a matrix in dimension *dim* over the field *F* (by default the rationals) that represents the permutation *perm* acting by permuting the basis vectors as it permutes points.

```
gap> PermutationMat((1,2,3),4);
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

34.14 TraceMat

`TraceMat(` *mat* `)`

`TraceMat` returns the trace of the square matrix *mat*. The trace is the sum of all entries on the diagonal of *mat*.

```
gap> TraceMat( [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ] );
15
gap> TraceMat( IdentityMat( 4, GF(2) ) );
0*Z(2)
```

34.15 DeterminantMat

`DeterminantMat(` *mat* `)`

`DeterminantMat` returns the determinant of the square matrix *mat*. The determinant is defined by
$\sum_{p \in Symm(n)} sign(p) \prod_{i=1}^{n} mat[i][i^p]$.

```
gap> DeterminantMat( [ [ 1, 2 ], [ 3, 4 ] ] );
-2
gap> DeterminantMat( [ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, Z(3) ] ] );
Z(3)
```

Note that `DeterminantMat` does not use the above definition to compute the result. Instead it performs a Gaussian elimination. For large rational matrices this may take very long, because the entries may become very large, even if the final result is a small integer.

34.16 RankMat

`RankMat(` *mat* `)`

`RankMat` returns the rank of the matrix *mat*. The rank is defined as the dimension of the vector space spanned by the rows of *mat*. It follows that a n by n matrix is invertible exactly if its rank is n.

```
gap> RankMat( [ [ 4, 1, 2 ], [ 3, -1, 4 ], [ -1, -2, 2 ] ] );
2
```

Note that `RankMat` performs a Gaussian elimination. For large rational matrices this may take very long, because the entries may become very large.

34.17 OrderMat

`OrderMat(` *mat* `)`

`OrderMat` returns the order of the invertible square matrix *mat*. The order *ord* is the smallest positive integer such that *mat*^*ord* is the identity.

```
gap> OrderMat( [ [ 0*Z(2), 0*Z(2), Z(2)^0 ],
>                [ Z(2)^0, Z(2)^0, 0*Z(2) ],
>                [ Z(2)^0, 0*Z(2), 0*Z(2) ] ] );
4
```

`OrderMat` first computes *ord1* such that the first standard basis vector is mapped by *mat*^*ord1* onto itself. It does this by applying *mat* repeatedly to the first standard basis vector. Then it computes *mat1* as *mat1*^*ord1*. Then it computes *ord2* such that the second standard basis vector is mapped by *mat1*^*ord2* onto itself. This process is repeated until all basis vectors are mapped onto themselves. `OrderMat` warns you that the order may be infinite, when it finds that the order must be larger than 1000.

34.18 TriangulizeMat

`TriangulizeMat(` *mat* `)`

`TriangulizeMat` brings the matrix *mat* into upper triangular form. Note that *mat* is changed. A matrix is in upper triangular form when the first nonzero entry in each row is one and lies further to the right than the first nonzero entry in the previous row. Furthermore, above the first nonzero entry in each row all entries are zero. Note that the matrix will have trailing zero rows if the rank of *mat* is not maximal. The rows of the resulting matrix span the same vectorspace than the rows of the original matrix *mat*. The function returns the indices of the lines of the orginal matrix corresponding to the non-zero lines of the triangulized matrix.

```
gap> m := [ [ 0, -3, -1 ], [ -3, 0, -1 ], [ 2, -2, 0 ] ];;
gap> TriangulizeMat( m ); m;
[ 2, 1 ]
[ [ 1, 0, 1/3 ], [ 0, 1, 1/3 ], [ 0, 0, 0 ] ]
```

Note that for large rational matrices `TriangulizeMat` may take very long, because the entries may become very large during the Gaussian elimination, even if the final result contains only small integers.

34.19 BaseMat

`BaseMat( ` *mat* `)`

`BaseMat` returns a standard base for the vector space spanned by the rows of the matrix *mat*. The standard base is in upper triangular form. That means that the first nonzero vector in each row is one and lies further to the right than the first nonzero entry in the previous row. Furthermore, above the first nonzero entry in each row all entries are zero.

```
gap> BaseMat( [ [ 0, -3, -1 ], [ -3, 0, -1 ], [ 2, -2, 0 ] ] );
[ [ 1, 0, 1/3 ], [ 0, 1, 1/3 ] ]
```

Note that for large rational matrices `BaseMat` may take very long, because the entries may become very large during the Gaussian elimination, even if the final result contains only small integers.

34.20 NullspaceMat

`NullspaceMat( ` *mat* `)`

`NullspaceMat` returns a base for the nullspace of the matrix *mat*. The nullspace is the set of vectors *vec* such that *vec* `*` *mat* is the zero vector. The returned base is the standard base for the nullspace (see 34.19).

```
gap> NullspaceMat( [ [ 2, -4, 1 ], [ 0, 0, -4 ], [ 1, -2, -1 ] ] );
[ [ 1, 3/4, -2 ] ]
```

Note that for large rational matrices `NullspaceMat` may take very long, because the entries may become very large during the Gaussian elimination, even if the final result only contains small integers.

34.21 SolutionMat

`SolutionMat( ` *mat*, *vec* `)`

`SolutionMat` returns one solution of the equation *x* `*` *mat* `=` *vec* or `false` if no such solution exists.

```
gap> SolutionMat( [ [ 2, -4, 1 ], [ 0, 0, -4 ], [ 1, -2, -1 ] ],
>                 [ 10, -20, -10 ] );
[ 5, 15/4, 0 ]
gap> SolutionMat( [ [ 2, -4, 1 ], [ 0, 0, -4 ], [ 1, -2, -1 ] ],
>                 [ 10, 20, -10 ] );
false
```

Note that for large rational matrices `SolutionMat` may take very long, because the entries may become very large during the Gaussian elimination, even if the final result only contains small integers.

34.22 DiagonalizeMat

`DiagonalizeMat( ` *mat* `)`

`DiagonalizeMat` transforms the integer matrix *mat* by multiplication with unimodular (i.e., determinant +1 or -1) integer matrices from the left and from the right into diagonal form (i.e., only diagonal entries are nonzero). Note that `DiagonalizeMat` changes *mat* and returns nothing. If there are several diagonal matrices to which *mat* is equivalent, it is not specified which one is computed, except that all zero entries on the diagonal are collected at the lower right end (see 34.23).

```
gap> m := [ [ 0, -1, 1 ], [ -2, 0, -2 ], [ 2, -2, 4 ] ];;
gap> DiagonalizeMat( m );  m;
[ [ 1, 0, 0 ], [ 0, 2, 0 ], [ 0, 0, 0 ] ]
```

Note that for large integer matrices `DiagonalizeMat` may take very long, because the entries may become very large during the computation, even if the final result only contains small integers.

34.23 ElementaryDivisorsMat

`ElementaryDivisorsMat(` *mat* `)`

`ElementaryDivisors` returns a list of the elementary divisors, i.e., the unique *d* with *d*`[`*i*`]` divides *d*`[`*i*`+1]` and *mat* is equivalent to a diagonal matrix with the elements *d*`[`*i*`]` on the diagonal (see 34.22).

```
gap> m := [ [ 0, -1, 1 ], [ -2, 0, -2 ], [ 2, -2, 4 ] ];;
gap> ElementaryDivisorsMat( m );
[ 1, 2, 0 ]
```

34.24 PrintArray

`PrintArray(` *mat* `)`

`PrintArray` displays the matrix *mat* in a pretty way.

```
gap> m := [[1,2,3,4],[5,6,7,8],[9,10,11,12]];
[ [ 1, 2, 3, 4 ], [ 5, 6, 7, 8 ], [ 9, 10, 11, 12 ] ]
gap> PrintArray( m );
[ [   1,   2,   3,   4 ],
  [   5,   6,   7,   8 ],
  [   9,  10,  11,  12 ] ]
```

Chapter 35

Integral matrices and lattices

This is a subset of the functions available in GAP4, ported to GAP3 to be used by CHEVIE.

35.1 NullspaceIntMat

`NullspaceIntMat(` *mat* `)`

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral nullspace of *mat*, i.e. of those vectors in the nullspace of *mat* that have integral entries.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> NullspaceMat(mat);
[ [ 1, 0, 3/4, -1/4, -3/4 ], [ 0, 1, -13/24, 1/8, -7/24 ] ]
gap> NullspaceIntMat(mat);
[ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ]
```

35.2 SolutionIntMat

`SolutionIntMat(` *mat*, *vec* `)`

If *mat* is a matrix with integral entries and *vec* a vector with integral entries, this function returns a vector *x* with integer entries that is a solution of the equation *x***mat*=*vec*. It returns `false` if no such vector exists.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionMat(mat,[95,115,182]);
[ 47/4, -17/2, 67/4, 0, 0 ]
gap> SolutionIntMat(mat,[95,115,182]);
[ 2285, -5854, 4888, -1299, 0 ]
```

35.3 SolutionNullspaceIntMat

`SolutionNullspaceIntMat(` *mat*, *vec* `)`

This function returns a list of length two, its first entry being the result of a call to `SolutionIntMat` with same arguments, the second the result of `NullspaceIntMat` applied to the matrix *mat*. The calculation is performed faster than if two separate calls would be used.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionNullspaceIntMat(mat,[95,115,182]);
[ [ 2285, -5854, 4888, -1299, 0 ],
  [ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ] ]
```

35.4 BaseIntMat

`BaseIntMat(` *mat* `)`

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral row space of *mat*, i.e. of the set of integral linear combinations of the rows of *mat*.

```
gap> mat:=[[1,2,7],[4,5,6],[10,11,19]];;
gap> BaseIntMat(mat);
[ [ 1, 2, 7 ], [ 0, 3, 7 ], [ 0, 0, 15 ] ]
```

35.5 BaseIntersectionIntMats

`BaseIntersectionIntMats(` *m*`,` *n* `)`

If *m* and *n* are matrices with integral entries, this function returns a list of vectors that forms a basis of the intersection of the integral row spaces of *m* and *n*.

```
gap> nat:=[[5,7,2],[4,2,5],[7,1,4]];;
gap> BaseIntMat(nat);
[ [ 1, 1, 15 ], [ 0, 2, 55 ], [ 0, 0, 64 ] ]
gap> BaseIntersectionIntMats(mat,nat);
[ [ 1, 5, 509 ], [ 0, 6, 869 ], [ 0, 0, 960 ] ]
```

35.6 ComplementIntMat

`ComplementIntMat(` *full*`,` *sub* `)`

Let *full* be a list of integer vectors generating an Integral module M and *sub* a list of vectors defining a submodule S. This function computes a free basis for M that extends S, that is, if the dimension of S is n it determines a basis $\{b_1,\ldots,b_m\}$ for M, as well as n integers x_i such that $x_i \mid x_j$ for $i < j$ and the n vectors $s_i := x_i \cdot b_i$ for $i = 1,\ldots,n$ form a basis for S.

It returns a record with the following components:

`complement`
: the vectors b_{n+1} up to b_m (they generate a complement to S).

`sub`
: the vectors s_i (a basis for S).

`moduli`
: the factors x_i.

```
gap> m:=IdentityMat(3);;
gap> n:=[[1,2,3],[4,5,6]];;
```

```
gap> ComplementIntMat(m,n);
rec( complement := [ [ 0, 0, 1 ] ], sub := [ [ 1, 2, 3 ], [ 0, 3, 6 ] ],
  moduli := [ 1, 3 ] )
```

35.7 TriangulizedIntegerMat

`TriangulizedIntegerMat(` *mat* `)`

Computes an integral upper triangular form of a matrix with integer entries.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> TriangulizedIntegerMat(m);
[ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
```

35.8 TriangulizedIntegerMatTransform

`TriangulizedIntegerMatTransform(` *mat* `)`

Computes an integral upper triangular form of a matrix with integer entries. It returns a record with a component `normal` (a matrix in upper triangular form) and a component `rowtrans` that gives the transformations done to the original matrix to bring it into upper triangular form.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> n:=TriangulizedIntegerMatTransform(m);
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1, rowtrans := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55
] ] )
gap> n.rowtrans*m=n.normal;
true
```

35.9 TriangulizeIntegerMat

`TriangulizeIntegerMat(` *mat* `)`

Changes *mat* to be in upper triangular form. (The result is the same as that of `TriangulizedIntegerMat`, but *mat* will be modified, thus using less memory.)

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> TriangulizeIntegerMat(m); m;
[ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
```

35.10 HermiteNormalFormIntegerMat

`HermiteNormalFormIntegerMat(` *mat* `)`

This operation computes the Hermite normal form of a matrix *mat* with integer entries. The Hermite Normal Form (HNF), H of an integer matrix, A is a row equivalent upper triangular form such that all off-diagonal entries are reduced modulo the diagonal entry of the column they are in. There exists a unique unimodular matrix Q such that $QA = H$.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> HermiteNormalFormIntegerMat(m);
[ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
```

35.11 HermiteNormalFormIntegerMatTransform

`HermiteNormalFormIntegerMatTransform(` *mat* `)`

This operation computes the Hermite normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix H of the Hermite normal form) and `rowtrans` (a unimodular matrix Q) such that $Q mat = H$

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> n:=HermiteNormalFormIntegerMatTransform(m);
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ] )
gap> n.rowtrans*m=n.normal;
true
```

35.12 SmithNormalFormIntegerMat

`SmithNormalFormIntegerMat(` *mat* `)`

This operation computes the Smith normal form of a matrix *mat* with integer entries. The Smith Normal Form,S, of an integer matrix A is the unique equivalent diagonal form with S_i dividing S_j for $i < j$. There exist unimodular integer matrices P, Q such that $PAQ = S$.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> SmithNormalFormIntegerMat(m);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

35.13 SmithNormalFormIntegerMatTransforms

`SmithNormalFormIntegerMatTransforms(` *mat* `)`

This operation computes the Smith normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix S), `rowtrans` (a matrix P), and `coltrans` (a matrix Q) such that $PmatQ = S$.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> n:=SmithNormalFormIntegerMatTransforms(m);
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ] )
```

```
gap> n.rowtrans*m*n.coltrans=n.normal;
true
```

35.14 DiagonalizeIntMat

`DiagonalizeIntMat(` *mat* `)`

This function changes *mat* to its SNF. (The result is the same as that of `SmithNormalFormIntegerMat`, but *mat* will be modified, thus using less memory.)

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> DiagonalizeIntMat(m);m;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

35.15 NormalFormIntMat

All the previous routines build on the following "workhorse routine

`NormalFormIntMat(` *mat*, *options* `)`

This general operation for computation of various Normal Forms is probably the most efficient.

Options bit values

- 0/1 Triangular Form / Smith Normal Form.
- 2 Reduce off diagonal entries.
- 4 Row Transformations.
- 8 Col Transformations.
- 16 Destructive (the original matrix may be destroyed)

Compute a Triangular, Hermite or Smith form of the $n \times m$ integer input matrix A. Optionally, compute $n \times n$ and $m \times m$ unimodular transforming matrices Q, P which satisfy $QA = H$ or $QAP = S$.

Note option is a value ranging from 0 - 15 but not all options make sense (eg reducing off diagonal entries with SNF option selected already). If an option makes no sense it is ignored.

Returns a record with component `normal` containing the computed normal form and optional components `rowtrans` and/or `coltrans` which hold the respective transformation matrix. Also in the record are components holding the sign of the determinant, signdet, and the Rank of the matrix, rank.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> NormalFormIntMat(m,0);  # Triangular, no transforms
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ], rank := 3,
  signdet := 1 )
gap> NormalFormIntMat(m,6);  # Hermite Normal Form with row transforms
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
```

```
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ] )
gap> NormalFormIntMat(m,13); # Smith Normal Form with both transforms
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ] )
gap> last.rowtrans*m*last.coltrans;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

35.16 AbelianInvariantsOfList

`AbelianInvariantsOfList(` *list* `)`

Given a list of positive integers, this routine returns a list of prime powers, such that the prime power factors of the entries in the list are returned in sorted form.

```
gap> AbelianInvariantsOfList([4,6,2,12]);
[ 2, 2, 3, 3, 4, 4 ]
```

35.17 Determinant of an integer matrix

`DeterminantIntMat(` *mat* `)`

Computes the determinant of an integer matrix using the same strategy as `NormalFormIntMat`. This method is faster in general for matrices greater than 20×20 but quite a lot slower for smaller matrices. It therefore passes the work to the more general `DeterminantMat` (see 34.15) for these smaller matrices.

35.18 Diaconis-Graham normal form

`DiaconisGraham(` *mat*, *moduli*`)`

Diaconis and Graham (see [DG99]) defined a normal form for generating sets of abelian groups. Here *moduli* should be a list of positive integers such that `moduli[i+1]` divides `moduli[i]` for all `i`, representing the abelian group $A = \mathbb{Z}/moduli[1] \times \ldots \times \mathbb{Z}/moduli[n]$. The integral matrix m should have n columns where `n=Length(moduli)`, and each line (with the i-th element taken `mod moduli[i]`) represents an element of the group A.

The function returns `false` if the set of elements of A represented by the lines of m does not generate A. Otherwise it returns a record `r` with fields

`r.normal` the Diaconis-Graham normal form, a matrix of same shape as `m` where either the first `n` lines are the identity matrix and the remaining lines are `0`, or `Length(m)=n` and `.normal` differs from the identity matrix only in the entry `.normal[n][n]`, which is prime to `moduli[n]`.

`r.rowtrans` a unimodular matrix such that `r.normal=List(r.rowtrans*m,v->Zip(v,moduli, function(x,y)return x mod y;end))`

Here is an example:

```
gap> DiaconisGraham([[3,0],[4,1]],[10,5]);
rec(
  rowtrans := [ [ -13, 10 ], [ 4, -3 ] ],
  normal := [ [ 1, 0 ], [ 0, 2 ] ] )
```

Chapter 36

Matrix Rings

A **matrix ring** is a ring of square matrices (see chapter 34). In GAP3 you can define matrix rings of matrices over each of the fields that GAP3 supports, i.e., the rationals, cyclotomic extensions of the rationals, and finite fields (see chapters 12, 13, and 18).

You define a matrix ring in GAP3 by calling `Ring` (see 5.2) passing the generating matrices as arguments.

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> m := Ring( m1, m2 );
Ring( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
    [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
    [ Z(3)^0, 0*Z(3), Z(3) ] ] )
gap> Size( m );
2187
```

However, currently GAP3 can only compute with finite matrix rings with a multiplicative neutral element (a **one**). Also computations with large matrix rings are not done very efficiently. We hope to improve this situation in the future, but currently you should be careful not to try too large matrix rings.

Because matrix rings are just a special case of domains all the set theoretic functions such as `Size` and `Intersection` are applicable to matrix rings (see chapter 4 and 36.1).

Also matrix rings are of course rings, so all ring functions such as `Units` and `IsIntegralRing` are applicable to matrix rings (see chapter 5 and 36.2).

36.1 Set Functions for Matrix Rings

All set theoretic functions described in chapter 4 use their default function for matrix rings currently. This means, for example, that the size of a matrix ring is computed by computing the set of all elements of the matrix ring with an orbit-like algorithm. Thus you should not try to work with too large matrix rings.

36.2 Ring Functions for Matrix Rings

As already mentioned in the introduction of this chapter matrix rings are after all rings. All ring functions such as `Units` and `IsIntegralRing` are thus applicable to matrix rings. This section describes how these functions are implemented for matrix rings. Functions not mentioned here inherit the default group methods described in the respective sections.

`IsUnit(` *R*`,` *m* `)`

A matrix is a unit in a matrix ring if its rank is maximal (see 34.16).

Chapter 37

Matrix Groups

A **matrix group** is a group of invertable square matrices (see chapter 34). In GAP3 you can define matrix groups of matrices over each of the fields that GAP3 supports, i.e., the rationals, cyclotomic extensions of the rationals, and finite fields (see chapters 12, 13, and 18).

You define a matrix group in GAP3 by calling `Group` (see 7.9) passing the generating matrices as arguments.

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> m := Group( m1, m2 );
Group( [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ],
  [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, 0*Z(3), Z(3) ] ] )
```

However, currently GAP3 can only compute with finite matrix groups. Also computations with large matrix groups are not done very efficiently. We hope to improve this situation in the future, but currently you should be careful not to try too large matrix groups.

Because matrix groups are just a special case of domains all the set theoretic functions such as `Size` and `Intersection` are applicable to matrix groups (see chapter 4 and 37.1).

Also matrix groups are of course groups, so all the group functions such as `Centralizer` and `DerivedSeries` are applicable to matrix groups (see chapter 7 and 37.2).

37.1 Set Functions for Matrix Groups

As already mentioned in the introduction of this chapter matrix groups are domains. All set theoretic functions such as `Size` and `Intersections` are thus applicable to matrix groups. This section describes how these functions are implemented for matrix groups. Functions not mentioned here either inherit the default group methods described in 7.114 or the default method mentioned in the respective sections.

To compute with a matrix group m, GAP3 computes the operation of the matrix group on the underlying vector space (more precisely the union of the orbits of the parent of m on the standard basis vectors). Then it works with the thus defined permutation group p, which is of course isomorphic to m, and finally translates the results back into the matrix group.

obj `in` *m*

To test if an object *obj* lies in a matrix group m, GAP3 first tests whether *obj* is a invertable square matrix of the same dimensions as the matrices of m. If it is, GAP3 tests whether *obj* permutes the vectors in the union of the orbits of m on the standard basis vectors. If it does, GAP3 takes this permutation and tests whether it lies in p.

`Size(` *m* `)`

To compute the size of the matrix group m, GAP3 computes the size of the isomorphic permutation group p.

`Intersection(` *m1* `,` *m2* `)`

To compute the intersection of two subgroups *m1* and *m2* with a common parent matrix group m, GAP3 intersects the images of the corresponding permutation subgroups *p1* and *p2* of p. Then it translates the generators of the intersection of the permutation subgroups back to matrices. The intersection of *m1* and *m2* is the subgroup of m generated by those matrices. If *m1* and *m2* do not have a common parent group, or if only one of them is a matrix group and the other is a set of matrices, the default method is used (see 4.12).

37.2 Group Functions for Matrix Groups

As already mentioned in the introduction of this chapter matrix groups are after all group. All group functions such as `Centralizer` and `DerivedSeries` are thus applicable to matrix groups. This section describes how these functions are implemented for matrix groups. Functions not mentioned here either inherit the default group methods described in the respective sections.

To compute with a matrix group m, GAP3 computes the operation of the matrix group on the underlying vector space (more precisely, if the vector space is small enough, it enumerates the space and acts on the whole space. Otherwise it takes the union of the orbits of the parent of m on the standard basis vectors). Then it works with the thus defined permutation group p, which is of course isomorphic to m, and finally translates the results back into the matrix group.

`Centralizer(` *m*, *u* `)`
`Normalizer(` *m*, *u* `)`
`SylowSubgroup(` *m*, *p* `)`
`ConjugacyClasses(` *m* `)`

This functions all work by solving the problem in the permutation group *p* and translating the result back.

`PermGroup(` *m* `)`

This function simply returns the permutation group defined above.

`Stabilizer(` *m*`,` *v* `)`

The stabilizer of a vector *v* that lies in the union of the orbits of the parent of *m* on the standard basis vectors is computed by finding the stabilizer of the corresponding point in the permutation group *p* and translating this back. Other stabilizers are computed with the default method (see 8.24).

`RepresentativeOperation(` *m*`,` *v1*`,` *v2* `)`

If *v1* and *v2* are vectors that both lie in the union of the orbits of the parent group of *m* on the standard basis vectors, `RepresentativeOperation` finds a permutation in *p* that takes the point corresponding to *v1* to the point corresponding to *v2*. If no such permutation exists, it returns `false`. Otherwise it translates the permutation back to a matrix.

`RepresentativeOperation(` *m*`,` *m1*`,` *m2* `)`

If *m1* and *m2* are matrices in *m*, `RepresentativeOperation` finds a permutation in *p* that conjugates the permutation corresponding to *m1* to the permutation corresponding to *m2*. If no such permutation exists, it returns `false`. Otherwise it translates the permutation back to a matrix.

37.3 Matrix Group Records

A group is represented by a record that contains information about the group. A matrix group record contains the following components in addition to those described in section 7.118.

`isMatGroup`
: always `true`.

If a permutation representation for a matrix group *m* is known it is stored in the following components.

`permGroupP`
: contains the permutation group representation of *m*.

`permDomain`
: contains the union of the orbits of the parent of *m* on the standard basis vectors.

Chapter 38

Group Libraries

When you start GAP3 it already knows several groups. Currently GAP3 initially knows the following groups:

- some basic groups, such as cyclic groups or symmetric groups (see 38.1),
- the primitive permutation groups of degree at most 50 (see 38.5),
- the transitive permutation groups of degree at most 15 (see 38.6),
- the solvable groups of size at most 100 (see 38.7),
- the 2-groups of size at most 256 (see 38.8),
- the 3-groups of size at most 729 (see 38.9),
- the irreducible solvable subgroups of $GL(n,p)$ for $n > 1$ and $p^n < 256$ (see 38.10),
- the finite perfect groups of size at most 10^6 (excluding 11 sizes) (see 38.11),
- the irreducible maximal finite integral matrix groups of dimension at most 24 (see 38.12),
- the crystallographic groups of dimension at most 4 (see 38.13).
- the groups of order at most 1000 except for 512 and 768 (see 38.14).

Each of the set of groups above is called a **library**. The whole set of groups that GAP3 knows initially is called the GAP3 **collection of group libraries**. There is usually no relation between the groups in the different libraries.

Several of the libraries are accessed in a uniform manner. For each of these libraries there is a so called **selection function** that allows you to select the list of groups that satisfy given criterias from a library. The **example function** allows you to select one group that satisfies given criteria from the library. The low-level **extraction function** allows you to extract a single group from a library, using a simple indexing scheme. These functions are described in the sections 38.2, 38.3, and 38.4.

Note that a system administrator may choose to install all, or only a few, or even none of the libraries. So some of the libraries mentioned below may not be available on your installation.

38.1 The Basic Groups Library

`CyclicGroup(` *n* `)`
`CyclicGroup(` *D*, *n* `)`

In the first form `CyclicGroup` returns the cyclic group of size n as a permutation group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `CyclicGroup` returns the cyclic group of size n as a group of elements of that type.

```
gap> c12 := CyclicGroup( 12 );
Group( ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12) )
gap> c105 := CyclicGroup( AgWords, 5*3*7 );
Group( c105_1, c105_2, c105_3 )
gap> Order(c105,c105.1); Order(c105,c105.2); Order(c105,c105.3);
105
35
7
```

`AbelianGroup(` *sizes* `)`
`AbelianGroup(` *D*, *sizes* `)`

In the first form `AbelianGroup` returns the abelian group $C_{sizes[1]} * C_{sizes[2]} * ... * C_{sizes[n]}$, where *sizes* must be a list of positive integers, as a permutation group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `AbelianGroup` returns the abelian group as a group of elements of this type.

```
gap> g := AbelianGroup( AgWords, [ 2, 3, 7 ] );
Group( a, b, c )
gap> Size( g );
42
gap> IsAbelian( g );
true
```

The default function `GroupElementsOps.AbelianGroup` uses the functions `CyclicGroup` and `DirectProduct` (see 7.99) to construct the abelian group.

`ElementaryAbelianGroup(` *n* `)`
`ElementaryAbelianGroup(` *D*, *n* `)`

In the first form `ElementaryAbelianGroup` returns the elementary abelian group of size n as a permutation group. n must be a positive prime power of course. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `ElementaryAbelianGroup` returns the elementary abelian group as a group of elements of this type.

```
gap> ElementaryAbelianGroup( 16 );
Group( (1,2), (3,4), (5,6), (7,8) )
gap> ElementaryAbelianGroup( AgWords, 3 ^ 10 );
Group( m59049_1, m59049_2, m59049_3, m59049_4, m59049_5, m59049_6,
m59049_7, m59049_8, m59049_9, m59049_10 )
```

The default function `GroupElementsOps.ElementaryAbelianGroup` uses `CyclicGroup` and `DirectProduct` (see 7.99 to construct the elementary abelian group.

`DihedralGroup(` *n* `)`
`DihedralGroup(` *D*, *n* `)`

In the first form `DihedralGroup` returns the dihedral group of size n as a permutation group. n must be a positive even integer. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `DihedralGroup` returns the dihedral group as a group of elements of this type.

```
gap> DihedralGroup( 12 );
Group( (1,2,3,4,5,6), (2,6)(3,5) )
```

`PolyhedralGroup(` *p*, *q* `)`
`PolyhedralGroup(` *D*, *p*, *q* `)`

In the first form `PolyhedralGroup` returns the polyhedral group of size p `*` q as a permutation group. p and q must be positive integers and there must exist a nontrivial p-th root of unity modulo every prime factor of q. In the second form D must be a domain of group elements, e.g., `Permutations` or `Words`, and `PolyhedralGroup` returns the polyhedral group as a group of elements of this type.

```
gap> PolyhedralGroup( 3, 13 );
Group( ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13), ( 2, 4,10)( 3, 7, 6)
( 5,13,11)( 8, 9,12) )
gap> Size( last );
39
```

`SymmetricGroup(` *d* `)`
`SymmetricGroup(` *D*, *d* `)`

In the first form `SymmetricGroup` returns the symmetric group of degree d as a permutation group. d must be a positive integer. In the second form D must be a domain of group elements, e.g., `Permutations` or `Words`, and `SymmetricGroup` returns the symmetric group as a group of elements of this type.

```
gap> SymmetricGroup( 8 );
Group( (1,8), (2,8), (3,8), (4,8), (5,8), (6,8), (7,8) )
gap> Size( last );
40320
```

`AlternatingGroup(` *d* `)`
`AlternatingGroup(` *D*, *d* `)`

In the first form `AlternatingGroup` returns the alternating group of degree d as a permutation group. d must be a positive integer. In the second form D must be a domain of group elements, e.g., `Permutations` or `Words`, and `AlternatingGroup` returns the alternating group as a group of elements of this type.

```
gap> AlternatingGroup( 8 );
Group( (1,2,8), (2,3,8), (3,4,8), (4,5,8), (5,6,8), (6,7,8) )
gap> Size( last );
20160
```

`GeneralLinearGroup(` n`,` q `)`
`GeneralLinearGroup(` D`,` n`,` q `)`

In the first form `GeneralLinearGroup` returns the general linear group $GL(n, q)$ as a matrix group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `GeneralLinearGroup` returns $GL(n, q)$ as a group of elements of that type.

```
gap> g := GeneralLinearGroup( 2, 4 ); Size( g );
GL(2,4)
180
```

`SpecialLinearGroup(` n`,` q `)`
`SpecialLinearGroup(` D`,` n`,` q `)`

In the first form `SpecialLinearGroup` returns the special linear group $SL(n, q)$ as a matrix group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `SpecialLinearGroup` returns $SL(n, q)$ as a group of elements of that type.

```
gap> g := SpecialLinearGroup( 3, 4 ); Size( g );
SL(3,4)
60480
```

`SymplecticGroup(` n`,` q `)`
`SymplecticGroup(` D`,` n`,` q `)`

In the first form `SymplecticGroup` returns the symplectic group $SP(n, q)$ as a matrix group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `SymplecticGroup` returns $SP(n, q)$ as a group of elements of that type.

```
gap> g := SymplecticGroup( 4, 2 ); Size( g );
SP(4,2)
720
```

`GeneralUnitaryGroup(` n`,` q `)`
`GeneralUnitaryGroup(` D`,` n`,` q `)`

In the first form `GeneralUnitaryGroup` returns the general unitary group $GU(n, q)$ as a matrix group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `GeneralUnitaryGroup` returns $GU(n, q)$ as a group of elements of that type.

```
gap> g := GeneralUnitaryGroup( 3, 3 ); Size( g );
GU(3,3)
24192
```

`SpecialUnitaryGroup(` n`,` q `)`
`SpecialUnitaryGroup(` D`,` n`,` q `)`

In the first form `SpecialUnitaryGroup` returns the special unitary group $SU(n, q)$ as a matrix group. In the second form D must be a domain of group elements, e.g., `Permutations` or `AgWords`, and `SpecialUnitaryGroup` returns $SU(n, q)$ as a group of elements of that type.

```
gap> g := SpecialUnitaryGroup( 3, 3 ); Size( g );
SU(3,3)
6048
```

`MathieuGroup(` *d* `)`

`MathieuGroup` returns the Mathieu group of degree *d* as a permutation group. *d* is expected to be 11, 12, 22, 23, or 24.

```
gap> g := MathieuGroup( 12 ); Size( g );
Group( ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11), ( 3, 7,11, 8)
( 4,10, 5, 6), ( 1,12)( 2,11)( 3, 6)( 4, 8)( 5, 9)( 7,10) )
95040
```

38.2 Selection Functions

`All`*Library*`Groups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

For each group library there is a **selection function**. This function allows you to select all groups from the library that have a given set of properties.

The name of the selection functions always begins with `All` and always ends with `Groups`. Inbetween is a name that hints at the nature of the group library. For example, the selection function for the library of all primitive groups of degree at most 50 (see 38.5) is called `AllPrimitiveGroups`, and the selection function for the library of all 2-groups of size at most 256 (see 38.8) is called `AllTwoGroups`.

These functions take an arbitrary number of pairs of arguments. The first argument in such a pair is a function that can be applied to the groups in the library, and the second argument is either a single value that this function must return in order to have this group included in the selection, or a list of such values.

For example

```
AllPrimitiveGroups( DegreeOperation,  [10..15],
                    Size,             [1..100],
                    IsAbelian,        false    );
```

should return a list of all primitive groups with degree between 10 and 15 and size less than 100 that are not abelian.

Thus the `AllPrimitiveGroups` behaves as if it was implemented by a function similar to the one defined below, where `PrimitiveGroupsList` is a list of all primitive groups. Note, in the definition below we assume for simplicity that `AllPrimitiveGroups` accepts exactly 4 arguments. It is of course obvious how to change this definition so that the function would accept a variable number of arguments.

```
AllPrimitiveGroups := function ( fun1, val1, fun2, val2 )
    local   groups, g, i;
    groups := [];
    for i  in [ 1 .. Length( PrimitiveGroupsList ) ] do
        g := PrimitiveGroupsList[i];
        if      fun1(g) = val1  or IsList(val1) and fun1(g) in val1
            and fun2(g) = val2  or IsList(val2) and fun2(g) in val2
```

```
            then
                Add( groups, g );
            fi;
        od;
        return groups;
    end;
```

Note that the real selection functions are considerably more difficult, to improve the efficiency. Most important, each recognizes a certain set of functions and handles those properties using an index (see 1.26).

38.3 Example Functions

`One`*Library*`Group(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

For each group library there is a **example function**. This function allows you to find one group from the library that has a given set of properties.

The name of the example functions always begins with `One` and always ends with `Group`. Inbetween is a name that hints at the nature of the group library. For example, the example function for the library of all primitive groups of degree at most 50 (see 38.5) is called `OnePrimitiveGroup`, and the example function for the library of all 2-groups of size at most 256 (see 38.8) is called `OneTwoGroup`.

These functions take an arbitrary number of pairs of arguments. The first argument in such a pair is a function that can be applied to the groups in the library, and the second argument is either a single value that this function must return in order to have this group returned by the example function, or a list of such values.

For example

```
    OnePrimitiveGroup( DegreeOperation,  [10..15],
                        Size,             [1..100],
                        IsAbelian,        false    );
```

should return one primitive group with degree between 10 and 15 and size size less than 100 that is not abelian.

Thus the `OnePrimitiveGroup` behaves as if it was implemented by a function similar to the one defined below, where `PrimitiveGroupsList` is a list of all primitive groups. Note, in the definition below we assume for simplicity that `OnePrimitiveGroup` accepts exactly 4 arguments. It is of course obvious how to change this definition so that the function would accept a variable number of arguments.

```
    OnePrimitiveGroup := function ( fun1, val1, fun2, val2 )
        local    g, i;
        for i  in [ 1 .. Length( PrimitiveGroupsList ) ] do
            g := PrimitiveGroupsList[i];
            if      fun1(g) = val1  or IsList(val1) and fun1(g) in val1
                and fun2(g) = val2  or IsList(val2) and fun2(g) in val2
            then
                return g;
            fi;
        od;
```

```
        return false;
    end;
```

Note that the real example functions are considerably more difficult, to improve the efficiency. Most important, each recognizes a certain set of functions and handles those properties using an index (see 1.26).

38.4 Extraction Functions

For each group library there is an **extraction function**. This function allows you to extract single groups from the library.

The name of the extraction function always ends with `Group` and begins with a name that hints at the nature of the library. For example the extraction function for the library of primitive groups (see 38.5) is called `PrimitiveGroup`, and the extraction function for the library of all 2-groups of size at most 256 (see 38.8) is called `TwoGroup`.

What arguments the extraction function accepts, and how they are interpreted is described in the sections that describe the individual group libraries.

For example

```
    PrimitiveGroup( 10, 4 );
```

returns the 4-th primitive group of degree 10.

The reason for the extraction function is as follows. A group library is usually not stored as a list of groups. Instead a more compact representation for the groups is used. For example the groups in the library of 2-groups are represented by 4 integers. The extraction function hides this representation from you, and allows you to access the group library as if it was a table of groups (two dimensional in the above example).

38.5 The Primitive Groups Library

This group library contains all primitive permutation groups of degree at most 50. There are a total of 406 such groups. Actually to be a little bit more precise, there are 406 inequivalent primitive operations on at most 50 points. Quite a few of the 406 groups are isomorphic.

`AllPrimitiveGroups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`AllPrimitiveGroups` returns a list containing all primitive groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first being a function, which will be applied to all groups in the library, and the second being a value or a list of values, that this function must return in order to have this group included in the list returned by `AllPrimitiveGroups`.

The first argument must be `DegreeOperation` and the second argument either a degree or a list of degrees, otherwise `AllPrimitiveGroups` will print a warning to the effect that the library contains only groups with degrees between 1 and 50.

```
gap> l := AllPrimitiveGroups( Size, 120, IsSimple, false );
#W  AllPrimitiveGroups: degree automatically restricted to [1..50]
[ S(5), PGL(2,5), S(5) ]
gap> List( l, g -> g.generators );
[ [ (1,2,3,4,5), (1,2) ], [ (1,2,3,4,5), (2,3,5,4), (1,6)(3,4) ],
  [ ( 1, 8)( 2, 5, 6, 3)( 4, 9, 7,10), ( 1, 5, 7)( 2, 9, 4)( 3, 8,10)
     ] ]
```

`OnePrimitiveGroup(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`OnePrimitiveGroup` returns one primitive group that has the properties given as argument. Each property is specified by passing a pair of arguments, the first being a function, which will be applied to all groups in the library, and the second being a value or a list of values, that this function must return in order to have this group returned by `OnePrimitiveGroup`. If no such group exists, `false` is returned.

The first argument must be `DegreeOperation` and the second argument either a degree or a list of degrees, otherwise `OnePrimitiveGroup` will print a warning to the effect that the library contains only groups with degrees between 1 and 50.

```
gap> g := OnePrimitiveGroup( DegreeOperation,5, IsSolvable,false );
A(5)
gap> Size( g );
60
```

`AllPrimitiveGroups` and `OnePrimitiveGroup` recognize the following functions and handle them usually quite efficient. `DegreeOperation`, `Size`, `Transitivity`, and `IsSimple`. You should pass those functions first, e.g., it is more efficient to say `AllPrimitiveGroups( Size,120 , IsAbelian,false )` than to say `AllPrimitiveGroups( IsAbelian,false, Size,120 )` (see 1.26).

`PrimitiveGroup(` *deg* `,` *nr* `)`

`PrimitiveGroup` returns the *nr*-th primitive group of degree *deg*. Both *deg* and *nr* must be positive integers. The primitive groups of equal degree are sorted with respect to their size, so for example `PrimitiveGroup(` *deg*`, 1 )` is the smallest primitive group of degree *deg*, e.g, the cyclic group of size *deg*, if *deg* is a prime. Primitive groups of equal degree and size are in no particular order.

```
gap>  g := PrimitiveGroup( 8, 1 );
AGL(1,8)
gap> g.generators;
[ (1,2,3,4,5,6,7), (1,8)(2,4)(3,7)(5,6) ]
```

Apart from the usual components described in 7.118, the group records returned by the above functions have the following components.

`transitivity`
: degree of transitivity of G.

`isSharpTransitive`
: `true` if G is sharply G`.transitivity`-fold transitive and `false` otherwise.

`isKPrimitive`
: `true` if G is k-fold primitive, and `false` otherwise.

`isOdd`
: `false` if G is a subgroup of the alternating group of degree G`.degree` and `true` otherwise.

`isFrobeniusGroup`
: `true` if G is a Frobenius group and `false` otherwise.

This library was computed by Charles Sims. The list of primitive permutation groups of degree at most 20 was published in [Sim70]. The library was brought into GAP3 format by Martin Schönert. He assumes the responsibility for all mistakes.

38.6 The Transitive Groups Library

The transitive groups library contains representatives for all transitive permutation groups of degree at most 22. Two permutations groups of the same degree are considered to be equivalent, if there is a renumbering of points, which maps one group into the other one. In other words, if they lie in the save conjugacy class under operation of the full symmetric group by conjugation.

There are a total of 4945 such groups up to degree 22.

`AllTransitiveGroups(` *fun1*`,` *val1*`,` *fun2*`,` *val2*`, ... )`

`AllTransitiveGroups` returns a list containing all transitive groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first being a function, and the second being a value or a list of values. `AllTransitiveGroups` will return all groups from the transitive groups library, for which all specified functions have the specified values.

If the degree is not restricted to 22 at most, `AllTransitiveGroups` will print a warning.

`OneTransitiveGroup(` *fun1*`,` *val1*`,` *fun2*`,` *val2*`, ... )`

`OneTransitiveGroup` returns one transitive group that has the properties given as argument. Each property is specified by passing a pair of arguments, the first being a function, and the second being a value or a list of values. `OneTransitiveGroup` will return one groups from the transitive groups library, for which all specified functions have the specified values. If no such group exists, `false` is returned.

If the degree is not restricted to 22 at most, `OneTransitiveGroup` will print a warning.

`AllTransitiveGroups` and `OneTransitiveGroup` recognize the following functions and get the corresponding properties from a precomputed list to speed up processing:
`DegreeOperation`, `Size`, `Transitivity`, and `IsPrimitive`. You do not need to pass those functions first, as the selection function picks the these properties first.

`TransitiveGroup(` *deg*`,` *nr* `)`

`TransitiveGroup` returns the *nr*-th transitive group of degree *deg*. Both *deg* and *nr* must be positive integers. The transitive groups of equal degree are sorted with respect to their size, so for example `TransitiveGroup(` *deg*`, 1 )` is the smallest transitive group of degree *deg*, e.g, the cyclic group of size *deg*, if *deg* is a prime. The ordering of the groups corresponds to the list in Butler/McKay [BM83].

This library was computed by Gregory Butler, John McKay, Gordon Royle and Alexander Hulpke. The list of transitive groups up to degree 11 was published in [BM83], the list of degree 12 was published in [Roy87], degree 14 and 15 were published in [But93].

The library was brought into GAP3 format by Alexander Hulpke, who is responsible for all mistakes.

```
gap> TransitiveGroup(10,22);
S(5)[x]2
gap> l:=AllTransitiveGroups(DegreeOperation,12,Size,1440,
```

```
> IsSolvable,false);
[ S(6)[x]2, M_10.2(12) = A_6.E_4(12) = [S_6[1/720]{M_10}S_6]2 ]
gap> List(l,IsSolvable);
[ false, false ]
```

`TransitiveIdentification(` G `)`

Let G be a permutation group, acting transitively on a set of up to 22 points. Then `TransitiveIdentification` will return the position of this group in the transitive groups library. This means, if G operates on m points and `TransitiveIdentification` returns n, then G is permutation isomorphic to the group `TransitiveGroup(m,n)`.

```
gap> TransitiveIdentification(Group((1,2),(1,2,3)));
2
```

38.7 The Solvable Groups Library

GAP3 has a library of the 1045 solvable groups of size between 2 and 100. The groups are from lists computed by M. Hall and J. K. Senior (size 64, see [HS64]), R. Laue (size 96, see [Lau82]) and J. Neubüser (other sizes, see [Neu67]).

`AllSolvableGroups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`AllSolvableGroups` returns a list containing all solvable groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first being a function, which will be applied to all the groups in the library, and the second being a value or a list of values, that this function must return in order to have this group included in the list returned by `AllSolvableGroups`.

```
gap> AllSolvableGroups(Size,24,IsNontrivialDirectProduct,false);
[ 12.2, grp_24_11, D24, Q8+S3, Sl(2,3), S4 ]
```

`OneSolvableGroup(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`OneSolvableGroup` returns a solvable group with the specified properties. Each property is specified by passing a pair of arguments, the first being a function, which will be applied to all the groups in the library, and the second being a value or a list of values, that this function must return in order to have this group returned by `OneSolvableGroup`. If no such group exists, `false` is returned.

```
gap> OneSolvableGroup(Size,100,x->Size(DerivedSubgroup(x)),10);
false
gap> OneSolvableGroup(Size,24,IsNilpotent,false);
S3x2^2
```

`AllSolvableGroups` and `OneSolvableGroup` recognize the following functions and handle them usually very efficiently:`Size`, `IsAbelian`, `IsNilpotent`, and `IsNonTrivialDirectProduct`.

`SolvableGroup(` *size* `,` *nr* `)`

`SolvableGroup` returns the *nr*-th group of size *size* in the library. `SolvableGroup` will signal an error if *size* is not between 2 and 100, or if *nr* is larger than the number of solvable groups of size *size*. The group is returned as finite polycyclic group (see 25).

```
gap> SolvableGroup( 32 , 15 );
Q8x4
```

38.8 The 2-Groups Library

The library of 2-groups contains all the 2-groups of size dividing 256. There are a total of 58760 such groups, 1 of size 2, 2 of size 4, 5 of size 8, 14 of size 16, 51 of size 32, 267 of size 64, 2328 of size 128, and 56092 of size 256.

`AllTwoGroups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`AllTwoGroups` returns the list of all the 2-groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first is a function that can be applied to each group, the second is either a single value or a list of values that the function must return in order to select that group.

```
gap> l := AllTwoGroups( Size, 256, Rank, 3, pClass, 2 );
[ Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ),
  Group( a1, a2, a3, a4, a5, a6, a7, a8 ) ]
gap> List( l, g -> Length( ConjugacyClasses( g ) ) );
[ 112, 88, 88, 88 ]
```

`OneTwoGroup(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`OneTwoGroup` returns a single 2-group that has the properties given as arguments. Each property is specified by passing a pair of arguments, the first is a function that can be applied to each group, the second is either a single value or a list of values that the function must return in order to select that group.

```
gap> g := OneTwoGroup( Size, [64..128], Rank, [2..3], pClass, 5 );
#I  size restricted to [ 64, 128 ]
Group( a1, a2, a3, a4, a5, a6 )
gap> Size( g );
64
gap> Rank( g );
2
```

`AllTwoGroups` and `OneTwoGroup` recognize the following functions and handle them usually very efficiently. `Size`, `Rank` for the rank of the Frattini quotient of the group, and `pClass` for the exponent-p class of the group. Note that `Rank` and `pClass` are dummy functions that can be used only in this context, i.e., they can not be applied to arbitrary groups.

`TwoGroup(` *size* `,` *nr* `)`

`TwoGroup` returns the *nr*-th group of size *size*. The group is returned as a finite polycyclic group (see 25). `TwoGroup` will signal an error if *size* is not a power of 2 between 2 and 256, or *nr* is larger than the number of groups of size *size*.

Within each size the following criteria have been used, in turn, to determine the index position of a group in the list

1 increasing generator number;

2 increasing exponent-2 class;

3 the position of its parent in the list of groups of appropriate size;

4 the list in which the Newman and O'Brien implementation of the p-group generation algorithm outputs the immediate descendants of a group.

```
gap> g := TwoGroup( 32, 45 );
Group( a1, a2, a3, a4, a5 )
gap> Rank( g );
4
gap> pClass( g );
2
gap> g.abstractRelators;
[ a1^2*a5^-1, a2^2, a2^-1*a1^-1*a2*a1, a3^2, a3^-1*a1^-1*a3*a1,
  a3^-1*a2^-1*a3*a2, a4^2, a4^-1*a1^-1*a4*a1, a4^-1*a2^-1*a4*a2,
  a4^-1*a3^-1*a4*a3, a5^2, a5^-1*a1^-1*a5*a1, a5^-1*a2^-1*a5*a2,
  a5^-1*a3^-1*a5*a3, a5^-1*a4^-1*a5*a4 ]
```

Apart from the usual components described in 7.118, the group records returned by the above functions have the following components.

`rank`
: rank of Frattini quotient of G.

`pclass`
: exponent-p class of G.

`abstractGenerators`
: a list of abstract generators of G (see 22.1).

`abstractRelators`
: a list of relators of G stored as words in the abstract generators.

Descriptions of the algorithms used in constructing the library data may be found in [O´Br90, O´Br91]. Using these techniques, a library was first prepared in 1987 by M.F. Newman and E.A. O'Brien; a partial description may be found in [NO89].

The library was brought into the GAP3 format by Werner Nickel, Alice Niemeyer, and E.A. O'Brien.

38.9 The 3-Groups Library

The library of 3-groups contains all the 3-groups of size dividing 729. There are a total of 594 such groups, 1 of size 3, 2 of size 9, 5 of size 27, 15 of size 81, 67 of size 243, and 504 of size 729.

`AllThreeGroups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`AllThreeGroups` returns the list of all the 3-groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first is a function that can be applied to each group, the second is either a single value or a list of values that the function must return in order to select that group.

```
gap> l := AllThreeGroups( Size, 243, Rank, [2..4], pClass, 3 );;
gap> Length ( l );
33
gap>  List( l, g -> Length( ConjugacyClasses( g ) ) );
[ 35, 35, 35, 35, 35, 35, 35, 243, 99, 99, 51, 51, 51, 51, 51, 51,
  51, 51, 99, 35, 243, 99, 99, 51, 51, 51, 51, 51, 35, 35, 35, 35, 35
 ]
```

`OneThreeGroup(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`OneThreeGroup` returns a single 3-group that has the properties given as arguments. Each property is specified by passing a pair of arguments, the first is a function that can be applied to each group, the second is either a single value or a list of values that the function must return in order to select that group.

```
gap> g := OneThreeGroup( Size, 729, Rank, 4, pClass, [3..5] );
Group( a1, a2, a3, a4, a5, a6 )
gap> IsAbelian( g );
true
```

`AllThreeGroups` and `OneThreeGroup` recognize the following functions and handle them usually very efficiently. `Size`, `Rank` for the rank of the Frattini quotient of the group, and `pClass` for the exponent-p class of the group. Note that `Rank` and `pClass` are dummy functions that can be used only in this context, i.e., they cannot be applied to arbitrary groups.

`ThreeGroup(` *size* `,` *nr* `)`

`ThreeGroup` returns the *nr*-th group of size *size*. The group is returned as a finite polycyclic group (see 25). `ThreeGroup` will signal an error if *size* is not a power of 3 between 3 and 729, or *nr* is larger than the number of groups of size *size*.

Within each size the following criteria have been used, in turn, to determine the index position of a group in the list

1 increasing generator number;

2 increasing exponent-3 class;

3 the position of its parent in the list of groups of appropriate size;

4 the list in which the Newman and O'Brien implementation of the p-group generation algorithm outputs the immediate descendants of a group.

```
gap> g := ThreeGroup( 243, 56 );
Group( a1, a2, a3, a4, a5 )
gap> pClass( g );
3
gap> g.abstractRelators;
[ a1^3, a2^3, a2^-1*a1^-1*a2*a1*a4^-1, a3^3, a3^-1*a1^-1*a3*a1,
  a3^-1*a2^-1*a3*a2*a5^-1, a4^3, a4^-1*a1^-1*a4*a1*a5^-1,
  a4^-1*a2^-1*a4*a2, a4^-1*a3^-1*a4*a3, a5^3, a5^-1*a1^-1*a5*a1,
  a5^-1*a2^-1*a5*a2, a5^-1*a3^-1*a5*a3, a5^-1*a4^-1*a5*a4 ]
```

Apart from the usual components described in 7.118, the group records returned by the above functions have the following components.

`rank`
: rank of Frattini quotient of G.

`pclass`
: exponent-p class of G.

`abstractGenerators`
: a list of abstract generators of G (see 22.1).

`abstractRelators`
: a list of relators of G stored as words in the abstract generators.

Descriptions of the algorithms used in constructing the library data may be found in [O´Br90, O´Br91].

The library was generated and brought into GAP3 format by E.A. O'Brien and Colin Rhodes. David Baldwin, M.F. Newman, and Maris Ozols have contributed in various ways to this project and to correctly determining these groups. The library design is modelled on and borrows extensively from the 2-groups library, which was brought into GAP3 format by Werner Nickel, Alice Niemeyer, and E.A. O'Brien.

38.10 The Irreducible Solvable Linear Groups Library

The **IrredSol** group library provides access to the irreducible solvable subgroups of $GL(n, p)$, where $n > 1$, p is prime and $p^n < 256$. The library contains exactly one member from each of the 370 conjugacy classes of such subgroups.

By well known theory, this library also doubles as a library of primitive solvable permutation groups of non-prime degree less than 256. To access the data in this form, you must first build the matrix group(s) of interest and then call the function
`PrimitivePermGroupIrreducibleMatGroup(` *matgrp* `)`
This function returns a permutation group isomorphic to the semidirect product of an irreducible matrix group (over a finite field) and its underlying vector space.

`AllIrreducibleSolvableGroups(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`AllIrreducibleSolvableGroups` returns a list containing all irreducible solvable linear groups that have the properties given as arguments. Each property is specified by passing a pair of arguments, the first being a function which will be applied to all groups in the library, and the second being a value or a list of values that this function must return in order to have this group included in the list returned by `AllIrreducibleSolvableGroups`.

```
gap> AllIrreducibleSolvableGroups( Dimension, 2,
>                                  CharFFE, 3,
>                                  Size, 8 );
[ Group( [ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ] ],
    [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
    [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ),
  Group( [ [ 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ],
    [ [ Z(3)^0, Z(3) ], [ Z(3), Z(3) ] ] ),
  Group( [ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, Z(3) ] ] ) ]
```

`OneIrreducibleSolvableGroup(` *fun1* `,` *val1* `,` *fun2* `,` *val2* `, ... )`

`OneIrreducibleSolvableGroup` returns one irreducible solvable linear group that has the properties given as arguments. Each property is specified by passing a pair of arguments, the first being a function which will be applied to all groups in the library, and the second being a value or a list of values that this function must return in order to have this group returned by `OneIrreducibleSolvableGroup`. If no such group exists, `false` is returned.

```
gap> OneIrreducibleSolvableGroup( Dimension, 4,
>                                 IsLinearlyPrimitive, false );
Group( [ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ],
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ],
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
```

```
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ] ] )
```

`AllIrreducibleSolvableGroups` and `OneIrreducibleSolvableGroup` recognize the following functions and handle them very efficiently (because the information is stored with the groups and so no computations are needed):`Dimension` for the linear degree, `CharFFE` for the field characteristic, `Size`, `IsLinearlyPrimitive`, and `MinimalBlockDimension`. Note that the last two are dummy functions that can be used only in this context. Their meaning is explained at the end of this section.

`IrreducibleSolvableGroup(` *n*, *p*, *i* `)`

`IrreducibleSolvableGroup` returns the i-th irreducible solvable subgroup of $GL($ n, p $)$. The irreducible solvable subgroups of $GL(n, p)$ are ordered with respect to the following criteria

1. increasing size;

2. increasing guardian number.

If two groups have the same size and guardian, they are in no particular order. (See the library documentation or [Sho92] for the meaning of guardian.)

```
gap> g := IrreducibleSolvableGroup( 3, 5, 12 );
Group( [ [ 0*Z(5), Z(5)^2, 0*Z(5) ], [ Z(5)^2, 0*Z(5), 0*Z(5) ],
    [ 0*Z(5), 0*Z(5), Z(5)^2 ] ],
[ [ 0*Z(5), Z(5)^0, 0*Z(5) ], [ 0*Z(5), 0*Z(5), Z(5)^0 ],
    [ Z(5)^0, 0*Z(5), 0*Z(5) ] ],
[ [ Z(5)^2, 0*Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^0, 0*Z(5) ],
    [ 0*Z(5), 0*Z(5), Z(5)^2 ] ],
[ [ Z(5)^0, 0*Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^2, 0*Z(5) ],
    [ 0*Z(5), 0*Z(5), Z(5)^2 ] ],
[ [ Z(5), 0*Z(5), 0*Z(5) ], [ 0*Z(5), Z(5), 0*Z(5) ],
    [ 0*Z(5), 0*Z(5), Z(5) ] ] )
```

Apart from the usual components described in 7.118, the group records returned by the above functions have the following components.

`size`
: size of G.

`isLinearlyPrimitive`
: `false` if G preserves a direct sum decomposition of the underlying vector space, and `true` otherwise.

`minimalBlockDimension`
: not bound if G is linearly primitive; otherwise equals the dimension of the blocks in an unrefinable system of imprimitivity for G.

This library was computed and brought into GAP3 format by Mark Short. Descriptions of the algorithms used in computing the library data can be found in [Sho92].

38.11 The Library of Finite Perfect Groups

The GAP3 library of finite perfect groups provides, up to isomorphism, a list of all perfect groups whose sizes are less than 10^6 excluding the following:

- For $n = 61440$, 122880, 172032, 245760, 344064, 491520, 688128, or 983040, the perfect groups of size n have not completely been determined yet. The library neither provides the number of these groups nor the groups themselves.
- For $n = 86016$, 368640, or 737280, the library does not yet contain the perfect groups of size n, it only provides their their numbers which are 52, 46, or 54, respectively.

Except for these eleven sizes, the list of altogether 1096 perfect groups in the library is complete. It relies on results of Derek F. Holt and Wilhelm Plesken which are published in their book *Perfect Groups* [HP89]. Moreover, they have supplied to us files with presentations of 488 of the groups. In terms of these, the remaining 607 nontrivial groups in the library can be described as 276 direct products, 107 central products, and 224 subdirect products. They are computed automatically by suitable GAP3 functions whenever they are needed.

We are grateful to Derek Holt and Wilhelm Plesken for making their groups available to the GAP3 community by contributing their files. It should be noted that their book contains a lot of further information for many of the library groups. So we would like to recommend it to any GAP3 user who is interested in the groups.

The library has been brought into GAP3 format by Volkmar Felsch.

Like most of the other GAP3 libraries, the library of finite perfect groups provides an extraction function, `PerfectGroup`. It returns the specified group in form of a finitely presented group which, in its group record, bears some additional information that allows you to easily construct an isomorphic permutation group of some appropriate degree by just calling the `PermGroup` function.

Further, there is a function `NumberPerfectGroups` which returns the number of perfect groups of a given size.

The library does not provide a selection or an example function. There is, however, a function `DisplayInformationPerfectGroups` which allows the display of some information about arbitrary library groups without actually loading the large files with their presentations, and without constructing the groups themselves.

Moereover, there are two functions which allow you to formulate loops over selected library groups. The first one is the `NumberPerfectLibraryGroups` function which, for any given size, returns the number of groups in the library which are of that size.

The second one is the `SizeNumbersPerfectGroups` function. It allows you to ask for all library groups which contain certain composition factors. The answer is provided in form of a list of pairs [*size*,*n*] where each such pair characterizes the n^{th} library group of size *size*. We will call such a pair [*size*,*n*] the **size number** of the respective perfect group. As the size numbers are accepted as input arguments by the `PerfectGroup` and the `DisplayInformationPerfectGroups` function, you may use their list to formulate a loop over the associated groups.

Now we shall give an individual description of each library function.

`NumberPerfectGroups(` *size* `)`

`NumberPerfectGroups` returns the number of non-isomorphic perfect groups of size *size* for each positive integer *size* up to 10^6 except for the eight sizes listed at the beginning of this section for which the number is not yet known. For these values as well as for any argument out of range it returns the value -1.

`NumberPerfectLibraryGroups(` *size* `)`

`NumberPerfectLibraryGroups` returns the number of perfect groups of size *size* which are available in the library of finite perfect groups.

The purpose of the function is to provide a simple way to formulate a loop over all library groups of a given size.

`SizeNumbersPerfectGroups(` *factor1* `,` *factor2* `... )`

`SizeNumbersPerfectGroups` returns a list of the *size numbers* (see above) of all library groups that contain the specified factors among their composition factors. Each argument must either be the name of a simple group or an integer expression which is the product of the sizes of one or more cyclic factors. The function ignores the order in which the argmuments are given and, in fact, replaces any list of more than one integer expression among the arguments by their product.

The following text strings are accepted as simple group names.

> `"A5"`, `"A6"`, `"A7"`, `"A8"`, `"A9"` or `"A(5)"`, `"A(6)"`, `"A(7)"`, `"A(8)"`, `"A(9)"` for the alternating groups A_n, $5 \leq n \leq 9$,
>
> `"L2(`*q*`)"` or `"L(2,`*q*`)"` for $PSL(2,q)$, where q is any prime power with $4 \leq q \leq 125$,
>
> `"L3(`*q*`)"` or `"L(3,`*q*`)"` for $PSL(3,q)$ with $2 \leq q \leq 5$,
>
> `"U3(`*q*`)"` or `"U(3,`*q*`)"` for $PSU(2,q)$ with $3 \leq q \leq 5$,
>
> `"U4(2)` or `"U(4,2)"` for $PSU(4,2)$,
>
> `"Sp4(4)"` or `"S(4,4)"` for the symplectic group $S(4,4)$,
>
> `"Sz(8)"` for the Suzuki group $Sz(8)$,
>
> `"M11"`, `"M12"`, `"M22"` or `"M(11)"`, `"M(12)"`, `"M(22)"` for the Matthieu groups M_{11}, M_{12}, and M_{22}, and
>
> `"J1"`, `"J2"` or `"J(1)"`, `"J(2)"` for the Janko groups J_1 and J_2.

Note that, for most of the groups, the preceding list offers two different names in order to be consistent with the notation used in [HP89] as well as with the notation used in the `DisplayCompositionSeries` command of GAP3. However, as the names are compared as text strings, you are restricted to the above choice. Even expressions like `"L2( 32 )"` or `"L2(2^5)"` are not accepted.

As the use of the term $PSU(n,q)$ is not unique in the literature, we state that here it denotes the factor group of $SU(n,q)$ by its centre, where $SU(n,q)$ is the group of all $n \times n$ unitary matrices with entries in $GF(q^2)$ and determinant 1.

The purpose of the function is to provide a simple way to formulate a loop over all library groups which contain certain composition factors.

`DisplayInformationPerfectGroups(` *size* `)`
`DisplayInformationPerfectGroups(` *size*`,` *n* `)`
`DisplayInformationPerfectGroups( [` *size*`,` *n* `] )`

`DisplayInformationPerfectGroups` displays some information about the library group G, say, which is specified by the size number [*size*,*n*] or by the two arguments *size* and n. If, in the second case, n is omitted, the function will loop over all library groups of size *size*.

The information provided for G includes the following items:

- a headline containing the size number $[size, n]$ of G in the form $size.n$ (the suffix $.n$ will be suppressed if, up to isomorphism, G is the only perfect group of size *size*),
- a message if G is simple or quasisimple, i. e., if the factor group of G by its centre is simple,
- the "description" of the structure of G as it is given by Holt and Plesken in [HP89] (see below),
- the size of the centre of G (suppressed, if G is simple),
- the prime decomposition of the size of G,
- orbit sizes for a faithful permutation representation of G which is provided by the library (see below),
- a reference to each occurrence of G in the tables of section 5.3 of [HP89]. Each of these references consists of a class number and an internal number (i, j) under which G is listed in that class. For some groups, there is more than one reference because these groups belong to more than one of the classes in the book.

Example:

```
gap> DisplayInformationPerfectGroups( 30720, 3 );
#I Perfect group 30720.3:  A5 ( 2^4 E N 2^1 E 2^4 ) A
#I   centre = 1  size = 2^11*3*5  orbit size = 240
#I   Holt-Plesken class 1 (9,3)
gap> DisplayInformationPerfectGroups( 30720, 6 );
#I Perfect group 30720.6:  A5 ( 2^4 x 2^4 ) C N 2^1
#I   centre = 2  size = 2^11*3*5  orbit size = 384
#I   Holt-Plesken class 1 (9,6)
gap> DisplayInformationPerfectGroups( Factorial( 8 ) / 2 );
#I Perfect group 20160.1:  A5 x L3(2) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 5 + 16
#I   Holt-Plesken class 31 (1,1) (occurs also in class 32)
#I Perfect group 20160.2:  A5 2^1 x L3(2)
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 7 + 24
#I   Holt-Plesken class 31 (1,2) (occurs also in class 32)
#I Perfect group 20160.3:  ( A5 x L3(2) ) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit size = 192
#I   Holt-Plesken class 31 (1,3)
#I Perfect group 20160.4:  simple group  A8
#I   size = 2^6*3^2*5*7  orbit size = 8
#I   Holt-Plesken class 26 (0,1)
#I Perfect group 20160.5:  simple group  L3(4)
#I   size = 2^6*3^2*5*7  orbit size = 21
```

```
#I   Holt-Plesken class 27 (0,1)
```

For any library group G, the library files do not only provide a presentation, but, in addition, a list of one or more subgroups $S_1, \ldots, S_r$ of G such that there is a faithful permutation representation of G of degree $\sum_{i=1}^{r} G\!:\!S_i$ on the set $\{S_i g \mid 1 \leq i \leq r,\ g \in G\}$ of the cosets of the S_i. The respective permutation group is available via the `PermGroup` function described below. The `DisplayInformationPerfectGroups` function displays only the available degree. The message

```
orbit size = 8
```

in the above example means that the available permutation representation is transitive and of degree 8, whereas the message

```
orbit sizes = 7 + 24
```

means that a nontransitive permutation representation is available which acts on two orbits of size 7 and 24 respectively.

The notation used in the "description" of a group is explained in section 5.1.2 of [HP89]. We quote the respective page from there:

'Within a class $Q\,\#\,p$, an isomorphism type of groups will be denoted by an ordered pair of integers (r, n), where $r \geq 0$ and $n > 0$. More precisely, the isomorphism types in $Q\ \#\ p$ of order $p^r|Q|$ will be denoted by $(r, 1)$, $(r, 2)$, $(r, 3)$, $\ldots$. Thus Q will always get the size number $(0, 1)$.

In addition to the symbol (r, n), the groups in $Q\,\#\,p$ will also be given a more descriptive name. The purpose of this is to provide a very rough idea of the structure of the group. The names are derived in the following manner. First of all, the isomorphism classes of irreducible F_pQ-modules M with $|Q||M| \leq 10^6$, where F_p is the field of order p, are assigned symbols. These will either be simply p^x, where x is the dimension of the module, or, if there is more than one isomorphism class of irreducible modules having the same dimension, they will be denoted by p^x, $p^{x'}$, etc. The one-dimensional module with trivial Q-action will therefore be denoted by p^1. These symbols will be listed under the description of Q. The group name consists essentially of a list of the composition factors working from the top of the group downwards; hence it always starts with the name of Q itself. (This convention is the most convenient in our context, but it is different from that adopted in the ATLAS (Conway et al. 1985), for example, where composition factors are listed in the reverse order. For example, we denote a group isomorphic to $SL(2, 5)$ by $A_5 2^1$ rather than $2 \cdot A_5$.)

Some other symbols are used in the name, in order to give some idea of the relationship between these composition factors, and splitting properties. We shall now list these additional symbols.

- $\times$ *between two factors denotes a direct product of F_pQ-modules or groups.*
- C *(for 'commutator') between two factors means that the second lies in the commutator subgroup of the first. Similarly, a segment of the form $(f_1 \times f_2)Cf_3$ would mean that the factors f_1 and f_2 commute modulo f_3 and f_3 lies in $[f_1, f_2]$.*
- A *(for 'abelian') between two factors indicates that the second is in the pth power (but not the commutator subgroup) of the first. 'A' may also follow the factors, if bracketed.*
- E *(for 'elementary abelian') between two factors indicates that together they generate an elementary abelian group (modulo subsequent factors), but that the resulting F_pQ-module extension does not split.*

N (for 'nonsplit') before a factor indicates that Q (or possibly its covering group) splits down as far at this factor but not over the factor itself. So 'Qf_1Nf_2' means that the normal subgroup f_1f_2 of the group has no complement but, modulo f_2, f_1, does have a complement.

Brackets have their obvious meaning. Summarizing, we have
$\times$ = *dirext product;*
C = *commutator subgroup;*
A = *abelian;*
E = *elementary abelian; and*
N = *nonsplit.*

Here are some examples.

(i) $A_5(2^4E2^1E2^4)A$ *means that the pairs* 2^4E2^1 *and* 2^1E2^4 *are both elementary abelian of exponent 4.*

(ii) $A_5(2^4E2^1A)C2^1$ *means that* $O_2(G)$ *is of symplectic type* 2^{1+5}*, with Frattini factor group of type* 2^4E2^1*. The 'A' after the* 2^1 *indicates that* G *has a central cyclic subgroup* 2^1A2^1 *of order 4.*

(iii) $L_3(2)((2^1E)\times(N2^3E2^{3'}A)C)2^{3'}$ *means that the* $2^{3'}$ *factor at the bottom lies in the commutator subgroup of the pair* $2^3E2^{3'}$ *in the middle, but the lower pair* $2^{3'}A2^{3'}$ *is abelian of exponent 4. There is also a submodule* $2^1E2^{3'}$*, and the covering group* $L_3(2)2^1$ *of* $L_3(2)$ *does not split over the* 2^3 *factor. (Since G is perfect, it goes without saying that the extension* $L_3(2)2^1$ *cannot split itself.)*

We must stress that this notation does not always succeed in being precise or even unambiguous, and the reader is free to ignore it if it does not seem helpful.'

If such a group description has been given in the book for G (and, in fact, this is the case for most of the library groups), it is displayed by the `DisplayInformationPerfectGroups` function. Otherwise the function provides a less explicit description of the (in these cases unique) Holt-Plesken class to which G belongs, together with a serial number if this is necessary to make it unique.

`PerfectGroup(` *size* `)`
`PerfectGroup(` *size*`,` *n* `)`
`PerfectGroup( [` *size*`,` *n* `] )`

`PerfectGroup` is the essential extraction function of the library. It returns a finitely presented group, G say, which is isomorphic to the library group specified by the size number [*size*,*n*] or by the two separate arguments *size* and n. In the second case, you may omit the parameter n. Then the default value is $n = 1$.

```
gap> G := PerfectGroup( 6048 );
PerfectGroup(6048)
gap> G.generators;
[ a, b ]
gap> G.relators;
[ a^2, b^6, a*b*a*b*a*b*a*b*a*b*a*b*a*b,
  a*b^2*a*b^2*a*b^2*a*b^-2*a*b^-2*a*b^-2,
```

```
  a*b*a*b^-2*a*b*a*b^-2*a*b*a*b^-2*a*b*a*b^-1*a*b^-1 ]
gap> G.size;
6048
gap> G.description;
"U3(3)"
gap> G.subgroups;
[ Subgroup( PerfectGroup(6048), [ a, b*a*b*a*b*a*b^3 ] ) ]
```

The generators and relators of G coincide with those given in [HP89].

Note that, besides the components that are usually initialized for any finitely presented group, the group record of G contains the following components:

`size`
: the size of G,

`isPerfect`
: always `true`,

`description`
: the description of G as described with the `DisplayInformationPerfectGroups` function above,

`source`
: some internal information used by the library functions,

`subgroups`
: a list of subgroups $S_1, \ldots, S_r$ of G such that G acts faithfully on on the union of the sets of all cosets of the S_i.

The last of these components exists only if G is one of the 488 nontrivial library groups which are given directly by a presentation on file, i. e., which are not constructed from other library groups in form of a direct, central, or subdirect product. It will be required by the following function.

`PermGroup(` *G* `)`

`PermGroup` returns a permutation group, P say, which is isomorphic to the given group G which is assumed to be a finitely presented perfect group that has been extracted from the library of finite perfect groups via the `PerfectGroup` function.

Let $S_1, \ldots, S_r$ be the subgroups listed in the component G`.subgroups` of the group record of G. Then the resulting group P is the permutation group of degree $\sum_{i=1}^{r} G:S_i$ which is induced by G on the set $\{S_i g \mid 1 \leq i \leq r, g \in G\}$ of all cosets of the S_i.

Example (continued):

```
gap> P := PermGroup( G );
PermGroup(PerfectGroup(6048))
gap> P.size;
6048
gap> P.degree;
28
```

Note that some of the library groups do not have a faithful permutation representation of small degree. Computations in these groups may be rather time consuming.

Example:

```
gap> P := PermGroup( PerfectGroup( 129024, 2 ) );
PermGroup(PerfectGroup(129024,2))
gap> P.degree;
14336
```

38.12 Irreducible Maximal Finite Integral Matrix Groups

A library of irreducible maximal finite integral matrix groups is provided with GAP3. It contains $\mathbb{Q}$-class representatives for all of these groups of dimension at most 24, and $\mathbb{Z}$-class representatives for those of dimension at most 11 or of dimension 13, 17, 19, or 23.

The groups provided in this library have been determined by Wilhelm Plesken, partially as joint work with Michael Pohst, or by members of his institute (Lehrstuhl B für Mathematik, RWTH Aachen). In particular, the data for the groups of dimensions 2 to 9 have been taken from the output of computer calculations which they performed in 1979 (see [PP77], [PP80]). The $\mathbb{Z}$-class representatives of the groups of dimension 10 have been determined and computed by Bernd Souvignier ([Sou94]), and those of dimensions 11, 13, and 17 have been recomputed for this library from the circulant Gram matrices given in [Ple85], using the stand-alone programs for the computation of short vectors and Bravais groups which have been developed in Plesken's institute. The $\mathbb{Z}$-class representatives of the groups of dimensions 19 and 23 had already been determined in [Ple85]. Gabriele Nebe has recomputed them for us. Her main contribution to this library, however, is that she has determined and computed the $\mathbb{Q}$-class representatives of the groups of non-prime dimensions between 12 and 24 (see [PN95], [NP95b], [Neb95]).

The library has been brought into GAP3 format by Volkmar Felsch. He has applied several GAP3 routines to check certain consistency of the data. However, the credit and responsibility for the lists remain with the authors. We are grateful to Wilhelm Plesken, Gabriele Nebe, and Bernd Souvignier for supplying their results to GAP3.

In the preceding acknowledgement, we used some notations that will also be needed in the sequel. We first define these.

Any integral matrix group G of dimension n is a subgroup of $GL_n(\mathbb{Z})$ as well as of $GL_n(\mathbb{Q})$ and hence lies in some conjugacy class of integral matrix groups under $GL_n(\mathbb{Z})$ and also in some conjugacy class of rational matrix groups under $GL_n(\mathbb{Q})$. As usual, we call these classes the $\mathbb{Z}$-class and the $\mathbb{Q}$-class of G, respectively. Note that any conjugacy class of subgroups of $GL_n(\mathbb{Q})$ contains at least one $\mathbb{Z}$-class of subgroups of $GL_n(\mathbb{Z})$ and hence can be considered as the $\mathbb{Q}$-class of some integral matrix group.

In the context of this library we are only concerned with $\mathbb{Z}$-classes and $\mathbb{Q}$-classes of subgroups of $GL_n(\mathbb{Z})$ which are irreducible and maximal finite in $GL_n(\mathbb{Z})$ (we will call them *i. m. f.* subgroups of $GL_n(\mathbb{Z})$). We can distinguish two types of these groups:

First, there are those i. m. f. subgroups of $GL_n(\mathbb{Z})$ which are also maximal finite subgroups of $GL_n(\mathbb{Q})$. Let us denote the set of their $\mathbb{Q}$-classes by $Q_1(n)$. It is clear from the above remark that $Q_1(n)$ just consists of the Q-classes of i. m. f. subgroups of $GL_n(\mathbb{Q})$.

Secondly, there is the set $Q_2(n)$ of the $\mathbb{Q}$-classes of the remaining i. m. f. subgroups of $GL_n(\mathbb{Z})$, i. e., of those which are not maximal finite subgroups of $GL_n(\mathbb{Q})$. For any such group G, say, there is at least one class $C \in Q_1(n)$ such that G is conjugate under $\mathbb{Q}$ to a proper subgroup of some group $H \in C$. In fact, the class C is uniquely determined for any group G occurring in the library (though there seems to be no reason to assume that this property should hold in general). Hence we may call C the *rational i. m. f. class* of G. Finally, we will denote the number of classes in $Q_1(n)$ and $Q_2(n)$ by $q_1(n)$ and $q_2(n)$, respectively.

As an example, let us consider the case $n = 4$. There are 6 $\mathbb{Z}$-classes of i. m. f. subgroups of $GL_4(\mathbb{Z})$ with representative subgroups $G_1, \ldots, G_6$ of isomorphsim types $G_1 \cong W(F_4)$,

$G_2 \cong D_{12} \wr C_2$, $G_3 \cong G_4 \cong C_2 \times S_5$, $G_5 \cong W(B_4)$, and $G_6 \cong (D_{12} \curlyvee D_{12}){:}C_2$. The corresponding $\mathbb{Q}$-classes, $R_1, \ldots, R_6$, say, are pairwise different except that R_3 coincides with R_4. The groups G_1, G_2, and G_3 are i. m. f. subgroups of $GL_4(\mathbb{Q})$, but G_5 and G_6 are not because they are conjugate under $GL_4(\mathbb{Q})$ to proper subgroups of G_1 and G_2, respectively. So we have $Q_1(4) = \{R_1, R_2, R_3\}$, $Q_2(4) = \{R_5, R_6\}$, $q_1(4) = 3$, and $q_2(4) = 2$.

The $q_1(n)$ $\mathbb{Q}$-classes of i. m. f. subgroups of $GL_n(\mathbb{Q})$ have been determined for each dimension $n \leq 24$. The current GAP3 library provides integral representative groups for all these classes. Moreover, all $\mathbb{Z}$-classes of i. m. f. subgroups of $GL_n(\mathbb{Z})$ are known for $n \leq 11$ and for $n \in \{13, 17, 19, 23\}$. For these dimensions, the library offers integral representative groups for all $\mathbb{Q}$-classes in $Q_1(n)$ and $Q_2(n)$ as well as for all $\mathbb{Z}$-classes of i. m. f. subgroups of $GL_n(\mathbb{Z})$.

Any group G of dimension n given in the library is represented as the automorphism group $G = \mathrm{Aut}(F, L) = \{g \in GL_n(\mathbb{Z}) \mid Lg = L \text{ and } gFg^{\mathrm{tr}} = F\}$ of a positive definite symmetric $n \times n$ matrix $F \in \mathbb{Z}^{n \times n}$ on an n-dimensional lattice $L \cong \mathbb{Z}^{1 \times n}$ (for details see e. g. [PN95]). GAP3 provides for G a list of matrix generators and the *Gram matrix* F.

The positive definite quadratic form defined by F defines a *norm* vFv^{tr} for each vector $v \in L$, and there is only a finite set of vectors of minimal norm. These vectors are often simply called the *"short vectors"*. Their set splits into orbits under G, and G being irreducible acts faithfully on each of these orbits by multiplication from the right. GAP3 provides for each of these orbits the orbit size and a representative vector.

Like most of the other GAP3 libraries, the library of i. m. f. integral matrix groups supplies an extraction function, `ImfMatGroup`. However, as the library involves only 423 different groups, there is no need for a selection or an example function. Instead, there are two functions, `ImfInvariants` and `DisplayImfInvariants`, which provide some $\mathbb{Z}$-class invariants that can be extracted from the library without actually constructing the representative groups themselves. The difference between these two functions is that the latter one displays the resulting data in some easily readable format, whereas the first one returns them as record components so that you can properly access them.

We shall give an individual description of each of the library functions, but first we would like to insert a short remark concerning their names: Any self-explaining name of a function handling *irreducible maximal finite integral matrix groups* would have to include this term in full length and hence would grow extremely long. Therefore we have decided to use the abbreviation `Imf` instead in order to restrict the names to some reasonable length.

The first three functions can be used to formulate loops over the classes.

`ImfNumberQQClasses(` *dim* `)`
`ImfNumberQClasses(` *dim* `)`
`ImfNumberZClasses(` *dim*`,` *q* `)`

`ImfNumberQQClasses` returns the number $q_1(dim)$ of $\mathbb{Q}$-classes of i. m. f. rational matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 24.

Note: In order to enable you to loop just over the classes belonging to $Q_1(dim)$, we have arranged the list of $\mathbb{Q}$-classes of dimension *dim* for any dimension *dim* in the library such that, whenever the classes of $Q_2(dim)$ are known, too, i. e., in the cases $dim \leq 11$ or $dim \in \{13, 17, 19, 23\}$, the classes of $Q_1(dim)$ precede those of $Q_2(dim)$ and hence are numbered from 1 to $q_1(dim)$.

`ImfNumberQClasses` returns the number of $\mathbb{Q}$-classes of groups of dimension *dim* which are available in the library. If $dim \leq 11$ or $dim \in \{13, 17, 19, 23\}$, this is the number $q_1(dim) + q_2(dim)$ of $\mathbb{Q}$-classes of i. m. f. subgroups of $GL_{dim}(\mathbb{Z})$. Otherwise, it is just the number $q_1(dim)$ of $\mathbb{Q}$-classes of i. m. f. subgroups of $GL_{dim}(\mathbb{Q})$. Valid values of *dim* are all positive integers up to 24.

`ImfNumberZClasses` returns the number of $\mathbb{Z}$-classes in the q^{th} $\mathbb{Q}$-class of i. m. f. integral matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 11 and all primes up to 23.

`DisplayImfInvariants(` *dim*`,` *q* `)`
`DisplayImfInvariants(` *dim*`,` *q*`,` *z* `)`

`DisplayImfInvariants` displays the following $\mathbb{Z}$-class invariants of the groups in the z^{th} $\mathbb{Z}$-class in the q^{th} $\mathbb{Q}$-class of i. m. f. integral matrix groups of dimension *dim*:

- its $\mathbb{Z}$-class number in the form *dim.q.z*, if *dim* is at most 11 or a prime, or its $\mathbb{Q}$-class number in the form *dim.q*, else,
- a message if the group is solvable,
- the size of the group,
- the isomorphism type of the group,
- the elementary divisors of the associated quadratic form,
- the sizes of the orbits of short vectors (these sizes are the degrees of the faithful permutation representations which you may construct using the `PermGroup` or `PermGroupImfGroup` commands below),
- the norm of the associated short vectors,
- only in case that the group is not an i. m. f. group in $GL_n(\mathbb{Q})$: an appropriate message, including the $\mathbb{Q}$-class number of the corresponding rational i. m. f. class.

If you specify the value 0 for any of the parameters *dim*, *q*, or *z*, the command will loop over all available dimensions, $\mathbb{Q}$-classes of given dimension, or $\mathbb{Z}$-classes within the given $\mathbb{Q}$-class, respectively. Otherwise, the values of the arguments must be in range. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i. e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the q^{th} $\mathbb{Q}$-class. The greatest legal value of *dim* is 24.

```
gap> DisplayImfInvariants( 3, 1, 0 );
#I Z-class 3.1.1:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = W(B3)
#I   elementary divisors = 1^3
#I   orbit size = 6, minimal norm = 1
#I Z-class 3.1.2:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1*4^2
#I   orbit size = 8, minimal norm = 3
#I Z-class 3.1.3:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1^2*4
```

```
#I   orbit size = 12, minimal norm = 2
gap> DisplayImfInvariants( 8, 15, 1 );
#I Z-class 8.15.1:  Solvable, size = 2^5*3^4
#I   isomorphism type = C2 x (S3 wr S3)
#I   elementary divisors = 1*3^3*9^3*27
#I   orbit size = 54, minimal norm = 8
#I   not maximal finite in GL(8,Q), rational imf class is 8.5
gap> DisplayImfInvariants( 20, 23 );
#I Q-class 20.23:  Size = 2^5*3^2*5*11
#I   isomorphism type = (PSL(2,11) x D12).C2
#I   elementary divisors = 1^18*11^2
#I   orbit size = 3*660 + 2*1980 + 2640 + 3960, minimal norm = 4
```

Note that the `DisplayImfInvariants` function uses a kind of shorthand to display the elementary divisors. E. g., the expression `1*3^3*9^3*27` in the preceding example stands for the elementary divisors $1, 3, 3, 3, 9, 9, 9, 27$. (See also the next example which shows that the `ImfInvariants` function provides the elementary divisors in form of an ordinary GAP3 list.)

In the description of the isomorphism types the following notations are used:

$A \,\mathrm{x}\, B$	denotes a direct product of a group A by a group B,
$A \,\mathrm{subd}\, B$	denotes a subdirect product of A by B,
$A \,\mathrm{Y}\, B$	denotes a central product of A by B,
$A \,\mathrm{wr}\, B$	denotes a wreath product of A by B,
$A : B$	denotes a split extension of A by B,
$A \,.\, B$	denotes just an extension of A by B (split or nonsplit).

The groups involved are

- the cyclic groups C_n, dihedral groups D_n, and generalized quaternion groups Q_n of order n, denoted by Cn, Dn, and Qn, respectively,
- the alternating groups A_n and symmetric groups S_n of degree n, denoted by An and Sn, respectively,
- the linear groups $GL_n(q)$, $PGL_n(q)$, $SL_n(q)$, and $PSL_n(q)$, denoted by GL(n,q), PGL(n,q), SL(n,q), and PSL(n,q), respectively,
- the unitary groups $SU_n(q)$ and $PSU_n(q)$, denoted by SU(n,q) and PSU(n,q), respectively,
- the symplectic groups $Sp(n, q)$, denoted by Sp(n,q),
- the orthogonal group $O_8^+(2)$, denoted by `O+(8,2)`,
- the extraspecial groups 2_+^{1+8}, 3_+^{1+2}, 3_+^{1+4}, and 5_+^{1+2}, denoted by `2+^(1+8)`, `3+^(1+2)`, `3+^(1+4)`, and `5+^(1+2)`, respectively,
- the Chevalley group $G_2(3)$, denoted by `G(2,3)`,
- the Weyl groups $W(A_n)$, $W(B_n)$, $W(D_n)$, $W(E_n)$, and $W(F_4)$, denoted by W(An), W(Bn), W(Dn), W(En), and `W(F4)`, respectively,
- the sporadic simple groups Co_1, Co_2, Co_3, HS, J_2, M_{12}, M_{22}, M_{23}, M_{24}, and Mc, denoted by `Co1`, `Co2`, `Co3`, `HS`, `J2`, `M12`, `M22`, `M23`, `M24`, and `Mc`, respectively,
- a point stabilizer of index 11 in M_{11}, denoted by `M10`.

As mentioned above, the data assembled by the `DisplayImfInvariants` command are "cheap data" in the sense that they can be provided by the library without loading any of its large matrix files or performing any matrix calculations. The following function allows you to get proper access to these cheap data instead of just displaying them.

`ImfInvariants(` *dim*`,` *q* `)`
`ImfInvariants(` *dim*`,` *q*`,` *z* `)`

`ImfInvariants` returns a record which provides some $\mathbb{Z}$-class invariants of the groups in the z^{th} $\mathbb{Z}$-class in the q^{th} $\mathbb{Q}$-class of i. m. f. integral matrix groups of dimension *dim*. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i. e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the q^{th} $\mathbb{Q}$-class. The greatest legal value of *dim* is 24.

The resulting record contains six or seven components:

`size`
: the size of any representative group G,

`isSolvable`
: is `true` if G is solvable,

`isomorphismType`
: a text string describing the isomorphism type of G (in the same notation as used by the `DisplayImfInvariants` command above),

`elementaryDivisors`
: the elementary divisors of the associated Gram matrix F (in the same format as the result of the `ElementaryDivisorsMat` function, see 34.23),

`minimalNorm`
: the norm of the associated short vectors,

`sizesOrbitsShortVectors`
: the sizes of the orbits of short vectors under F,

`maximalQClass`
: the $\mathbb{Q}$-class number of an i. m. f. group in $GL_n(\mathbb{Q})$ that contains G as a subgroup (only in case that not G itself is an i. m. f. subgroup of $GL_n(\mathbb{Q})$).

Note that four of these data, namely the group size, the solvability, the isomorphism type, and the corresponding rational i. m. f. class, are not only $\mathbb{Z}$-class invariants, but also $\mathbb{Q}$-class invariants.

Note further that, though the isomorphism type is a $\mathbb{Q}$-class invariant, you will sometimes get different descriptions for different $\mathbb{Z}$-classes of the same $\mathbb{Q}$-class (as, e. g., for the classes 3.1.1 and 3.1.2 in the last example above). The purpose of this behaviour is to provide some more information about the underlying lattices.

```
gap> ImfInvariants( 8, 15, 1 );
rec(
  size := 2592,
  isSolvable := true,
  isomorphismType := "C2 x (S3 wr S3)",
```

```
    elementaryDivisors := [ 1, 3, 3, 3, 9, 9, 9, 27 ],
    minimalNorm := 8,
    sizesOrbitsShortVectors := [ 54 ],
    maximalQClass := 5 )
  gap> ImfInvariants( 24, 1 ).size;
  10409396852733332453861621760000
  gap> ImfInvariants( 23, 5, 2 ).sizesOrbitsShortVectors;
  [ 552, 53130 ]
  gap> for i in [ 1 .. ImfNumberQClasses( 22 ) ] do
  >    Print( ImfInvariants( 22, i ).isomorphismType, "\n" ); od;
  C2 wr S22 = W(B22)
  (C2 x PSU(6,2)).S3
  (C2 x S3) wr S11 = (C2 x W(A2)) wr S11
  (C2 x S12) wr C2 = (C2 x W(A11)) wr C2
  C2 x S3 x S12 = C2 x W(A2) x W(A11)
  (C2 x HS).C2
  (C2 x Mc).C2
  C2 x S23 = C2 x W(A22)
  C2 x PSL(2,23)
  C2 x PSL(2,23)
  C2 x PGL(2,23)
  C2 x PGL(2,23)
```

`ImfMatGroup(` *dim*, *q* `)`
`ImfMatGroup(` *dim*, *q*, *z* `)`

`ImfMatGroup` is the essential extraction function of this library. It returns a representative group, G say, of the z^{th} $\mathbb{Z}$-class in the q^{th} $\mathbb{Q}$-class of i. m. f. integral matrix groups of dimension *dim*. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i. e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the q^{th} $\mathbb{Q}$-class. The greatest legal value of *dim* is 24.

```
  gap> G := ImfMatGroup( 5, 1, 3 );
  ImfMatGroup(5,1,3)
  gap> for m in G.generators do PrintArray( m ); od;
  [ [  -1,   0,   0,   0,   0 ],
    [   0,   1,   0,   0,   0 ],
    [   0,   0,   0,   1,   0 ],
    [  -1,  -1,  -1,  -1,   2 ],
    [  -1,   0,   0,   0,   1 ] ]
  [ [  0,  1,  0,  0,  0 ],
    [  0,  0,  1,  0,  0 ],
    [  0,  0,  0,  1,  0 ],
    [  1,  0,  0,  0,  0 ],
    [  0,  0,  0,  0,  1 ] ]
```

The group record of G contains the usual components of a matrix group record. In addition, it includes the same six or seven records as the resulting record of the `ImfInvariants` function described above, namely the components `size`, `isSolvable`, `isomorphismType`, `elementaryDivisors`, `minimalNorm`, and `sizesOrbitsShortVectors` and, if G is not a rational i. m. f. group, `maximalQClass`. Moreover, there are the two components

`form`
: the associated Gram matrix F,

`repsOrbitsShortVectors`
: representatives of the orbits of short vectors under F.

The last of these components will be required by the `PermGroup` function below.

Example:

```
gap> G.size;
3840
gap> G.isomorphismType;
"C2 wr S5 = C2 x W(D5)"
gap> PrintArray( G.form );
[ [  4,  0,  0,  0,  2 ],
  [  0,  4,  0,  0,  2 ],
  [  0,  0,  4,  0,  2 ],
  [  0,  0,  0,  4,  2 ],
  [  2,  2,  2,  2,  5 ] ]
gap> G.elementaryDivisors;
[ 1, 4, 4, 4, 4 ]
gap> G.minimalNorm;
4
```

If you want to perform calculations in such a matrix group G you should be aware of the fact that GAP3 offers much more efficient permutation group routines than matrix group routines. Hence we recommend that you do your computations, whenever it is possible, in the isomorphic permutation group that is induced by the action of G on one of the orbits of the associated short vectors. You may call one of the following functions to get such a permutation group.

`PermGroup(` G `)`

`PermGroup` returns the permutation group which is induced by the given i. m. f. integral matrix group G on an orbit of minimal size of G on the set of short vectors (see also `PermGroupImfGroup` below).

The permutation representation is computed by first constructing the specified orbit, S say, of short vectors and then computing the permutations which are induced on S by the generators of G. We would like to warn you that in case of a large orbit this procedure may be space and time consuming. Fortunately, there are only five $\mathbb{Q}$-classes in the library for which the smallest orbit of short vectors is of size greater than 20000, the worst case being the orbit of size 196560 for the Leech lattice ($dim = 24$, $q = 3$).

As mentioned above, `PermGroup` constructs the required permutation group, P say, as the image of G under the isomorphism between the matrices in G and their action on S. Moreover, it constructs the inverse isomorphism from P to G, φ say, and returns it in the group record component P.`bijection` of P. In fact, φ is constructed by determining a $\mathbb{Q}$-base $B \subset S$ of $\mathbb{Q}^{1\times dim}$ in S and, in addition, the associated base change matrix M which transforms B into the standard base of $\mathbb{Z}^{1\times dim}$. Then the image $\varphi(p)$ of any permutation $p \in P$ can be easily computed: If, for $1 \leq i \leq dim$, b_i is the position number in S of the i^{th} base vector in B, it suffices to look up the vector whose position number in S is the image of b_i under p and to multiply this vector by M to get the i^{th} row of $\varphi(p)$.

You may use φ at any time to compute the images in G of permutations in P or to compute the preimages in P of matrices in G.

The record of P contains, in addition to the usual components of permutation group records, some special components. The most important of those are:

`isomorphismType`
: a text string describing the isomorphism type of P (in the same notation as used by the `DisplayImfInvariants` command above),

`matGroup`
: the associated matrix group G,

`bijection`
: the isomorphism φ from P to G,

`orbitShortVectors`
: the orbit S of short vectors (needed for φ),

`baseVectorPositions`
: the position numbers of the base vectors in B with respect to S (needed for φ),

`baseChangeMatrix`
: the base change matrix M (needed for φ).

As an example, let us compute a set R of matrix generators for the solvable residuum of the group G that we have constructed in the preceding example.

```
gap> # Perform the computations in an isomorphic permutation group.
gap> P := PermGroup( G );
PermGroup(ImfMatGroup(5,1,3))
gap> P.generators;
[ ( 1, 7, 6)( 2, 9)( 4, 5,10), ( 2, 3, 4, 5)( 6, 9, 8, 7) ]
gap> D := DerivedSubgroup( P );
Subgroup( PermGroup(ImfMatGroup(5,1,3)),
[ ( 1, 2,10, 9)( 3, 8)( 4, 5)( 6, 7),
  ( 1, 6)( 2, 7, 9, 4)( 3, 8)( 5,10), ( 1, 5,10, 6)( 2, 8, 9, 3) ] )
gap> Size( D );
960
gap> IsPerfect( D );
true
gap> # Now move the results back to the matrix group.
gap> phi := P.bijection;;
gap> R := List( D.generators, p -> Image( phi, p ) );;
gap> for m in R do PrintArray( m ); od;
[ [  -1,  -1,  -1,  -1,   2 ],
  [   0,  -1,   0,   0,   0 ],
  [   0,   0,   0,   1,   0 ],
  [   0,   0,   1,   0,   0 ],
```

```
      [  -1,  -1,   0,   0,   1 ] ]
    [ [   0,   0,  -1,   0,   0 ],
      [   0,  -1,   0,   0,   0 ],
      [   1,   0,   0,   0,   0 ],
      [  -1,  -1,  -1,  -1,   2 ],
      [   0,  -1,  -1,   0,   1 ] ]
    [ [   0,  -1,   0,   0,   0 ],
      [   1,   0,   0,   0,   0 ],
      [   0,   0,   1,   0,   0 ],
      [  -1,  -1,  -1,  -1,   2 ],
      [   0,  -1,   0,  -1,   1 ] ]
gap> # The PreImage function allows us to use the inverse of phi.
gap> PreImage( phi, R[1] ) = D.generators[1];
true
```

`PermGroupImfGroup(` G`,` n `)`

`PermGroupImfGroup` returns the permutation group which is induced by the given i. m. f. integral matrix group G on its n^{th} orbit of short vectors. The only difference to the above `PermGroup` function is that you can specify the orbit to be used. In fact, as the orbits of short vectors are sorted by increasing sizes, the function `PermGroup(` G `)` has been implemented such that it is equivalent to `PermGroupImfGroup(` G`, 1 )`.

```
gap> ImfInvariants( 12, 9 ).sizesOrbitsShortVectors;
[ 120, 300 ]
gap> G := ImfMatGroup( 12, 9 );
ImfMatGroup(12,9)
gap> P1 := PermGroupImfGroup( G, 1 );
PermGroup(ImfMatGroup(12,9))
gap> P1.degree;
120
gap> P2 := PermGroupImfGroup( G, 2 );
PermGroupImfGroup(ImfMatGroup(12,9),2)
gap> P2.degree;
300
```

38.13 The Crystallographic Groups Library

GAP3 provides a library of crystallographic groups of dimensions 2, 3, and 4 which covers many of the data that have been listed in the book "Crystallographic groups of four-dimensional space" [BBN+78]. It has been brought into GAP3 format by Volkmar Felsch.

How to access the data of the book

Among others, the library offers functions which provide access to the data listed in the Tables 1, 5, and 6 of [BBN+78]:

- The information on the crystal families listed in Table 1 can be reproduced using the `DisplayCrystalFamily` function.
- Similarly, the `DisplayCrystalSystem` function can be used to reproduce the information on the crystal systems provided in Table 1.
- The information given in the $\mathbb{Q}$-class headlines of Table 1 can be displayed by the `DisplayQClass` function, whereas the `FpGroupQClass` function can be used to reproduce the presentations that are listed in Table 1 for the $\mathbb{Q}$-class representatives.
- The information given in the $\mathbb{Z}$-class headlines of Table 1 will be covered by the results of the `DisplayZClass` function, and the matrix generators of the $\mathbb{Z}$-class representatives can be constructed by calling the `MatGroupZClass` function.
- The `DisplaySpaceGroupType` and the `DisplaySpaceGroupGenerators` functions can be used to reproduce all of the information on the space-group types that is provided in Table 1.
- The normalizers listed in Table 5 can be reproduced by calling the `NormalizerZClass` function.
- Finally, the `CharTableQClass` function will compute the character tables listed in Table 6, whereas the isomorphism types given in Table 6 may be obtained by calling the `DisplayQClass` function.

The display functions mentioned in the above list print their output with different indentation. So, calling them in a suitably nested loop, you may produce a listing in which the information about the objects of different type will be properly indented as has been done in Table 1 of [BBN+78].

Representation of space groups in GAP3

Probably the most important function in the library is the `SpaceGroup` function which provides representatives of the affine classes of space groups. A space group of dimension n is represented by an $(n+1)$-dimensional rational matrix group as follows.

If S is an n-dimensional space group, then each element $\alpha \in S$ is an affine mapping $\alpha\colon V \to V$ of an n-dimensional $\mathbb{R}$-vector space V onto itself. Hence α can be written as the sum of an appropriate invertible linear mapping $\varphi\colon V \to V$ and a translation by some translation vector $t \in V$ such that, if we write mappings from the left, we have $\alpha(v) = \varphi(v) + t$ for all $v \in V$.

If we fix a basis of V and then replace each $v \in V$ by the column vector of its coefficients with respect to that basis (and hence V by the isomorphic column vector space $\mathbb{R}^{n\times 1}$), we can describe the linear mapping φ involved in α by an $n \times n$ matrix $M_\varphi \in GL_n(\mathbb{R})$ which acts by multiplication from the left on the column vectors in $\mathbb{R}^{n\times 1}$. Hence, if we identify V with $\mathbb{R}^{n\times 1}$, we have $\alpha(v) = M_\varphi \cdot v + t$ for all $v \in \mathbb{R}^{n\times 1}$.

Moreover, if we extend each column vector $v \in \mathbb{R}^{n\times 1}$ to a column $\begin{bmatrix} v \\ 1 \end{bmatrix}$ of length $n+1$ by adding an entry 1 in the last position and if we define an $(n+1) \times (n+1)$ matrix $M_\alpha = \left[\begin{array}{c|c} M_\varphi & t \\ \hline 0 & 1 \end{array}\right]$, we have $\begin{bmatrix} \alpha(v) \\ 1 \end{bmatrix} = M_\alpha \cdot \begin{bmatrix} v \\ 1 \end{bmatrix}$ for all $v \in \mathbb{R}^{n\times 1}$. This means that we can represent the space group S by the isomorphic group $M(S) = \{M_\alpha \mid \alpha \in S\}$. The submatrices M_φ occurring in the elements of $M(S)$ form an $n \times n$ matrix group $P(S)$, the "point group" of $M(S)$. In fact, we can choose the basis of $\mathbb{R}^{n\times 1}$ such that $M_\varphi \in GL_n(\mathbb{Z})$ and $t \in \mathbb{Q}^{n\times 1}$ for all $M_\alpha \in M(S)$. In particular, the space group representatives that are normally used by the crystallographers are of this form, and the book [BBN+78] uses the same convention.

Before we describe all available library functions in detail, we have to add three remarks.

Remark 1

The concepts used in this section are defined in chapter 1 (Basic definitions) of [BBN+78]. However, note that the definition of the concept of a crystal system given on page 16 of that book relies on the following statement about $\mathbb{Q}$-classes:

> For a $\mathbb{Q}$-class C there is a unique holohedry H such that each f. u. group in C is a subgroup of some f. u. group in H, but is not a subgroup of any f. u. group belonging to a holohedry of smaller order.

This statement is correct for dimensions 1, 2, 3, and 4, and hence the definition of "crystal system" given on page 16 of [BBN+78] is known to be unambiguous for these dimensions. However, there is a counterexample to this statement in seven-dimensional space so that the definition breaks down for some higher dimensions.

Therefore, the authors of the book have since proposed to replace this definition of "crystal system" by the following much simpler one, which has been discussed in more detail in [NPW81]. To formulate it, we use the intersections of $\mathbb{Q}$-classes and Bravais flocks as introduced on page 17 of [BBN+78], and we define the classification of the set of all $\mathbb{Z}$-classes into crystal systems as follows:

> **Definition**: A crystal system (introduced as an equivalence class of $\mathbb{Z}$-classes) consists of full geometric crystal classes. The $\mathbb{Z}$-classes of two (geometric) crystal classes belong to the same crystal system if and only if these geometric crystal classes intersect the same set of Bravais flocks of $\mathbb{Z}$-classes.

¿From this definition of a crystal system of $\mathbb{Z}$-classes one then obtains crystal systems of f. u. groups, of space-group types, and of space groups in the same manner as with the preceding definitions in the book.

The new definition is unambiguous for all dimensions. Moreover, it can be checked from the tables in the book that it defines the same classification as the old one for dimensions 1, 2, 3, and 4.

It should be noted that the concept of crystal family is well-defined independently of the dimension if one uses the "more natural" second definition of it at the end of page 17. Moreover, the first definition of crystal family on page 17 defines the same concept as the second one if the now proposed definition of crystal system is used.

Remark 2

The second remark just concerns a different terminology in the tables of [BBN+78] and in the current library. In group theory, the number of elements of a finite group usually is called the "order" of the group. This notation has been used throughout in the book. Here, however, we will follow the GAP3 conventions and use the term "size" instead.

Remark 3

The third remark concerns a problem in the use of the space groups that should be well understood.

There is an alternative to the representation of the space group elements by matrices of the form $\left[\begin{array}{c|c} M_\varphi & t \\ \hline 0 & 1 \end{array}\right]$ as described above. Instead of considering the coefficient vectors as columns we may consider them as rows. Then we can associate to each affine mapping $\alpha \in S$ an $(n+1)\times(n+1)$ matrix $\overline{M}_\alpha = \left[\begin{array}{c|c} \overline{M}_{\overline{\varphi}} & 0 \\ \hline \bar{t} & 1 \end{array}\right]$ with $\overline{M}_{\overline{\varphi}} \in GL_n(\mathbb{R})$ and $\bar{t} \in \mathbb{R}^{1\times n}$ such that $[\alpha(\overline{v}),1] = [\overline{v},1]\cdot\overline{M}_\alpha$ for all $\overline{v} \in \mathbb{R}^{1\times n}$, and we may represent S by the matrix group $\overline{M}(S) = \{\overline{M}_\alpha \mid \alpha \in S\}$. Again, we can choose the basis of $\mathbb{R}^{1\times n}$ such that $\overline{M}_{\overline{\varphi}} \in GL_n(\mathbb{Z})$ and $\bar{t} \in \mathbb{Q}^{1\times n}$ for all $\overline{M}_\alpha \in \overline{M}(S)$.

From the mathematical point of view, both approaches are equivalent. In particular, $M(S)$ and $\overline{M}(S)$ are isomorphic, for instance via the isomorphism τ mapping $M_\alpha \in M(S)$ to $(M_\alpha^{\mathrm{tr}})^{-1}$. Unfortunately, however, neither of the two is a good choice for our GAP3 library.

The first convention, using matrices which act on column vectors from the left, is not consistent with the fact that actions in GAP3 are usually from the right.

On the other hand, if we choose the second convention, we run into a problem with the names of the space groups as introduced in [BBN+78]. Any such name does not just describe the abstract isomorphism type of the respective space group S, but reflects properties of the matrix group $M(S)$. In particular, it contains as a leading part the name of the $\mathbb{Z}$-class of the associated point group $P(S)$. Since the classification of space groups by affine equivalence is tantamount to their classification by abstract isomorphism, $\overline{M}(S)$ lies in the same affine class as $M(S)$ and hence should get the same name as $M(S)$. But the point group $P(S)$ that occurs in that name is not always $\mathbb{Z}$-equivalent to the point group $\overline{P}(S)$ of $\overline{M}(S)$. For example, the isomorphism τ: $M(S) \to \overline{M}(S)$ defined above maps the $\mathbb{Z}$-class representative with the parameters $[3,7,3,2]$ (in the notation described below) to the $\mathbb{Z}$-class representative with the parameters $[3,7,3,3]$. In other words: The space group names introduced for the groups $M(S)$ in [BBN+78] lead to confusing inconsistencies if assigned to the groups $\overline{M}(S)$.

In order to avoid this confusion we decided that the first convention is the lesser evil. So the GAP3 library follows the book, and if you call the `SpaceGroup` function you will get the same space group representatives as given there. This does not cause any problems as long as you do calculations within these groups treating them just as matrix groups of certain isomorphism types. However, if it is necesary to consider the action of a space group as affine mappings on the natural lattice, you need to use the transposed representation of the space group. For this purpose the library offers the `TransposedSpaceGroup` function which not only transposes the matrices, but also adapts appropriately the associated group presentation.

Both these functions are described in detail in the following.

The library functions

`NrCrystalFamilies(` *dim* `)`

`NrCrystalFamilies` returns the number of crystal families in case of dimension *dim*. It can be used to formulate loops over the crystal families.

There are 4, 6, and 23 crystal families of dimension 2, 3, and 4, respectively.

```
gap> n := NrCrystalFamilies( 4 );
23
```

`DisplayCrystalFamily(` *dim*`,` *family* `)`

`DisplayCrystalFamily` displays for the specified crystal family essentially the same information as is provided for that family in Table 1 of [BBN+78], namely

- the family name,
- the number of parameters,
- the common rational decomposition pattern,
- the common real decomposition pattern,
- the number of crystal systems in the family, and
- the number of Bravais flocks in the family.

For details see [BBN+78].

```
gap> DisplayCrystalFamily( 4, 17 );
#I Family XVII: cubic orthogonal; 2 free parameters;
#I  Q-decomposition pattern 1+3; R-decomposition pattern 1+3;
#I  2 crystal systems; 6 Bravais flocks
gap> DisplayCrystalFamily( 4, 18 );
#I Family XVIII: octagonal; 2 free parameters;
#I  Q-irreducible; R-decomposition pattern 2+2;
#I  1 crystal system; 1 Bravais flock
gap> DisplayCrystalFamily( 4, 21 );
#I Family XXI: di-isohexagonal orthogonal; 1 free parameter;
#I  R-irreducible; 2 crystal systems; 2 Bravais flocks
```

`NrCrystalSystems(` *dim* `)`

`NrCrystalSystems` returns the number of crystal systems in case of dimension *dim*. It can be used to formulate loops over the crystal systems.

There are 4, 7, and 33 crystal systems of dimension 2, 3, and 4, respectively.

```
gap> n := NrCrystalSystems( 2 );
4
```

The following two functions are functions of crystal systems.

Each crystal system is characterized by a pair (*dim*, *system*) where *dim* is the associated dimension, and *system* is the number of the crystal system.

`DisplayCrystalSystem(` *dim*, *system* `)`

`DisplayCrystalSystem` displays for the specified crystal system essentially the same information as is provided for that system in Table 1 of [BBN+78], namely

- the number of ℚ-classes in the crystal system and
- the identification number, i. e., the tripel (*dim*, *system*, *q-class*) described below, of the ℚ-class that is the holohedry of the crystal system.

For details see [BBN+78].

```
gap> for sys in [ 1 .. 4 ] do  DisplayCrystalSystem( 2, sys );  od;
#I  Crystal system 1: 2 Q-classes; holohedry (2,1,2)
#I  Crystal system 2: 2 Q-classes; holohedry (2,2,2)
#I  Crystal system 3: 2 Q-classes; holohedry (2,3,2)
#I  Crystal system 4: 4 Q-classes; holohedry (2,4,4)
```

`NrQClassesCrystalSystem(` *dim*, *system* `)`

`NrQClassesCrystalSystem` returns the number of ℚ-classes within the given crystal system. It can be used to formulate loops over the ℚ-classes.

The following five functions are functions of ℚ-classes.

In general, the parameters characterizing a ℚ-class will form a triple (*dim*, *system*, *q-class*) where *dim* is the associated dimension, *system* is the number of the associated crystal system, and *q-class* is the number of the ℚ-class within the crystal system. However, in case of dimensions 2 or 3, a ℚ-class may also be characterized by a pair (*dim*, *IT-number*) where *IT-number* is the number in the International Tables for Crystallography [Hah83] of any space-group type lying in (a ℤ-class of) that ℚ-class, or just by the Hermann-Mauguin symbol of any space-group type lying in (a ℤ-class of) that ℚ-class.

The Hermann-Mauguin symbols which we use in GAP3 are the short Hermann-Mauguin symbols defined in the 1983 edition of the International Tables [Hah83], but any occurring indices are expressed by ordinary integers, and bars are replaced by minus signs. For example, the Hermann-Mauguin symbol $P\overline{4}2_1m$ will be represented by the string `"P-421m"`.

`DisplayQClass(` *dim*, *system*, *q-class* `)`
`DisplayQClass(` *dim*, *IT-number* `)`
`DisplayQClass(` *Hermann-Mauguin-symbol* `)`

`DisplayQClass` displays for the specified ℚ-class essentially the same information as is provided for that ℚ-class in Table 1 of [BBN+78] (except for the defining relations given there), namely

- the size of the groups in the ℚ-class,
- the isomorphism type of the groups in the ℚ-class,
- the Hurley pattern,
- the rational constituents,
- the number of ℤ-classes in the ℚ-class, and
- the number of space-group types in the ℚ-class.

For details see [BBN+78].

```
gap> DisplayQClass( "p2" );
#I   Q-class H (2,1,2): size 2; isomorphism type 2.1 = C2;
#I    Q-constituents 2*(2,1,2); cc; 1 Z-class; 1 space group
gap> DisplayQClass( "R-3" );
#I   Q-class (3,5,2): size 6; isomorphism type 6.1 = C6;
#I    Q-constituents (3,1,2)+(3,4,3); ncc; 2 Z-classes; 2 space grps
gap> DisplayQClass( 3, 195 );
#I   Q-class (3,7,1): size 12; isomorphism type 12.5 = A4;
#I    C-irreducible; 3 Z-classes; 5 space grps
gap> DisplayQClass( 4, 27, 4 );
#I   Q-class H (4,27,4): size 20; isomorphism type 20.3 = D10xC2;
#I    Q-irreducible; 1 Z-class; 1 space group
gap> DisplayQClass( 4, 29, 1 );
#I  *Q-class (4,29,1): size 18; isomorphism type 18.3 = D6xC3;
#I    R-irreducible; 3 Z-classes; 5 space grps
```

Note in the preceding examples that, as pointed out above, the term "size" denotes the order of a representative group of the specified ℚ-class and, of course, not the (infinite) class length.

`FpGroupQClass(` *dim*`,` *system*`,` *q-class* `)`
`FpGroupQClass(` *dim*`,` *IT-number* `)`
`FpGroupQClass(` *Hermann-Mauguin-symbol* `)`

`FpGroupQClass` returns a finitely presented group F, say, which is isomorphic to the groups in the specified ℚ-class.

The presentation of that group is the same as the corresponding presentation given in Table 1 of [BBN+78] except for the fact that its generators are listed in reverse order. The reason for this change is that, whenever the group in question is solvable, the resulting generators form an AG system (as defined in GAP3) if they are numbered "from the top to the bottom", and the presentation is a polycyclic power commutator presentation. The `AgGroupQClass` function described next will make use of this fact in order to construct an ag group isomorphic to F.

Note that, for any ℤ-class in the specified ℚ-class, the matrix group returned by the `MatGroupZClass` function (see below) not only is isomorphic to F, but also its generators satisfy the defining relators of F.

Besides of the usual components, the group record of F will have an additional component F`.crQClass` which saves a list of the parameters that specify the given ℚ-class.

```
gap> F := FpGroupQClass( 4, 20, 3 );
```

```
FpGroupQClass( 4, 20, 3 )
gap> F.generators;
[ f.1, f.2 ]
gap> F.relators;
[ f.1^2*f.2^-3, f.2^6, f.2^-1*f.1^-1*f.2*f.1*f.2^-4 ]
gap> F.size;
12
gap> F.crQClass;
[ 4, 20, 3 ]
```

`AgGroupQClass(` *dim*`,` *system*`,` *q-class* `)`
`AgGroupQClass(` *dim*`,` *IT-number* `)`
`AgGroupQClass(` *Hermann-Mauguin-symbol* `)`

`AgGroupQClass` returns an ag group A, say, isomorphic to the groups in the specified $\mathbb{Q}$-class, if these groups are solvable, or the value `false` (together with an appropriate warning), otherwise.

A is constructed by first establishing a finitely presented group (as it would be returned by the `FpGroupQClass` function described above) and then constructing from it an isomorphic ag group. If the underlying AG system is not yet a PAG system (see sections 24.1 and 25.1), it will be refined appropriately (and a warning will be displayed).

Besides of the usual components, the group record of A will have an additional component A`.crQClass` which saves a list of the parameters that specify the given $\mathbb{Q}$-class.

```
gap> A := AgGroupQClass( 4, 31, 3 );
#I  Warning: a non-solvable group can't be represented as an ag group
false
gap> A := AgGroupQClass( 4, 20, 3 );
#I  Warning: the presentation has been extended to get a PAG system
AgGroupQClass( 4, 20, 3 )
gap> A.generators;
[ f.1, f.21, f.22 ]
gap> A.size;
12
gap> A.crQClass;
[ 4, 20, 3 ]
```

`CharTableQClass(` *dim*`,` *system*`,` *q-class* `)`
`CharTableQClass(` *dim*`,` *IT-number* `)`
`CharTableQClass(` *Hermann-Mauguin-symbol* `)`

`CharTableQClass` returns the character table T, say, of a representative group of (a $\mathbb{Z}$-class of) the specified $\mathbb{Q}$-class.

Although the set of characters can be considered as an invariant of the specified $\mathbb{Q}$-class, the resulting table will depend on the order in which GAP3 sorts the conjugacy classes of elements and the irreducible characters and hence, in general, will not coincide with the corresponding table presented in [BBN+78].

`CharTableQClass` proceeds as follows. If the groups in the given $\mathbb{Q}$-class are solvable, then it first calls the `AgGroupQClass` and `RefinedAgSeries` functions to get an isomorphic ag group with a PAG system, and then it calls the `CharTable` function to compute the character table of that ag group. In the case of one of the five $\mathbb{Q}$-classes of dimension 4 whose groups are not solvable, it first calls the `FpGroupQClass` function to get an isomorphic finitely presented group, then it constructs a specially chosen faithful permutation representation of low degree for that group, and finally it determines the character table of the resulting permutation group again by calling the `CharTable` function.

In general, the above strategy will be much more efficient than the alternative possibilities of calling the `CharTable` function for a finitely presented group provided by the `FpGroupQClass` function or for a matrix group provided by the `MatGroupZClass` function.

```
gap> T := CharTableQClass( 4, 20, 3 );;
gap> DisplayCharTable( T );
CharTableQClass( 4, 20, 3 )

     2  2  1  1  2  2  2
     3  1  1  1  1  .  .

       1a 3a 6a 2a 4a 4b
    2P 1a 3a 3a 1a 2a 2a
    3P 1a 1a 2a 2a 4b 4a
    5P 1a 3a 6a 2a 4a 4b

X.1     1  1  1  1  1  1
X.2     1  1  1  1 -1 -1
X.3     1  1 -1 -1  A -A
X.4     1  1 -1 -1 -A  A
X.5     2 -1  1 -2  .  .
X.6     2 -1 -1  2  .  .

A = E(4)
  = ER(-1) = i
```

`NrZClassesQClass(` *dim*`,` *system*`,` *q-class* `)`
`NrZClassesQClass(` *dim*`,` *IT-number* `)`
`NrZClassesQClass(` *Hermann-Mauguin-symbol* `)`

`NrZClassesQClass` returns the number of $\mathbb{Z}$-classes within the given $\mathbb{Q}$-class. It can be used to formulate loops over the $\mathbb{Z}$-classes.

The following functions are functions of $\mathbb{Z}$-classes.

In general, the parameters characterizing a $\mathbb{Z}$-class will form a quadruple `(`*dim*`,` *system*`,` *q-class*`,` *z-class*`)` where *dim* is the associated dimension, *system* is the number of the associated crystal system, *q-class* is the number of the associated $\mathbb{Q}$-class within the crystal system, and *z-class* is the number of the $\mathbb{Z}$-class within the $\mathbb{Q}$-class. However, in case of dimensions 2 or 3, a $\mathbb{Z}$-class may also be characterized by a pair `(`*dim*`,` *IT-number*`)` where

IT-number is the number in the International Tables [Hah83] of any space-group type lying in that ℤ-class, or just by the Hermann-Mauguin symbol of any space-group type lying in that ℤ-class.

`DisplayZClass(` *dim*`,` *system*`,` *q-class*`,` *z-class* `)`
`DisplayZClass(` *dim*`,` *IT-number* `)`
`DisplayZClass(` *Hermann-Mauguin-symbol* `)`

`DisplayZClass` displays for the specified ℤ-class essentially the same information as is provided for that ℤ-class in Table 1 of [BBN+78] (except for the generating matrices of a class representative group given there), namely

- for dimensions 2 and 3, the Hermann-Mauguin symbol of a representative space-group type which belongs to that ℤ-class,
- the Bravais type,
- some decomposability information,
- the number of space-group types belonging to the ℤ-class,
- the size of the associated cohomology group.

For details see [BBN+78].

```
gap> DisplayZClass( 2, 3 );
#I    Z-class (2,2,1,1) = Z(pm): Bravais type II/I; fully Z-reducible;
#I     2 space groups; cohomology group size 2
gap> DisplayZClass( "F-43m" );
#I    Z-class (3,7,4,2) = Z(F-43m): Bravais type VI/II; Z-irreducible;
#I     2 space groups; cohomology group size 2
gap> DisplayZClass( 4, 2, 3, 2 );
#I    Z-class B (4,2,3,2): Bravais type II/II; Z-decomposable;
#I     2 space groups; cohomology group size 4
gap> DisplayZClass( 4, 21, 3, 1 );
#I   *Z-class (4,21,3,1): Bravais type XVI/I; Z-reducible;
#I     1 space group; cohomology group size 1
```

`MatGroupZClass(` *dim*`,` *system*`,` *q-class*`,` *z-class* `)`
`MatGroupZClass(` *dim*`,` *IT-number* `)`
`MatGroupZClass(` *Hermann-Mauguin-symbol* `)`

`MatGroupZClass` returns a $dim \times dim$ matrix group M, say, which is a representative of the specified ℤ-class. Its generators satisfy the defining relators of the finitely presented group which may be computed by calling the `FpGroupQClass` function (see above) for the ℚ-class which contains the given ℤ-class.

The generators of M are the same matrices as those given in Table 1 of [BBN+78]. Note, however, that they will be listed in reverse order to keep them in parallel to the abstract generators provided by the `FpGroupQClass` function (see above).

Besides of the usual components, the group record of M will have an additional component M`.crZClass` which saves a list of the parameters that specify the given ℤ-class. (In fact, in order to make the resulting group record consistent with those returned by the

NormalizerZClass or ZClassRepsDadeGroup functions described below, it also will have an additional component M.crConjugator containing just the identity element of M.)

```
gap> M := MatGroupZClass( 4, 20, 3, 1 );
MatGroupZClass( 4, 20, 3, 1 )
gap> for g in M.generators do
>  Print( "\n" ); PrintArray( g ); od; Print( "\n" );

[ [   0,   1,   0,   0 ],
  [  -1,   0,   0,   0 ],
  [   0,   0,  -1,  -1 ],
  [   0,   0,   0,   1 ] ]

[ [  -1,   0,   0,   0 ],
  [   0,  -1,   0,   0 ],
  [   0,   0,  -1,  -1 ],
  [   0,   0,   1,   0 ] ]

gap> M.size;
12
gap> M.crZClass;
[ 4, 20, 3, 1 ]
```

NormalizerZClass(*dim*, *system*, *q-class*, *z-class*)
NormalizerZClass(*dim*, *IT-number*)
NormalizerZClass(*Hermann-Mauguin-symbol*)

NormalizerZClass returns the normalizer N, say, in $GL(dim,\mathbb{Z})$ of the representative $dim \times dim$ matrix group which is constructed by the MatGroupZClass function (see above).

If the size of N is finite, then N again lies in some $\mathbb{Z}$-class. In this case, the group record of N will contain two additional components N.crZClass and N.crConjugator which provide the parameters of that $\mathbb{Z}$-class and a matrix $g \in GL(dim,\mathbb{Z})$, respectively, such that $N = g^{-1}Rg$, where R is the representative group of that $\mathbb{Z}$-class.

```
gap> N := NormalizerZClass( 4, 20, 3, 1 );
NormalizerZClass( 4, 20, 3, 1 )
gap> for g in N.generators do
>  Print( "\n" ); PrintArray( g ); od; Print( "\n" );

[ [   1,   0,   0,   0 ],
  [   0,   1,   0,   0 ],
  [   0,   0,   1,   0 ],
  [   0,   0,  -1,  -1 ] ]

[ [   1,   0,   0,   0 ],
  [   0,  -1,   0,   0 ],
  [   0,   0,  -1,  -1 ],
  [   0,   0,   1,   0 ] ]
```

```
[ [   0,   1,   0,   0 ],
  [  -1,   0,   0,   0 ],
  [   0,   0,   1,   0 ],
  [   0,   0,   0,   1 ] ]

[ [  -1,   0,   0,   0 ],
  [   0,  -1,   0,   0 ],
  [   0,   0,  -1,   0 ],
  [   0,   0,   0,  -1 ] ]

gap> N.size;
96
gap> N.crZClass;
[ 4, 20, 22, 1 ]
gap> N.crConjugator = N.identity;
true
gap> L := NormalizerZClass( 3, 42 );
NormalizerZClass( 3, 3, 2, 4 )
gap> L.size;
16
gap> L.crZClass;
[ 3, 4, 7, 2 ]
gap> L.crConjugator;
[ [ 0, 0, -1 ], [ 1, 0, 0 ], [ 0, -1, -1 ] ]
gap> M := NormalizerZClass( "C2/m" );
Group( [ [ -1, 0, 0 ], [ 0, -1, 0 ], [ 0, 0, -1 ] ],
[ [ 0, -1, 0 ], [ -1, 0, 0 ], [ 0, 0, -1 ] ],
[ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 1 ] ],
[ [ -1, 0, 0 ], [ 0, -1, 0 ], [ -1, -1, 1 ] ],
[ [ 0, 1, -1 ], [ 1, 0, -1 ], [ 0, 0, -1 ] ] )
gap> M.size;
"infinity"
gap> IsBound( M.crZClass );
false
```

`NrSpaceGroupTypesZClass(` *dim*, *system*, *q-class*, *z-class* `)`
`NrSpaceGroupTypesZClass(` *dim*, *IT-number* `)`
`NrSpaceGroupTypesZClass(` *Hermann-Mauguin-symbol* `)`

`NrSpaceGroupTypes` returns the number of space-group types within the given $\mathbb{Z}$-class. It can be used to formulate loops over the space-group types.

```
gap> N := NrSpaceGroupTypesZClass( 4, 4, 1, 1 );
13
```

Some of the $\mathbb{Z}$-classes of dimension d, say, are "maximal" in the sense that the groups in these classes are maximal finite subgroups of $GL(d, \mathbb{Z})$. Generalizing a term which is being used for dimension 4, we call the representatives of these maximal $\mathbb{Z}$-classes the "**Dade groups**" of dimension d.

`NrDadeGroups(` *dim* `)`

`NrDadeGroups` returns the number of Dade groups of dimension *dim*. It can be used to formulate loops over the Dade groups.

There are 2, 4, and 9 Dade groups of dimension 2, 3, and 4, respectively.

```
gap> NrDadeGroups( 4 );
9
```

`DadeGroup(` *dim*`,` *n* `)`

`DadeGroup` returns the nth Dade group of dimension *dim*.

```
gap> D := DadeGroup( 4, 7 );
MatGroupZClass( 4, 31, 7, 2 )
```

`DadeGroupNumbersZClass(` *dim*`,` *system*`,` *q-class*`,` *z-class* `)`
`DadeGroupNumbersZClass(` *dim*`,` *IT-number* `)`
`DadeGroupNumbersZClass(` *Hermann-Mauguin-symbol* `)`

`DadeGroupNumbersZClass` returns the set of all those integers n_i for which the n_ith Dade group of dimension *dim* contains a subgroup which, in $GL(dim,\mathbb{Z})$, is conjugate to the representative group of the given $\mathbb{Z}$-class.

```
gap> dadeNums := DadeGroupNumbersZClass( 4, 4, 1, 2 );
[ 1, 5, 8 ]
gap> for d in dadeNums do
>     D := DadeGroup( 4, d );
>     Print( D, " of size ", Size( D ), "\n" );
> od;
MatGroupZClass( 4, 20, 22, 1 ) of size 96
MatGroupZClass( 4, 30, 13, 1 ) of size 288
MatGroupZClass( 4, 32, 21, 1 ) of size 384
```

`ZClassRepsDadeGroup(` *dim*`,` *system*`,` *q-class*`,` *z-class*`,` *n* `)`
`ZClassRepsDadeGroup(` *dim*`,` *IT-number*`,` *n* `)`
`ZClassRepsDadeGroup(` *Hermann-Mauguin-symbol*`,` *n* `)`

`ZClassRepsDadeGroup` determines in the nth Dade group of dimension *dim* all those conjugacy classes whose groups are, in $GL(dim,\mathbb{Z})$, conjugate to the $\mathbb{Z}$-class representative group R, say, of the given $\mathbb{Z}$-class. It returns a list of representative groups of these conjugacy classes.

Let M be any group in the resulting list. Then the group record of M provides two components M`.crZClass` and M`.crConjugator` which contain the list of $\mathbb{Z}$-class parameters of R and a suitable matrix g from $GL(dim,\mathbb{Z})$, respectively, such that M equals $g^{-1}Rg$.

```
gap> DadeGroupNumbersZClass( 2, 2, 1, 2 );
[ 1, 2 ]
gap> ZClassRepsDadeGroup( 2, 2, 1, 2, 1 );
[ MatGroupZClass( 2, 2, 1, 2 )^[ [ 0, 1 ], [ -1, 0 ] ] ]
gap> ZClassRepsDadeGroup( 2, 2, 1, 2, 2 );
```

```
[ MatGroupZClass( 2, 2, 1, 2 )^[ [ 1, -1 ], [ 0, -1 ] ],
  MatGroupZClass( 2, 2, 1, 2 )^[ [ 1, 0 ], [ -1, 1 ] ] ]
gap> R := last[2];;
gap> R.crZClass;
[ 2, 2, 1, 2 ]
gap> R.crConjugator;
[ [ 1, 0 ], [ -1, 1 ] ]
```

The following functions are functions of space-group types.

In general, the parameters characterizing a space-group type will form a quintuple `(`*dim*`,` *system*`,` *q-class*`,` *z-class*`,` *sg-type*`)` where *dim* is the associated dimension, *system* is the number of the associated crystal system, *q-class* is the number of the associated $\mathbb{Q}$-class within the crystal system, *z-class* is the number of the $\mathbb{Z}$-class within the $\mathbb{Q}$-class, and *sg-type* is the space-group type within the $\mathbb{Z}$-class. However, in case of dimensions 2 or 3, you may instead specify a $\mathbb{Z}$-class by a pair `(`*dim*`,` *IT-number*`)` or by its Hermann-Mauguin symbol (as described above). Then the function will handle the first space-group type within that $\mathbb{Z}$-class, i. e., *sg-type* $= 1$, that is, the corresponding symmorphic space group (split extension).

`DisplaySpaceGroupType(` *dim*`,` *system*`,` *q-class*`,` *z-class*`,` *sg-type* `)`
`DisplaySpaceGroupType(` *dim*`,` *IT-number* `)`
`DisplaySpaceGroupType(` *Hermann-Mauguin-symbol* `)`

`DisplaySpaceGroupType` displays for the specified space-group type some of the information which is provided for that space-group type in Table 1 of [BBN+78], namely

- the orbit size associated with that space-group type and,
- for dimensions 2 and 3, the *IT-number* and the Hermann-Mauguin symbol.

For details see [BBN+78].

```
gap> DisplaySpaceGroupType( 2, 17 );
#I     Space-group type (2,4,4,1,1); IT(17) = p6mm; orbit size 1
gap> DisplaySpaceGroupType( "Pm-3" );
#I     Space-group type (3,7,2,1,1); IT(200) = Pm-3; orbit size 1
gap> DisplaySpaceGroupType( 4, 32, 10, 2, 4 );
#I    *Space-group type (4,32,10,2,4); orbit size 18
gap> DisplaySpaceGroupType( 3, 6, 1, 1, 4 );
#I    *Space-group type (3,6,1,1,4); IT(169) = P61, IT(170) = P65;
#I      orbit size 2; fp-free
```

`DisplaySpaceGroupGenerators(` *dim*`,` *system*`,` *q-class*`,` *z-class*`,` *sg-type* `)`
`DisplaySpaceGroupGenerators(` *dim*`,` *IT-number* `)`
`DisplaySpaceGroupGenerators(` *Hermann-Mauguin-symbol* `)`

`DisplaySpaceGroupGenerators` displays the non-translation generators of a representative space group of the specified space-group type without actually constructing that matrix group.

In more details: Let $n = dim$ be the given dimension, and let $M_1, \ldots, M_r$ be the generators of the representative $n \times n$ matrix group of the given $\mathbb{Z}$-class (this is the group which you will get if you call the `MatGroupZClass` function (see above) for that $\mathbb{Z}$-class). Then, for the given space-group type, the `SpaceGroup` function described below will construct as representative of that space-group type an $(n+1) \times (n+1)$ matrix group which is generated by the n translations which are induced by the (standard) basis vectors of the n-dimensional Euclidian space, and r additional matrices $S_1, \ldots, S_r$ of the form $S_i = \left[\begin{array}{c|c} M_i & t_i \\ \hline 0 & 1 \end{array}\right]$, where the $n \times n$ submatrices M_i are as defined above, and the t_i are n-columns with rational entries. The `DisplaySpaceGroupGenerators` function saves time by not constructing the group, but just displaying the r matrices $S_1, \ldots, S_r$.

```
gap> DisplaySpaceGroupGenerators( "P61" );
#I  The non-translation generators of SpaceGroup( 3, 6, 1, 1, 4 ) are

[ [   -1,    0,    0,    0 ],
  [    0,   -1,    0,    0 ],
  [    0,    0,    1,  1/2 ],
  [    0,    0,    0,    1 ] ]

[ [    0,   -1,    0,    0 ],
  [    1,   -1,    0,    0 ],
  [    0,    0,    1,  1/3 ],
  [    0,    0,    0,    1 ] ]
```

`SpaceGroup(` *dim*, *system*, *q-class*, *z-class*, *sg-type* `)`
`SpaceGroup(` *dim*, *IT-number* `)`
`SpaceGroup(` *Hermann-Mauguin-symbol* `)`

`SpaceGroup` returns a $(dim+1) \times (dim+1)$ matrix group S, say, which is a representative of the given space-group type (see also the description of the `DisplaySpaceGroupGenerators` function above).

```
gap> S := SpaceGroup( "P61" );
SpaceGroup( 3, 6, 1, 1, 4 )
gap> for s in S.generators do
>  Print( "\n" ); PrintArray( s ); od; Print( "\n" );

[ [   -1,    0,    0,    0 ],
  [    0,   -1,    0,    0 ],
  [    0,    0,    1,  1/2 ],
  [    0,    0,    0,    1 ] ]

[ [    0,   -1,    0,    0 ],
  [    1,   -1,    0,    0 ],
  [    0,    0,    1,  1/3 ],
  [    0,    0,    0,    1 ] ]

[ [  1,  0,  0,  1 ],
```

```
    [  0,  1,  0,  0 ],
    [  0,  0,  1,  0 ],
    [  0,  0,  0,  1 ] ]

  [ [  1,  0,  0,  0 ],
    [  0,  1,  0,  1 ],
    [  0,  0,  1,  0 ],
    [  0,  0,  0,  1 ] ]

  [ [  1,  0,  0,  0 ],
    [  0,  1,  0,  0 ],
    [  0,  0,  1,  1 ],
    [  0,  0,  0,  1 ] ]

gap> S.crSpaceGroupType;
[ 3, 6, 1, 1, 4 ]
```

Besides of the usual components, the resulting group record of S contains an additional component S`.crSpaceGroupType` which saves a list of the parameters that specify the given space-group type.

Moreover, it contains, in form of a finitely presented group, a presentation of S which is satisfied by the matrix generators. If the factor group of S by its translation normal subgroup is solvable then this presentation is chosen such that it is a polycyclic power commutator presentation. The proper way to access this presentation is to call the following function.

`FpGroup(` *S* `)`

`FpGroup` returns a finitely presented group G, say, which is isomorphic to S, where S is expected to be a space group. It is chosen such that there is an isomrphism from G to S which maps each generator of G onto the corresponding generator of S. This means, in particular, that the matrix generators of S satisfy the relators of G.

```
gap> G := FpGroup( S );
Group( g1, g2, g3, g4, g5 )
gap> for rel in G.relators do Print( rel, "\n" ); od;
g1^2*g5^-1
g2^3*g5^-1
g2^-1*g1^-1*g2*g1
g3^-1*g1^-1*g3*g1*g3^2
g3^-1*g2^-1*g3*g2*g4*g3^2
g4^-1*g1^-1*g4*g1*g4^2
g4^-1*g2^-1*g4*g2*g4*g3^-1
g4^-1*g3^-1*g4*g3
g5^-1*g1^-1*g5*g1
g5^-1*g2^-1*g5*g2
g5^-1*g3^-1*g5*g3
g5^-1*g4^-1*g5*g4
gap> # Verify that the matrix generators of S satisfy the relators of G.
gap> ForAll( G.relators,
```

```
>  rel -> MappedWord( rel, G.generators, S.generators ) = S.identity );
true
```

`TransposedSpaceGroup(` *dim*`,` *system*`,` *q-class*`,` *z-class*`,` *sg-type* `)`
`TransposedSpaceGroup(` *dim*`,` *IT-number* `)`
`TransposedSpaceGroup(` *Hermann-Mauguin-symbol* `)`
`TransposedSpaceGroup(` *S* `)`

`TransposedSpaceGroup` returns a matrix group T, say, whose generators are just the transposed generators (in the same order) of the corresponding space group S specified by the arguments. As for S, you may get a finite presentation for T via the `FpGroup` function.

The purpose of this function is explicitly discussed in the introduction to this section.

```
gap> T := TransposedSpaceGroup( S );
TransposedSpaceGroup( 3, 6, 1, 1, 4 )
gap> for m in T.generators do
>  Print( "\n" ); PrintArray( m ); od; Print( "\n" );

[ [   -1,    0,    0,    0 ],
  [    0,   -1,    0,    0 ],
  [    0,    0,    1,    0 ],
  [    0,    0,  1/2,    1 ] ]

[ [    0,    1,    0,    0 ],
  [   -1,   -1,    0,    0 ],
  [    0,    0,    1,    0 ],
  [    0,    0,  1/3,    1 ] ]

[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  1,  0,  0,  1 ] ]

[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  1,  0,  1 ] ]

[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  1,  1 ] ]
```

38.14 The Small Groups Library

This library contains all groups of order at most 1000 except for 512 and 768 up to isomorphism. There are a total of 174366 such groups.

`SmallGroup(` *size* `,` *i* `)`

The function `SmallGroup(` *size* `,` *i* `)` returns the *i*th group of order *size* in the catalogue. It will return an AgGroup, if the group is soluble and a PermGroup otherwise.

`NumberSmallGroups(` *size* `)`

The function `NumberSmallGroups(` *size* `)` returns the number of groups of the order *size*.

`AllSmallGroups(` *size* `)`

The function `AllSmallGroups(` *size* `)` returns the list of all groups of the order *size*.

`UnloadSmallGroups(` *list of sizes* `)`

It is possible to work with the catalogue of groups of small order just using the functions described above. However, the catalogue is rather large even though the groups are stored in a very compact description. Thus it might be helpful for a space efficient usage of the catalogue, to know a little bit about unloading parts of the catalogue by hand.

At the first call of one of the functions described above, the groups of order *size* are loaded and stored in a compact description. GAP will not unload them itsself again. Thus if one calls one of the above functions for a lot of different orders, then all the groups of these orders are stored. Even though the description of the groups is space efficient, this might use a lot of space. For example, if one uses the above functions to load the complete catalogue, then GAP will grow to about 12 MB of workspace.

Thus it might be interesting to unload the groups of some orders again, if they are not used anymore. This can be done by calling the function `UnloadSmallGroups(` *list of sizes* `)`

If the groups of order *size* are unloaded by hand, then GAP will of course load them again at the next call of `SmallGroup(` *size* `,` *i* `)` or one of the other functions described at the beginning of this section.

`IdGroup(` *G* `)`

Let G be a PermGroup or AgGroup of order at most 1000, but not of order 256, 512 or 768. Then the function call `IdGroup(` *G* `)` returns a tuple $[size, i]$ meaning that G is isomorphic to the i-th group in the catalogue of groups of order *size*.

Note that this package calls and uses the ANUPQ share library of GAP in a few cases.

Chapter 39

Algebras

This chapter introduces the data structures and functions for algebras in GAP3. The word **algebra** in this manual means always **associative algebra**.

At the moment GAP3 supports only finitely presented algebras and matrix algebras. For details about implementation and special functions for the different types of algebras, see 39.1 and the chapters 40 and 41.

The treatment of algebras is very similar to that of groups. For example, algebras in GAP3 are always finitely generated, since for many questions the generators play an important role. If you are not familiar with the concepts that are used to handle groups in GAP3 it might be useful to read the introduction and the overview sections in chapter 7.

Algebras are created using `Algebra` (see 39.4) or `UnitalAlgebra` (see 39.5), subalgebras of a given algebra using `Subalgebra` (see 39.8) or `UnitalSubalgebra` (see 39.9). See 39.3, and the corresponding section 7.6 in the chapter about groups for details about the distinction between parent algebras and subalgebras.

The first sections of the chapter describe the data structures (see 39.1) and the concepts of unital algebras (see 39.2) and parent algebras (see 39.3).

The next sections describe the functions for the construction of algebras, and the tests for algebras (see 39.4, 39.5, 39.6, 39.7, 39.8, 39.9, 39.10, 39.11, 39.12, 39.13, 39.14).

The next sections describe the different types of functions for algebras (see 39.15, 39.16, 39.17, 39.18, 39.19, 39.20, 39.21).

The next sections describe the operation of algebras (see 39.22, 39.23).

The next sections describe algebra homomorphisms (see 39.24, 39.25).

The next sections describe algebra elements (see 39.26, 39.27).

The last section describes the implementation of the data structures (see 39.28).

At the moment there is no implementation for ideals, cosets, and factors of algebras in GAP3, and the only available algebra homomorphisms are operation homomorphisms.

Also there is no implementation of bases for general algebras, this will be available as soon as it is for general vector spaces.

39.1 More about Algebras

Let F be a field. A ring A is called an F**-algebra** if A is an F-vector space. All algebras in GAP3 are **associative**, that is, the multiplication is associative.

An algebra always contains a **zero element** that can be obtained by subtracting an arbitrary element from itself. A discussion of **identity elements** of algebras (and of the consequences for the implementation in GAP3) can be found in 39.2.

Elements of the field F are not regarded as elements of A. The practical reason (besides the obvious mathematical one) for this is that even if the identity matrix is contained in the matrix algebra A it is not possible to write `1 + a` for adding the identity matrix to the algebra element `a`, since independent of the algebra A the meaning in GAP3 is already defined as to add `1` to all positions of the matrix `a`. Thus one has to write `One( A ) + a` or `a^0 + a` instead.

The natural **operation domains** for algebras are modules (see 39.22, and chapter 42).

39.2 Algebras and Unital Algebras

Not all algebras contain a (left and right) multiplicative neutral **identity element**, but if an algebra contains such an identity element it is unique.

If an algebra A contains a multiplicative neutral element then in general it cannot be derived from an arbitrary element a of A by forming a/a or a^0, since these operations may be not defined for the algebra A.

More precisely, it may be possible to invert a or raise it to the zero-th power, but A is not necessarily closed under these operations. For example, if a is a square matrix in GAP3 then we can form a^0 which is the identity matrix of the same size and over the same field as a.

On the other hand, an algebra may have a multiplicative neutral element that is **not** equal to the zero-th power of elements (see 39.16).

In many cases, however, the zero-th power of algebra elements is well-defined, with the result again in the algebra. This holds for example for all finitely presented algebras (see chapter 40) and all those matrix algebras whose generators are the generators of a finite group.

For practical purposes it is useful to distinguish general **algebras** and **unital algebras**.

A unital algebra in GAP3 is an algebra U that is **known to contain** zero-th powers of elements, and all functions may assume this. A not unital algebra A may contain zero-th powers of elements or not, and no function for A should assume existence or nonexistence of these elements in A. So it may be possible to view A as a unital algebra using `AsUnitalAlgebra(` A `)` (see 39.12), and of course it is always possible to view a unital algebra as algebra using `AsAlgebra(` U `)` (see 39.11).

A can have unital subalgebras, and of course U can have subalgebras that are not unital.

The images of unital algebras under operation homomorphisms are either unital or trivial, since the identity of the source acts trivially, so its image under the homomorphism is the identity of the image.

The following example shows the main differences between algebras and unital algebras.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
```

```
gap> alg1:= Algebra( Rationals, [ a ] );
Algebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> id:= a^0;
[ [ 1, 0 ], [ 0, 1 ] ]
gap> id in alg1;
false
gap> alg2:= UnitalAlgebra( Rationals, [ a ] );
UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> id in alg2;
true
gap> alg3:= AsAlgebra( alg2 );
Algebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ]
 ] )
gap> alg3 = alg2;
true
gap> AsUnitalAlgebra( alg1 );
Error, <D> is not unital
```

We see that if we want the identity matrix to be contained in an algebra that is not known to be unital, it might be necessary to add it to the generators. If we would not have the possibility to define unital algebras, this would lead to the strange situations that a two-generator algebra means an algebra generated by one nonidentity generator and the identity matrix, or that an algebra is free on the set X but is generated as algebra by the set X plus the identity.

39.3 Parent Algebras and Subalgebras

GAP3 distinguishs between parent algebras and subalgebras of parent algebras. The concept is the same as that for groups (see 7.6), so here it is only sketched.

Each subalgebra belongs to a unique parent algebra, the so-called **parent** of the subalgebra. A parent algebra is its own parent.

Parent algebras are constructed by `Algebra` and `UnitalAlgebra`, subalgebras are constructed by `Subalgebra` and `UnitalSubalgebra`. The parent of the first argument of `Subalgebra` will be the parent of the constructed subalgebra.

Those algebra functions that take more than one algebra as argument require that the arguments have a common parent. Take for instance `Centralizer`. It takes two arguments, an algebra A and an algebra B, where either A is a parent algebra, and B is a subalgebra of this parent algebra, or A and B are subalgebras of a common parent algebra P, and returns the centralizer of B in A. This is represented as a subalgebra of the common parent of A and B. Note that a subalgebra of a parent algebra need not be a proper subalgebra.

An exception to this rule is again the set theoretic function `Intersection` (see 4.12), which allows to intersect algebras with different parents.

Whenever you have two subalgebras which have different parent algebras but have a common superalgebra A you can use `AsSubalgebra` or `AsUnitalSubalgebra` (see 39.13, 39.14) in order to construct new subalgebras which have a common parent algebra A.

Note that subalgebras of unital algebras need not be unital (see 39.2).

The following sections describe the functions related to this concept (see 39.4, 39.5, 39.6, 39.7, 39.11, 39.12, 39.8, 39.9, 39.13, 39.14, and also 7.7, 7.8).

39.4 Algebra

`Algebra(` *U* `)`

returns a parent algebra A which is isomorphic to the parent algebra or subalgebra U.

`Algebra(` *F*`,` *gens* `)`
`Algebra(` *F*`,` *gens*`,` *zero* `)`

returns a parent algebra over the field F and generated by the algebra elements in the list *gens*. The zero element of this algebra may be entered as *zero*; this is necessary whenever *gens* is empty.

```
gap> a:= [ [ 1 ] ];;
gap> alg:= Algebra( Rationals, [ a ] );
Algebra( Rationals, [ [ [ 1 ] ] ] )
gap> alg.name:= "alg";;
gap> sub:= Subalgebra( alg, [] );
Subalgebra( alg, [  ] )
gap> Algebra( sub );
Algebra( Rationals, [ [ [ 0 ] ] ] )
gap> Algebra( Rationals, [], 0*a );
Algebra( Rationals, [ [ [ 0 ] ] ] )
```

The algebras returned by `Algebra` are not unital. For constructing unital algebras, use 39.5 `UnitalAlgebra`.

39.5 UnitalAlgebra

`UnitalAlgebra(` *U* `)`

returns a unital parent algebra A which is isomorphic to the parent algebra or subalgebra U. If U is not unital it is checked whether the zero-th power of elements is contained in U, and if not an error is signalled.

`UnitalAlgebra(` *F*`,` *gens* `)`
`UnitalAlgebra(` *F*`,` *gens*`,` *zero* `)`

returns a unital parent algebra over the field F and generated by the algebra elements in the list *gens*. The zero element of this algebra may be entered as *zero*; this is necessary whenever *gens* is empty.

```
gap> alg1:= UnitalAlgebra( Rationals, [ NullMat( 2, 2 ) ] );
UnitalAlgebra( Rationals, [ [ [ 0, 0 ], [ 0, 0 ] ] ] )
gap> alg2:= UnitalAlgebra( Rationals, [], NullMat( 2, 2 ) );
UnitalAlgebra( Rationals, [ [ [ 0, 0 ], [ 0, 0 ] ] ] )
gap> alg3:= Algebra( alg1 );
Algebra( Rationals, [ [ [ 0, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ]
 ] )
gap> alg1 = alg3;
true
```

```
gap> AsUnitalAlgebra( alg3 );
UnitalAlgebra( Rationals,
[ [ [ 0, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] )
```

The algebras returned by `UnitalAlgebra` are unital. For constructing algebras that are not unital, use 39.4 `Algebra`.

39.6 IsAlgebra

`IsAlgebra(` *obj* `)`

returns `true` if *obj*, which can be an object of arbitrary type, is a parent algebra or a subalgebra and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsAlgebra( FreeAlgebra( GF(2), 0 ) );
true
gap> IsAlgebra( 1/2 );
false
```

39.7 IsUnitalAlgebra

`IsUnitalAlgebra(` *obj* `)`

returns `true` if *obj*, which can be an object of arbitrary type, is a unital parent algebra or a unital subalgebra and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsUnitalAlgebra( FreeAlgebra( GF(2), 0 ) );
true
gap> IsUnitalAlgebra( Algebra( Rationals, [ [ [ 1 ] ] ] ) );
false
```

Note that the function does **not** check whether *obj* is an algebra that contains the zero-th power of elements, but just checks whether *obj* is an algebra with flag `isUnitalAlgebra`.

39.8 Subalgebra

`Subalgebra(` A`,` *gens* `)`

returns the subalgebra of the algebra A generated by the elements in the list *gens*.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
gap> b:= [ [ 0, 0 ], [ 0, 1 ] ] ;;
gap> alg:= Algebra( Rationals, [ a, b ] );;
gap> alg.name:= "alg";;
gap> s:= Subalgebra( alg, [ a ] );
Subalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> s = alg;
false
gap> s:= UnitalSubalgebra( alg, [ a ] );
UnitalSubalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> s = alg;
```

```
true
```

Note that `Subalgebra`, `UnitalSubalgebra`, `AsSubalgebra` and `AsUnitalSubalgebra` are the only functions in which the name `Subalgebra` does not refer to the mathematical terms subalgebra and superalgebra but to the implementation of algebras as subalgebras and parent algebras.

39.9 UnitalSubalgebra

`UnitalSubalgebra(` *A*`,` *gens* `)`

returns the unital subalgebra of the algebra A generated by the elements in the list *gens*. If A is not (known to be) unital then first it is checked that A really contains the zero-th power of elements.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
gap> b:= [ [ 0, 0 ], [ 0, 1 ] ] ;;
gap> alg:= Algebra( Rationals, [ a, b ] );;
gap> alg.name:= "alg";;
gap> s:= Subalgebra( alg, [ a ] );
Subalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> s = alg;
false
gap> s:= UnitalSubalgebra( alg, [ a ] );
UnitalSubalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> s = alg;
true
```

Note that `Subalgebra`, `UnitalSubalgebra`, `AsSubalgebra` and `AsUnitalSubalgebra` are the only functions in which the name `Subalgebra` does not refer to the mathematical terms subalgebra and superalgebra but to the implementation of algebras as subalgebras and parent algebras.

39.10 IsSubalgebra

`IsSubalgebra(` *A*`,` *U* `)`

returns `true` if U is a subalgebra of A and `false` otherwise.

Note that A and U must have a common parent algebra. This function returns `true` if and only if the set of elements of U is a subset of the set of elements of A.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
gap> b:= [ [ 0, 0 ], [ 0, 1 ] ] ;;
gap> alg:= Algebra( Rationals, [ a, b ] );;
gap> alg.name:= "alg";;
gap> IsSubalgebra( alg, alg );
true
gap> s:= UnitalSubalgebra( alg, [ a ] );
UnitalSubalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> IsSubalgebra( alg, s );
true
```

39.11 AsAlgebra

`AsAlgebra(` *D* `)`
`AsAlgebra(` *F*`,` *D* `)`

Let D be a domain. `AsAlgebra` returns an algebra A over the field F such that the set of elements of D is the same as the set of elements of A if this is possible. If D is an algebra the argument F may be omitted, the coefficients field of D is taken as coefficients field of F in this case.

If D is a list of algebra elements these elements must form a algebra. Otherwise an error is signalled.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ] * Z(2);;
gap> AsAlgebra( GF(2), [ a, 0*a ] );
Algebra( GF(2), [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ] ] )
```

Note that this function returns a parent algebra or a subalgebra of a parent algebra depending on D. In order to convert a subalgebra into a parent algebra you must use `Algebra` or `UnitalAlgebra` (see 39.4, 39.5).

39.12 AsUnitalAlgebra

`AsUnitalAlgebra(` *D* `)`
`AsUnitalAlgebra(` *F*`,` *D* `)`

Let D be a domain. `AsUnitalAlgebra` returns a unital algebra A over the field F such that the set of elements of D is the same as the set of elements of A if this is possible. If D is an algebra the argument F may be omitted, the coefficients field of D is taken as coefficients field of F in this case.

If D is a list of algebra elements these elements must form a unital algebra. Otherwise an error is signalled.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ] * Z(2);;
gap> AsUnitalAlgebra( GF(2), [ a, a^0, 0*a, a^0-a ] );
UnitalAlgebra( GF(2), [ [ [ 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ] ] )
```

Note that this function returns a parent algebra or a subalgebra of a parent algebra depending on D. In order to convert a subalgebra into a parent algebra you must use `Algebra` or `UnitalAlgebra` (see 39.4, 39.5).

39.13 AsSubalgebra

`AsSubalgebra(` *A*`,` *U* `)`

Let A be a parent algebra and U be a parent algebra or a subalgebra with a possibly different parent algebra, such that the generators of U are elements of A. `AsSubalgebra` returns a new subalgebra S such that S has parent algebra A and is generated by the generators of U.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
gap> b:= [ [ 0, 0 ], [ 0, 1 ] ] ;;
```

```
gap> alg:= Algebra( Rationals, [ a, b ] );;
gap> alg.name:= "alg";;
gap> s:= Algebra( Rationals, [ a ] );
Algebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> AsSubalgebra( alg, s );
Subalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
```

Note that `Subalgebra`, `UnitalSubalgebra`, `AsSubalgebra` and `AsUnitalSubalgebra` are the only functions in which the name `Subalgebra` does not refer to the mathematical terms subalgebra and superalgebra but to the implementation of algebras as subalgebras and parent algebras.

39.14 AsUnitalSubalgebra

`AsUnitalSubalgebra(` A, U `)`

Let A be a parent algebra and U be a parent algebra or a subalgebra with a possibly different parent algebra, such that the generators of U are elements of A. `AsSubalgebra` returns a new unital subalgebra S such that S has parent algebra A and is generated by the generators of U. If U or A do not contain the zero-th power of elements an error is signalled.

```
gap> a:= [ [ 1, 0 ], [ 0, 0 ] ];;
gap> b:= [ [ 0, 0 ], [ 0, 1 ] ];;
gap> alg:= Algebra( Rationals, [ a, b ] );;
gap> alg.name:= "alg";;
gap> s:= UnitalAlgebra( Rationals, [ a ] );
UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> AsSubalgebra( alg, s );
Subalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] )
gap> AsUnitalSubalgebra( alg, s );
UnitalSubalgebra( alg, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
```

Note that `Subalgebra`, `UnitalSubalgebra`, `AsSubalgebra` and `AsUnitalSubalgebra` are the only functions in which the name `Subalgebra` does not refer to the mathematical terms subalgebra and superalgebra but to the implementation of algebras as subalgebras and parent algebras.

39.15 Operations for Algebras

A `^` n

The operator `^` evaluates to the n-fold direct product of A, viewed as a free A-module.

```
gap> a:= FreeAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ a.1, a.2 ] )
gap> a^2;
Module( UnitalAlgebra( GF(2), [ a.1, a.2 ] ),
[ [ a.one, a.zero ], [ a.zero, a.one ] ] )
```

a `in` A

The operator `in` evaluates to `true` if a is an element of A and `false` otherwise. a must be an element of the parent algebra of A.

```
gap> a.1^3 + a.2 in a;
true
gap> 1 in a;
false
```

39.16 Zero and One for Algebras

`Zero(` A `)`

returns the additive neutral element of the algebra A.

`One(` A `)`

returns the (right and left) multiplicative neutral element of the algebra A if this exists, and `false` otherwise. If A is a unital algebra then this element is obtained on raising an arbitrary element to the zero-th power (see 39.2).

```
gap> a:= Algebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );
Algebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> Zero( a );
[ [ 0, 0 ], [ 0, 0 ] ]
gap> One( a );
[ [ 1, 0 ], [ 0, 0 ] ]
gap> a:= UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );
UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> Zero( a );
[ [ 0, 0 ], [ 0, 0 ] ]
gap> One( a );
[ [ 1, 0 ], [ 0, 1 ] ]
```

39.17 Set Theoretic Functions for Algebras

As already mentioned in the introduction of the chapter, algebras are domains. Thus all set theoretic functions, for example `Intersection` and `Size` can be applied to algebras. All set theoretic functions not mentioned here are not treated specially for algebras.

`Elements(` A `)`

computes the elements of the algebra A using a Dimino algorithm. The default function for algebras computes a vector space basis at the same time.

`Intersection(` A`,` H `)`

returns the intersection of A and H either as set of elements or as an algebra record.

`IsSubset(` A`,` H `)`

If A and H are algebras then `IsSubset` tests whether the generators of H are elements of A. Otherwise `DomainOps.IsSubset` is used.

`Random(` A `)`

returns a random element of the algebra A. This requires the computation of a vector space basis.

See also 41.5, 40.6 for the set theoretic functions for the different types of algebras.

39.18 Property Tests for Algebras

The following property tests (cf. 7.45) are available for algebras.

`IsAbelian( A )`

returns `true` if the algebra A is abelian and `false` otherwise. An algebra A is **abelian** if and only if for every $a, b \in A$ the equation $a * b = b * a$ holds.

`IsCentral( A, U )`

returns `true` if the algebra A centralizes the algebra U and `false` otherwise. An algebra A **centralizes** an algebra U if and only if for all $a \in A$ and for all $u \in U$ the equation $a * u = u * a$ holds. Note that U need not to be a subalgebra of A but they must have a common parent algebra.

`IsFinite( A )`

returns `true` if the algebra A is finite, and `false` otherwise.

`IsTrivial( A )`

returns `true` if the algebra A consists only of the zero element, and `false` otherwise. If A is a unital algebra it is of course never trivial.

All tests expect a parent algebra or subalgebra and return `true` if the algebra has the property and `false` otherwise. Some functions may not terminate if the given algebra has an infinite set of elements. A warning may be printed in such cases.

```
gap> IsAbelian( FreeAlgebra( GF(2), 2 ) );
false
gap> a:= UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );
UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> a.name:= "a";;
gap> s1:= Subalgebra( a, [ One(a) ] );
Subalgebra( a, [ [ [ 1, 0 ], [ 0, 1 ] ] ] )
gap> IsCentral( a, s1 ); IsFinite( s1 );
true
false
gap> s2:= Subalgebra( a, [] );
Subalgebra( a, [  ] )
gap> IsFinite( s2 ); IsTrivial( s2 );
true
true
```

39.19 Vector Space Functions for Algebras

A finite dimensional F-algebra A is always a finite dimensional F-vector space. Thus in GAP3, an algebra is a vector space (see 9.2), and vector space functions such as `Base` and `Dimension` are applicable to algebras.

```
gap> a:= UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );
UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] )
gap> Dimension( a );
2
gap> Base( a );
```

```
[ [ [ 1, 0 ], [ 0, 1 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
```

The vector space structure is used also by the set theoretic functions.

39.20 Algebra Functions for Algebras

The functions desribed in this section compute certain subalgebras of a given algebra, e.g., `Centre` computes the centre of an algebra.

They return algebra records as described in 39.28 for the computed subalgebras. Some functions may not terminate if the given algebra has an infinite set of elements, while other functions may signal an error in such cases.

Here the term "subalgebra" is used in a mathematical sense. But in GAP3, every algebra is either a parent algebra or a subalgebra of a unique parent algebra. If you compute the centre C of an algebra U with parent algebra A then C is a subalgebra of U but its parent algebra is A (see 39.3).

`Centralizer(` A`,` x `)`

`Centralizer(` A`,` U `)`

returns the centralizer of an element x in A where x must be an element of the parent algebra of A, resp. the centralizer of the algebra U in A where both algebras must have a common parent.

The **centralizer** of an element x in A is defined as the set C of elements c of A such that c and x commute.

The **centralizer** of an algebra U in A is defined as the set C of elements c of A such that c commutes with every element of U.

```
gap> a:= MatAlgebra( GF(2), 2 );;
gap> a.name:= "a";;
gap> m:= [ [ 1, 1 ], [ 0, 1 ] ] * Z(2);;
gap> Centralizer( a, m );
UnitalSubalgebra( a, [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2) ] ] ] )
```

`Centre(` A `)`

returns the centre of A (that is, the centralizer of A in A).

```
gap> c:= Centre( a );
UnitalSubalgebra( a, [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ] )
```

`Closure(` U`,` a `)`

`Closure(` U`,` S `)`

Let U be an algebra with parent algebra A and let a be an element of A. Then `Closure` returns the closure C of U and a as subalgebra of A. The closure C of U and a is the subalgebra generated by U and a.

Let U and S be two algebras with a common parent algebra A. Then `Closure` returns the subalgebra of A generated by U and S.

```
gap> Closure( c, m );
UnitalSubalgebra( a, [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ] )
```

39.21 TrivialSubalgebra

`TrivialSubalgebra( U )`

Let U be an algebra with parent algebra A. Then `TrivialSubalgebra` returns the trivial subalgebra T of U, as subalgebra of A.

```
gap> a:= MatAlgebra( GF(2), 2 );;
gap> a.name:= "a";;
gap> TrivialSubalgebra( a );
Subalgebra( a, [  ] )
```

39.22 Operation for Algebras

`Operation( A, M )`

Let A be an F-algebra for a field F, and M an A-module of F-dimension n. With respect to a chosen F-basis of M, the action of an element of A on M can be described by an $n \times n$ matrix over F. This induces an algebra homomorphism from A onto a matrix algebra A_M, with action on its natural module equivalent to the action of A on M. The matrix algebra A_M can be computed as `Operation( A, M )`.

`Operation( A, B )`

returns the operation of the algebra A on an A-module M with respect to the vector space basis B of M.

Note that contrary to the situation for groups, the operation domains of algebras are not lists of elements but domains.

For constructing the algebra homomorphism from A onto A_M, and the module homomorphism from M onto the equivalent A_M-module, see 39.23 and 42.17, respectively.

```
gap> a:= UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );;
gap> m:= Module( a, [ [ 1, 0 ] ] );;
gap> op:= Operation( a, m );
UnitalAlgebra( Rationals, [ [ [ 1 ] ] ] )
gap> mat1:= PermutationMat( (1,2,3), 3, GF(2) );;
gap> mat2:= PermutationMat(   (1,2), 3, GF(2) );;
gap> u:= Algebra( GF(2), [ mat1, mat2 ] );; u.name:= "u";;
gap> nat:= NaturalModule( u );; nat.name:= "nat";;
gap> q:= nat / FixedSubmodule( nat );;
gap> op1:= Operation( u, q );
UnitalAlgebra( GF(2), [ [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ],
  [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ] )
gap> b:= Basis( q, [ [ 0, 1, 1 ], [ 0, 0, 1 ] ] * Z(2) );;
gap> op2:= Operation( u, b );
UnitalAlgebra( GF(2), [ [ [ Z(2)^0, Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ],
  [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ] )
gap> IsEquivalent( NaturalModule( op1 ), NaturalModule( op2 ) );
true
```

If the dimension of M is zero then the elements of A_M cannot be represented as GAP3 matrices. The result is a null algebra, see 41.9, `NullAlgebra`.

39.23 OperationHomomorphism for Algebras

`OperationHomomorphism(` *A*`,` *B* `)`

returns the algebra homomorphism (see 39.24) with source A and range B, provided that B is a matrix algebra that was constructed as operation of A on a suitable module M using `Operation(` *A*`,` *M* `)`, see 39.22.

```
gap> ophom:= OperationHomomorphism( a, op );
OperationHomomorphism( UnitalAlgebra( Rationals,
[ [ [ 1, 0 ], [ 0, 0 ] ] ] ), UnitalAlgebra( Rationals,
[ [ [ 1 ] ] ] ) )
gap> Image( ophom, a.1 );
[ [ 1 ] ]
gap> Image( ophom, Zero( a ) );
[ [ 0 ] ]
gap> PreImagesRepresentative( ophom, [ [ 2 ] ] );
[ [ 2, 0 ], [ 0, 2 ] ]
```

39.24 Algebra Homomorphisms

An **algebra homomorphism** ϕ is a mapping that maps each element of an algebra A, called the source of ϕ, to an element of an algebra B, called the range of ϕ, such that for each pair $x, y \in A$ we have $(xy)^\phi = x^\phi y^\phi$ and $(x+y)^\phi = x^\phi + y^\phi$.

An algebra homomorphism of unital algebras is **unital** if the zero-th power of elements in the source is mapped to the zero-th power of elements in the range.

At the moment, only operation homomorphisms are supported in GAP3 (see 39.23).

39.25 Mapping Functions for Algebra Homomorphisms

This section describes how the mapping functions defined in chapter 43 are implemented for algebra homomorphisms. Those functions not mentioned here are implemented by the default functions described in the respective sections.

`Image(` *hom* `)`
`Image(` *hom*`,` *H* `)`
`Images(` *hom*`,` *H* `)`

The image of a subalgebra under a algebra homomorphism is computed by computing the images of a set of generators of the subalgebra, and the result is the subalgebra generated by those images.

`PreImagesRepresentative(` *hom*`,` *elm* `)`

```
gap> a:= UnitalAlgebra( Rationals, [ [ [ 1, 0 ], [ 0, 0 ] ] ] );;
gap> a.name:= "a";;
gap> m:= Module( a, [ [ 1, 0 ] ] );;
gap> op:= Operation( a, m );
UnitalAlgebra( Rationals, [ [ [ 1 ] ] ] )
```

```
gap> ophom:= OperationHomomorphism( a, op );
OperationHomomorphism( a, UnitalAlgebra( Rationals, [ [ [ 1 ] ] ] ) )
gap> Image( ophom, a.1 );
[ [ 1 ] ]
gap> Image( ophom, Zero( a ) );
[ [ 0 ] ]
gap> PreImagesRepresentative( ophom, [ [ 2 ] ] );
[ [ 2, 0 ], [ 0, 2 ] ]
```

39.26 Algebra Elements

This section describes the operations and functions available for algebra elements.

Note that algebra elements may exist independently of an algebra, e.g., you can write down two matrices and compute their sum and product without ever defining an algebra that contains them.

Comparisons of Algebra Elements

g = *h*

evaluates to `true` if the algebra elements *g* and *h* are equal and to `false` otherwise.

g <> *h*

evaluates to `true` if the algebra elements *g* and *h* are not equal and to `false` otherwise.

g < *h*
g <= *h*
g >= *h*
g > *h*

The operators `<`, `<=`, `>=` and `>` evaluate to `true` if the algebra element *g* is strictly less than, less than or equal to, greater than or equal to and strictly greater than the algebra element *h*. There is no general ordering on all algebra elements, so *g* and *h* should lie in the same parent algebra. Note that for elements of finitely presented algebra, comparison means comparison with respect to the underlying free algebra (see 40.9).

Arithmetic Operations for Algebra Elements

a * *b*
a + *b*
a - *b*

The operators `*`, `+` and `-` evaluate to the product, sum and difference of the two algebra elements *a* and *b*. The operands must of course lie in a common parent algebra, otherwise an error is signalled.

a / *c*

returns the quotient of the algebra element *a* by the nonzero element *c* of the base field of the algebra.

a ^ *i*

returns the i-th power of an algebra element a and a positive integer i. If i is zero or negative, perhaps the result is not defined, or not contained in the algebra generated by a.

list + *a*
a + *list*
list * *a*
a * *list*

In this form the operators `+` and `*` return a new list where each entry is the sum resp. product of a and the corresponding entry of *list*. Of course addition resp. multiplication must be defined between a and each entry of *list*.

39.27 IsAlgebraElement

`IsAlgebraElement(` *obj* `)`

returns `true` if *obj*, which may be an object of arbitrary type, is an algebra element, and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsAlgebraElement( (1,2) );
false
gap> IsAlgebraElement( NullMat( 2, 2 ) );
true
gap> IsAlgebraElement( FreeAlgebra( Rationals, 1 ).1 );
true
```

39.28 Algebra Records

Algebras and their subalgebras are represented by records. Once an algebra record is created you may add record components to it but you must **not** alter information already present.

Algebra records must always contain the components `isDomain` and `isAlgebra`. Subalgebras contain an additional component `parent`. The components `generators`, `zero` and `one` are not necessarily contained.

The contents of important record components of an algebra A is described below.

The **category components** are

`isDomain`
: is `true`.

`isAlgebra`
: is `true`.

`isUnitalAlgebra`
: is present (and then `true`) if A is a unital algebra.

The **identification components** are

`field`
: is the coefficient field of A.

`generators`
: is a list of algebra generators. Duplicate generators are allowed, also the algebra

zero may be among the generators. Note that once created this entry must never be changed, as most of the other entries depend on `generators`. If `generators` is not bound it can be computed using `Generators`.

`parent`
: if present this contains the algebra record of the parent algebra of a subalgebra A, otherwise A itself is a parent algebra.

`zero`
: is the additive neutral element of A, can be computed using `Zero`.

The component `operations` contains the **operations record** of A. This will usually be one of `AlgebraOps`, `UnitalAlgebraOps`, or a record for more specific algebras.

39.29 FFList

`FFList(` *F* `)`

returns for a finite field F a list l of all elements of F in an ordering that is compatible with the ordering of field elements in the MeatAxe share library (see chapter 69).

The element of F corresponding to the number n is l`[` n`+1 ]`, and the canonical number of the field element z is `Position(` l`,` z `) -1`.

```
gap> FFList( GF( 8 ) );
[ 0*Z(2), Z(2)^0, Z(2^3), Z(2^3)^3, Z(2^3)^2, Z(2^3)^6, Z(2^3)^4,
  Z(2^3)^5 ]
```

(This program was originally written by Meinolf Geck.)

Chapter 40

Finitely Presented Algebras

This chapter contains the description of functions dealing with finitely presented algebras. The first section informs about the data structures (see 40.1), the next sections tell how to construct free and finitely presented algebras (see 40.2, 40.3), and what functions can be applied to them (see 40.4, 40.6, 40.5, 40.7), and the final sections introduce functions for elements of finitely presented algebras (see 40.8, 40.9, 40.10, 40.11).

For a detailed description of operations of finitely presented algebras on modules, see chapter 73.

40.1 More about Finitely Presented Algebras

Free Algebras

Let X be a finite set, and F a field. The **free algebra** A on X over F can be regarded as the semigroup ring of the free monoid on X over F. Addition and multiplication of elements are performed by dealing with sums of words in abstract generators, with coefficients in F.

Free algebras and also their subalgebras in GAP3 are always **unital**, that is, for an element a in a subalgebra A of a free algebra always the element a^0 lies in A (see 39.2). Thus the free algebra on the empty set over a field F is defined to consist of all elements fe where f is in F, and e is the multiplicative neutral element, corresponding to the empty word.

Free algebras are useful when dealing with other algebras, like matrix algebras, since they allow to handle expressions in terms of the generators. This is just a generalization of handling words in abstract generators and concrete group elements in parallel, as is done for example in `MappedWord` (see 22.12) or functions that construct images and preimages under homomorphisms. This mechanism is also provided for the records representing matrices in the MeatAxe share library (see chapter 69).

Finitely Presented Algebras

A **finitely presented algebra** is defined as quotient A/I of a free algebra A by a two-sided ideal I in A that is generated by a finite set S of elements in F.

Thus computations with finitely presented algebras are similar to those with finitely presented groups. For example, in general it is impossible to decide whether two elements of the free algebra A are equal modulo I.

For finitely presented groups a permutation representation on the cosets of a subgroup of finite index can be computed by the Todd-Coxeter coset enumeration method. An analogue of this method for finitely presented algebras is Steve Linton's Vector Enumeration method that tries to compute a matrix representation of the action on a quotient of a free module of the algebra. This method is available in GAP3 as a share library (see chapter 73, and the references there), and this makes finitely presented algebra in GAP3 more than an object one can only use for the obvious arithmetics with elements of free algebras.

GAP3 only handles the data structures, all the work in done by the standalone program. Thus all functions for finitely presented algebras, like `Size`, delegate the work to the Vector Enumeration program.

Note that (contrary to the situation in finitely presented groups, and several places in Vector Enumeration) relators are meant to be equal to **zero**, not to the identity. Two examples for this. If `x^2 - a.one` is a relator in the presentation of the algebra `a`, with `x` a generator, then `x` is an involution. If `x^2` is a relator then `x` is nilpotent. If the generator `x` occurs in relators of the form `x * v - a.one` and `w * x - a.one`, for `v` and `w` elements of the free algebra, then `x` is known to be invertible.

The Vector Enumeration package is loaded automatically as soon as it is needed. You can also load it explicitly using

```
gap> RequirePackage( "ve" );
```

Elements of Finitely Presented Algebras

The elements of finitely presented algebras in GAP3 are records that store lists of coefficients and of words in abstract generators. Note that the elements of the ground field are not regarded as elements of the algebra, especially the identity and zero element are denoted by `a.one` and `a.zero`, respectively. Functions and operators for elements of finitely presented algebras are listed in 40.9.

Implementation of Functions for Finitely Presented Algebras

Every question about a finitely presented algebra A that cannot be answered from the presentation directly is delegated to an isomorphic matrix algebra M using the Vector Enumeration share library. This may be impossible because the dimension of an isomorphic matrix algebra is too large. But for small A it seems to be valuable.

For example, if one asks for the size of A, Vector Enumeration tries to find such a matrix algebra M, and then GAP3 computes its size. M and the isomorphism between A and M are stored in the component A`.matAlgebraA`, so Vector Enumeration is called only once for A.

40.2 FreeAlgebra

`FreeAlgebra(` *F*`,` *rank* `)`
`FreeAlgebra(` *F*`,` *rank*`,` *name* `)`
`FreeAlgebra(` *F*`,` *name1*`,` *name2*`, ... )`

return a free algebra with ground field F. In the first two forms an algebra on *rank* free generators is returned, their names will be *name*`.1`, …, *name*`.`*rank*, the default for *name* is the string `"a"`.

```
gap> a:= FreeAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ a.1, a.2 ] )
gap> b:= FreeAlgebra( Rationals, "x", "y" );
UnitalAlgebra( Rationals, [ x, y ] )
gap> x:= b.1;
x
```

Finitely presented algebras are constructed from free algebras via factoring by a suitable ideal (see 40.5).

40.3 FpAlgebra

`FpAlgebra(` *A* `)`

returns a finitely presented algebra isomorphic to the algebra A. At the moment this is implemented only for matrix algebras and finitely presented algebras.

```
gap> a:= FreeAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ a.1, a.2 ] )
gap> a:= a / [ a.one+a.1^2, a.one+a.2^2, a.one+(a.1*a.2)^3 ];;
gap> a.name:= "a";; s:= Subalgebra( a, [ a.2 ] );;
gap> f:= FpAlgebra( s );
UnitalAlgebra( GF(2), [ a.1 ] )
gap> PrintDefinitionFpAlgebra( f, "f" );
f:= FreeAlgebra( GF(2), "a.1" );
f:= f / [ f.one+f.1^2 ];
```

`FpAlgebra(` *F*`,` *fpgroup* `)`

returns the **group algebra** of the finitely presented group *fpgroup* over the field F, this is the algebra of formal linear combinations of elements of *fpgroup*, with coefficients in F; in this case the number of algebra generators is twice the number of group generators, the first half corresponding to the group generators, the second half to their inverses.

```
gap> f:= FreeGroup( 2 );;
gap> s3:= f / [ f.1^2, f.2^2, (f.1*f.2)^3 ];
Group( f.1, f.2 )
gap> a:= FpAlgebra( GF(2), s3 );
UnitalAlgebra( GF(2), [ a.1, a.2, a.3, a.4 ] )
```

40.4 IsFpAlgebra

`IsFpAlgebra(` *obj* `)`

returns `true` if *obj* is a finitely presented algebra, and `false` otherwise.

```
gap> IsFpAlgebra( FreeAlgebra( GF(2), 0 ) );
true
gap> IsFpAlgebra( last );
false
```

40.5 Operators for Finitely Presented Algebras

A / *relators*

returns a finitely presented algebra that is the quotient of the free algebra A (see 40.2) by the two-sided ideal in A spanned by the elements in the list *relators*.

This is the general method to construct finitely presented algebras in GAP3. For the special case of group algebras of finitely presented groups see 40.3.

A ^ *n*

returns a free A-module of dimension n (see chapter 42) for the finitely presented algebra A.

```
gap> f:= FreeAlgebra( Rationals, 2 );
UnitalAlgebra( Rationals, [ a.1, a.2 ] )
gap> a:= f / [ f.1^2 - f.one, f.2^2 - f.one, (f.1*f.2)^2 - f.one ];
UnitalAlgebra( Rationals, [ a.1, a.2 ] )
gap> a = f;
false
gap> a^2;
Module( UnitalAlgebra( Rationals, [ a.1, a.2 ] ),
[ [ a.one, a.zero ], [ a.zero, a.one ] ] )
```

a `in` *A*

returns `true` if a is an element of the finitely presented algebra A, and `false` otherwise. Note that the answer may require the computation of an isomorphic matrix algebra if A is not a parent algebra.

```
gap> a.1 in a;
true
gap> f.1 in a;
false
gap> 1 in a;
false
```

40.6 Functions for Finitely Presented Algebras

The following functions are overlaid in the operations record of finitely presented algebras.

The **set theoretic functions**
: `Elements`, `Intersection`, `IsFinite`, `IsSubset`, `Size`;

the **vector space functions**
: `Base`, `Coefficients`, and `Dimension`,

Note that at the moment no basis records (see 33.2) for finitely presented algebras are supported.

and the **algebra functions**
: `Closure`, `IsAbelian`, `IsTrivial`, `Operation` (see 39.22, 73.1, 73.3), `Subalgebra`, and `TrivialSubalgebra`.

Note that these functions try to compute a faithful matrix representation of the algebra using the Vector Enumeration share library (see chapter 73).

40.7 PrintDefinitionFpAlgebra

`PrintDefinitionFpAlgebra(` *A*, *name* `)`

prints the assignment of the finitely presented algebra *A* to the variable name *name*. Using the call as an argument of `PrintTo` (see 3.15), this can be used to save *A* to a file.

```
gap> a:= FreeAlgebra( GF(2), "x", "y" );
UnitalAlgebra( GF(2), [ x, y ] )
gap> a:= a / [ a.1^2-a.one, a.2^2-a.one, (a.1*a.2)^3 - a.one ];
UnitalAlgebra( GF(2), [ x, y ] )
gap> PrintDefinitionFpAlgebra( a, "b" );
b:= FreeAlgebra( GF(2), "x", "y" );
b:= b / [ b.one+b.1^2, b.one+b.2^2, b.one+b.1*b.2*b.1*b.2*b.1*b.2 ];
gap> PrintTo( "algebra", PrintDefinitionFpAlgebra( a, "b" ) );
```

40.8 MappedExpression

`MappedExpression(` *expr*, *gens1*, *gens2* `)`

For an arithmetic expression *expr* in terms of *gens1*, `MappedExpression` returns the corresponding expression in terms of *gens2*.

gens1 may be a list of abstract generators (in this case the result is the same as the object returned by 22.12 `MappedWord`), or of generators of a finitely presented algebra.

```
gap> a:= FreeAlgebra( Rationals, 2 );;
gap> a:= a / [ a.1^2 - a.one, a.2^2 - a.one, (a.1*a.2)^2 - a.one ];;
gap> matgens:= [ [[0,0,0,1],[0,0,1,0],[0,1,0,0],[1,0,0,0]],
>                [[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]] ];;
gap> permgens:= [ (1,4)(2,3), (1,2)(3,4) ];;
gap> MappedExpression( a.1^2 + a.1, a.generators, matgens );
[ [ 1, 0, 0, 1 ], [ 0, 1, 1, 0 ], [ 0, 1, 1, 0 ], [ 1, 0, 0, 1 ] ]
gap> MappedExpression( a.1 * a.2, a.generators, permgens );
(1,3)(2,4)
```

Note that this can be done also in terms of (algebra or group) homomorphisms (see 39.24).

`MappedExpression` may raise elements in *gens2* to the zero-th power.

40.9 Elements of Finitely Presented Algebras

Zero and One of Finitely Presented Algebras

A finitely presented algebra *A* contains a zero element *A*`.zero`. If the number of generators of *A* is not zero, the multiplicative neutral element of *A* is *A*`.one`, which is the zero-th power of any nonzero element of *A*.

Comparisons of Elements of Finitely Presented Algebras

x = *y*
x < *y*

Elements of the same algebra can be compared in order to form sets. **Note** that probably it will be necessary to compute an isomorphic matrix representation in order to decide equality if x and y are not elements of a free algebra.

```
gap> a:= FreeAlgebra( Rationals, 1 );;
gap> a:= a / [ a.1^2 - a.one ];
UnitalAlgebra( Rationals, [ a.1 ] )
gap> [ a.1^3 = a.1, a.1^3 > a.1, a.1 > a.one, a.zero > a.one ];
[ true, false, false, false ]
```

Arithmetic Operations for Elements of Finitely Presented Algebras

x + *y*
x - *y*
x * *y*
x ^ *n*
x / *c*

The usual arithmetical operations for ring elements apply to elements of finitely presented algebras. Exponentiation `^` can be used to raise an element x to the n-th power. Division `/` is only defined for denominators in the base field of the algebra.

```
gap> a:= FreeAlgebra( Rationals, 2 );;
gap> x:= a.1 - a.2;
a.1+-1*a.2
gap> x^2;
a.1^2+-1*a.1*a.2+-1*a.2*a.1+a.2^2
gap> y:= 4 * x - a.1;
3*a.1+-4*a.2
gap> y^2;
9*a.1^2+-12*a.1*a.2+-12*a.2*a.1+16*a.2^2
```

`IsFpAlgebraElement(` *obj* `)`

returns `true` if *obj* is an element of a finitely presented algebra, and `false` otherwise.

```
gap> IsFpAlgebraElement( a.zero );
true
gap> IsFpAlgebraElement( a.field.zero );
false
```

`FpAlgebraElement(` *A*`,` *coeff*`,` *words* `)`

Elements of finitely presented algebras normally arise from arithmetical operations. It is, however, possible to construct directly the element of the finitely presented algebra A that is the sum of the words in the list *words*, with coefficients given by the list *coeff*, by calling `FpAlgebraElement(` *A*`,` *coeff*`,` *words* `)`. **Note** that this function does **not** check whether some of the words are equal, or whether all coefficients are nonzero. So one should probably not use it.

```
gap> a;
UnitalAlgebra( Rationals, [ a.1, a.2 ] )
gap> FpAlgebraElement( a, [ 1, 1 ], a.generators );
a.1+a.2
gap> FpAlgebraElement( a, [ 1, 1, 1 ], List( [ 1..3 ], i -> a.1^i ) );
a.1+a.1^2+a.1^3
```

40.10 ElementAlgebra

`ElementAlgebra(` A`,` nr `)`

returns the nr-th element in terms of the generators of the free algebra A over the finite field F, with respect to the following ordering.

We form the elements as linear combinations with coefficients in the base field of A, with respect to the basis defined by the ordering of words according to length and lexicographic order; this sequence starts as follows.

a_1^0, a_1, a_2, $\ldots$, a_n, a_1^2, a_1a_2, a_1a_3, $\ldots$, a_1a_n, a_2a_1, $\ldots$, a_2a_n, $\ldots$, a_n^2, a_1^3, $a_1^2a_2$, $\ldots$, $a_1^2a_n$, $a_1a_2a_1$, $\ldots$

Let n be the number of generators of A, q the size of F, and $nr = \sum_{i=0}^{k} a_i q^i$ the q-adic expression of nr. Then the a_i-th element of A`.field` is the coefficient of the i-th base element in the required algebra element. The ordering of field elements is the same as that defined in the MeatAxe package, that is, `FFList(` F `)[` m`+1 ]` (see 39.29) is the m-th element of the field F.

```
gap> a:= FreeAlgebra( GF(2), 2 );;
gap> List( [ 10 .. 20 ], x -> ElementAlgebra( a, x ) );
[ a.1+a.1^2, a.one+a.1+a.1^2, a.2+a.1^2, a.one+a.2+a.1^2,
  a.1+a.2+a.1^2, a.one+a.1+a.2+a.1^2, a.1*a.2, a.one+a.1*a.2,
  a.1+a.1*a.2, a.one+a.1+a.1*a.2, a.2+a.1*a.2 ]
gap> ElementAlgebra( a, 0 );
a.zero
```

The function can be applied also if A is an arbitrary finitely presented algebra or a matrix algebra. In these cases the result is the element of the algebra obtained on replacing the generators of the corresponding free algebra by the generators of A.

Note that the zero-th power of elements may be needed, which is not necessarily an element of a matrix algebra.

```
gap> a:= UnitalAlgebra( GF(2), GL(2,2).generators );
UnitalAlgebra( GF(2), [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ] )
gap> ElementAlgebra( a, 17 );
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ]
```

The number of an element a can be computed using 40.11.

40.11 NumberAlgebraElement

`NumberAlgebraElement(` a `)`

returns the number n such that the element a of the finitely presented algebra A is the n-th element of A in the sense of 40.10, that is, a `= ElementAlgebra(` A`,` n `)`.

```
gap> a:= FreeAlgebra( GF(2), 2 );;
gap> NumberAlgebraElement( ( a.1 + a.one )^4 );
32769
gap> NumberAlgebraElement( a.zero );
0
gap> NumberAlgebraElement( a.one );
1
```

Note that A`.field` must be finite.

Chapter 41

Matrix Algebras

This chapter describes the data structures and functions for matrix algebras in GAP3. See chapter 39 for the description of all those aspects that concern general algebras.

First the objects of interest in this chapter are introduced (see 41.1, 41.2).

The next sections describe functions for matrix algebras, first those that can be applied not only for matrix algebras (see 41.3, 41.4, 41.5, 41.6, 41.7), and then specific matrix algebra functions (see 41.8, 41.9, 41.10, 41.11).

41.1 More about Matrix Algebras

A **matrix algebra** is an algebra (see 39.1) the elements of which are matrices.

There is a canonical isomorphism of a matrix algebra onto a row space (see chapter 33) that maps a matrix to the concatenation of its rows. This makes all computations with matrix algebras that use its vector space structure as efficient as the corresponding computation with a row space. For example the computation of a vector space basis, of coefficients with respect to such a basis, and of representatives under the action on a vector space by right multiplication.

If one is interested in matrix algebras as domains themselves then one should think of this algebra as of a row space that admits a multiplication. For example, the convention for row spaces that the coefficients field must contain the field of the vector elements also applies to matrix algebras. And the concept of vector space bases is the same as that for row spaces (see 41.2).

In the chapter about modules (see chapter 42) it is stated that modules are of interest mainly as operation domains of algebras. Here we can state that matrix algebras are of interest mainly because they describe modules. For some of the functions it is not obvious whether they are functions for modules or for algebras or for the matrices that generate an algebra. For example, one usually talks about the fingerprint of an A-module M, but this is in fact computed as the list of nullspace dimensions of generators of a certain matrix algebra, namely the induced action of A on M as is computed using `Operation(` A, M `)` (see 41.10, 39.22).

41.2 Bases for Matrix Algebras

As stated in section 41.1, the implementation of bases for matrix algebras follows that of row space bases, see 33.2 for the details. Consequently there are two types of bases, arbitrary bases and semi-echelonized bases, where the latter type can be defined as follows. Let φ be the vector space homomorphism that maps a matrix in the algebra A to the concatenation of its rows, and let $B = (b_1, b_2, \ldots, b_n)$ be a vector space basis of A, then B is called **semi-echelonized** if and only if the row space basis $(\varphi(b_1), \varphi(b_2), \ldots, \varphi(b_n))$ is semi-echelonized, in the sense of 33.2. The **canonical basis** is defined analogeously.

Due to the multiplicative structure that allows to view a matrix algebra A as an A-module with action via multiplication from the right, there is additionally the notion of a **standard basis** for A, which is essentially described in 42.13. The default way to compute a vector space basis of a matrix algebra from a set of generating matrices is to compute this standard basis and a semi-echelonized basis in parallel.

If the matrix algebra A is unital then every semi-echelonized basis and also the standard basis have `One(` *A* `)` as first basis vector.

41.3 IsMatAlgebra

`IsMatAlgebra(` *obj* `)`

returns `true` if *obj*, which may be an object of arbitrary type, is a matrix algebra and `false` otherwise.

```
gap> IsMatAlgebra( FreeAlgebra( GF(2), 0 ) );
false
gap> IsMatAlgebra( Algebra( Rationals, [[[1]]] ) );
true
```

41.4 Zero and One for Matrix Algebras

`Zero(` *A* `)`

returns the square zero matrix of the same dimension and characteristic as the elements of A. This matrix is thought only for testing whether a matrix is zero, usually all its rows will be **identical** in order to save space. So you should **not** use this zero matrix for other purposes; use 34.4 `NullMat` instead.

`One(` *A* `)`

returns for a unital matrix algebra A the identity matrix of the same dimension and characteristic as the elements of A; for a not unital matrix algebra A the (left and right) multiplicative neutral element (if exists) is computed by solving a linear equation system.

41.5 Functions for Matrix Algebras

`Closure`, `Elements`, `IsFinite`, and `Size` are the only **set theoretic functions** that are overlaid in the operations records for matrix algebras and unital matrix algebras. See 39.17 for an overview of set theoretic functions for general algebras.

No **vector space functions** are overlaid in the operations records for matrix algebras and unital matrix algebras. The **functions for vector space bases** are mainly the same as those for row space bases (see 41.2).

For other functions for matrix algebras, see 41.6.

41.6 Algebra Functions for Matrix Algebras

`Centralizer(` *A*`,` *a* `)`
`Centralizer(` *A*`,` *S* `)`

returns the element or subalgebra centralizer in the matrix algebra A. Centralizers in matrix algebras are computed by solving a linear equation system.

`Centre(` *A* `)`

returns the centre of the matrix algebra A, which is computed by solving a linear equation system.

`FpAlgebra(` *A* `)`

returns a finitely presented algebra that is isomorphic to A. The presentation is computed using the structure constants, thus a vector space basis of A has to be computed. If A contains no multiplicative neutral element (see 41.4) an error is signalled. (At the moment the implementation is really simpleminded.)

```
gap> a:= UnitalAlgebra( Rationals, [[[0,1],[0,0]]] );
UnitalAlgebra( Rationals, [ [ [ 0, 1 ], [ 0, 0 ] ] ] )
gap> FpAlgebra( a );
UnitalAlgebra( Rationals, [ a.1 ] )
gap> last.relators;
[ a.1^2 ]
```

41.7 RepresentativeOperation for Matrix Algebras

`RepresentativeOperation(` *A*`,` *v1*`,` *v2* `)`

returns the element in the matrix algebra A that maps *v1* to *v2* via right multiplication if such an element exists, and `false` otherwise. *v1* and *v2* may be vectors or matrices of same dimension.

```
gap> a:= MatAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ] )
gap> v1:= [ 1, 0 ] * Z(2);; v2:= [ 1, 1 ] * Z(2);;
gap> RepresentativeOperation( a, v1, v2 );
[ [ Z(2)^0, Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ]
gap> t:= TrivialSubalgebra( a );;
gap> RepresentativeOperation( t, v1, v2 );
false
```

41.8 MatAlgebra

`MatAlgebra(` *F*`,` *n* `)`

returns the full matrix algebra of n by n matrices over the field F.

```
gap> a:= MatAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ] )
gap> Size( a );
16
```

41.9 NullAlgebra

`NullAlgebra(` *F* `)`

returns a trivial algebra (that is, it contains only the zero element) over the field F. This occurs in a natural way whenever `Operation` (see 39.22) constructs a faithful representation of the zero module.

Here we meet the strange situation that an operation algebra does not consist of matrices, since in GAP3 a matrix always has a positive number of rows and columns. The element of a `NullAlgebra(` *F* `)` is the object `EmptyMat` that acts (trivially) on empty lists via right multiplication.

```
gap> a:= NullAlgebra( GF(2) );
NullAlgebra( GF(2) )
gap> Size( a );
1
gap> Elements( a );
[ EmptyMat ]
gap> [] * EmptyMat;
[  ]
gap> IsAlgebra( a );
true
```

41.10 Fingerprint

`Fingerprint(` *A* `)`
`Fingerprint(` *A*, *list* `)`

returns the fingerprint of the matrix algebra A, i.e., a list of nullities of six "standard" words in A (for 2-generator algebras only) or of the words with numbers in *list*.

```
gap> m1:= PermutationMat( (1,2,3,4,5), 5, GF(2) );;
gap> m2:= PermutationMat( (1,2)      , 5, GF(2) );;
gap> a:= Algebra( GF(2), [ m1, m2 ] );;
gap> Fingerprint( a );
[ 1, 1, 1, 3, 0, 4 ]
```

Let a and b be the generators of a 2-generator matix algebra. The six standard words used by `Fingerprint` are $w_1, w_2, \ldots, w_6$ where

$$\begin{array}{rclrcl} w_1 & = & ab+a+b, & w_2 & = & w_1+ab^2, \\ w_3 & = & a+bw_2, & w_4 & = & b+w_3, \\ w_5 & = & ab+w_4, & w_6 & = & a+w_5 \end{array}$$

41.11 NaturalModule

`NaturalModule(` *A* `)`

returns the **natural module** M of the matrix algebra A. If A consists of n by n matrices, and F is the coefficients field of A then M is an n-dimensional row space over the field F, viewed as A-right module (see 42.4).

```
gap> a:= MatAlgebra( GF(2), 2 );;
gap> a.name:= "a";;
gap> m:= NaturalModule( a );
Module( a, [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
```

Chapter 42

Modules

This chapter describes the data structures and functions for modules in GAP3.

After the introduction of the data structures (see 42.1, 42.2, 42.3) the functions for constructing modules and submodules (see 42.4, 42.5, 42.6, 42.7, 42.8) and testing for modules (see 42.9, 42.10) are described.

The next sections describe operations and functions for modules (see 42.11, 42.12, 42.13, 42.14, 42.16).

The next section describes available module homomorphisms. At the moment only operation homomorphisms are supported (see 42.17).

The last sections describe the implementation of the data structures (see 42.18, 42.19).

Many examples in this chapter use the natural permutation module for the symmetric group S_3. If you want to run the examples you must first define this module, as is done using the following commands.

```
gap> mat1:= PermutationMat( (1,2,3), 3, GF(2) );;
gap> mat2:= PermutationMat(   (1,2), 3, GF(2) );;
gap> a:= UnitalAlgebra( GF(2), [ mat1, mat2 ] );; a.name:= "a";;
gap> nat:= NaturalModule( a );;
gap> nat.name:= "nat";;
```

There is no possibility to compute the lattice of submodules with the implementations in GAP3. However, it is possible to use the MeatAxe share library (see chapter 69) to compute the lattice, and then (perhaps) to carry back interesting parts to GAP3 format using 69.2 `GapObject`.

42.1 More about Modules

Let R be a ring. An R-**module** (or, more exactly, an R-right module) is an additive abelian group on that R acts from the right.

A module is of interest mainly as operation domain of an algebra (see chapter 39). Thus it is the natural place to store information about the operation of the algebra, for example whether it is irreducible. But since a module is a domain it has also properties of its own, independent of the algebra.

According to the different types of algebras in GAP3, namely matrix algebras and finitely presented algebras, at the moment two types of modules are supported in GAP3, namely **row modules** and their quotients for matrix algebras and **free modules** and their submodules and quotients for finitely presented algebras. See 42.2 and 42.3 for more information.

For modules, the same concept of parent and substructures holds as for row spaces. That is, a module is stored either as a submodule of a module, or it is not (see 42.5, 42.7 for the details).

Also the concept of factor structures and cosets is the same as that for row spaces (see 33.4, 33.3), especially the questions about a factor module is mainly delegated to the numerator and the denominator, see also 42.11.

42.2 Row Modules

A **row module** for a matrix algebra A is a row space over a field F on that A acts from the right via matrix multiplication. All operations, set theoretic functions and vector space functions for row spaces are applicable to row modules, and the conventions for row spaces also hold for row modules (see chapter 33). For the notion of a standard basis of a module, see 42.13.

It should be mentioned, however, that the functions and their results have to be interpreted in the module context. For example, `Generators` returns a list of module generators not vector space generators (see 42.8), and `Closure` or `Sum` for modules return a module (namely the smallest module generated by the arguments).

Quotient modules $Q = V/W$ of row modules are quotients of row spaces V, W that are both (row) modules for the same matrix algebra A. All operations and functions for quotient spaces are applicable. The element of such quotient modules are **module cosets**, in addition to the operations and functions for row space cosets they can be multiplied by elements of the acting algebra.

42.3 Free Modules

A **free module** of dimension n for an algebra A consists of all n-tuples of elements of A, the action of A is defined as component-wise multiplication from the right. Submodules and quotient modules are defined in the obvious way.

In GAP3, elements of free modules are stored as lists of algebra elements. Thus there is no difference to row modules with respect to addition of elements, and operation of the algebra. However, the applicable functions are different.

At the moment, only free modules for finitely presented algebras are supported in GAP3, and only very few functions are available for free modules at the moment. Especially the set theoretic and vector space functions do not work for free modules and their submodules and quotients.

Free modules were only introduced as operation domains of finitely presented algebras.

A ^ n

returns a free module of dimension n for the algebra A.

```
gap> a:= FreeAlgebra( Rationals, 2 );; a.name:= "a";;
gap> a^2;
Module( a, [ [ a.one, a.zero ], [ a.zero, a.one ] ] )
```

42.4 Module

`Module(` R, *gens* `)`
`Module(` R, *gens*, *zero* `)`
`Module(` R, *gens*, `"basis")`

returns the module for the ring R that is generated by the elements in the list *gens*. If *gens* is empty then the zero element *zero* of the module must be entered.

If the third argument is the string `"basis"` then the generators *gens* are assumed to form a vector space basis.

```
gap> a:= UnitalAlgebra( GF(2), GL(2,2).generators );;
gap> a.name:="a";;
gap> m1:= Module( a, [ a.1[1] ] );
Module( a, [ [ Z(2)^0, Z(2)^0 ] ] )
gap> Dimension( m1 );
2
gap> Basis( m1 );
SemiEchelonBasis( Module( a, [ [ Z(2)^0, Z(2)^0 ] ] ),
[ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] )
gap> m2:= Module( a, a.2, "basis" );;
gap> Basis( m2 );
Basis( Module( a, [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] ),
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ] )
gap> a.2;
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ]
gap> m1 = m2;
true
```

42.5 Submodule

`Submodule(` M, *gens* `)`

returns the submodule of the parent of the module M that is generated by the elements in the list *gens*. If M is a factor module, *gens* may also consist of representatives instead of the cosets themselves.

```
gap> a:= UnitalAlgebra( GF(2), [ mat1, mat2 ] );; a.name:= "a";;
gap> nat:= NaturalModule( a );;
gap> nat.name:= "nat";;
gap> s:= Submodule( nat, [ [ 1, 1, 1 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> Dimension( s );
1
```

42.6 AsModule

`AsModule(` *M* `)`

returns a module that is isomorphic to the module or submodule *M*.

```
gap> s:= Submodule( nat, [ [ 1, 1, 1 ] * Z(2) ] );;
gap> s2:= AsModule( s );
Module( a, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> s = s2;
true
```

42.7 AsSubmodule

`AsSubmodule(` *M*, *U* `)`

returns a submodule of the parent of *M* that is isomorphic to the module *U* which can be a parent module or a submodule with a different parent.

Note that the same ring must act on *M* and *U*.

```
gap> s2:= Module( a, [ [ 1, 1, 1 ] * Z(2) ] );;
gap> s:= AsSubmodule( nat, s2 );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> s = s2;
true
```

42.8 AsSpace for Modules

`AsSpace(` *M* `)`

returns a (quotient of a) row space that is equal to the (quotient of a) row module *M*.

```
gap> s:= Submodule( nat, [ [ 1, 1, 0 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] )
gap> Dimension( s );
2
gap> AsSpace( s );
RowSpace( GF(2),
[ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ] )
gap> q:= nat / s;
nat / [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ]
gap> AsSpace( q );
RowSpace( GF(2),
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ] ) /
[ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
```

42.9 IsModule

`IsModule(` *obj* `)`

returns `true` if *obj*, which may be an object of arbitrary type, is a module, and `false` otherwise.

```
gap> IsModule( nat );
true
gap> IsModule( AsSpace( nat ) );
false
```

42.10 IsFreeModule

`IsFreeModule(` *obj* `)`

returns `true` if *obj*, which may be an object of arbitrary type, is a free module, and `false` otherwise.

```
gap> IsFreeModule( nat );
false
gap> IsFreeModule( a^2 );
true
```

42.11 Operations for Row Modules

Here we mention only those facts about operations that have to be told in addition to those for row spaces (see 33.7).

Comparisons of Modules

M1 `=` *M2*
M1 `<` *M2*

Equality and ordering of (quotients of) row modules are defined as equality resp. ordering of the modules as vector spaces (see 33.7).

This means that equal modules may be inequivalent as modules, and even the acting rings may be different. For testing equivalence of modules, see 42.14.

```
gap> s:= Submodule( nat, [ [ 1, 1, 1 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> s2:= Submodule( nat, [ [ 1, 1, 0 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] )
gap> s = s2;
false
gap> s < s2;
true
```

Arithmetic Operations of Modules

M1 `+` *M2*
: returns the sum of the two modules *M1* and *M2*, that is, the smallest module containing both *M1* and *M2*. Note that the same ring must act on *M1* and *M2*.

M1 `/` *M2*
: returns the factor module of the module *M1* by its submodule *M2*. Note that the same ring must act on *M1* and *M2*.

```
gap> s1:= Submodule( nat, [ [ 1, 1, 1 ] * Z(2) ] );
```

```
Submodule( nat, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> q:= nat / s1;
nat / [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> s2:= Submodule( nat, [ [ 1, 1, 0 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] )
gap> s3:= s1 + s2;
Submodule( nat,
[ [ Z(2)^0, Z(2)^0, Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0 ] ] )
gap> s3 = nat;
true
```

For forming the sum and quotient of row spaces, see 33.7.

42.12 Functions for Row Modules

As stated in 42.2, row modules behave like row spaces with respect to **set theoretic** and **vector space** functions (see 33.8).

The functions in the following sections use the module structure (see 42.13, 42.14, 42.15, 42.16, 42.17).

42.13 StandardBasis for Row Modules

`StandardBasis(` *M* `)`
`StandardBasis(` *M* `,` *seedvectors* `)`

returns the standard basis of the row module *M* with respect to the seed vectors in the list *seedvectors*. If no second argument is given the generators of *M* are taken.

The **standard basis** is defined as follows. Take the first seed vector v, apply the generators of the ring R acting on M in turn, and if the image is linearly independent of the basis vectors found up to this time, it is added to the basis. When the space becomes stable under the action of R, proceed with the next seed vector, and so on.

Note that you do not get a basis of the whole module if all seed vectors lie in a proper submodule.

```
gap> s:= Submodule( nat, [ [ 1, 1, 0 ] * Z(2) ] );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] )
gap> b:= StandardBasis( s );
StandardBasis( Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] ) )
gap> b.vectors;
[ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> StandardBasis( s, [ [ 0, 1, 1 ] * Z(2) ] );
StandardBasis( Submodule( nat, [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ] ),
[ [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0 ] ] )
```

42.14 IsEquivalent for Row Modules

`IsEquivalent(` *M1* `,` *M2* `)`

Let *M1* and *M2* be modules acted on by rings R_1 and R_2, respectively, such that mapping the generators of R_1 to the generators of R_2 defines a ring homomorphism. Furthermore let at least one of *M1*, *M2* be irreducible. Then `IsEquivalent(` *M1* `,` *M2* `)` returns `true` if the actions on *M1* and *M2* are equivalent, and `false` otherwise.

```
gap> rand:= RandomInvertableMat( 3, GF(2) );;
gap> b:= UnitalAlgebra( GF(2), List( a.generators, x -> x^rand ) );;
gap> m:= NaturalModule( b );;
gap> IsEquivalent( nat / FixedSubmodule( nat ),
>                  m / FixedSubmodule( m ) );
true
```

42.15 IsIrreducible for Row Modules

`IsIrreducible(` *M* `)`

returns `true` if the (quotient of a) row module *M* is irreducible, and `false` otherwise.

```
gap> IsIrreducible( nat );
false
gap> IsIrreducible( nat / FixedSubmodule( nat ) );
true
```

42.16 FixedSubmodule

`FixedSubmodule(` *M* `)`

returns the submodule of fixed points in the module *M* under the action of the generators of *M*`.ring`.

```
gap> fix:= FixedSubmodule( nat );
Submodule( nat, [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] )
gap> Dimension( fix );
1
```

42.17 Module Homomorphisms

Let M_1 and M_2 be modules acted on by the rings R_1 and R_2 (via exponentiation), and φ a ring homomorphism from R_1 to R_2. Any linear map $\psi = \psi_\varphi$ from M_1 to M_2 with the property that $(m^r)^\psi = (m^\psi)^{(r^\varphi)}$ is called a **module homomorphism**.

At the moment only the following type of module homomorphism is available in GAP3. Suppose you have the module M_1 for the algebra R_1. Then you can construct the operation algebra R_2:= `Operation`(R_1, M_1), and the module for R_2 isomorphic to M_1 as M_2:= `OperationModule`(R_2).

Then `OperationHomomorphism`(M_1, M_2) can be used to construct the module homomorphism from M_1 to M_2.

```
gap> s:= Submodule( nat, [ [ 1, 1, 0 ] *Z(2) ] );; s.name:= "s";;
gap> op:= Operation( a, s ); op.name:="op";;
UnitalAlgebra( GF(2), [ [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0 ] ] ] )
gap> opmod:= OperationModule( op ); opmod.name:= "opmod";;
Module( op, [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] )
```

```
gap> modhom:= OperationHomomorphism( s, opmod );
OperationHomomorphism( s, opmod )
gap> b:= Basis( s );
SemiEchelonBasis( s,
[ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ] )
```

Images and preimages of elements under module homomorphisms are computed using `Image` and `PreImagesRepresentative`, respectively. If M_1 is a row module this is done by using the knowledge of images of a basis, if M_1 is a (quotient of a) free module then the algebra homomorphism and images of the generators of M_1 are used. The computation of preimages requires in both cases the knowledge of representatives of preimages of a basis of M_2.

```
gap> im:= List( b.vectors, x -> Image( modhom, x ) );
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
gap> List( im, x -> PreImagesRepresentative( modhom, x ) );
[ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
```

42.18 Row Module Records

Module records contain at least the components

`isDomain`
: always `true`,

`isModule`
: always `true`,

`isVectorSpace`
: always `true`, since modules are vector spaces,

`ring`
: the ring acting on the module,

`field`
: the coefficients field, is the same as R`.field` where R is the `ring` component of the module,

`operations`
: the operations record of the module.

The following components are optional, but if they are not present then the corresponding function in the `operations` record must know how to compute them.

`generators`
: a list of **module** generators (not necessarily of vector space generators),

`zero`
: the zero element of the module.

`basis`
: a vector space basis of the module (see also 33.2),

Factors of row modules have the same components as quotients of row spaces (see 33.30), except that of course they have an appropriate `operations` record.

Additionally factors of row modules have the components `isModule`, `isFactorModule` (both always `true`). Parent modules also have the `ring` component, which is the same ring as the ring component of numerator and denominator.

42.19 Module Homomorphism Records

Module homomorphism records have at least the following components.

`isGeneralMapping`
: `true`,

`isMapping`
: `true`,

`isHomomorphism`
: `true`,

`domain`
: `Mappings`,

`source`
: the source of the homomorphism, a module M_1,

`range`
: the range of the homomorphism, a module M_2,

`preImage`
: the module M_1,

`basisImage`
: a vector space basis of the image of M_1,

`preimagesBasis`
: a list of preimages of the basis vectors in `basisImage`

`operations`
: the operations record of the homomorphism.

If the source is a (factor of a) free module then there are also the components

`genimages`
: a list of images of the generators of the source,

`alghom`
: the underlying algebra homomorphism from the ring acting on M_1 to the ring acting on M_2.

If the source is a (factor of a) row module then there are also the components

`basisSource`
: a vector space basis of M_1,

`imagesBasis`
: a list of images of the basis vectors in `basisSource`.

Chapter 43

Mappings

A **mapping** is an object that maps each element of its source to a value in its range.

Precisely, a mapping is a triple. The **source** is a set of objects. The **range** is another set of objects. The **relation** is a subset S of the cartesian product of the source with the range, such that for each element elm of the source there is exactly one element img of the range, so that the pair (elm, img) lies in S. This img is called the image of elm under the mapping, and we say that the mapping maps elm to img.

A **multi valued mapping** is an object that maps each element of its source to a set of values in its range.

Precisely, a multi valued mapping is a triple. The **source** is a set of objects. The **range** is another set of objects. The **relation** is a subset S of the cartesian product of the source with the range. For each element elm of the source the set img such that the pair (elm, img) lies in S is called the set of **images** of elm under the mapping, and we say that the mapping maps elm to this set.

Thus a mapping is a special case of a multi valued mapping where the set of images of each element of the source contains exactly one element, which is then called the image of the element under the mapping.

Mappings are created by **mapping constructors** such as `MappingByFunction` (see 43.18) or `NaturalHomomorphism` (see 7.110).

This chapter contains sections that describe the functions that test whether an object is a mapping (see 43.1), whether a mapping is single valued (see 43.2), and the various functions that test if such a mapping has a certain property (see 43.3, 43.4, 43.5, 44.1, 44.2, 44.3, 44.4, 44.3, and 44.6).

Next this chapter contains functions that describe how mappings are compared (see 43.6) and the operations that are applicable to mappings (see 43.7).

Next this chapter contains sections that describe the functions that deal with the images and preimages of elements under mappings (see 43.8, 43.9, 43.10, 43.11, 43.12, and 43.13).

Next this chapter contains sections that describe the functions that compute the composition of two mappings, the power of a mapping, the inverse of a mapping, and the identity mapping on a certain domain (see 43.14, 43.15, 43.16, and 43.17).

Finally this chapter also contains a section that describes how mappings are represented internally (see 43.19).

The functions described in this chapter are in the file *libname*/"mapping.g".

43.1 IsGeneralMapping

`IsGeneralMapping( obj )`

`IsGeneralMapping` returns `true` if the object *obj* is a mapping (possibly multi valued) and `false` otherwise.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsGeneralMapping( p4 );
true
gap> IsGeneralMapping( InverseMapping( p4 ) );
true     # note that the inverse mapping is multi valued
gap> IsGeneralMapping( x -> x^4 );
false     # a function is not a mapping
```

See 43.18 for the definition of `MappingByFunction` and 43.16 for `InverseMapping`.

43.2 IsMapping

`IsMapping( map )`

`IsMapping` returns `true` if the general mapping *map* is single valued and `false` otherwise. Signals an error if *map* is not a general mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsMapping( p4 );
true
gap> IsMapping( InverseMapping( p4 ) );
false     # note that the inverse mapping is multi valued
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsMapping( p5 );
true
gap> IsMapping( InverseMapping( p5 ) );
true     # p5 is a bijection
```

`IsMapping` first tests if the flag *map*`.isMapping` is bound. If the flag is bound, it returns its value. Otherwise it calls *map*`.operations.IsMapping(` *map* `)`, remembers the returned value in *map*`.isMapping`, and returns it.

The default function called this way is `MappingOps.IsMapping`, which computes the sets of images of all the elements in the source of *map*, provided this is finite, and returns `true` if all those sets have size one. Look in the index under **IsMapping** to see for which mappings this function is overlaid.

43.3 IsInjective

`IsInjective(` *map* `)`

`IsInjective` returns `true` if the mapping *map* is injective and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping *map* is **injective** if for each element *img* of the range there is at most one element *elm* of the source that *map* maps to *img*.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsInjective( p4 );
false
gap> IsInjective( InverseMapping( p4 ) );
Error, <map> must be a single valued mapping
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsInjective( p5 );
true
gap> IsInjective( InverseMapping( p5 ) );
true    # p5 is a bijection
```

`IsInjective` first tests if the flag *map*`.isInjective` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.isInjective(` *map* `)`, remembers the returned value in *map*`.isInjective`, and returns it.

The default function called this way is `MappingOps.IsInjective`, which compares the sizes of the source and image of *map*, and returns `true` if they are equal (see 43.8). Look in the index under **IsInjective** to see for which mappings this function is overlaid.

43.4 IsSurjective

`IsSurjective(` *map* `)`

`IsSurjective` returns `true` if the mapping *map* is surjective and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping *map* is **surjective** if for each element *img* of the range there is at least one element *elm* of the source that *map* maps to *img*.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
```

```
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsSurjective( p4 );
false
gap> IsSurjective( InverseMapping( p4 ) );
Error, <map> must be a single valued mapping
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsSurjective( p5 );
true
gap> IsSurjective( InverseMapping( p5 ) );
true    # p5 is a bijection
```

`IsSurjective` first tests if the flag *map*`.isSurjective` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsSurjective(` *map* `)`, remembers the returned value in *map*`.isSurjective`, and returns it.

The default function called this way is `MappingOps.IsSurjective`, which compares the sizes of the range and image of *map*, and returns `true` if they are equal (see 43.8). Look in the index under **IsSurjective** to see for which mappings this function is overlaid.

43.5 IsBijection

`IsBijection(` *map* `)`

`IsBijection` returns `true` if the mapping *map* is a bijection and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping *map* is a **bijection** if for each element *img* of the range there is exactly one element *elm* of the source that *map* maps to *img*. We also say that *map* is **bijective**.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsBijection( p4 );
false
gap> IsBijection( InverseMapping( p4 ) );
Error, <map> must be a single valued mapping
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsBijection( p5 );
true
gap> IsBijection( InverseMapping( p5 ) );
true    # p5 is a bijection
```

`IsBijection` first tests if the flag *map*`.isBijection` is bound. If the flag is bound, it returns its value. Otherwise it calls *map*`.operations.IsBijection(` *map* `)`, remembers the returned value in *map*`.isBijection`, and returns it.

The default function called this way is `MappingOps.IsBijection`, which calls `IsInjective` and `IsSurjective`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done by `IsInjective` and `IsSurjective`.

43.6 Comparisons of Mappings

map1 `=` *map2*
map1 `<>` *map2*

The equality operator `=` applied to two mappings *map1* and *map2* evaluates to `true` if the two mappings are equal and to `false` otherwise. The unequality operator `<>` applied to two mappings *map1* and *map2* evaluates to `true` if the two mappings are not equal and to `false` otherwise. A mapping can also be compared with another object that is not a mapping, of course they are never equal.

Two mappings are considered equal if and only if their sources are equal, their ranges are equal, and for each elment *elm* of the source `Images(` *map1* `,` *elm* `)` is equal to `Images(` *map2* `,` *elm* `)` (see 43.9).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> p13 := MappingByFunction( g, g, x -> x^13 );
MappingByFunction( g, g, function ( x )
    return x ^ 13;
end )
gap> p4 = p13;
false
gap> p13 = IdentityMapping( g );
true
```

map1 `<` *map2*
map1 `<=` *map2*
map1 `>` *map2*
map1 `>=` *map2*

The operators `<`, `<=`, `>`, and `>=` applied to two mappings evaluates to `true` if *map1* is less than, less than or equal to, greater than, or greater than or equal to *map2* and `false` otherwise. A mapping can also be compared with another object that is not a mapping, everything except booleans, lists, and records is smaller than a mapping.

If the source of *map1* is less than the source of *map2*, then *map1* is considered to be less than *map2*. If the sources are equal and the range of *map1* is less than the range of *map2*, then *map1* is considered to be less than *map2*. If the sources and the ranges are equal the mappings are compared lexicographically with respect to the sets of images of the elements of the source under the mappings.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> p4 < p5;
true     # since (5,6,7) is the smallest nontrivial element of g
         # and the image of (5,6,7) under p4 is smaller
         # than the image of (5,6,7) under p5
```

The operator `=` calls *map2*`.operations.=(` *map1*`,` *map2* `)` and returns this value. The operator `<>` also calls *map2*`.operations.=(` *map1*`,` *map2* `)` and returns the logical **not** of this value.

The default function called this way is `MappingOps.=`, which first compares the sources of *map1* and *map2*, then, if they are equal, compares the ranges of *map1* and *map2*, and then, if both are equal and the source is finite, compares the images of all elements of the source under *map1* and *map2*. Look in the index under **equality** to see for which mappings this function is overlaid.

The operator `<` calls *map2*`.operations.<(` *map1*`,` *map2* `)` and returns this value. The operator `<=` calls *map2*`.operations.<(` *map2*`,` *map1* `)` and returns the logical **not** of this value. The operator `>` calls *map2*`.operations.<(` *map2*`,` *map1* `)` and returns this value. The operator `>=` calls *map2*`.operations.<(` *map1*`,` *map2* `)` and returns the logical **not** of this value.

The default function called this way is `MappingOps.<`, which first compares the sources of *map1* and *map2*, then, if they are equal, compares the ranges of *map1* and *map2*, and then, if both are equal and the source is finite, compares the images of all elements of the source under *map1* and *map2*. Look in the index under **ordering** to see for which mappings this function is overlaid.

43.7 Operations for Mappings

map1 `*` *map2*

The product operator `*` applied to two mappings *map1* and *map2* evaluates to the product of the two mappings, i.e., the mapping *map* that maps each element *elm* of the source of *map1* to the value `(`*elm* `^` *map1*`)` `^` *map2*. Note that the range of *map1* must be a subset of the source of *map2*. If *map1* and *map2* are homomorphisms then so is the result. This can also be expressed as `CompositionMapping(` *map2*`,` *map1* `)` (see 43.14). Note that the arguments of `CompositionMapping` are reversed.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
```

```
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> p20 := p4 * p5;
CompositionMapping( MappingByFunction( g, g, function ( x )
    return x ^ 5;
end ), MappingByFunction( g, g, function ( x )
    return x ^ 4;
end ) )
```

list `*` *map*
map `*` *list*

As with every other type of group elements a mapping *map* can also be multiplied with a list of mappings *list*. The result is a new list, such that each entry is the product of the corresponding entry of *list* with *map* (see 27.13).

elm `^` *map*

The power operator `^` applied to an element *elm* and a mapping *map* evaluates to the image of *elm* under *map*, i.e., the element of the range to which *map* maps *elm*. Note that *map* must be a single valued mapping, a multi valued mapping is not allowed (see 43.9). This can also be expressed as `Image(` *map*`,` *elm* `)` (see 43.8).

```
gap> (1,2,3,4) ^ p4;
()
gap> (2,4)(5,6,7) ^ p20;
(5,7,6)
```

map `^ 0`

The power operator `^` applied to a mapping *map*, for which the range must be a subset of the source, and the integer `0` evaluates to the identity mapping on the source of *map*, i.e., the mapping that maps each element of the source to itself. If *map* is a homomorphism then so is the result. This can also be expressed as `IdentityMapping(` *map*`.source )` (see 43.17).

```
gap> p20 ^ 0;
IdentityMapping( g )
```

map `^` *n*

The power operator `^` applied to a mapping *map*, for which the range must be a subset of the source, and an positive integer *n* evaluates to the *n*-fold composition of *map*. If *map* is a homomorphism then so is the result. This can also be expressed as `PowerMapping(` *map*`,` *n* `)` (see 43.15).

```
gap> p16 := p4 ^ 2;
CompositionMapping( CompositionMapping( IdentityMapping( g ), MappingB\
yFunction( g, g, function ( x )
    return x ^ 4;
end ) ), CompositionMapping( IdentityMapping( g ), MappingByFunction( \
g, g, function ( x )
    return x ^ 4;
```

```
end ) ) )
gap> p16 = MappingByFunction( g, g, x -> x^16 );
true
```

bij `^ -1`

The power operator `^` applied to a bijection *bij* and the integer `-1` evaluates to the inverse mapping of *bij*, i.e., the mapping that maps each element *img* of the range of *bij* to the uniq element *elm* of the source of *bij* that maps to *img*. Note that *bij* must be a bijection, a mapping that is not a bijection is not allowed. This can also be expressed as `InverseMapping(` *bij* `)` (see 43.16).

```
gap> p5 ^ -1;
InverseMapping( MappingByFunction( g, g, function ( x )
    return x ^ 5;
end ) )
gap> p4 ^ -1;
Error, <lft> must be a bijection
```

bij `^` *z*

The power operator `^` applied to a bijection *bij*, for which the source and the range must be equal, and an integer *z* returns the *z*-fold composition of *bij*. If *z* is 0 or positive see above, if *z* is negative, this is equivalent to `(`*bij* `^ -1) ^ -`*z*. If *bij* is an automorphism then so is the result.

aut1 `^` *aut2*

The power operator `^` applied to two automorphisms *aut1* and *aut2*, which must have equal sources (and thus ranges) returns the conjugate of *aut1* by *aut2*, i.e., *aut2* `^ -1 *` *aut1* `*` *aut2*. The result if of course again an automorphism.

The operator `*` calls *map2*`.operations.*(` *map1*`,` *map2* `)` and returns this value.

The default function called this way is `MappingOps.*` which calls `CompositionMapping` to do the work. This function is seldom overlaid, since `CompositionMapping` does all the interesting work.

The operator `^` calls *map*`.operations.^(` *map1*`,` *map2* `)` and returns this value.

The default function called this way is `MappingOps.^`, which calls `Image`, `IdentityMapping`, `InverseMapping`, or `PowerMapping` to do the work. This function is seldom overlaid, since `Image`, `IdentityMapping`, `InverseMapping`, and `PowerMapping` do all the interesting work.

43.8 Image

`Image(` *map*`,` *elm* `)`

In this form `Image` returns the image of the element *elm* of the source of the mapping *map* under *map*, i.e., the element of the range to which *map* maps *elm*. Note that *map* must be a single valued mapping, a multi valued mapping is not allowed (see 43.9). This can also be expressed as *elm* `^` *map* (see 43.7).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
```

```
        return x ^ 4;
    end )
    gap> Image( p4, (2,4)(5,6,7) );
    (5,6,7)
    gap> p5 := MappingByFunction( g, g, x -> x^5 );
    MappingByFunction( g, g, function ( x )
        return x ^ 5;
    end )
    gap> Image( p5, (2,4)(5,6,7) );
    (2,4)(5,7,6)
```

`Image(` *map*`,` *elms* `)`

In this form `Image` returns the image of the set of elements *elms* of the source of the mapping *map* under *map*, i.e., set of images of the elements in *elms*. *elms* may be a proper set (see 28) or a domain (see 4). The result will be a subset of the range of *map*, either as a proper set or as a domain. Again *map* must be a single valued mapping, a multi valued mapping is not allowed (see 43.9).

```
    gap> Image( p4, Subgroup( g, [ (2,4), (5,6,7) ] ) );
    [ (), (5,6,7), (5,7,6) ]
    gap> Image( p5, [ (5,6,7), (2,4) ] );
    [ (5,7,6), (2,4) ]
```

Note that in the first example, the result is returned as a proper set, even though it is mathematically a subgroup. This is because `p4` is not known to be a homomorphism, even though it is one.

`Image(` *map* `)`

In this form `Image` returns the image of the mapping *map*, i.e., the subset of element of the range of *map* that are actually values of *map*. Note that in this case the argument may also be a multi valued mapping.

```
    gap> Image( p4 );
    [ (), (5,6,7), (5,7,6) ]
    gap> Image( p5 ) = g;
    true
```

`Image` firsts checks in which form it is called.

In the first case it calls *map*`.operations.ImageElm(` *map*`,` *elm* `)` and returns this value.

The default function called this way is `MappingOps.ImageElm`, which checks that *map* is indeed a single valued mapping, calls `Images(` *map*`,` *elm* `)`, and returns the single element of the set returned by `Images`. Look in the index under **Image** to see for which mappings this function is overlaid.

In the second case it calls *map*`.operations.ImageSet(` *map*`,` *elms* `)` and returns this value.

The default function called this way is `MappingOps.ImageSet`, which checks that *map* is indeed a single valued mapping, calls `Images(` *map*`,` *elms* `)`, and returns this value. Look in the index under **Image** to see for which mappings this function is overlaid.

In the third case it tests if the field *map*`.image` is bound. If this field is bound, it simply returns this value. Otherwise it calls *map*`.operations.ImageSource(` *map* `)`, remembers the returned value in *map*`.image`, and returns it.

The default function called this way is `MappingOps.ImageSource`, which calls `Images(` *map*`,` *map*`.source )`, and returns this value. This function is seldom overlaid, since all the work is done by *map*`.operations.ImagesSet`.

43.9 Images

`Images(` *map*`,` *elm* `)`

In this form `Images` returns the set of images of the element *elm* in the source of the mapping *map* under *map*. *map* may be a multi valued mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> i4 := InverseMapping( p4 );
InverseMapping( MappingByFunction( g, g, function ( x )
    return x ^ 4;
end ) )
gap> IsMapping( i4 );
false    # i4 is multi valued
gap> Images( i4, () );
[ (), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4), (1,4,3,2),
  (1,4)(2,3) ]
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> i5 := InverseMapping( p5 );
InverseMapping( MappingByFunction( g, g, function ( x )
    return x ^ 5;
end ) )
gap> Images( i5, () );
[ () ]
```

`Images(` *map*`,` *elms* `)`

In this form `Images` returns the set of images of the set of elements *elms* in the source of *map* under *map*. *map* may be a multi valued mapping. In any case `Images` returns a set of elements of the range of *map*, either as a proper set (see 28) or as a domain (see 4).

```
gap> Images( i4, [ (), (5,6,7) ] );
[ (), (5,6,7), (2,4), (2,4)(5,6,7), (1,2)(3,4), (1,2)(3,4)(5,6,7),
  (1,2,3,4), (1,2,3,4)(5,6,7), (1,3), (1,3)(5,6,7), (1,3)(2,4),
  (1,3)(2,4)(5,6,7), (1,4,3,2), (1,4,3,2)(5,6,7), (1,4)(2,3),
  (1,4)(2,3)(5,6,7) ]
gap> Images( i5, [ (), (5,6,7) ] );
[ (), (5,7,6) ]
```

`Images` first checks in which form it is called.

In the first case it calls *map*`.operations.ImagesElm(` *map*`,` *elm* `)` and returns this value.

The default function called this way is `MappingOps.ImagesElm`, which just raises an error, since their is no default way to compute the images of an element under a mapping about which nothing is known. Look in the index under **Images** to see how images are computed for the various mappings.

In the second case it calls *map*`.operations.ImagesSet(` *map*`,` *elms* `)` and returns this value.

The default function called this way is `MappingOps.ImagesSet`, which returns the union of the images of all the elements in the set *elms*. Look in the index under **Images** to see for which mappings this function is overlaid.

43.10 ImagesRepresentative

`ImagesRepresentative(` *map*`,` *elm* `)`

`ImagesRepresentative` returns a representative of the set of images of *elm* under *map*, i.e., a single element *img*, such that *img* `in Images(` *map*`,` *elm* `)` (see 43.9). *map* may be a multi valued mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> i4 := InverseMapping( p4 );
InverseMapping( MappingByFunction( g, g, function ( x )
    return x ^ 4;
end ) )
gap> IsMapping( i4 );
false    # i4 is multi valued
gap> ImagesRepresentative( i4, () );
()
```

`ImagesRepresentative` calls
map`.operations.ImagesRepresentative(` *map*`,` *elm* `)` and returns this value.

The default function called this way is `MappingOps.ImagesRepresentative`, which calls `Images(` *map*`,` *elm* `)` and returns the first element in this set. Look in the index under **ImagesRepresentative** to see for which mappings this function is overlaid.

43.11 PreImage

`PreImage(` *bij*`,` *img* `)`

In this form `PreImage` returns the preimage of the element *img* of the range of the bijection *bij* under *bij*. The preimage is the unique element of the source of *bij* that is mapped by *bij* to *img*. Note that *bij* must be a bijection, a mapping that is not a bijection is not allowed (see 43.12).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
```

```
        return x ^ 4;
    end )
    gap> PreImage( p4, (5,6,7) );
    Error, <bij> must be a bijection, not an arbitrary mapping
    gap> p5 := MappingByFunction( g, g, x -> x^5 );
    MappingByFunction( g, g, function ( x )
        return x ^ 5;
    end )
    gap> PreImage( p5, (2,4)(5,6,7) );
    (2,4)(5,7,6)
```

`PreImage(` *bij* `,` *imgs* `)`

In this form `PreImage` returns the preimage of the elements *imgs* of the range of the bijection *bij* under *bij*. The primage of *imgs* is the set of all preimages of the elements in *imgs*. *imgs* may be a proper set (see 28.2) or a domain (see 4). The result will be a subset of the source of *bij*, either as a proper set or as a domain. Again *bij* must be a bijection, a mapping that is not a bijection is not allowed (see 43.12).

```
    gap> PreImage( p4, [ (), (5,6,7) ] );
    [ (), (5,6,7), (2,4), (2,4)(5,6,7), (1,2)(3,4), (1,2)(3,4)(5,6,7),
      (1,2,3,4), (1,2,3,4)(5,6,7), (1,3), (1,3)(5,6,7), (1,3)(2,4),
      (1,3)(2,4)(5,6,7), (1,4,3,2), (1,4,3,2)(5,6,7), (1,4)(2,3),
      (1,4)(2,3)(5,6,7) ]
    gap> PreImage( p5, Subgroup( g, [ (5,7,6), (2,4) ] ) );
    [ (), (5,6,7), (5,7,6), (2,4), (2,4)(5,6,7), (2,4)(5,7,6) ]
```

`PreImage(` *map* `)`

In this form `PreImage` returns the preimage of the mapping *map*. The preimage is the set of elements *elm* of the source of *map* that are actually mapped to at least one element, i.e., for which `PreImages(` *map*`,` *elm* `)` is nonempty. Note that in this case the argument may be an arbitrary mapping (especially a multi valued one).

```
    gap> PreImage( p4 ) = g;
    true
```

`PreImage` firsts checks in which form it is called.

In the first case it calls *bij*`.operations.PreImageElm(` *bij* `,` *elm* `)` and returns this value.

The default function called this way is `MappingOps.PreImageElm`, which checks that *bij* is indeed a bijection, calls `PreImages(` *bij* `,` *elm* `)`, and returns the single element of the set returned by `PreImages`. Look in the index under **PreImage** to see for which mappings this function is overlaid.

In the second case it calls *bij*`.operations.PreImageSet(` *bij* `,` *elms* `)` and returns this value.

The default function called this way is `MappingOps.PreImageSet`, which checks that *map* is indeed a bijection, calls `PreImages(` *bij* `,` *elms* `)`, and returns this value. Look in the index under **PreImage** to see for which mappings this is overlaid.

In the third case it tests if the field *map*`.preImage` is bound. If this field is bound, it simply returns this value. Otherwise it calls *map*`.operations.PreImageRange(` *map* `)`, remembers the returned value in *map*`.preImage`, and returns it.

The default function called this way is `MappingOps.PreImageRange`, which calls `PreImages(` *map*`,` *map*`.source )`, and returns this value. This function is seldom overlaid, since all the work is done by *map*`.operations.PreImagesSet`.

43.12 PreImages

`PreImages(` *map*`,` *img* `)`

In the first form `PreImages` returns the set of elements from the source of the mapping *map* that are mapped by *map* to the element *img* in the range of *map*, i.e., the set of elements *elm* such that *img* `in Images(` *map*`,` *elm* `)` (see 43.9). *map* may be a multi valued mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> PreImages( p4, (5,6,7) );
[ (5,6,7), (2,4)(5,6,7), (1,2)(3,4)(5,6,7), (1,2,3,4)(5,6,7),
  (1,3)(5,6,7), (1,3)(2,4)(5,6,7), (1,4,3,2)(5,6,7),
  (1,4)(2,3)(5,6,7) ]
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> PreImages( p5, (2,4)(5,6,7) );
[ (2,4)(5,7,6) ]
```

`PreImages(` *map*`,` *imgs* `)`

In the second form `PreImages` returns the set of all preimages of the elements in the set of elements *imgs*, i.e., the union of the preimages of the single elements of *imgs*. *map* may be a multi valued mapping.

```
gap> PreImages( p4, [ (), (5,6,7) ] );
[ (), (5,6,7), (2,4), (2,4)(5,6,7), (1,2)(3,4), (1,2)(3,4)(5,6,7),
  (1,2,3,4), (1,2,3,4)(5,6,7), (1,3), (1,3)(5,6,7), (1,3)(2,4),
  (1,3)(2,4)(5,6,7), (1,4,3,2), (1,4,3,2)(5,6,7), (1,4)(2,3),
  (1,4)(2,3)(5,6,7) ]
gap> PreImages( p5, [ (), (5,6,7) ] );
[ (), (5,7,6) ]
```

`PreImages` first checks in which form it is called.

In the first case it calls *map*`.operations.PreImagesElm(` *map*`,` *img* `)` and returns this value.

The default function called this way is `MappingOps.PreImagesElm`, which runs through all elements of the source of *map*, if it is finite, and returns the set of those that have *img* in their images. Look in the index under **PreImages** to see for which mappings this function is overlaid.

In the second case if calls *map*`.operations.PreImagesSet(` *map*`,` *imgs* `)` and returns this value.

The default function called this way is `MappingOps.PreImagesSet`, which returns the union of the preimages of all the elements of the set *imgs*. Look in the index under **PreImages** to see for which mappings this function is overlaid.

43.13 PreImagesRepresentative

`PreImagesRepresentative(` *map*`,` *img* `)`

`PreImagesRepresentative` returns an representative of the set of preimages of *img* under *map*, i.e., a single element *elm*, such that *img* `in Images(` *map*`,` *elm* `)` (see 43.9).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> PreImagesRepresentative( p4, (5,6,7) );
(5,6,7)
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> PreImagesRepresentative( p5, (2,4)(5,6,7) );
(2,4)(5,7,6)
```

`PreImagesRepresentative` calls
map`.operations.PreImagesRepresentative(` *map*`,` *img* `)` and returns this value.

The default function called this way is `MappingOps.PreImagesRepresentative`, which calls `PreImages(` *map*`,` *img* `)` and returns the first element in this set. Look in the index under **PreImagesRepresentative** to see for which mappings this function is overlaid.

43.14 CompositionMapping

`CompositionMapping(` *map1*`..` `)`

`CompositionMapping` returns the composition of the mappings *map1*, *map2*, etc. where the range of each mapping must be a subset of the source of the previous mapping. The mappings need not be single valued mappings, i.e., multi valued mappings are allowed.

The composition of *map1* and *map2* is the mapping *map* that maps each element *elm* of the source of *map2* to `Images(` *map1*`, Images(` *map2*`,` *elm* `) )`. If *map1* and *map2* are single valued mappings this can also be expressed as *map2* `*` *map1* (see 43.7). Note the reversed operands.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
```

```
    return x ^ 5;
end )
gap> p20 := CompositionMapping( p4, p5 );
CompositionMapping( MappingByFunction( g, g, function ( x )
    return x ^ 4;
end ), MappingByFunction( g, g, function ( x )
    return x ^ 5;
end ) )
gap> (2,4)(5,6,7) ^ p20;
(5,7,6)
```

`CompositionMapping` calls
map2`.operations.CompositionMapping(` *map1*`,` *map2* `)` and returns this value.

The default function called this way is `MappingOps.CompositionMapping`, which constructs a new mapping *com*. This new mapping remembers *map1* and *map2* as its factors in *com*`.map1` and *com*`.map2` and delegates everything to them. For example to compute `Images(` *com*`,` *elm* `)`, *com*`.operations.ImagesElm` calls `Images(` *com*`.map1, Images(` *com*`.map2,` *elm* `) )`. Look in the index under **CompositionMapping** to see for which mappings this function is overlaid.

43.15 PowerMapping

`PowerMapping(` *map*`,` *n* `)`

`PowerMapping` returns the *n*-th power of the mapping *map*. *map* must be a mapping whose range is a subset of its source. *n* must be a nonnegative integer. *map* may be a multi valued mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> p16 := p4 ^ 2;
CompositionMapping( CompositionMapping( IdentityMapping( g ), MappingB\
yFunction( g, g, function ( x )
    return x ^ 4;
end ) ), CompositionMapping( IdentityMapping( g ), MappingByFunction( \
g, g, function ( x )
    return x ^ 4;
end ) ) )
gap> p16 = MappingByFunction( g, g, x -> x^16 );
true
```

`PowerMapping` calls *map*`.operations.PowerMapping(` *map*`,` *n* `)` and returns this value.

The default function called this way is `MappingOps.PowerMapping`, which computes the power using a binary powering algorithm, `IdentityMapping`, and `CompositionMapping`. This function is seldom overlaid, because `CompositionMapping` does the interesting work.

43.16 InverseMapping

`InverseMapping(` *map* `)`

`InverseMapping` returns the inverse mapping of the mapping *map*. The inverse mapping *inv* is a mapping with source *map*`.range`, range *map*`.source`, such that each element *elm* of its source is mapped to the set `PreImages(` *map*`,` *elm* `)` (see 43.12). *map* may be a multi valued mapping.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> i4 := InverseMapping( p4 );
InverseMapping( MappingByFunction( g, g, function ( x )
    return x ^ 4;
end ) )
gap> Images( i4, () );
[ (), (2,4), (1,2)(3,4), (1,2,3,4), (1,3), (1,3)(2,4), (1,4,3,2),
  (1,4)(2,3) ]
```

`InverseMapping` first tests if the field *map*`.inverseMapping` is bound. If the field is bound, it returns its value. Otherwise it calls *map*`.operations.InverseMapping(` *map* `)`, remembers the returned value in *map*`.inverseMapping`, and returns it.

The default function called this way is `MappingOps.InverseMapping`, which constructs a new mapping *inv*. This new mapping remembers *map* as its own inverse mapping in *inv*`.inverseMapping` and delegates everything to it. For example to compute `Image(` *inv*`,` *elm* `)`, *inv*`.operations.ImageElm` calls `PreImage(`*inv*`.inverseMapping,`*elm*`)`. Special types of mappings will overlay this default function with more efficient functions.

43.17 IdentityMapping

`IdentityMapping(` *D* `)`

`IdentityMapping` returns the identity mapping on the domain *D*.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> i := IdentityMapping( g );
IdentityMapping( g )
gap> (1,2,3,4) ^ i;
(1,2,3,4)
gap> IsBijection( i );
true
```

`IdentityMapping` calls *D*`.operations.IdentityMapping(` *D* `)` and returns this value.

The functions usually called this way are `GroupOps.IdentityMapping` if the domain *D* is a group and `FieldOps.IdentityMapping` if the domain *D* is a field.

43.18 MappingByFunction

`MappingByFunction( D, E, fun )`

`MappingByFunction` returns a mapping *map* with source D and range E such that each element d of D is mapped to the element *fun*(d), where *fun* is a GAP3 function.

```
gap> g := Group( (1,2,3,4), (1,2) );;  g.name := "g";;
gap> m := MappingByFunction( g, g, x -> x^2 );
MappingByFunction( g, g, function ( x )
    return x ^ 2;
end )
gap> (1,2,3) ^ m;
(1,3,2)
gap> IsHomomorphism( m );
false
```

`MappingByFunction` constructs the mapping in the obvious way. For example the image of an element under *map* is simply computed by applying *fun* to the element.

43.19 Mapping Records

A mapping *map* is represented by a record with the following components

`isGeneralMapping`
: always `true`, indicating that this is a general mapping.

`source`
: the source of the mapping, i.e., the set of elements to which the mapping can be applied.

`range`
: the range of the mapping, i.e., a set in which all value of the mapping lie.

The following entries are optional. The functions with the corresponding names will generally test if they are present. If they are then their value is simply returned. Otherwise the functions will perform the computation and add those fields to those record for the next time.

`isMapping`
: `true` if *map* is a single valued mapping and `false` otherwise.

`isInjective`
: `isSurjective`
`isBijection`
`isHomomorphism`
`isMonomorphism`
`isEpimorphism`
`isIsomorphism`
`isEndomorphism`
`isAutomorphism`
`true` if *map* has the corresponding property and `false` otherwise.

`preImage`
: the subset of *map*`.source` of elements *pre* that are actually mapped to at least one element, i.e., for which `Images(` *map*`,` *pre* `)` is nonempty.

`image`
: the subset of *map*`.range` of the elements *img* that are actually values of the mapping, i.e., for which `PreImages(` *map*`,` *img* `)` is nonempty.

`inverseMapping`
: the inverse mapping of *map*. This is a mapping from *map*`.range` to *map*`.source` that maps each element *img* to the set `PreImages(` *map*`,` *img* `)`.

The following entry is optional. It must be bound only if the inverse of *map* is indeed a single valued mapping.

`inverseFunction`
: the inverse function of *map*.

The following entry is optional. It must be bound only if *map* is a homomorphism.

`kernel`
: the elements of *map*`.source` that are mapped to the identity element of *map*`.range`.

Chapter 44

Homomorphisms

An important special class of mappings are homomorphisms.

A mapping *map* is a **homomorphism** if the source and the range are domains of the same category, and *map* respects their structure. For example, if both source and range are groups and for each x, y in the source $(xy)^{map} = x^{map}y^{map}$, then *map* is a group homomorphism.

GAP3 currently supports field and group homomorphisms (see 6.13, 7.106).

Homomorphism are created by **homomorphism constructors**, which are ordinary GAP3 functions that return homomorphisms, such as `FrobeniusAutomorphism` (see 18.11) or `NaturalHomomorphism` (see 7.110).

The first section in this chapter describes the function that tests whether a mapping is a homomorphism (see 44.1). The next sections describe the functions that test whether a homomorphism has certain properties (see 44.2, 44.3, 44.4, 44.5, and 44.6). The last section describes the function that computes the kernel of a homomorphism (see 44.7).

Because homomorphisms are just a special case of mappings all operations and functions described in chapter 43 are applicable to homomorphisms. For example, the image of an element *elm* under a homomorphism *hom* can be computed by *elm* `^` *hom* (see 43.7).

44.1 IsHomomorphism

`IsHomomorphism(` *map* `)`

`IsHomomorphism` returns `true` if the mapping *map* is a homomorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping *map* is a **homomorphism** if the source and the range are sources of the same category, and *map* respects the structure. For example, if both source and range are groups and for each x, y in the source $(xy)^{map} = x^{map}y^{map}$, then *map* is a homomorphism.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
```

```
gap> IsHomomorphism( p4 );
true
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsHomomorphism( p5 );
true
gap> p6 := MappingByFunction( g, g, x -> x^6 );
MappingByFunction( g, g, function ( x )
    return x ^ 6;
end )
gap> IsHomomorphism( p6 );
false
```

`IsHomomorphism` first tests if the flag *map*`.isHomomorphism` is bound. If the flag is bound, it returns its value. Otherwise it calls *map*`.source.operations.IsHomomorphism(` *map* `)`, remembers the returned value in *map*`.isHomomorphism`, and returns it.

The functions usually called this way are `IsGroupHomomorphism` if the source of *map* is a group and `IsFieldHomomorphism` if the source of *map* is a field (see 7.107, 6.14).

44.2 IsMonomorphism

`IsMonomorphism(` *map* `)`

`IsMonomorphism` returns `true` if the mapping *map* is a monomorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping is a **monomorphism** if it is an injective homomorphism (see 43.3, 44.1).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsMonomorphism( p4 );
false
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsMonomorphism( p5 );
true
```

`IsMonomorphism` first test if the flag *map*`.isMonomorphism` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsMonomorphism(` *map* `)`, remembers the returned value in *map*`.isMonomorphism`, and returns it.

The default function called this way is `MappingOps.IsMonomorphism`, which calls the functions `IsInjective` and `IsHomomorphism`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done in `IsInjective` and `IsHomomorphism`.

44.3 IsEpimorphism

`IsEpimorphism(` *map* `)`

`IsEpimorphism` returns `true` if the mapping *map* is an epimorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping is an **epimorphism** if it is an surjective homomorphism (see 43.4, 44.1).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsEpimorphism( p4 );
false
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsEpimorphism( p5 );
true
```

`IsEpimorphism` first test if the flag *map*`.isEpimorphism` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsEpimorphism(` *map* `)`, remembers the returned value in *map*`.isEpimorphism`, and returns it.

The default function called this way is `MappingOps.IsEpimorphism`, which calls the functions `IsSurjective` and `IsHomomorphism`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done in `IsSurjective` and `IsHomomorphism`.

44.4 IsIsomorphism

`IsIsomorphism(` *map* `)`

`IsIsomorphism` returns `true` if the mapping *map* is an isomorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping is an **isomorphism** if it is a bijective homomorphism (see 43.5, 44.1).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsIsomorphism( p4 );
false
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsIsomorphism( p5 );
```

```
true
```

`IsIsomorphism` first test if the flag *map*`.isIsomorphism` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsIsomorphism(` *map* `)`, remembers the returned value in *map*`.isIsomorphism`, and returns it.

The default function called this way is `MappingOps.IsIsomorphism`, which calls the functions `IsInjective`, `IsSurjective`, and `IsHomomorphism`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done in `IsInjective`, `IsSurjective`, and `IsHomomorphism`.

44.5 IsEndomorphism

`IsEndomorphism(` *map* `)`

`IsEndomorphism` returns `true` if the mapping *map* is a endomorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping is an **endomorphism** if it is a homomorphism (see 44.1) and the range is a subset of the source.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsEndomorphism( p4 );
true
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsEndomorphism( p5 );
true
```

`IsEndomorphism` first test if the flag *map*`.isEndomorphism` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsEndomorphism(` *map* `)`, remembers the returned value in *map*`.isEndomorphism`, and returns it.

The default function called this way is `MappingOps.IsEndomorphism`, which tests if the range is a subset of the source, calls `IsHomomorphism`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done in `IsSubset` and `IsHomomorphism`.

44.6 IsAutomorphism

`IsAutomorphism(` *map* `)`

`IsAutomorphism` returns `true` if the mapping *map* is an automorphism and `false` otherwise. Signals an error if *map* is a multi valued mapping.

A mapping is an **automorphism** if it is an isomorphism where the source and the range are equal (see 44.4, 44.5).

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
```

```
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> IsAutomorphism( p4 );
false
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> IsAutomorphism( p5 );
true
```

`IsAutomorphism` first test if the flag *map*`.isAutomorphism` is bound. If the flag is bound, it returns this value. Otherwise it calls *map*`.operations.IsAutomorphism(` *map* `)`, remembers the returned value in *map*`.isAutomorphism`, and returns it.

The default function called this way is `MappingOps.IsAutomorphism`, which calls the functions `IsEndomorphism` and `IsBijection`, and returns the logical **and** of the results. This function is seldom overlaid, because all the interesting work is done in `IsEndomorphism` and `IsBijection`.

44.7 Kernel

`Kernel(` *hom* `)`

`Kernel` returns the kernel of the homomorphism *hom*. The kernel is usually returned as a source, though in some cases it might be returned as a proper set.

The kernel is the set of elements that are mapped *hom* to the identity element of *hom*`.range`, i.e., to *hom*`.range.identity` if *hom* is a group homomorphism, and to *hom*`.range.zero` if *hom* is a ring or field homomorphism. The kernel is a substructure of *hom*`.source`.

```
gap> g := Group( (1,2,3,4), (2,4), (5,6,7) );;  g.name := "g";;
gap> p4 := MappingByFunction( g, g, x -> x^4 );
MappingByFunction( g, g, function ( x )
    return x ^ 4;
end )
gap> Kernel( p4 );
Subgroup( g, [ (1,2,3,4), (1,4)(2,3) ] )
gap> p5 := MappingByFunction( g, g, x -> x^5 );
MappingByFunction( g, g, function ( x )
    return x ^ 5;
end )
gap> Kernel( p5 );
Subgroup( g, [  ] )
```

`Kernel` first tests if the field *hom*`.kernel` is bound. If the field is bound it returns its value. Otherwise it calls *hom*`.source.operations.Kernel(` *hom* `)`, remembers the returned value in *hom*`.kernel`, and returns it.

The functions usually called this way from the dispatcher are `KernelGroupHomomorphism` and `KernelFieldHomomorphism` (see 7.108, 6.15).

Chapter 45

Booleans

The two **boolean** values are `true` and `false`. They stand for the **logical** values of the same name. They appear mainly as values of the conditions in `if`-statements and `while`-loops.

This chapter contains sections describing the operations that are available for the boolean values (see 45.1, 45.2).

Further this chapter contains a section about the function `IsBool` (see 45.3). Note that it is a convention that the name of a function that tests a property, and returns `true` and `false` according to the outcome, starts with `Is`, as in `IsBool`.

45.1 Comparisons of Booleans

bool1 `=` *bool2*, *bool1* `<>` *bool2*

The equality operator `=` evaluates to `true` if the two boolean values *bool1* and *bool2* are equal, i.e., both are `true` or both are `false`, and `false` otherwise. The inequality operator `<>` evaluates to `true` if the two boolean values *bool1* and *bool2* are different and `false` otherwise. This operation is also called the **exclusive or**, because its value is `true` if exactly one of *bool1* or *bool2* is `true`.

You can compare boolean values with objects of other types. Of course they are never equal.

```
gap> true = false;
false
gap> false = (true = false);
true
gap> true <> 17;
true
```

bool1 `<` *bool2*, *bool1* `<=` *bool2*,
bool1 `>` *bool2*, *bool1* `>=` *bool2*

The operators `<`, `<=`, `>`, and `=>` evaluate to `true` if the boolean value *bool1* is less than, less than or equal to, greater than, and greater than or equal to the boolean value *bool2*. The ordering of boolean values is defined by `true < false`.

You can compare boolean values with objects of other types. Integers, rationals, cyclotomics, permutations, and words are smaller than boolean values. Objects of the other types, i.e., functions, lists, and records are larger.

```
gap> true < false;
true
gap> false >= true;
true
gap> 17 < true;
true
gap> true < [17];
true
```

45.2 Operations for Booleans

bool1 `or` *bool2*

The logical operator `or` evaluates to `true` if at least one of the two boolean operands *bool1* and *bool2* is `true` and to `false` otherwise.

`or` first evaluates *bool1*. If the value is neither `true` nor `false` an error is signalled. If the value is `true`, then `or` returns `true` **without** evaluating *bool2*. If the value is `false`, then `or` evaluates *bool2*. Again, if the value is neither `true` nor `false` an error is signalled. Otherwise `or` returns the value of *bool2*. This **short-circuited** evaluation is important if the value of *bool1* is `true` and evaluation of *bool2* would take much time or cause an error.

`or` is associative, i.e., it is allowed to write *b1* `or` *b2* `or` *b3*, which is interpreted as `(`*b1* `or` *b2*`)` `or` *b3*. `or` has the lowest precedence of the logical operators. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> true or false;
true
gap> false or false;
false
gap> i := -1;;  l := [1,2,3];;
gap> if i <= 0 or l[i] = false  then Print("aha\n");  fi;
aha     #  no error, because l[i] is not evaluated
```

bool1 `and` *bool2*

The logical operator `and` evaluates to `true` if both boolean operands *bool1* and *bool2* are `true` and to `false` otherwise.

`and` first evaluates *bool1*. If the value is neither `true` nor `false` an error is signalled. If the value is `false`, then `and` returns `false` **without** evaluating *bool2*. If the value is `true`, then `and` evaluates *bool2*. Again, if the value is neither `true` nor `false` an error is signalled. Otherwise `and` returns the value of *bool2*. This **short-circuited** evaluation is important if the value of *bool1* is `false` and evaluation of *bool2* would take much time or cause an error.

`and` is associative, i.e., it is allowed to write *b1* `and` *b2* `and` *b3*, which is interpreted as `(`*b1* `and` *b2*`)` `and` *b3*. `and` has higher precedence than the logical `or` operator, but lower than the unary logical `not` operator. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> true and false;
```

```
false
gap> true and true;
true
gap> false and 17;
false      # is no error, because 17 is never looked at
```

`not` *bool*

The logical operator `not` returns `true` if the boolean value *bool* is `false` and `true` otherwise. An error is signalled if *bool* does not evaluate to `true` or `false`.

`not` has higher precedence than the other logical operators, `or` and `and`. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> not true;
false
gap> not false;
true
```

45.3 IsBool

`IsBool(` *obj* `)`

`IsBool` returns `true` if *obj*, which may be an object of an arbitrary type, is a boolean value and `false` otherwise. `IsBool` will signal an error if *obj* is an unbound variable.

```
gap> IsBool( true );
true
gap> IsBool( false );
true
gap> IsBool( 17 );
false
```

Chapter 46

Records

Records are next to lists the most important way to collect objects together. A record is a collection of **components**. Each component has a unique **name**, which is an identifier that distinguishes this component, and a **value**, which is an object of arbitrary type. We often abbreviate **value of a component** to **element**. We also say that a record **contains** its elements. You can access and change the elements of a record using its name.

Record literals are written by writing down the components in order between `rec(` and `)`, and separating them by commas `,`. Each component consists of the name, the assignment operator `:=`, and the value. The **empty record**, i.e., the record with no components, is written as `rec()`.

```
gap> rec( a := 1, b := "2" );      # a record with two components
rec(
  a := 1,
  b := "2" )
gap> rec( a := 1, b := rec( c := 2 ) );      # record may contain records
rec(
  a := 1,
  b := rec(
      c := 2 ) )
```

Records usually contain elements of various types, i.e., they are usually not homogeneous like lists.

The first section in this chapter tells you how you can access the elements of a record (see 46.1).

The next sections tell you how you can change the elements of a record (see 46.2 and 46.3).

The next sections describe the operations that are available for records (see 46.4, 46.5, 46.6, and 46.7).

The next section describes the function that tests if an object is a record (see 46.8).

The next sections describe the functions that test whether a record has a component with a given name, and delete such a component (see 46.9 and 46.10). Those functions are also applicable to lists (see chapter 27).

The final sections describe the functions that create a copy of a record (see 46.11 and 46.12). Again those functions are also applicable to lists (see chapter 27).

46.1 Accessing Record Elements

rec`.`*name*

The above construct evaluates to the value of the record component with the name *name* in the record *rec*. Note that the *name* is not evaluated, i.e., it is taken literal.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a;
1
gap> r.b;
2
```

rec`.(`*name*`)`

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then evaluates to the element of the record *rec* whose name is, as a string, equal to *name*.

```
gap> old := rec( a := 1, b := 2 );;
gap> new := rec();
rec(
   )
gap> for i  in RecFields( old )  do
>        new.(i) := old.(i);
>    od;
gap> new;
rec(
  a := 1,
  b := 2 )
```

If *rec* does not evaluate to a record, or if *name* does not evaluate to a string, or if *rec*`.`*name* is unbound, an error is signalled. As usual you can leave the break loop (see 3.2) with `quit;`. On the other hand you can return a result to be used in place of the record element by `return` *expr*`;`.

46.2 Record Assignment

rec`.`*name* `:=` *obj*`;`

The record assignment assigns the object *obj*, which may be an object of arbitrary type, to the record component with the name *name*, which must be an identifier, of the record *rec*. That means that accessing the element with name *name* of the record *rec* will return *obj* after this assignment. If the record *rec* has no component with the name *name*, the record is automatically extended to make room for the new component.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a := 10;;  r;
rec(
  a := 10,
  b := 2 )
gap> r.c := 3;;  r;
```

```
rec(
  a := 10,
  b := 2,
  c := 3 )
```

The function `IsBound` (see 46.9) can be used to test if a record has a component with a certain name, the function `Unbind` (see 46.10) can be used to remove a component with a certain name again.

Note that assigning to a record changes the record. The ability to change an object is only available for lists and records (see 46.3).

rec.(*name*) = *obj*;

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then assigns *obj* to the record component of the record *rec* whose name is, as a string, equal to *name*.

If *rec* does not evaluate to a record, *name* does not evaluate to a string, or *obj* is a call to a function that does not return a value, e.g., `Print` (see 3.14), an error is signalled. As usual you can leave the break loop (see 3.2) with `quit;`. On the other hand you can continue the assignment by returning a record in the first case, a string in the second, or an object to be assigned in the third, using `return` *expr*;.

46.3 Identical Records

With the record assignment (see 46.2) it is possible to change a record. The ability to change an object is only available for lists and records. This section describes the semantic consequences of this fact.

You may think that in the following example the second assignment changes the integer, and that therefore the above sentence, which claimed that only records and lists can be changed, is wrong.

```
i := 3;
i :=  i + 1;
```

But in this example not the **integer** `3` is changed by adding one to it. Instead the **variable** `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the following example

```
r := rec( a := 1 );
r := rec( a := 1, b := 2 );
```

The second assignment does not change the first record, instead it assigns a new record to the variable `r`. On the other hand, in the following example the record is changed by the second assignment.

```
r := rec( a := 1 );
r.b := 2;
```

To understand the difference first think of a variable as a name for an object. The important point is that a record can have several names at the same time. An assignment *var* `:=` *record*; means in this interpretation that *var* is a name for the object *record*. At the end of the following example r2 still has the value rec(a := 1) as this record has not been changed and nothing else has been assigned to r2.

```
r1 := rec( a := 1 );
r2 := r1;
r1 := rec( a := 1, b := 2 );
```

But after the following example the record for which r2 is a name has been changed and thus the value of r2 is now rec(a := 1, b := 2).

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

We shall say that two records are **identical** if changing one of them by a record assignment also changes the other one. This is slightly incorrect, because if **two** records are identical, there are actually only two names for **one** record. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical records must be equal, because there is only one records with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two records are identical.

If you enter a record literal then the record denoted by this literal is a new record that is not identical to any other record. Thus in the following example r1 and r2 are not identical, though they are equal of course.

```
r1 := rec( a := 1 );
r2 := rec( a := 1 );
```

Also in the following example, no records in the list l are identical.

```
l := [];
for i  in [1..10]  do
    l[i] := rec( a := 1 );
od;
```

If you assign a record to a variable no new record is created. Thus the record value of the variable on the left hand side and the record on the right hand side of the assignment are identical. So in the following example r1 and r2 are identical records.

```
r1 := rec( a := 1 );
r2 := r1;
```

If you pass a record as argument, the old record and the argument of the function are identical. Also if you return a record from a function, the old record and the value of the function call are identical. So in the following example r1 and r2 are identical record

```
r1 := rec( a := 1 );
f := function ( r )  return r;  end;
r2 := f( r1 );
```

The functions Copy and ShallowCopy (see 46.11 and 46.12) accept a record and return a new record that is equal to the old record but that is **not** identical to the old record. The difference between Copy and ShallowCopy is that in the case of ShallowCopy the corresponding elements of the new and the old records will be identical, whereas in the case of `Copy` they will only be equal. So in the following example `r1` and `r2` are not identical records.

```
r1 := rec( a := 1 );
r2 := Copy( r1 );
```

If you change a record it keeps its identity. Thus if two records are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two records that are not identical will never become identical if you change one of them. So in the following example both `r1` and `r2` are changed, and are still identical.

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

46.4 Comparisons of Records

rec1 `=` *rec2*
rec1 `<>` *rec2*

The equality operator `=` returns `true` if the record *rec1* is equal to the record *rec2* and `false` otherwise. The inequality operator `<>` returns `true` if the record *rec1* is not equal to *rec2* and `false` otherwise.

Usually two records are considered equal, if for each component of one record the other record has a component of the same name with an equal value and vice versa. You can also compare records with other objects, they are of course different, unless the record has a special comparison function (see below) that says otherwise.

```
gap> rec( a := 1, b := 2 ) = rec( b := 2, a := 1 );
true
gap> rec( a := 1, b := 2 ) = rec( a := 2, b := 1 );
false
gap> rec( a := 1 ) = rec( a := 1, b := 2 );
false
gap> rec( a := 1 ) = 1;
false
```

However a record may contain a special `operations` record that contains a function that is called when this record is an operand of a comparison. The precise mechanism is as follows. If the operand of the equality operator `=` is a record, and if this record has an element with the name `operations` that is a record, and if this record has an element with the name `=` that is a function, then this function is called with the operands of `=` as arguments, and the value of the operation is the result returned by this function. In this case a record may also be equal to an object of another type if this function says so. It is probably not a good idea to define a comparison function in such a way that the resulting relation is not an equivalence relation, i.e., not reflexive, symmetric, and transitive. Note that there is no corresponding `<>` function, because *left* `<>` *right* is implemented as `not` *left* `=` *right*.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation (see 1.30). Note that the `=` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueOps := rec( );;
```

```
gap> ResidueOps.\= := function ( l, r )
>   return (l.modulus = r.modulus)
>     and (l.representative-r.representative) mod l.modulus = 0;
> end;;
gap> Residue := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueOps );
> end;;
gap> l := Residue( 13, 23 );;
gap> r := Residue( -10, 23 );;
gap> l = r;
true
gap> l = Residue( 10, 23 );
false
```

rec1 `<`*rec2*
rec1 `<=` *rec2*
rec1 `>` *rec2*
rec1 `>=` *rec2*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the record *rec1* is less than, less than or equal to, greater than, and greater than or equal to the record *rec2*, and to `false` otherwise.

To compare records we imagine that the components of both records are sorted according to their names. Then the records are compared lexicographically with unbound elements considered smaller than anything else. Precisely one record *rec1* is considered less than another record *rec2* if *rec2* has a component with name *name2* and either *rec1* has no component with this name or *rec1*`.`*name2* `<`*rec2*`.`*name2* and for each component of *rec1* with name *name1* `<`*name2* *rec2* has a component with this name and *rec1*`.`*name1* `=` *rec2*`.`*name1*. Records may also be compared with objects of other types, they are greater than anything else, unless the record has a special comparison function (see below) that says otherwise.

```
gap> rec( a := 1, b := 2 ) < rec( b := 2, a := 1 );
false     # they are equal
gap> rec( a := 1, b := 2 ) < rec( a := 2, b := 0 );
true     # the a elements are compared first and 1 is less than 2
gap> rec( a := 1 ) < rec( a := 1, b := 2 );
true     # unbound is less than 2
gap> rec( a := 1 ) < rec( a := 0, b := 2 );
false     # the a elements are compared first and 0 is less than 1
gap> rec( b := 1 ) < rec( b := 0, a := 2 );
true     # the a-s are compared first and unbound is less than 2
gap> rec( a := 1 ) < 1;
false     # other objects are less than records
```

However a record may contain a special `operations` record that contains a function that is called when this record is an operand of a comparison. The precise mechanism is as follows. If the operand of the equality operator `<` is a record, and if this record has an element with the name `operations` that is a record, and if this record has an element with the name `<`

that is a function, then this function is called with the operands of `<` as arguments, and the value of the operation is the result returned by this function. In this case a record may also be smaller than an object of another type if this function says so. It is probably not a good idea to define a comparison function in such a way that the resulting relation is not an ordering relation, i.e., not antisymmetric, and transitive. Note that there are no corresponding `<=`, `>`, and `>=` functions, since those operations are implemented as `not` *right* `<`*left*, *right* `<`*left*, and `not` *left* `<`*right* respectively.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation (see 1.30). Note that the `<` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueOps := rec( );;
gap> ResidueOps.\< := function ( l, r )
>   if l.modulus <> r.modulus  then
>     Error("<l> and <r> must have the same modulus");
>   fi;
>   return   l.representative mod l.modulus
>          < r.representative mod r.modulus;
> end;;
gap> Residue := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueOps );
> end;;
gap> l := Residue( 13, 23 );;
gap> r := Residue( -1, 23 );;
gap> l < r;
true     # 13 is less than 22
gap> l < Residue( 10, 23 );
false     # 10 is less than 13
```

46.5 Operations for Records

Usually no operations are defined for record. However a record may contain a special `operations` record that contains functions that are called when this record is an operand of a binary operation. This mechanism is detailed below for the addition.

obj `+` *rec*, *rec* `+` *obj*

If either operand is a record, and if this record contains an element with name `operations` that is a record, and if this record in turn contains an element with the name `+` that is a function, then this function is called with the two operands as arguments, and the value of the addition is the value returned by that function. If both operands are records with such a function *rec*`.operations.+`, then the function of the **right** operand is called. If either operand is a record, but neither operand has such a function *rec*`.operations.+`, an error is signalled.

obj `-` *rec*, *rec* `-` *obj*
obj `*` *rec*, *rec* `*` *obj*

obj `/` *rec*, *rec* `/` *obj*
obj `mod` *rec*, *rec* `mod` *obj*
obj `^` *rec*, *rec* `^` *obj*

This is evaluated similar, but the functions must obviously be called `-`, `*`, `/`, `mod`, `^` respectively.

The following example shows one piece of the definition of a residue classes, using record operations. Of course this is far from a complete implementation (see 1.30). Note that the `*` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueOps := rec( );;
gap> ResidueOps.\* := function ( l, r )
>   if l.modulus <> r.modulus  then
>     Error("<l> and <r> must have the same modulus");
>   fi;
>   return rec(
>     representative := (l.representative * r.representative)
>                       mod l.modulus,
>     modulus        := l.modulus,
>     operations     := ResidueOps );
> end;;
gap> Residue := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueOps );
> end;;
gap> l := Residue( 13, 23 );;
gap> r := Residue( -1, 23 );;
gap> s := l * r;
rec(
  representative := 10,
  modulus := 23,
  operations := rec(
      \* := function ( l, r ) ... end ) )
```

46.6 In for Records

element `in` *rec*

Usually the membership test is only defined for lists. However a record may contain a special `operations` record, that contains a function that is called when this record is the right operand of the `in` operator. The precise mechanism is as follows.

If the right operand of the `in` operator is a record, and if this record contains an element with the name `operations` that is a record, and if this record in turn contains an element with the name `in` that is a function, then this function is called with the two operands as arguments, and the value of the membership test is the value returned by that function. The function should of course return `true` or `false`.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation (see 1.30). Note that the `in` must be quoted, so that it is taken as an identifier (see 2.5).

```
gap> ResidueOps := rec( );;
gap> ResidueOps.\in := function ( l, r )
>   if IsInt( l )  then
>     return (l - r.representative) mod r.modulus = 0;
>   else
>     return false;
>   fi;
> end;;
gap> Residue:= function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueOps );
> end;;
gap> l := Residue( 13, 23 );;
gap> -10 in l;
true
gap> 10 in l;
false
```

46.7 Printing of Records

`Print(` *rec* `)`

If a record is printed by `Print` (see 3.14, 3.15, and 3.16) or by the main loop (see 3.1), it is usually printed as record literal, i.e., as a sequence of components, each in the format *name* `:=` *value*, separated by commas and enclosed in `rec(` and `)`.

```
gap> r := rec();;  r.a := 1;;  r.b := 2;;
gap> r;
rec(
  a := 1,
  b := 2 )
```

But if the record has an element with the name `operations` that is a record, and if this record has an element with the name `Print` that is a function, then this function is called with the record as argument. This function must print whatever the printed representation of the record should look like.

The following example shows one piece of the definition of residue classes, using record operations. Of course this is far from a complete implementation (see 1.30). Note that it is typical for records that mimic group elements to print as a function call that, when evaluated, will create this group element record.

```
gap> ResidueOps := rec( );;
gap> ResidueOps.Print := function ( r )
>   Print( "Residue( ",
>               r.representative mod r.modulus, ", ",
```

```
>               r.modulus, " )" );
> end;;
gap> Residue := function ( representative, modulus )
>   return rec(
>     representative := representative,
>     modulus        := modulus,
>     operations     := ResidueOps );
> end;;
gap> l := Residue( 33, 23 );
Residue( 10, 23 )
```

46.8 IsRec

`IsRec(` *obj* `)`

`IsRec` returns `true` if the object *obj*, which may be an object of arbitrary type, is a record, and `false` otherwise. Will signal an error if *obj* is a variable with no assigned value.

```
gap> IsRec( rec( a := 1, b := 2 ) );
true
gap> IsRec( IsRec );
false
```

46.9 IsBound

`IsBound(` *rec*`.`*name* `)`
`IsBound(` *list*`[`*n*`]` `)`

In the first form `IsBound` returns `true` if the record *rec* has a component with the name *name*, which must be an ident and `false` otherwise. *rec* must evaluate to a record, otherwise an error is signalled.

In the second form `IsBound` returns `true` if the list *list* has a element at the position n, and `false` otherwise. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> IsBound( r.a );
true
gap> IsBound( r.c );
false
gap> l := [  , 2, 3,  , 5,  , 7,  ,  ,   , 11 ];;
gap> IsBound( l[7] );
true
gap> IsBound( l[4] );
false
gap> IsBound( l[101] );
false
```

Note that `IsBound` is special in that it does not evaluate its argument, otherwise it would always signal an error when it is supposed to return `false`.

46.10 Unbind

`Unbind(` *rec.name* `)`
`Unbind(` *list*`[`*n*`] )`

In the first form `Unbind` deletes the component with the name *name* in the record *rec*. That is, after execution of `Unbind`, *rec* no longer has a record component with this name. Note that it is not an error to unbind a nonexisting record component. *rec* must evaluate to a record, otherwise an error is signalled.

In the second form `Unbind` deletes the element at the position *n* in the list *list*. That is, after execution of `Unbind`, *list* no longer has an assigned value at the position *n*. Note that it is not an error to unbind a nonexisting list element. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> Unbind( r.a );  r;
rec(
  b := 2 )
gap> Unbind( r.c );  r;
rec(
  b := 2 )
gap> l := [  , 2, 3,  5,  , 7,  ,  ,  , 11 ];;
gap> Unbind( l[3] );  l;
[ , 2,, 5,, 7,,,, 11 ]
gap> Unbind( l[4] );  l;
[ , 2,,,, 7,,,, 11 ]
```

Note that `Unbind` does not evaluate its argument, otherwise there would be no way for `Unbind` to tell which component to remove in which record, because it would only receive the value of this component.

46.11 Copy

`Copy(` *obj* `)`

`Copy` returns a copy *new* of the object *obj*. You may apply `Copy` to objects of any type, but for objects that are not lists or records `Copy` simply returns the object itself.

For lists and records the result is a **new** list or record that is **not identical** to any other list or record (see 27.9 and 46.3). This means that you may modify this copy *new* by assignments (see 27.6 and 46.2) or by adding elements to it (see 27.7 and 27.8), without modifying the original object *obj*.

```
gap> list1 := [ 1, 2, 3 ];;
gap> list2 := Copy( list1 );
[ 1, 2, 3 ]
gap> list2[1] := 0;;  list2;
[ 0, 2, 3 ]
gap> list1;
[ 1, 2, 3 ]
```

That `Copy` returns the object itself if it is not a list or a record is consistent with this definition, since there is no way to change the original object *obj* by modifying *new*, because in fact there is no way to change the object *new*.

`Copy` basically executes the following code for lists, and similar code for records.

```
new := [];
for i  in [1..Length(obj)]  do
    if IsBound(obj[i])  then
        new[i] := Copy( obj[i] );
    fi;
od;
```

Note that `Copy` recursively copies all elements of the object *obj*. If you only want to copy the top level use `ShallowCopy` (see 46.12).

```
gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;
gap> list2 := Copy( list1 );
[ [ 1, 2 ], [ 3, 4 ] ]
gap> list2[1][1] := 0;;  list2;
[ [ 0, 2 ], [ 3, 4 ] ]
gap> list1;
[ [ 1, 2 ], [ 3, 4 ] ]
```

The above code is not entirely correct. If the object *obj* contains a list or record twice this list or record is not copied twice, as would happen with the above definition, but only once. This means that the copy *new* and the object *obj* have exactly the same structure when view as a general graph.

```
gap> sub := [ 1, 2 ];;  list1 := [ sub, sub ];;
gap> list2 := Copy( list1 );
[ [ 1, 2 ], [ 1, 2 ] ]
gap> list2[1][1] := 0;;  list2;
[ [ 0, 2 ], [ 0, 2 ] ]
gap> list1;
[ [ 1, 2 ], [ 1, 2 ] ]
```

46.12 ShallowCopy

`ShallowCopy(` *obj* `)`

`ShallowCopy` returns a copy of the object *obj*. You may apply `ShallowCopy` to objects of any type, but for objects that are not lists or records `ShallowCopy` simply returns the object itself.

For lists and records the result is a **new** list or record that is **not identical** to any other list or record (see 27.9 and 46.3). This means that you may modify this copy *new* by assignments (see 27.6 and 46.2) or by adding elements to it (see 27.7 and 27.8), without modifying the original object *obj*.

```
gap> list1 := [ 1, 2, 3 ];;
gap> list2 := ShallowCopy( list1 );
[ 1, 2, 3 ]
gap> list2[1] := 0;;  list2;
```

```
[ 0, 2, 3 ]
gap> list1;
[ 1, 2, 3 ]
```

That `ShallowCopy` returns the object itself if it is not a list or a record is consistent with this definition, since there is no way to change the original object *obj* by modifying *new*, because in fact there is no way to change the object *new*.

`ShallowCopy` basically executes the following code for lists, and similar code for records.

```
new := [];
for i  in [1..Length(obj)]  do
    if IsBound(obj[i])  then
        new[i] := obj[i];
    fi;
od;
```

Note that `ShallowCopy` only copies the top level. The subobjects of the new object *new* are identical to the corresponding subobjects of the object *obj*. If you want to copy recursively use `Copy` (see 46.11).

46.13 RecFields

`RecFields(` *rec* `)`

`RecFields` returns a list of strings corresponding to the names of the record components of the record *rec*.

```
gap> r := rec( a := 1, b := 2 );;
gap> RecFields( r );
[ "a", "b" ]
```

Note that you cannot use the string result in the ordinary way to access or change a record component. You must use the *rec*`.(`*name*`)` construct (see 46.1 and 46.2).

Chapter 47

Combinatorics

This chapter describes the functions that deal with combinatorics. We mainly concentrate on two areas. One is about **selections**, that is the ways one can select elements from a set. The other is about **partitions**, that is the ways one can partition a set into the union of pairwise disjoint subsets.

First this package contains various functions that are related to the number of selections from a set (see 47.1, 47.2) or to the number of partitions of a set (see 47.3, 47.4, 47.5). Those numbers satisfy literally thousands of identities, which we do no mention in this document, for a thorough treatment see [GKP90].

Then this package contains functions to compute the selections from a set (see 47.6), ordered selections, i.e., selections where the order in which you select the elements is important (see 47.7), selections with repetitions, i.e., you are allowed to select the same element more than once (see 47.8) and ordered selections with repetitions (see 47.9).

As special cases of ordered combinations there are functions to compute all permutations (see 47.10), all fixpointfree permutations (see 47.11) of a list.

This package also contains functions to compute partitions of a set (see 47.12), partitions of an integer into the sum of positive integers (see 47.13, 47.15) and ordered partitions of an integer into the sum of positive integers (see 47.14).

Moreover, it provides three functions to compute Fibonacci numbers (see 47.22), Lucas sequences (see 47.23), or Bernoulli numbers (see 47.24).

Finally, there is a function to compute the number of permutations that fit a given 1-0 matrix (see 47.25).

All these functions are in the file `"LIBNAME/combinat.g"`.

47.1 Factorial

`Factorial(` *n* `)`

`Factorial` returns the **factorial** $n!$ of the positive integer n, which is defined as the product $1 * 2 * 3 * .. * n$.

$n!$ is the number of permutations of a set of n elements. $1/n!$ is the coefficient of x^n in the formal series e^x, which is the generating function for factorial.

```
gap> List( [0..10], Factorial );
[ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
gap> Factorial( 30 );
265252859812191058636308480000000
```

`PermutationsList` (see 47.10) computes the set of all permutations of a list.

47.2 Binomial

`Binomial(` n, k `)`

`Binomial` returns the **binomial coefficient** $\binom{n}{k}$ of integers n and k, which is defined as $n!/(k!(n-k)!)$ (see 47.1). We define $\binom{0}{0} = 1$, $\binom{n}{k} = 0$ if $k < 0$ or $n < k$, and $\binom{n}{k} = (-1)^k \binom{-n+k-1}{k}$ if $n < 0$, which is consistent with $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

$\binom{n}{k}$ is the number of combinations with k elements, i.e., the number of subsets with k elements, of a set with n elements. $\binom{n}{k}$ is the coefficient of the term x^k of the polynomial $(x+1)^n$, which is the generating function for $\binom{n}{*}$, hence the name.

```
gap> List( [0..4], k->Binomial( 4, k ) );
[ 1, 4, 6, 4, 1 ]      # Knuth calls this the trademark of Binomial
gap> List( [0..6], n->List( [0..6], k->Binomial( n, k ) ) );;
gap> PrintArray( last );
[ [   1,   0,   0,   0,   0,   0,   0 ],      # the lower triangle is
  [   1,   1,   0,   0,   0,   0,   0 ],      # called Pascal's triangle
  [   1,   2,   1,   0,   0,   0,   0 ],
  [   1,   3,   3,   1,   0,   0,   0 ],
  [   1,   4,   6,   4,   1,   0,   0 ],
  [   1,   5,  10,  10,   5,   1,   0 ],
  [   1,   6,  15,  20,  15,   6,   1 ] ]
gap> Binomial( 50, 10 );
10272278170
```

`NrCombinations` (see 47.6) is the generalization of `Binomial` for multisets. `Combinations` (see 47.6) computes the set of all combinations of a multiset.

47.3 Bell

`Bell(` n `)`

`Bell` returns the **Bell number** $B(n)$. The Bell numbers are defined by $B(0) = 1$ and the recurrence $B(n+1) = \sum_{k=0}^{n} \binom{n}{k} B(k)$.

$B(n)$ is the number of ways to partition a set of n elements into pairwise disjoint nonempty subsets (see 47.12). This implies of course that $B(n) = \sum_{k=0}^{n} S_2(n,k)$ (see 47.5). $B(n)/n!$ is the coefficient of x^n in the formal series e^{e^x-1}, which is the generating function for $B(n)$.

```
gap> List( [0..6], n -> Bell( n ) );
[ 1, 1, 2, 5, 15, 52, 203 ]
gap> Bell( 14 );
190899322
```

47.4 Stirling1

`Stirling1(` n, k `)`

`Stirling1` returns the **Stirling number of the first kind** $S_1(n,k)$ of the integers n and k. Stirling numbers of the first kind are defined by $S_1(0,0)=1$, $S_1(n,0)=S_1(0,k)=0$ if $n,k <> 0$ and the recurrence $S_1(n,k)=(n-1)S_1(n-1,k)+S_1(n-1,k-1)$.

$S_1(n,k)$ is the number of permutations of n points with k cycles. Stirling numbers of the first kind appear as coefficients in the series $n!\binom{x}{n}=\sum_{k=0}^{n} S_1(n,k)x^k$ which is the generating function for Stirling numbers of the first kind. Note the similarity to $x^n = \sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$ (see 47.5). Also the definition of S_1 implies $S_1(n,k)=S_2(-k,-n)$ if $n,k<0$. There are many formulae relating Stirling numbers of the first kind to Stirling numbers of the second kind, Bell numbers, and Binomial numbers.

```
gap> List( [0..4], k->Stirling1( 4, k ) );
[ 0, 6, 11, 6, 1 ]      # Knuth calls this the trademark of S_1
gap> List( [0..6], n->List( [0..6], k->Stirling1( n, k ) ) );;
gap> PrintArray( last );
[ [    1,    0,    0,    0,    0,    0,    0 ],     # Note the similarity
  [    0,    1,    0,    0,    0,    0,    0 ],     # with Pascal's
  [    0,    1,    1,    0,    0,    0,    0 ],     # triangle for the
  [    0,    2,    3,    1,    0,    0,    0 ],     # Binomial numbers
  [    0,    6,   11,    6,    1,    0,    0 ],
  [    0,   24,   50,   35,   10,    1,    0 ],
  [    0,  120,  274,  225,   85,   15,    1 ] ]
gap> Stirling1(50,10);
101623020926367490059043797119309944043405505380503665627365376
```

47.5 Stirling2

`Stirling2(` n, k `)`

`Stirling2` returns the **Stirling number of the second kind** $S_2(n,k)$ of the integers n and k. Stirling numbers of the second kind are defined by $S_2(0,0)=1$, $S_2(n,0)=S_2(0,k)=0$ if $n,k <> 0$ and the recurrence $S_2(n,k)=kS_2(n-1,k)+S_2(n-1,k-1)$.

$S_2(n,k)$ is the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets (see 47.12). Stirling numbers of the second kind appear as coefficients in the expansion of $x^n=\sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$. Note the similarity to $n!\binom{x}{n}=\sum_{k=0}^{n} S_1(n,k)x^k$ (see 47.4). Also the definition of S_2 implies $S_2(n,k)=S_1(-k,-n)$ if $n,k<0$. There are many formulae relating Stirling numbers of the second kind to Stirling numbers of the first kind, Bell numbers, and Binomial numbers.

```
gap> List( [0..4], k->Stirling2( 4, k ) );
[ 0, 1, 7, 6, 1 ]      # Knuth calls this the trademark of S_2
gap> List( [0..6], n->List( [0..6], k->Stirling2( n, k ) ) );;
gap> PrintArray( last );
[ [   1,   0,   0,   0,   0,   0,   0 ],     # Note the similarity with
  [   0,   1,   0,   0,   0,   0,   0 ],     # Pascal's triangle for
  [   0,   1,   1,   0,   0,   0,   0 ],     # the Binomial numbers
  [   0,   1,   3,   1,   0,   0,   0 ],
```

```
  [  0,  1,  7,  6,  1,  0,  0 ],
  [  0,  1, 15, 25, 10,  1,  0 ],
  [  0,  1, 31, 90, 65, 15,  1 ] ]
gap> Stirling2( 50, 10 );
26154716515862881292012777396577993781727011
```

47.6 Combinations

`Combinations(` *mset* `)`
`Combinations(` *mset*`,` *k* `)`

`NrCombinations(` *mset* `)`
`NrCombinations(` *mset*`,` *k* `)`

In the first form `Combinations` returns the set of all combinations of the multiset *mset*. In the second form `Combinations` returns the set of all combinations of the multiset *mset* with k elements.

In the first form `NrCombinations` returns the number of combinations of the multiset *mset*. In the second form `NrCombinations` returns the number of combinations of the multiset *mset* with k elements.

A **combination** of *mset* is an unordered selection without repetitions and is represented by a sorted sublist of *mset*. If *mset* is a proper set, there are $\binom{|mset|}{k}$ (see 47.2) combinations with k elements, and the set of all combinations is just the **powerset** of *mset*, which contains all **subsets** of *mset* and has cardinality $2^{|mset|}$.

```
gap> Combinations( [1,2,2,3] );
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 2 ], [ 1, 2, 2, 3 ], [ 1, 2, 3 ],
  [ 1, 3 ], [ 2 ], [ 2, 2 ], [ 2, 2, 3 ], [ 2, 3 ], [ 3 ] ]
gap> NrCombinations( [1..52], 5 );
2598960    # number of different hands in a game of poker
```

The function `Arrangements` (see 47.7) computes ordered selections without repetitions, `UnorderedTuples` (see 47.8) computes unordered selections with repetitions and `Tuples` (see 47.9) computes ordered selections with repetitions.

47.7 Arrangements

`Arrangements(` *mset* `)`
`Arrangements(` *mset*`,` *k* `)`

`NrArrangements(` *mset* `)`
`NrArrangements(` *mset*`,` *k* `)`

In the first form `Arrangements` returns the set of arrangements of the multiset *mset*. In the second form `Arrangements` returns the set of all arrangements with k elements of the multiset *mset*.

In the first form `NrArrangements` returns the number of arrangements of the multiset *mset*. In the second form `NrArrangements` returns the number of arrangements with k elements of the multiset *mset*.

An **arrangement** of *mset* is an ordered selection without repetitions and is represented by a list that contains only elements from *mset*, but maybe in a different order. If *mset* is a proper set there are $|mset|!/(|mset|-k)!$ (see 47.1) arrangements with k elements.

As an example of arrangements of a multiset, think of the game Scrabble. Suppose you have the six characters of the word `settle` and you have to make a four letter word. Then the possibilities are given by

```
gap> Arrangements( ["s","e","t","t","l","e"], 4 );
[ [ "e", "e", "l", "s" ], [ "e", "e", "l", "t" ],
  [ "e", "e", "s", "l" ], [ "e", "e", "s", "t" ],
  # 96 more possibilities
  [ "t", "t", "s", "e" ], [ "t", "t", "s", "l" ] ]
```

Can you find the five proper English words, where `lets` does not count? Note that the fact that the list returned by `Arrangements` is a proper set means in this example that the possibilities are listed in the same order as they appear in the dictionary.

```
gap> NrArrangements( ["s","e","t","t","l","e"] );
523
```

The function `Combinations` (see 47.6) computes unordered selections without repetitions, `UnorderedTuples` (see 47.8) computes unordered selections with repetitions and `Tuples` (see 47.9) computes ordered selections with repetitions.

47.8 UnorderedTuples

`UnorderedTuples(` *set*`,` *k* `)`

`NrUnorderedTuples(` *set*`,` *k* `)`

`UnorderedTuples` returns the set of all unordered tuples of length k of the set *set*.

`NrUnorderedTuples` returns the number of unordered tuples of length k of the set *set*.

An **unordered tuple** of length k of *set* is a unordered selection with repetitions of *set* and is represented by a sorted list of length k containing elements from *set*. There are $\binom{|set|+k-1}{k}$ (see 47.2) such unordered tuples.

Note that the fact that `UnOrderedTuples` returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* k times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

As an example for unordered tuples think of a poker-like game played with 5 dice. Then each possible hand corresponds to an unordered five-tuple from the set [1..6]

```
gap> NrUnorderedTuples( [1..6], 5 );
252
gap> UnorderedTuples( [1..6], 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 2 ], [ 1, 1, 1, 1, 3 ],
  [ 1, 1, 1, 1, 4 ], [ 1, 1, 1, 1, 5 ], [ 1, 1, 1, 1, 6 ],
  # 99 more tuples
  [ 1, 3, 4, 5, 6 ], [ 1, 3, 4, 6, 6 ], [ 1, 3, 5, 5, 5 ],
  # 99 more tuples
  [ 3, 3, 4, 4, 5 ], [ 3, 3, 4, 4, 6 ], [ 3, 3, 4, 5, 5 ],
  # 39 more tuples
  [ 5, 5, 6, 6, 6 ], [ 5, 6, 6, 6, 6 ], [ 6, 6, 6, 6, 6 ] ]
```

The function `Combinations` (see 47.6) computes unordered selections without repetitions, `Arrangements` (see 47.7) computes ordered selections without repetitions and `Tuples` (see 47.9) computes ordered selections with repetitions.

47.9 Tuples

`Tuples(` *set*`,` *k* `)`

`NrTuples(` *set*`,` *k* `)`

`Tuples` returns the set of all ordered tuples of length k of the set *set*.

`NrTuples` returns the number of all ordered tuples of length k of the set *set*.

An **ordered tuple** of length k of *set* is an ordered selection with repetition and is represented by a list of length k containing elements of *set*. There are $|set|^k$ such ordered tuples.

Note that the fact that `Tuples` returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* k times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

```
gap> Tuples( [1,2,3], 2 );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 3, 1 ], [ 3, 2 ], [ 3, 3 ] ]
gap> NrTuples( [1..10], 5 );
100000
```

`Tuples(`*set*`,`*k*`)` can also be viewed as the k-fold cartesian product of *set* (see 27.26).

The function `Combinations` (see 47.6) computes unordered selections without repetitions, `Arrangements` (see 47.7) computes ordered selections without repetitions, and finally the function `UnorderedTuples` (see 47.8) computes unordered selections with repetitions.

47.10 PermutationsList

`PermutationsList(` *mset* `)`

`NrPermutationsList(` *mset* `)`

`PermutationsList` returns the set of permutations of the multiset *mset*.

`NrPermutationsList` returns the number of permutations of the multiset *mset*.

A **permutation** is represented by a list that contains exactly the same elements as *mset*, but possibly in different order. If *mset* is a proper set there are $|mset|!$ (see 47.1) such permutations. Otherwise if the first elements appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|mset|!/(k_1!k_2!..)$, which is sometimes called multinomial coefficient.

```
gap> PermutationsList( [1,2,3] );
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 2, 1, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ],
  [ 3, 2, 1 ] ]
gap> PermutationsList( [1,1,2,2] );
[ [ 1, 1, 2, 2 ], [ 1, 2, 1, 2 ], [ 1, 2, 2, 1 ], [ 2, 1, 1, 2 ],
  [ 2, 1, 2, 1 ], [ 2, 2, 1, 1 ] ]
gap> NrPermutationsList( [1,2,2,3,3,3,4,4,4,4] );
12600
```

The function `Arrangements` (see 47.7) is the generalization of `PermutationsList` that allows you to specify the size of the permutations. `Derangements` (see 47.11) computes permutations that have no fixpoints.

47.11 Derangements

`Derangements(` *list* `)`

`NrDerangements(` *list* `)`

`Derangements` returns the set of all derangements of the list *list*.

`NrDerangements` returns the number of derangements of the list *list*.

A **derangement** is a fixpointfree permutation of *list* and is represented by a list that contains exactly the same elements as *list*, but in such an order that the derangement has at no position the same element as *list*. If the list *list* contains no element twice there are exactly $|list|!(1/2! - 1/3! + 1/4! - ..(-1)^n/n!)$ derangements.

Note that the ratio `NrPermutationsList([1..n])/NrDerangements([1..n])`, which is $n!/(n!(1/2! - 1/3! + 1/4! - ..(-1)^n/n!))$ is an approximation for the base of the natural logarithm $e = 2.7182818285$, which is correct to about n digits.

As an example of derangements suppose that you have to send four different letters to four different people. Then a derangement corresponds to a way to send those letters such that no letter reaches the intended person.

```
gap> Derangements( [1,2,3,4] );
[ [ 2, 1, 4, 3 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 3, 1, 4, 2 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 3, 1, 2 ],
  [ 4, 3, 2, 1 ] ]
gap> NrDerangements( [1..10] );
1334961
gap> Int( 10^7*NrPermutationsList([1..10])/last );
27182816
gap> Derangements( [1,1,2,2,3,3] );
[ [ 2, 2, 3, 3, 1, 1 ], [ 2, 3, 1, 3, 1, 2 ], [ 2, 3, 1, 3, 2, 1 ],
  [ 2, 3, 3, 1, 1, 2 ], [ 2, 3, 3, 1, 2, 1 ], [ 3, 2, 1, 3, 1, 2 ],
  [ 3, 2, 1, 3, 2, 1 ], [ 3, 2, 3, 1, 1, 2 ], [ 3, 2, 3, 1, 2, 1 ],
  [ 3, 3, 1, 1, 2, 2 ] ]
gap> NrDerangements( [1,2,2,3,3,3,4,4,4,4] );
338
```

The function `PermutationsList` (see 47.10) computes all permutations of a list.

47.12 PartitionsSet

`PartitionsSet(` *set* `)`
`PartitionsSet(` *set*, *k* `)`

`NrPartitionsSet(` *set* `)`
`NrPartitionsSet(` *set*, *k* `)`

In the first form `PartitionsSet` returns the set of all unordered partitions of the set *set*. In the second form `PartitionsSet` returns the set of all unordered partitions of the set *set* into k pairwise disjoint nonempty sets.

In the first form `NrPartitionsSet` returns the number of unordered partitions of the set *set*. In the second form `NrPartitionsSet` returns the number of unordered partitions of the set *set* into k pairwise disjoint nonempty sets.

An **unordered partition** of *set* is a set of pairwise disjoint nonempty sets with union *set* and is represented by a sorted list of such sets. There are $B(|set|)$ (see 47.3) partitions of the set *set* and $S_2(|set|, k)$ (see 47.5) partitions with k elements.

```
gap> PartitionsSet( [1,2,3] );
[ [ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1 ], [ 2, 3 ] ], [ [ 1, 2 ], [ 3 ] ],
  [ [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 2 ] ] ]
gap> PartitionsSet( [1,2,3,4], 2 );
[ [ [ 1 ], [ 2, 3, 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ],
  [ [ 1, 2, 3 ], [ 4 ] ], [ [ 1, 2, 4 ], [ 3 ] ],
  [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 3, 4 ], [ 2 ] ],
  [ [ 1, 4 ], [ 2, 3 ] ] ]
gap> NrPartitionsSet( [1..6] );
203
gap> NrPartitionsSet( [1..10], 3 );
9330
```

Note that `PartitionsSet` does currently not support multisets and that there is currently no ordered counterpart.

47.13 Partitions

`Partitions(` *n* `)`
`Partitions(` *n*`,` *k* `)`

`NrPartitions(` *n* `)`
`NrPartitions(` *n*`,` *k* `)`

In the first form `Partitions` returns the set of all (unordered) partitions of the positive integer n. In the second form `Partitions` returns the set of all (unordered) partitions of the positive integer n into sums with k summands.

In the first form `NrPartitions` returns the number of (unordered) partitions of the positive integer n. In the second form `NrPartitions` returns the number of (unordered) partitions of the positive integer n into sums with k summands.

An **unordered partition** is an unordered sum $n = p_1+p_2+..+p_k$ of positive integers and is represented by the list $p = [p_1, p_2, .., p_k]$, in nonincreasing order, i.e., $p_1 >= p_2 >= .. >= p_k$. We write $p \vdash n$. There are approximately $E^{\pi\sqrt{2/3n}}/4\sqrt{3}n$ such partitions.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore $p(n) := NrPartitions(n)$ is the number of conjugacy classes of the symmetric group on n points.

Ramanujan found the identities $p(5i+4) = 0$ mod 5, $p(7i+5) = 0$ mod 7 and $p(11i+6) = 0$ mod 11 and many other fascinating things about the number of partitions.

Do not call `Partitions` with an n much larger than 40, in which case there are 37338 partitions, since the list will simply become too large.

```
gap> Partitions( 7 );
[ [ 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1, 1 ], [ 2, 2, 1, 1, 1 ],
  [ 2, 2, 2, 1 ], [ 3, 1, 1, 1, 1 ], [ 3, 2, 1, 1 ], [ 3, 2, 2 ],
  [ 3, 3, 1 ], [ 4, 1, 1, 1 ], [ 4, 2, 1 ], [ 4, 3 ], [ 5, 1, 1 ],
```

```
  [ 5, 2 ], [ 6, 1 ], [ 7 ] ]
gap> Partitions( 8, 3 );
[ [ 3, 3, 2 ], [ 4, 2, 2 ], [ 4, 3, 1 ], [ 5, 2, 1 ], [ 6, 1, 1 ] ]
gap> NrPartitions( 7 );
15
gap> NrPartitions( 100 );
190569292
```

The function `OrderedPartitions` (see 47.14) is the ordered counterpart of `Partitions`.

47.14 OrderedPartitions

`OrderedPartitions(` *n* `)`
`OrderedPartitions(` *n*, *k* `)`

`NrOrderedPartitions(` *n* `)`
`NrOrderedPartitions(` *n*, *k* `)`

In the first form `OrderedPartitions` returns the set of all ordered partitions of the positive integer n. In the second form `OrderedPartitions` returns the set of all ordered partitions of the positive integer n into sums with k summands.

In the first form `NrOrderedPartitions` returns the number of ordered partitions of the positive integer n. In the second form `NrOrderedPartitions` returns the number of ordered partitions of the positive integer n into sums with k summands.

An **ordered partition** is an ordered sum $n = p_1 + p_2 + .. + p_k$ of positive integers and is represented by the list $[p_1, p_2, .., p_k]$. There are totally 2^{n-1} ordered partitions and $\binom{n-1}{k-1}$ (see 47.2) partitions with k summands.

Do not call `OrderedPartitions` with an n larger than 15, the list will simply become too large.

```
gap> OrderedPartitions( 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 1, 3 ],
  [ 1, 2, 1, 1 ], [ 1, 2, 2 ], [ 1, 3, 1 ], [ 1, 4 ], [ 2, 1, 1, 1 ],
  [ 2, 1, 2 ], [ 2, 2, 1 ], [ 2, 3 ], [ 3, 1, 1 ], [ 3, 2 ],
  [ 4, 1 ], [ 5 ] ]
gap> OrderedPartitions( 6, 3 );
[ [ 1, 1, 4 ], [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 4, 1 ], [ 2, 1, 3 ],
  [ 2, 2, 2 ], [ 2, 3, 1 ], [ 3, 1, 2 ], [ 3, 2, 1 ], [ 4, 1, 1 ] ]
gap> NrOrderedPartitions(20);
524288
```

The function `Partitions` (see 47.13) is the unordered counterpart of `OrderedPartitions`.

47.15 RestrictedPartitions

`RestrictedPartitions(` *n*, *set* `)`
`RestrictedPartitions(` *n*, *set*, *k* `)`

`NrRestrictedPartitions(` *n*, *set* `)`
`NrRestrictedPartitions(` *n*, *set*, *k* `)`

In the first form `RestrictedPartitions` returns the set of all restricted partitions of the positive integer n with the summands of the partition coming from the set *set*. In the second form `RestrictedPartitions` returns the set of all partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set *set*.

In the first form `NrRestrictedPartitions` returns the number of restricted partitions of the positive integer n with the summands coming from the set *set*. In the second form `NrRestrictedPartitions` returns the number of restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set *set*.

A **restricted partition** is like an ordinary partition (see 47.13) an unordered sum $n = p_1 + p_2 + .. + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, .., p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set *set*, while for ordinary partitions they may be elements from `[1..n]`.

```
gap> RestrictedPartitions( 8, [1,3,5,7] );
[ [ 1, 1, 1, 1, 1, 1, 1, 1 ], [ 3, 1, 1, 1, 1, 1 ], [ 3, 3, 1, 1 ],
  [ 5, 1, 1, 1 ], [ 5, 3 ], [ 7, 1 ] ]
gap> NrRestrictedPartitions( 50, [1,5,10,25,50] );
50
```

The last example tells us that there are 50 ways to return 50 cent change using 1, 5, 10 cent coins, quarters and halfdollars.

47.16 SignPartition

`SignPartition(` *pi* `)`

returns the sign of a permutation with cycle structure *pi*.

```
gap> SignPartition([6,5,4,3,2,1]);
-1
```

This function actually describes a homomorphism of the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n (see 20.6). Partitions of sign 1 are called **even** partitions while partitions of sign -1 are called **odd**.

47.17 AssociatedPartition

`AssociatedPartition(` *pi* `)`

returns the associated partition of the partition *pi*.

```
gap> AssociatedPartition([4,2,1]);
[ 3, 2, 1, 1 ]
gap> AssociatedPartition([6]);
[ 1, 1, 1, 1, 1, 1 ]
```

The **associated partition** of a partition *pi* is defined to be the partition belonging to the transposed of the Young diagram of *pi*.

47.18 BetaSet

`BetaSet(` *p* `)`

Here p is a partition (a non-increasing list of positive integers). `BetaSet` returns the corresponding nomalized Beta set.

```
gap> BetaSet([3,3,1]);
  [ 1, 4, 5 ]
```

A beta-set is a set of positive integers, up to the **shift** equivalence relation. This equivalence relation is the transitive closure of the elementary equivalence of $[s_1, \ldots, s_n]$ and $[0, 1 + s_1, \ldots, 1 + s_n]$. An equivalence class has exactly one member which does not contain 0:it is called the normalized beta-set. To a partition $p_1 \geq p_2 \geq \ldots \geq p_n > 0$ is associated a beta-set, whose normalized representative is $p_n, p_{n-1} + 1, \ldots, p_1 + n - 1$.

47.19 Dominates

`Dominates(`μ`, `ν`)`

The dominance ordering is an important partial order in representation theory. `Dominates(`μ`, `ν`)` returns `true` if either μ=ν or for all $i \geq 1$, $\sum_{j=1}^{i} \mu_j \geq \sum_{j=1}^{i} \nu_j$, and `false` otherwise.

```
gap> Dominates([5,4],[4,4,1]);
true
```

47.20 PowerPartition

`PowerPartition(` *pi*`, ` *k* `)`

returns the partition corresponding to the k-th power of a permutation with cycle structure *pi*.

```
gap> PowerPartition([6,5,4,3,2,1], 3);
[ 5, 4, 2, 2, 2, 2, 1, 1, 1, 1 ]
```

Each part l of *pi* is replaced by $d = \gcd(l, k)$ parts l/d. So if *pi* is a partition of n then pi^k also is a partition of n. `PowerPartition` describes the powermap of symmetric groups.

47.21 PartitionTuples

`PartitionTuples(` n`, ` r `)`

`NrPartitionTuples(` n`, ` r `)`

`PartitionTuples(` n`, ` r `)` returns the list of all r–tuples of partitions that together partition n. `NrPartitionTuples` just returns their number.

```
gap> PartitionTuples(3, 2);
[ [ [ 1, 1, 1 ], [  ] ], [ [ 1, 1 ], [ 1 ] ], [ [ 1 ], [ 1, 1 ] ],
  [ [  ], [ 1, 1, 1 ] ], [ [ 2, 1 ], [  ] ], [ [ 1 ], [ 2 ] ],
  [ [ 2 ], [ 1 ] ], [ [  ], [ 2, 1 ] ], [ [ 3 ], [  ] ],
  [ [  ], [ 3 ] ] ]
gap> NrPartitionTuples(3,2);
10
```

r–tuples of partitions describe the classes and the characters of wreath products of groups with r conjugacy classes with the symmetric group S_n.

47.22 Fibonacci

`Fibonacci(` n `)`

`Fibonacci` returns the nth number of the **Fibonacci sequence**. The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1}F_{-n}$, which is consistent with the recurrence relation.

Using generating functions one can prove that $F_n = \phi^n - 1/\phi^n$, where ϕ is $(\sqrt{5}+1)/2$, i.e., one root of $x^2 - x - 1 = 0$. Fibonacci numbers have the property $Gcd(F_m, F_n) = F_{Gcd(m,n)}$. But a pair of Fibonacci numbers requires more division steps in Euclid's algorithm (see 5.26) than any other pair of integers of the same size. `Fibonnaci(`k`)` is the special case `Lucas(1,-1,`k`)[1]` (see 47.23).

```
gap> Fibonacci( 10 );
55
gap> Fibonacci( 35 );
9227465
gap> Fibonacci( -10 );
-55
```

47.23 Lucas

`Lucas(` P`,` Q`,` k `)`

`Lucas` returns the k-th values of the **Lucas sequence** with parameters P and Q, which must be integers, as a list of three integers.

Let α, β be the two roots of $x^2 - Px + Q$ then we define
$Lucas(P,Q,k)[1] = U_k = (\alpha^k - \beta^k)/(\alpha - \beta)$ and
$Lucas(P,Q,k)[2] = V_k = (\alpha^k + \beta^k)$ and as a convenience
$Lucas(P,Q,k)[3] = Q^k$.

The following recurrence relations are easily derived from the definition
$U_0 = 0, U_1 = 1, U_k = PU_{k-1} - QU_{k-2}$ and
$V_0 = 2, V_1 = P, V_k = PV_{k-1} - QV_{k-2}$.
Those relations are actually used to define `Lucas` if $\alpha = \beta$.

Also the more complex relations used in `Lucas` can be easily derived
$U_{2k} = U_k V_k, U_{2k+1} = (PU_{2k} + V_{2k})/2$ and
$V_{2k} = V_k^2 - 2Q^k, V_{2k+1} = ((P^2 - 4Q)U_{2k} + PV_{2k})/2$.

`Fibonnaci(`k`)` (see 47.22) is simply `Lucas(1,-1,`k`)[1]`. In an abuse of notation, the sequence `Lucas(1,-1,`k`)[2]` is sometimes called the Lucas sequence.

```
gap> List( [0..10], i->Lucas(1,-2,i)[1] );
[ 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341 ]     # 2^k - (-1)^k)/3
gap> List( [0..10], i->Lucas(1,-2,i)[2] );
[ 2, 1, 5, 7, 17, 31, 65, 127, 257, 511, 1025 ]     # 2^k + (-1)^k
gap> List( [0..10], i->Lucas(1,-1,i)[1] );
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]     # Fibonacci sequence
gap> List( [0..10], i->Lucas(2,1,i)[1] );
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]     # the roots are equal
```

47.24 Bernoulli

`Bernoulli( n )`

`Bernoulli` returns the n-th **Bernoulli number** B_n, which is defined by $B_0 = 1$ and $B_n = -\sum_{k=0}^{n-1} \binom{n+1}{k} B_k/(n+1)$.

$B_n/n!$ is the coefficient of x^n in the power series of $x/e^x - 1$. Except for $B_1 = -1/2$ the Bernoulli numbers for odd indices m are zero.

```
gap> Bernoulli( 4 );
-1/30
gap> Bernoulli( 10 );
5/66
gap> Bernoulli( 12 );
-691/2730     # there is no simple pattern in Bernoulli numbers
gap> Bernoulli( 50 );
495057205241079648212477525/66     # and they grow fairly fast
```

47.25 Permanent

`Permanent( mat )`

`Permanent` returns the **permanent** of the matrix *mat*. The permanent is defined by $\sum_{p \in Symm(n)} \prod_{i=1}^{n} mat[i][i^p]$.

Note the similarity of the definition of the permanent to the definition of the determinant. In fact the only difference is the missing sign of the permutation. However the permanent is quite unlike the determinant, for example it is not multilinear or alternating. It has however important combinatorical properties.

```
gap> Permanent( [[0,1,1,1],
>                [1,0,1,1],
>                [1,1,0,1],
>                [1,1,1,0]] );
9    # inefficient way to compute NrDerangements([1..4])
gap> Permanent( [[1,1,0,1,0,0,0],
>                [0,1,1,0,1,0,0],
>                [0,0,1,1,0,1,0],
>                [0,0,0,1,1,0,1],
>                [1,0,0,0,1,1,0],
>                [0,1,0,0,0,1,1],
>                [1,0,1,0,0,0,1]] );
24    # 24 permutations fit the projective plane of order 2
```

Chapter 48

Tables of Marks

The concept of a table of marks was introduced by W. Burnside in his book *Theory of Groups of Finite Order* [Bur55]. Therefore a table of marks is sometimes called a Burnside matrix.

The table of marks of a finite group G is a matrix whose rows and columns are labelled by the conjugacy classes of subgroups of G and where for two subgroups A and B the (A, B)–entry is the number of fixed points of B in the transitive action of G on the cosets of A in G. So the table of marks characterizes all permutation representations of G.

Moreover, the table of marks gives a compact description of the subgroup lattice of G, since from the numbers of fixed points the numbers of conjugates of a subgroup B contained in a subgroup A can be derived.

This chapter describes a function (see 48.4) which restores a table of marks from the GAP3 library of tables of marks (see 48.3) or which computes the table of marks for a given group from the subgroup lattice of that group. Moreover this package contains a function to display a table of marks (see 48.12), a function to check the consistency of a table of marks (see 48.11), functions which switch between several forms of representation (see 48.5, 48.6, 48.8, and 48.9), functions which derive information about the group from the table of marks (see 48.10, 48.13, 48.14, 48.15, 48.16, 48.17, 48.18, 48.19, 48.20, 48.21, and 48.22), and some functions for the generic construction of a table of marks (see 48.23, 48.24, and 48.25).

The functions described in this chapter are implemented in the file `LIBNAME/"tom.g"`.

48.1 More about Tables of Marks

Let G be a finite group with n conjugacy classes of subgroups $C_1, \ldots, C_n$ and representatives $H_i \in C_i$, $i = 1, \ldots, n$. The **table of marks** of G is defined to be the $n \times n$ matrix $M = (m_{ij})$ where m_{ij} is the number of fixed points of the subgroup H_j in the action of G on the cosets of H_i in G.

Since H_j can only have fixed points if it is contained in a one point stablizer the matrix M is lower triangular if the classes C_i are sorted according to the following condition; if H_i is contained in a conjugate of H_j then $i \leq j$.

Moreover, the diagonal entries m_{ii} are nonzero since m_{ii} equals the index of H_i in its normalizer in G. Hence M is invertible. Since any transitive action of G is equivalent to an action on the cosets of a subgroup of G, one sees that the table of marks completely characterizes permutation representations of G.

The entries m_{ij} have further meanings. If H_1 is the trivial subgroup of G then each mark m_{i1} in the first column of M is equal to the index of H_i in G since the trivial subgroup fixes all cosets of H_i. If $H_n = G$ then each m_{nj} in the last row of M is equal to 1 since there is only one coset of G in G. In general, m_{ij} equals the number of conjugates of H_i which contain H_j, multiplied by the index of H_i in its normalizer in G. Moreover, the number c_{ij} of conjugates of H_j which are contained in H_i can be derived from the marks m_{ij} via the formula

$$c_{ij} = \frac{m_{ij} m_{j1}}{m_{i1} m_{jj}}.$$

Both the marks m_{ij} and the numbers of subgroups c_{ij} are needed for the functions described in this chapter.

48.2 Table of Marks Records

A table of marks is represented by a record. This record has at least a component `subs` which is a list where for each conjugacy class of subgroups the class numbers of its subgroups are stored. These are exactly the positions in the corresponding row of the table of marks which have nonzero entries.

The marks themselves can be stored in the component `marks` which is a list that contains for each entry in the component `subs` the corresponding nonzero value of the table of marks.

The same information is, however, given by the three components `nrSubs`, `length`, and `order`, where `nrSubs` is a list which contains for each entry in the component `subs` the corresponding number of conjugates which are contained in a subgroup, `length` is a list which contains for each class of subgroups its length, and `order` is a list which contains for each class of subgroups their order.

So a table of marks consists either of the components `subs` and `marks` or of the components `subs`, `nrSubs`, `length`, and `order`. The functions `Marks` (see 48.5) and `NrSubs` (see 48.6) will derive one representation from the other when needed.

Additional information about a table of marks is needed by some functions. The class numbers of normalizers are stored in the component `normalizer`. The number of the derived subgroup of the whole group is stored in the component `derivedSubgroup`.

48.3 The Library of Tables of Marks

This package of functions comes together with a library of tables of marks. The library files are stored in a directory `TOMNAME`. The file `TOMNAME/"tmprimar.tom"` is the primary file of the library of tables of marks. It contains the information where to find a table and the function `TomLibrary` which restores a table from the library.

The secondary files are

```
tmaltern.tom  tmmath24.tom  tmsuzuki.tom  tmunitar.tom
tmlinear.tom  tmmisc.tom    tmsporad.tom  tmsymple.tom
```

The list `TOMLIST` contains for each table an entry with its name and the name of the file where it is stored.

A table of marks which is restored from the library will be stored as a component of the record `TOM`.

48.4 TableOfMarks

`TableOfMarks(` *str* `)`

If the argument *str* given to `TableOfMarks` is a string then `TableOfMarks` will search the library of tables of marks (see 48.3) for a table with name *str*. If such a table is found then `TableOfMarks` will return a copy of that table. Otherwise `TableOfMarks` will return `false`.

```
gap> a5 := TableOfMarks( "A5" );
rec(
  derivedSubgroup := 9,
  normalizer := [ 9, 4, 6, 8, 7, 6, 7, 8, 9 ],
  nrSubs := [ [ 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 3, 1 ], [ 1, 1 ],
      [ 1, 3, 1, 1 ], [ 1, 5, 1, 1 ], [ 1, 3, 4, 1, 1 ],
      [ 1, 15, 10, 5, 6, 10, 6, 5, 1 ] ],
  order := [ 1, 2, 3, 4, 5, 6, 10, 12, 60 ],
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 5 ],
      [ 1, 2, 3, 6 ], [ 1, 2, 5, 7 ], [ 1, 2, 3, 4, 8 ],
      [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ],
  length := [ 1, 15, 10, 5, 6, 10, 6, 5, 1 ] )
gap> TableOfMarks( "A10" );
#W  TableOfMarks: no table of marks A10 found.
false
```

`TableOfMarks(` *grp* `)`

If `TableOfMarks` is called with a group *grp* as its argument then the table of marks of that group will be computed and returned in the compressed format. The computation of the table of marks requires the knowledge of the complete subgroup lattice of the group *grp*. If the lattice is not yet known then it will be constructed (see 7.75). This may take a while if the group *grp* is large.

Moreover, as the `Lattice` command is involved the applicability of `TableOfMarks` underlies the same restrictions with respect to the soluble residuum of *grp* as described in section 7.75. The result of `TableOfMarks` is assigned to the component `tableOfMarks` of the group record *grp*, so that the next call to `TableOfMarks` with the same argument can just return this component `tableOfMarks`.

Warning: Note that `TableOfMarks` has changed with the release GAP3 3.2. It now returns the table of marks in compressed form. However, you can apply the `MatTom` command (see 48.8) to convert it into the square matrix which was returned by `TableOfMarks` in GAP3 version 3.1.

```
gap> alt5 := AlternatingPermGroup( 5 );;
gap> TableOfMarks( alt5 );
rec(
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 5 ],
```

```
        [ 1, 2, 3, 6 ], [ 1, 2, 5, 7 ], [ 1, 2, 3, 4, 8 ],
        [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ],
    marks := [ [ 60 ], [ 30, 2 ], [ 20, 2 ], [ 15, 3, 3 ], [ 12, 2 ],
        [ 10, 2, 1, 1 ], [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ],
        [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ] )
gap> last = alt5.tableOfMarks;
true
```

For a pretty print display of a table of marks see 48.12.

48.5 Marks

`Marks(` *tom* `)`

`Marks` returns the list of lists of marks of the table of marks *tom*. If these are not yet stored in the component `marks` of *tom* then they will be computed and assigned to the component `marks`.

```
gap> Marks( a5 );
[ [ 60 ], [ 30, 2 ], [ 20, 2 ], [ 15, 3, 3 ], [ 12, 2 ],
  [ 10, 2, 1, 1 ], [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

48.6 NrSubs

`NrSubs(` *tom* `)`

`NrSubs` returns the list of lists of numbers of subgroups of the table of marks *tom*. If these are not yet stored in the component `nrSubs` of *tom* then they will be computed and assigned to the component `nrSubs`.

`NrSubs` also has to compute the orders and lengths from the marks.

```
gap> NrSubs( a5 );
[ [ 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 3, 1 ], [ 1, 1 ], [ 1, 3, 1, 1 ],
  [ 1, 5, 1, 1 ], [ 1, 3, 4, 1, 1 ], [ 1, 15, 10, 5, 6, 10, 6, 5, 1 ]
 ]
```

48.7 WeightsTom

`WeightsTom(` *tom* `)`

`WeightsTom` extracts the weights from a table of marks *tom*, i.e., the diagonal entries, indicating the index of a subgroup in its normalizer.

```
gap> wt := WeightsTom( a5 );
[ 60, 2, 2, 3, 2, 1, 1, 1, 1 ]
```

This information may be used to obtain the numbers of conjugate supergroups from the marks.

```
gap> marks := Marks( a5 );;
gap> List( [ 1 .. 9 ], x -> marks[x] / wt[x] );
[ [ 1 ], [ 15, 1 ], [ 10, 1 ], [ 5, 1, 1 ], [ 6, 1 ], [ 10, 2, 1, 1 ],
  [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

48.8 MatTom

`MatTom( tom )`

`MatTom` produces a square matrix corresponding to the table of marks *tom* in compressed form. For large tables this may need a lot of space.

```
gap> MatTom( a5 );
[ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
  [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
  [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
  [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

48.9 TomMat

`TomMat( mat )`

Given a matrix *mat* which contains the marks of a group as its entries, `TomMat` will produce the corresponding table of marks record.

```
gap> mat:=
> [ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
>   [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
>   [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
>   [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
>   [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ];;
gap> TomMat( mat );
rec(
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 5 ],
      [ 1, 2, 3, 6 ], [ 1, 2, 5, 7 ], [ 1, 2, 3, 4, 8 ],
      [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ],
  marks := [ [ 60 ], [ 30, 2 ], [ 20, 2 ], [ 15, 3, 3 ], [ 12, 2 ],
      [ 10, 2, 1, 1 ], [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ],
      [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ] )
gap> TomMat( IdentityMat( 7 ) );
rec(
  subs := [ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7 ] ],
  marks := [ [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ], [ 1 ] ] )
```

48.10 DecomposedFixedPointVector

`DecomposedFixedPointVector( tom, fix )`

Let the group with table of marks *tom* act as a permutation group on its conjugacy classes of subgroups, then *fix* is assumed to be a vector of fixed point numbers, i.e., a vector which contains for each class of subgroups the number of fixed points under that action. `DecomposedFixedPointVector` returns the decomposition of *fix* into rows of the table of marks. This decomposition corresponds to a decomposition of the action into transitive constituents. Trailing zeros in *fix* may be omitted.

```
gap> DecomposedFixedPointVector( a5, [ 16, 4, 1, 0, 1, 1, 1 ] );
```

```
[ ,,,,, 1, 1 ]
```

The vector *fix* may be any vector of integers. The resulting decomposition, however, will not be integral, in general.

```
gap> DecomposedFixedPointVector( a5, [ 0, 0, 0, 0, 1, 1 ] );
[ 2/5, -1, -1/2,, 1/2, 1 ]
```

48.11 TestTom

`TestTom(` *tom* `)`

`TestTom` decomposes all tensor products of rows of the table of marks *tom*. It returns `true` if all decomposition numbers are nonnegative integers and `false` otherwise. This provides a strong consistency check for a table of marks.

```
gap> TestTom( a5 );
true
```

48.12 DisplayTom

`DisplayTom(` *tom* `)`

`DisplayTom` produces a formatted output for the table of marks *tom*. Each line of output begins with the number of the corresponding class of subgroups. This number is repeated if the output spreads over several pages.

```
gap> DisplayTom( a5 );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
5:  12 . . . 2
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1
```

`DisplayTom(` *tom*`,` *arec* `)`

In this form `DisplayTom` takes a record *arec* as an additional parameter. If this record has a component `classes` which contains a list of class numbers then only the rows and columns of the matrix corresponding to this list are printed.

```
gap> DisplayTom( a5, rec( classes := [ 1, 2, 3, 4, 8 ] ) );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
8:   5 1 2 1 1
```

The record *arec* may also have a component `form` which enables the printing of tables of numbers of subgroups. If *arec*.`form` has the value `"subgroups"` then at position (i, j) the number of conjugates of H_j contained in H_i will be printed. If it has the value `"supergroups"` then at position (i, j) the number of conjugates of H_i which contain H_j will be printed.

```
gap> DisplayTom( a5, rec( form := "subgroups" ) );
1:  1
2:  1  1
3:  1  .  1
4:  1  3  . 1
5:  1  .  . . 1
6:  1  3  1 . .  1
7:  1  5  . . 1  . 1
8:  1  3  4 1 .  . . 1
9:  1 15 10 5 6 10 6 5 1

gap> DisplayTom( a5, rec( form := "supergroups" ) );
1:   1
2:  15 1
3:  10 . 1
4:   5 1 . 1
5:   6 . . . 1
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1
```

48.13 NormalizerTom

`NormalizerTom(` *tom*`,` *u* `)`

`NormalizerTom` tries to find conjugacy class of the normalizer of a subgroup with class number u. It will return the list of class numbers of those subgroups which have the right size and contain the subgroup and all subgroups which clearly contain it as a normal subgroup. If the normalizer is uniquely determined by these conditions then only its class number will be returned. `NormalizerTom` should never return an empty list.

```
gap> NormalizerTom( a5, 4 );
8
```

The example shows that a subgroup with class number 4 in A_5 (which is a Kleinan four group) is normalized by a subgroup in class 8. This class contains the subgroups of A_5 which are isomorphic to A_4.

48.14 IntersectionsTom

`IntersectionsTom(` *tom*`,` *a*`,` *b* `)`

The intersections of the two conjugacy classes of subgroups with class numbers a and b, respectively, are determined by the decomposition of the tensor product of their rows of marks. `IntersectionsTom` returns this decomposition.

```
gap> IntersectionsTom( a5, 8, 8 );
[ ,, 1,,,,, 1 ]
```

Any two subgroups of class number 8 (A_4) of A_5 are either equal and their intersection has again class number 8, or their intersection has class number 3, and is a cyclic subgroup of order 3.

48.15 IsCyclicTom

`IsCyclicTom(` *tom*, *n* `)`

A subgroup is cyclic if and only if the sum over the corresponding row of the inverse table of marks is nonzero (see [Ker91], page 125). Thus we only have to decompose the corresponding idempotent.

```
gap> for i in [ 1 .. 6 ] do
> Print( i, ": ", IsCyclicTom(a5, i), "  " );
> od;  Print( "\n" );
1: true  2: true  3: true  4: false  5: true  6: false
```

48.16 FusionCharTableTom

`FusionCharTableTom(` *tbl*, *tom* `)`

`FusionCharTableTom` determines the fusion of the classes of elements from the character table *tbl* into classes of cyclic subgroups on the table of marks *tom*.

```
gap> a5c := CharTable( "A5" );;
gap> fus := FusionCharTableTom( a5c, a5 );
[ 1, 2, 3, 5, 5 ]
```

48.17 PermCharsTom

`PermCharsTom(` *tom*, *fus* `)`

`PermCharsTom` reads the list of permutation characters from the table of marks *tom*. It therefore has to know the fusion map *fus* which sends each conjugacy class of elements of the group to the conjugacy class of subgroups they generate.

```
gap> PermCharsTom( a5, fus );
[ [ 60, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0 ], [ 20, 0, 2, 0, 0 ],
  [ 15, 3, 0, 0, 0 ], [ 12, 0, 0, 2, 2 ], [ 10, 2, 1, 0, 0 ],
  [ 6, 2, 0, 1, 1 ], [ 5, 1, 2, 0, 0 ], [ 1, 1, 1, 1, 1 ] ]
```

48.18 MoebiusTom

`MoebiusTom(` *tom* `)`

`MoebiusTom` computes the Möbius values both of the subgroup lattice of the group with table of marks *tom* and of the poset of conjugacy classes of subgroups. It returns a record where the component `mu` contains the Möbius values of the subgroup lattice, and the component `nu` contains the Möbius values of the poset. Moreover, according to a conjecture of Isaacs et al. (see [HIÖ89], [Pah93]), the values on the poset of conjugacy classes are derived from those of the subgroup lattice. These theoretical values are returned in the component `ex`. For that computation, the derived subgroup must be known in the component `derivedSubgroup` of *tom*. The numbers of those subgroups where the theoretical value does not coincide with the actual value are returned in the component `hyp`.

```
gap> MoebiusTom( a5 );
rec(
```

```
    mu := [ -60, 4, 2,,, -1, -1, -1, 1 ],
    nu := [ -1, 2, 1,,, -1, -1, -1, 1 ],
    ex := [ -60, 4, 2,,, -1, -1, -1, 1 ],
    hyp := [  ] )
```

48.19 CyclicExtensionsTom

`CyclicExtensionsTom( tom, p )`

According to A. Dress [Dre69], two columns of the table of marks *tom* are equal modulo the prime *p* if and only if the corresponding subgroups are connected by a chain of normal extensions of order *p*. `CyclicExtensionsTom` returns the classes of this equivalence relation.

This information is not used by `NormalizerTom` although it might give additional restrictions in the search of normalizers.

```
gap> CyclicExtensionsTom( a5, 2 );
[ [ 1, 2, 4 ], [ 3, 6 ], [ 5, 7 ], [ 8 ], [ 9 ] ]
```

48.20 IdempotentsTom

`IdempotentsTom( tom )`

`IdempotentsTom` returns the list of idempotents of the integral Burnside ring described by the table of marks *tom*. According to A. Dress [Dre69], these idempotents correspond to the classes of perfect subgroups, and each such idempotent is the characteristic function of all those subgroups which arise by cyclic extension from the corresponding perfect subgroup.

```
gap> IdempotentsTom( a5 );
[ 1, 1, 1, 1, 1, 1, 1, 1, 9 ]
```

48.21 ClassTypesTom

`ClassTypesTom( tom )`

`ClassTypesTom` distinguishes isomorphism types of the classes of subgroups of the table of marks *tom* as far as this is possible. Two subgroups are clearly not isomorphic if they have different orders. Moreover, isomorphic subgroups must contain the same number of subgroups of each type.

The types are represented by numbers. `ClassTypesTom` returns a list which contains for each class of subgroups its corresponding number.

```
gap> a6 := TableOfMarks( "A6" );;
gap> ClassTypesTom( a6 );
[ 1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8, 9, 10, 11, 11, 12, 13, 13, 14, 15,
  15, 16 ]
```

48.22 ClassNamesTom

`ClassNamesTom( tom )`

`ClassNamesTom` constructs generic names for the conjugacy classes of subgroups of the table of marks *tom*.

In general, the generic name of a class of non–cyclic subgroups consists of three parts, "(*order*)", "_{*type*}", and "*letter*", and hence has the form "(*order*)_{*type*}*letter*", where *order* indicates the order of the subgroups, *type* is a number that distinguishes different types of subgroups of the same order, and *letter* is a letter which distinguishes classes of subgroups of the same type and order. The type of a subgroup is determined by the numbers of its subgroups of other types (see 48.21). This is slightly weaker than isomorphism.

The letter is omitted if there is only one class of subgroups of that order and type, and the type is omitted if there is only one class of that order. Moreover, the braces round the type are omitted if the type number has only one digit.

For classes of cyclic subgoups, the parentheses round the order and the type are omitted. Hence the most general form of their generic names is "*order letter*". Again, the letter is omitted if there is only one class of cyclic subgroups of that order.

```
gap> ClassNamesTom( a6 );
[ "1", "2", "3a", "3b", "5", "4", "(4)_2a", "(4)_2b", "(6)a", "(6)b",
  "(9)", "(10)", "(8)", "(12)a", "(12)b", "(18)", "(24)a", "(24)b",
  "(36)", "(60)a", "(60)b", "(360)" ]
```

48.23 TomCyclic

`TomCyclic( n )`

`TomCyclic` constructs the table of marks of the cyclic group of order n. A cyclic group of order n has as its subgroups for each divisor d of n a cyclic subgroup of order d. The record which is returned has an additional component `name` where for each subgroup its order is given as a string.

```
gap> c6 := TomCyclic( 6 );
rec(
  name := [ "1", "2", "3", "6" ],
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 3, 4 ] ],
  marks := [ [ 6 ], [ 3, 3 ], [ 2, 2 ], [ 1, 1, 1, 1 ] ] )
gap> DisplayTom( c6 );
1:  6
2:  3 3
3:  2 . 2
4:  1 1 1 1
```

48.24 TomDihedral

`TomDihedral( m )`

`TomDihedral` constructs the table of marks of the dihedral group of order m. For each divisor d of m, a dihedral group of order $m = 2n$ contains subgroups of order d according to the following rule. If d is odd and divides n then there is only one cyclic subgroup of order d. If d is even and divides n then there are a cyclic subgroup of order d and two classes of dihedral subgroups of order d which are cyclic, too, in the case $d = 2$, see example below). Otherwise, (i.e. if d does not divide n, there is just one class of dihedral subgroups of order d.

```
gap> d12 := TomDihedral( 12 );
```

```
rec(
  name := [ "1", "2", "D_{2}a", "D_{2}b", "3", "D_{4}", "6",
      "D_{6}a", "D_{6}b", "D_{12}" ],
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ],
      [ 1, 2, 3, 4, 6 ], [ 1, 2, 5, 7 ], [ 1, 3, 5, 8 ],
      [ 1, 4, 5, 9 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ] ],
  marks := [ [ 12 ], [ 6, 6 ], [ 6, 2 ], [ 6, 2 ], [ 4, 4 ],
      [ 3, 3, 1, 1, 1 ], [ 2, 2, 2, 2 ], [ 2, 2, 2, 2 ],
      [ 2, 2, 2, 2 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ] )
gap> DisplayTom( d12 );
 1:  12
 2:   6 6
 3:   6 . 2
 4:   6 . . 2
 5:   4 . . . 4
 6:   3 3 1 1 . 1
 7:   2 2 . . 2 . 2
 8:   2 . 2 . 2 . . 2
 9:   2 . . 2 2 . . . 2
10:   1 1 1 1 1 1 1 1 1 1
```

48.25 TomFrobenius

`TomFrobenius(` p, q `)`

`TomFrobenius` computes the table of marks of a Frobenius group of order pq, where p is a prime and q divides $p-1$.

```
gap> f20 := TomFrobenius( 5, 4 );
rec(
  name := [ "1", "2", "4", "5:1", "5:2", "5:4" ],
  subs := [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ], [ 1, 4 ], [ 1, 2, 4, 5 ],
      [ 1, 2, 3, 4, 5, 6 ] ],
  marks :=
   [ [ 20 ], [ 10, 2 ], [ 5, 1, 1 ], [ 4, 4 ], [ 2, 2, 2, 2 ],
      [ 1, 1, 1, 1, 1, 1 ] ] )
gap> DisplayTom( f20 );
1:  20
2:  10 2
3:   5 1 1
4:   4 . . 4
5:   2 2 . 2 2
6:   1 1 1 1 1 1
```

Chapter 49

Character Tables

This chapter contains

the introduction of GAP3 character tables (see 49.1, 49.2, 49.3, 49.4, 49.5, 49.6, 49.7, 49.8, 49.9) and some conventions for their usage (see 49.10),

the description how to construct or get character tables (see 49.11, 49.12; for the contents of the table library, see Chapter 53), matrix representations (see 49.25).

the description of some functions which give information about the conjugacy classes of character tables, that is, to compute classlengths (see 49.27), inverse classes (see 49.28) and classnames (see 49.29), structure constants (see 49.30, 49.31, 49.32), the set of real classes (see 49.33), orbits of the Galois group on the classes (see 49.34) and roots of classes (see 49.35),

the description how character tables or parts of them can be displayed (see 49.37) and sorted (see 49.38, 49.39, 49.40).

the description of functions which compute the automorphism group of a matrix (see 49.41) or character table (see 49.42), or which compute permutations relating permutation equivalent matrices (see 49.43) or character tables (see 49.44),

the description of functions which get fusions from and store fusions on tables (see 49.45, 49.46, 49.47),

the description of the interface between GAP3 and the MOC3 system (see 49.48, 49.49, 49.50, 49.51, 49.52, 49.53), and of a function which converts GAP3 tables to CAS tables (see 49.54).

This chapter does **not** contain information about

functions to construct characters (see Chapter 51), or functions to construct and use maps (see Chapter 52).

For some elaborate examples how character tables are handled in GAP3, see 1.25.

49.1 Some Notes on Character Theory in GAP

It seems to be necessary to state some basic facts –and maybe warnings– at the beginning of the character theory package. This holds for people who are familiar with character theory because there is no global reference on computational character theory, although there are many papers on this topic, like [NPP84] or [LP91]. It holds, however, also for people who are familiar with GAP3 because the general concept of categories and domains (see 1.23 and chapter 4) plays no important role here –we will justify this later in this section.

Intuitively, characters of the finite group G can be thought of as certain mappings defined on G, with values in the complex number field; the set of all characters of G forms a semiring with addition and multiplication both defined pointwise, which is embedded in the ring of generalized (or virtual) characters in the natural way. A $\mathbf{Z}$–basis of this ring, and also a vector space base of the vector space of class functions, is given by the irreducible characters.

At this stage one could ask where there is a problem, since all these algebraic structures are supported by GAP3, as is described in chapters 4, 5, 9, 43, and others.

Now, we first should say that characters are **not** implemented as mappings, that there are **no** GAP3 domains denoting character rings, and that a character table is **not** a domain.

For computations with characters of a finite group G with n conjugacy classes, say, we fix an order of the classes, and then identify each class with its position according to this order. Each character of G will be represented as list of length n where at the i–th position the character value for elements of the i–th class is stored. Note that we do not need to know the conjugacy classes of G physically, even our "knowledge" of the group may be implicit in the sense that e.g. we know how many classes of involutions G has, and which length these classes have, but we never have seen an element of G, or a presentation or representation of G. This allows to work with the character tables of very large groups, e.g., of the so–called monster, where GAP3 has no chance to work with the group.

As a consequence, also other information involving characters is given implicitly. For example, we can talk about the kernel of a character not as a group but as a list of classes (more exactly: a list of their positions according to the order of classes) forming this kernel; we can deduce the group order, the contained cyclic subgroups and so on, but we do not get the group itself.

Characters are one kind of class functions, and we also represent general class functions as lists. Two important kinds of these functions which are not characters are power maps and fusion maps. The k–th power map maps each class to the class of k–th powers of its elements, the corresponding list contains at each position the position of the image. A subgroup fusion map between the classes of a subgroup H of G and the classes of G maps each class c of H to that class of G that contains c; if we know only the character tables of the two groups, this means with respect to a fixed embedding of H in G.

So the data mainly consist of lists, and typical calculations with character tables are more or less loops over these lists. For example, the known scalar product of two characters χ, ψ of G given by

$$[\chi, \psi] = \frac{1}{|G|} \sum_{g \in G} \chi(g) \psi(g^{-1})$$

can be written as

```
Sum( [1..n], i -> t.classes[i]*chi[i]*GaloisCyc(psi[i],-1) );
```

where `t.classes` is the list of classlengths, and `chi`, `psi` are the lists corresponding to χ, ψ, respectively. Characters, classlengths, element orders, power maps, fusion maps and other information about a group is stored in a common character table record just to avoid confusion, not to indicate an algebraic structure (which would mean a domain in the sense of GAP3).

A character table is not determined by something similar to generators for groups or rings in GAP3 where other components (the knowledge about the domain) is stored for the sake of efficiency. In many situations one works with incomplete tables or preliminary tables which are, strictly speaking, no character tables but shall be handled like character tables. Moreover, the correctness or even the consistency of a character table is hard to prove. Thus it is not sufficient to view a character table as a black box, and to get information about it using a few property test functions. In fact there are very few functions that return character tables or that are property tests. Most GAP3 functions dealing with character tables return class functions, or lists of them, or information about class functions. For that, GAP3 directly accesses the components of the table record, and the user will have to look at the record components, too, in order to put the pieces of the puzzle together, and to decide how to go on.

So it is not easy to say what a character table **is**; it **describes** some properties of the underlying group, and it describes them in a rather abstract way. Also GAP3 does not know whether or not a list **is** a character, it will e.g. regard a list with all entries equal to 1 as the trivial character if it is passed to a function that expects characters.

It is one of the advantages of character theory that after one has translated a problem concerning groups into a problem concerning their character tables the calculations are mostly simple. For example, one can often prove that a group is a Galois group over the rationals using calculations of structure constants that can be computed from the character table, and informations on (the character tables of) maximal subgroups.

In this kind of problems the translation back to the group is just an interpretation by the user, it does not take place in GAP3. At the moment, the only interface between handling groups and handling character tables is the fixed order of conjugacy classes.

Note that algebraic structures are not of much interest in character theory. The main reason for this is that we have no homomorphisms since we need not to know anything about the group multiplication.

49.2 Character Table Records

For GAP3, a character table is any record that has the components `centralizers` and `identifier` (see 49.4).

There are three different but very similar types of character tables in GAP3, namely ordinary tables, Brauer tables and generic tables. Generic tables are described in Chapter 50. Brauer tables are defined and stored relative to ordinary tables, so they will be described in 49.3, and we start with ordinary tables.

You may store arbitrary information on an ordinary character table, but these are the only fields used by GAP3 functions:

`centralizers`
: the list of centralizer orders which should be positive integers

`identifier`
: a string that identifies the table, sometimes also called he table name; it is used for fusions (see below), programs for generic tables (see chapter 50) and for access to library tables (see 49.12, 53.1)

`order`
: the group order, a positive integer; in most cases, it is equal to `centralizers[1]`

`classes`
: the lengths of conjugacy classes, a list of positive integers

`orders`
: the list of representative orders

`powermap`
: a list where at position p, if bound, the p-th powermap is stored; the p-th powermap is a -possibly parametrized- map (see 52.1)

`fusions`
: a list of records which describe the fusions into other character tables, that is subgroup fusions and factor fusions; any record has fields `name` (the `identifier` component of the destination table) and `map` (a list of images for the classes, it may be parametrized (see 52.1)); if there are different fusions with same destination table, the field `specification` is used to distinguish them; optional fields are `type` (a string that is `"normal"` for normal subgroup fusions and `"factor"` for factor fusions) and `text` (a string with information about the fusion)

`fusionsource`
: a list of table names of those tables which contain a fusion into the actual table

`irreducibles`
: a list of irreducible characters (see below)

`irredinfo`
: a list of records with information about `irreducibles`, usual entries are `indicator`, `pblock` and `charparam` (see 51.7, 51.6, 50); if the field `irreducibles` is sorted using 49.38, the `irredinfo` field is sorted, too. So any information about `irreducibles` should be stored here.

`projectives`
: (only for ATLAS tables, see 53.3) a list of records, each with fields `name` (of the table of a covering group) and `chars` (a list of –in general not all– faithful irreducibles of the covering group)

`permutation`
: the actual permutation of the classes (see 49.10, 49.39)

`classparam`
: a list of parameter values specifying the classes of tables constructed via specialisation of a generic character table (see chapter 50)

`classtext`
: a list of additional information about the conjugacy classes (e.g. representatives of the class for matrix groups or permutation groups)

`text`
: a string containing information about the table; these are e.g. its source (see Chapter 53), the tests it has passed (`1.o.r.` for the test of orthogonality, `pow[`*p*`]` for the construction of the p-th powermap, `DEC` for the decomposition of ordinary characters in Brauer characters), and choices made without loss of generality where possible

`automorphisms`
: the permutation group of column permutations preserving the set `irreducibles` (see 49.41, 49.42)

`classnames`
: a list of names for the classes, a string each (see 49.29)

classnames
: for each entry *clname* in `classnames`, a field *tbl.clname* that has the position of *clname* in `classnames` as value (see 49.29)

`operations`
: a record with fields `Print` (see 49.37) and `ScalarProduct` (see 51.1); the default value of the `operations` field is `CharTableOps` (see 49.7)

`CAS`
: a list of records, each with fields `permchars`, `permclasses` (both permutations), `name` and eventually `text` and `classtext`; application of the two permutations to `irreducibles` and classes yields the original CAS library table with name `name` and text `text` (see 53.5)

`libinfo`
: a record with components `othernames` and perhaps `CASnames` which are all admissible names of the table (see 49.12); using these records, the list `LIBLIST.ORDINARY` can be constructed from the library using `MakeLIBLIST` (see 53.6)

`group`
: the group the table belongs to; if the table was computed using `CharTable` (see 49.12) then this component holds the group, with conjugacy classes sorted compatible with the columns of the table

Note that tables in library files may have different format (see chapter 53).

This is a typical example of a character table, first the "naked" record, then the displayed version:

```
gap> t:= CharTable( "2.A5" );; PrintCharTable( t );
rec( text := "origin: ATLAS of finite groups, tests: 1.o.r., pow[2,3,5\
]", centralizers := [ 120, 120, 4, 6, 6, 10, 10, 10, 10
 ], powermap := [ , [ 1, 1, 2, 4, 4, 8, 8, 6, 6 ],
  [ 1, 2, 3, 1, 2, 8, 9, 6, 7 ],, [ 1, 2, 3, 4, 5, 1, 2, 1, 2 ]
 ], fusions := [ rec(
      name := "A5",
      map := [ 1, 1, 2, 3, 3, 4, 4, 5, 5 ] ), rec(
      name := "2.A5.2",
      map := [ 1, 2, 3, 4, 5, 6, 7, 6, 7 ] ), rec(
      name := "2.J2",
      map := [ 1, 2, 5, 8, 9, 16, 17, 18, 19 ],
```

```
      text := [ 'f', 'u', 's', 'i', 'o', 'n', ' ', 'o', 'f', ' ',
          'm', 'a', 'x', 'i', 'm', 'a', 'l', ' ', '2', '.', 'A', '5',
          ' ', 'd', 'e', 't', 'e', 'r', 'm', 'i', 'n', 'e', 'd', ' ',
          'b', 'y', ' ', 't', 'h', 'e', ' ', '3', 'B', ' ', 'e', 'l',
          'e', 'm', 'e', 'n', 't', 's' ] ) ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 3, 3, -1, 0, 0, -E(5)-E(5)^4, -E(5)-E(5)^4, -E(5)^2-E(5)^3,
      -E(5)^2-E(5)^3 ],
  [ 3, 3, -1, 0, 0, -E(5)^2-E(5)^3, -E(5)^2-E(5)^3, -E(5)-E(5)^4,
      -E(5)-E(5)^4 ], [ 4, 4, 0, 1, 1, -1, -1, -1, -1 ],
  [ 5, 5, 1, -1, -1, 0, 0, 0, 0 ],
  [ 2, -2, 0, -1, 1, E(5)+E(5)^4, -E(5)-E(5)^4, E(5)^2+E(5)^3,
      -E(5)^2-E(5)^3 ],
  [ 2, -2, 0, -1, 1, E(5)^2+E(5)^3, -E(5)^2-E(5)^3, E(5)+E(5)^4,
      -E(5)-E(5)^4 ], [ 4, -4, 0, 1, -1, -1, 1, -1, 1 ],
  [ 6, -6, 0, 0, 0, 1, -1, 1, -1 ] ], automorphisms := Group( (6,8)
(7,9) ), construction := function ( tbl )
    ConstructProj( tbl );
end, irredinfo := [ rec(
      pblock := [ , 1, 1,, 1 ] ), rec(
      pblock := [ , 1, 2,, 1 ] ), rec(
      pblock := [ , 1, 3,, 1 ] ), rec(
      pblock := [ , 2, 1,, 1 ] ), rec(
      pblock := [ , 1, 1,, 2 ] ), rec(
      pblock := [ , 1, 4,, 3 ] ), rec(
      pblock := [ , 1, 4,, 3 ] ), rec(
      pblock := [ , 2, 4,, 3 ] ), rec(
      pblock := [ , 1, 5,, 3 ] )
 ], identifier := "2.A5", operations := CharTableOps, fusionsource :=
[ "P2/G1/L1/V1/ext2", "P2/G1/L1/V1/ext3", "P2/G2/L1/V1/ext2",
  "P2/G2/L1/V1/ext3", "P2/G2/L1/V2/ext2" ], name := "2.A5", size :=
120, order := 120, classes := [ 1, 1, 30, 20, 20, 12, 12, 12, 12
 ], orders := [ 1, 2, 4, 3, 6, 5, 10, 5, 10 ] )

gap> DisplayCharTable( t );
2.A5

     2  3  3  2  1  1   1   1   1   1
     3  1  1  .  1  1   .   .   .   .
     5  1  1  .  .  .   1   1   1   1

       1a 2a 4a 3a 6a  5a 10a  5b 10b
    2P 1a 1a 2a 3a 3a  5b  5b  5a  5a
    3P 1a 2a 4a 1a 2a  5b 10b  5a 10a
    5P 1a 2a 4a 3a 6a  1a  2a  1a  2a

X.1     1  1  1  1  1   1   1   1   1
X.2     3  3 -1  .  .   A   A  *A  *A
```

```
    X.3     3  3 -1  .  .  *A  *A   A   A
    X.4     4  4  .  1  1  -1  -1  -1  -1
    X.5     5  5  1 -1 -1   .   .   .   .
    X.6     2 -2  . -1  1  -A   A -*A  *A
    X.7     2 -2  . -1  1 -*A  *A  -A   A
    X.8     4 -4  .  1 -1  -1   1  -1   1
    X.9     6 -6  .  .  .   1  -1   1  -1

    A = -E(5)-E(5)^4
      = (1-ER(5))/2 = -b5
```

49.3 Brauer Table Records

Brauer table records are similar to the records which represent ordinary character tables. They contain many of the well–known record components, like `identifier`, `centralizers`, `irreducibles` etc.; but there are two kinds of differences:

First, the operations record is `BrauerTableOps` instead of `CharTableOps` (see 49.7). Second, there are two extra components, namely

`ordinary`, which contains the ordinary character table corresponding to the Brauer table, and

`blocks`, which reflects the **block** information; it is a list of records with components

`defect`
: the defect of the block,

`ordchars`
: a list of integers indexing the ordinary irreducibles in the block,

`modchars`
: a list of integers indexing the Brauer characters in the block,

`basicset`
: a list of integers indexing the ordinary irreducibles of a basic set; **note** that the indices refer to the positions in the whole `irreducibles` list of the ordinary table, not to the positions in the block,

`decinv`
: the inverse of the restriction of the decomposition matrix of the block to the basic set given by the `basicset` component, and possibly

`brauertree`
: if exists, a list that represents the decomposition matrix which in this case is viewed as incidence matrix of a tree (the so–called Brauer tree); the entries of the list correspond to the edges of the tree, they refer to positions in the block, not in the whole `irreducibles` list of the tables. Brauer trees are mainly used to store the information in a more compact way than by decomposition matrices, planar embeddings etc. are not (or not yet) included.

Note that Brauer tables in the library have different format (see 53.6).

We give an example:

```
gap> PrintCharTable( CharTable( "M11" ) mod 11 );
rec( identifier := "M11mod11", text := "origin: modular ATLAS of finit\
e groups, tests: DEC, TENS", prime := 11, size :=
7920, centralizers := [ 7920, 48, 18, 8, 5, 6, 8, 8 ], orders :=
[ 1, 2, 3, 4, 5, 6, 8, 8 ], classes :=
[ 1, 165, 440, 990, 1584, 1320, 990, 990 ], powermap :=
[ , [ 1, 1, 3, 2, 5, 3, 4, 4 ], [ 1, 2, 1, 4, 5, 2, 7, 8 ],,
  [ 1, 2, 3, 4, 1, 6, 8, 7 ],,,,,, [ 1, 2, 3, 4, 5, 6, 7, 8 ]
 ], fusions := [ rec(
      name := "M11",
      map := [ 1, 2, 3, 4, 5, 6, 7, 8 ],
      type := "choice" ) ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1 ], [ 9, 1, 0, 1, -1, -2, -1, -1 ],
  [ 10, -2, 1, 0, 0, 1, E(8)+E(8)^3, -E(8)-E(8)^3 ],
  [ 10, -2, 1, 0, 0, 1, -E(8)-E(8)^3, E(8)+E(8)^3 ],
  [ 11, 3, 2, -1, 1, 0, -1, -1 ], [ 16, 0, -2, 0, 1, 0, 0, 0 ],
  [ 44, 4, -1, 0, -1, 1, 0, 0 ], [ 55, -1, 1, -1, 0, -1, 1, 1 ]
 ], irredinfo := [ rec(
       ), rec(
       ), rec(
       ), rec(
       ), rec(
       ), rec(
       ), rec(
       ), rec(
       ) ], blocks := [ rec(
      defect := 1,
      ordchars := [ 1, 2, 3, 4, 6, 7, 9 ],
      modchars := [ 1, 2, 3, 4, 6 ],
      decinv :=
       [ [ 1, 0, 0, 0, 0 ], [ -1, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ],
          [ 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 1 ] ],
      basicset := [ 1, 2, 3, 4, 6 ],
      brauertree :=
       [ [ 1, 2 ], [ 2, 7 ], [ 3, 7 ], [ 4, 7 ], [ 5 .. 7 ] ] ), rec(
      defect := 0,
      ordchars := [ 5 ],
      modchars := [ 5 ],
      decinv := [ [ 1 ] ],
      basicset := [ 5 ] ), rec(
      defect := 0,
      ordchars := [ 8 ],
      modchars := [ 7 ],
      decinv := [ [ 1 ] ],
      basicset := [ 8 ] ), rec(
      defect := 0,
      ordchars := [ 10 ],
      modchars := [ 8 ],
```

```
      decinv := [ [ 1 ] ],
      basicset := [ 10 ] )
 ], ordinary := CharTable( "M11" ), operations := BrauerTableOps, orde\
r := 7920, name := "M11mod11", automorphisms := Group( (7,8) ) )
```

49.4 IsCharTable

`IsCharTable( obj )`

returns `true` if *obj* is a record with fields `centralizers` (a list) and `identifier` (a string), otherwise it returns `false`.

```
gap> IsCharTable( rec( centralizers:= [ 2,2 ], identifier:= "C2" ) );
true
```

There is one exception: If the record does not contain an `identifier` component, but a `name` component instead, then the function returns `true`. Note, however, that this exception will disappear in forthcoming GAP3 versions.

49.5 PrintCharTable

`PrintCharTable( tbl )`

prints the information stored in the character table *tbl* in a format that is GAP3 readable. The call can be used as argument of `PrintTo` in order to save the table to a file.

```
gap> t:= CharTable( "Cyclic", 3 );
CharTable( "C3" )
gap> PrintCharTable( t );
rec( identifier := "C3", name := "C3", size := 3, order :=
3, centralizers := [ 3, 3, 3 ], orders := [ 1, 3, 3 ], powermap :=
[ ,, [ 1, 1, 1 ] ], irreducibles :=
[ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ]
 ], classparam := [ [ 1, 0 ], [ 1, 1 ], [ 1, 2 ] ], irredinfo :=
[ rec(
      charparam := [ 1, 0 ] ), rec(
      charparam := [ 1, 1 ] ), rec(
      charparam := [ 1, 2 ] )
 ], text := "computed using generic character table for cyclic groups"\
, classes := [ 1, 1, 1 ], operations := CharTableOps, fusions :=
[  ], fusionsource := [  ], projections := [  ], projectionsource :=
[  ] )
```

49.6 TestCharTable

`TestCharTable( tbl )`

checks the character table *tbl*

if *tbl*`.centralizers`, *tbl*`.classes`, *tbl*`.orders` and the entries of *tbl*`.powermap` have same length,
if the product of *tbl*`.centralizers[`*i*`]` with *tbl*`.classes[`*i*`]` is equal to *tbl*`.order`,
if *tbl*`.orders[`*i*`]` divides *tbl*`.centralizers[`*i*`]`,

if the entries of *tbl*.`classnames` and the corresponding record fields are consistent,
if the first orthogonality relation for *tbl*.`irreducibles` is satisfied,
if the centralizers agree with the sums of squared absolute values of *tbl*.`irreducibles` and
if powermaps and representative orders are consistent.

If no inconsistency occurs, `true` is returned, otherwise each error is signalled, and `false` is returned at the end.

```
gap> t:= CharTable("A5");; TestCharTable(t);
true
gap> t.irreducibles[2]:= t.irreducibles[3] - t.irreducibles[1];;
gap> TestCharTable(t);
#E TestCharTable(A5): Scpr( ., X[2], X[1] ) = -1
#E TestCharTable(A5): Scpr( ., X[2], X[2] ) = 2
#E TestCharTable(A5): Scpr( ., X[3], X[2] ) = 1
#E TestCharTable(A5): centralizer orders inconsistent with irreducibles
false
```

49.7 Operations Records for Character Tables

Although a character table is not a domain (see 49.1), it needs an operations record. That for **ordinary character tables** is `CharTableOps`, that for **Brauer tables** is `BrauerTableOps`. The functions in these records are listed in section 49.8.

In the following two cases it may be useful to overlay these functions.

Character tables are printed using the `Print` component, one can for example replace the default `Print` by 49.37 `DisplayCharTable`.

Whenever a library function calls the scalar product this is the `ScalarProduct` field of the operations record, so one can replace the default function (see 51.1) by a more efficient one for special cases.

49.8 Functions for Character Tables

The following polymorphic functions are overlaid in the `operations` record of character tables. They are listed in alphabetical order.

`AbelianInvariants(` *tbl* `)`

`Agemo(` *tbl*`,` *p* `)`

`Automorphisms(` *tbl* `)`

`Centre(` *tbl* `)`

`CharacterDegrees(` *tbl* `)`

`DerivedSubgroup(` *tbl* `)`

`Display(` *tbl* `)`

`ElementaryAbelianSeries(` *tbl* `)`

`Exponent(` *tbl* `)`

`FittingSubgroup(` *tbl* `)`

`FrattiniSubgroup(` *tbl* `)`

`FusionConjugacyClasses(` *tbl1* `,` *tbl2* `)`

`Induced`

`IsAbelian(` *tbl* `)`

`IsCyclic(` *tbl* `)`

`IsNilpotent(` *tbl* `)`

`IsSimple(` *tbl* `)`

`IsSolvable(` *tbl* `)`

`IsSupersolvable(` *tbl* `)`

`LowerCentralSeries(` *tbl* `)`

`MaximalNormalSubgroups(` *tbl* `)`

`NoMessageScalarProduct(` *tbl*`,` *chi1* `,` *chi2* `)`

`NormalClosure(` *tbl*`,` *classes* `)`

`NormalSubgroups(` *tbl* `)`

`Print(` *tbl* `)`

`Restricted`

`ScalarProduct(` *tbl*`,` *chi1* `,` *chi2* `)`

`Size(` *tbl* `)`

`SizesConjugacyClasses(` *tbl* `)`

`SupersolvableResiduum(` *tbl* `)`

`UpperCentralSeries(` *tbl* `)`

49.9 Operators for Character Tables

The following operators are defined for character tables.

tbl1 `*` *tbl2*
: direct product of two character tables (see 49.17),

tbl `/` *list*
: table of the factor group modulo the classes in the list *list* (see 49.15),

tbl `mod` *p*
: *p*–modular table corresponding to *tbl* (see 49.12).

49.10 Conventions for Character Tables

The following few conventions should be noted:

The identity element is expected to be in the first class.

Characters are lists of cyclotomics (see Chapter 13) or unknowns (see chapter 17); they do not physically "belong" to a table, so when necessary, functions "regard" them as characters of a table which is given as another parameter.

Conversely , most functions that take a character table as a parameter and work with characters expect these characters as a parameter, too.

Some functions, however, expect the characters to be stored in the `irreducibles` field of the table (e.g. 49.6 `TestCharTable`) or allow application either to a list of characters given by a parameter or to the `irreducibles` field (e.g. 51.7 `Indicator`) if this parameter is missing.

The trivial character need not be the first one in a list of characters.

Sort convention: Whenever 49.39 `SortClassesCharTable` or 49.40 `SortCharTable` is used to sort the classes of a character table, the fusions into that table are **not** adjusted; only the `permutation` field of the sorted table will be actualized.

If one handles fusions only using 49.45 `GetFusionMap` and 49.46 `StoreFusion`, the maps are adjusted automatically with respect to the value of the field `permutation` of the destination table. So one should not change this field by hand. Fusion maps that are entered explicitly (e.g. because they are not stored on a table) are expected to be sorted, they will not be adjusted.

49.11 Getting Character Tables

There are in general four different ways to get a character table which GAP3 already "knows": You can either

read a file that contains the table record,
construct the table using generic formulae,
derive it from known tables or
use a presentation or representation of the group.

The first two methods are used by 49.12 `CharTable`. For the conception of generic character tables, see chapter 50. **Note** that library files often contain something that is much different from the tables returned by `CharTable`, see chapter 53. Especially see 53.2.

As for the third method, some generic ways to derive a character table are implemented:
One can obtain it as table of a factor group where the table of the group is given (see 49.15),
for given tables the table of the direct product can be constructed (see 49.17),
the restriction of a table to the p-regular classes can be formed (see 49.19),
for special cases, an isoclinic table of a given table can be constructed (see 49.20),
the splitting and fusion of classes may be viewed as a generic process (see 49.21, 49.22).

At the moment, for the last method there are algorithms dealing with arbitrary groups (see 49.12), and with finite polycyclic groups with special properties (see 49.26).

Note that whenever fusions between tables occur in these functions, they are stored on the concerned tables, and the `fusionsource` fields are updated (see 49.2).

49.12 CharTable

`CharTable(` *G* `)`
`CharTable(` *tblname* `)`
`CharTable(` *series*`,` *parameter1*`,` *parameter2* `...` `)`

`CharTable(` *G* `)`

returns the character table of the group G. If G`.name` is bound, the table is baptized the same. Otherwise it is given the identifier component `""` (empty string). This is necessary since every character table needs an identifier in GAP3 (see 49.4).

`CharTable` first computes the linear characters, using the commutator factor group. If irreducible characters are missing afterwards, they are computed using the algorithm of Dixon and Schneider (see [Dix67] and [Sch90]).

```
gap> M11 := Group((1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6));;
gap> M11.name := "M11";;
gap> PrintCharTable( CharTable( M11 ) );
rec( size := 7920, centralizers := [ 7920, 11, 11, 8, 48, 8, 8, 18,
  5, 6 ], orders := [ 1, 11, 11, 4, 2, 8, 8, 3, 5, 6 ], classes :=
[ 1, 720, 720, 990, 165, 990, 990, 440, 1584, 1320 ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 10, -1, -1, 2, 2, 0, 0, 1, 0, -1 ],
  [ 10, -1, -1, 0, -2, E(8)+E(8)^3, -E(8)-E(8)^3, 1, 0, 1 ],
  [ 10, -1, -1, 0, -2, -E(8)-E(8)^3, E(8)+E(8)^3, 1, 0, 1 ],
  [ 11, 0, 0, -1, 3, -1, -1, 2, 1, 0 ],
  [ 16, E(11)^2+E(11)^6+E(11)^7+E(11)^8+E(11)^10,
      E(11)+E(11)^3+E(11)^4+E(11)^5+E(11)^9, 0, 0, 0, 0, -2, 1, 0 ],
  [ 16, E(11)+E(11)^3+E(11)^4+E(11)^5+E(11)^9,
      E(11)^2+E(11)^6+E(11)^7+E(11)^8+E(11)^10, 0, 0, 0, 0, -2, 1, 0 ]
    , [ 44, 0, 0, 0, 4, 0, 0, -1, -1, 1 ],
  [ 45, 1, 1, 1, -3, -1, -1, 0, 0, 0 ],
  [ 55, 0, 0, -1, -1, 1, 1, 1, 0, -1 ]
 ], operations := CharTableOps, identifier := "M11", order :=
7920, name := "M11", powermap :=
[ , [ 1, 3, 2, 5, 1, 4, 4, 8, 9, 8 ], [ 1, 2, 3, 4, 5, 6, 7, 1, 9, 5 ]
   ,, [ 1, 2, 3, 4, 5, 7, 6, 8, 1, 10 ],,
  [ 1, 3, 2, 4, 5, 7, 6, 8, 9, 10 ],,,,
  [ 1, 1, 1, 4, 5, 6, 7, 8, 9, 10 ] ], galomorphisms := Group(
( 6, 7),
( 2, 3) ), text := "origin: Dixon's Algorithm", group := M11 )
```

The columns of the table will be sorted in the same order, as the classes of the group, thus allowing a bijection between group and table. If the conjugacy classes are bound in G`.conjugacyClasses` the order is not changed. Otherwise the routine itself computes the classes. One can sort them in the canonical way, using `SortClassesCharTable` (see 49.39). If an entry G`.charTable` exists the routine uses information contained in this table. This also provides a facility for entering known characters, but then the user assumes responsibility for the correctness of the characters (There is little use in providing the trivial character to the routine).

Note: The algorithm binds the record component `galomorphisms` of the character table. This is a permutation group generated by the **Galois**-morphisms only. If there is no `automorphisms` component in the table then this group is used by routines like `SubgroupFusion`.

The computation of character tables needs to identify the classes of group elements very often, so it can be helpful to store a class list of all group elements. Since this is obviously limited by the group size, it is controlled by the global variable `LARGEGROUPORDER`, which is set by standard to 10000. If the group is smaller, the class map is stored. Otherwise each occuring element is identified individually.

Limitations: At the moment there is a limitation to the group size given by the following condition: the routine computes in a prime field of size p. p is a prime number, such that the exponent of the group divides $(p-1)$ and such that $2\sqrt{|G|} < p$. At the moment, GAP3 provides only prime fields up to size 65535.

The routine also sets up a component G`.dixon`. Using this component, routines that identify classes, for example `FusionConjugacyClasses`, will work much faster. When interrupting the algorithm, however, a neccessary cleanup has not taken place. Thus you should call `Unbind(` G`.dixon )` to avoid possible further confusion. This is also a good idea because G`.dixon` may become very large. When the computation by `CharTable` is complete, this record is shrunk to an acceptable size, something that could not be done when interrupting.

`CharTable(` *tblname* `)`

If the only parameter is a string *tblname* and this is an admissible name of a library table, `CharTable` returns this library table, otherwise `false`. A call of `CharTable` may cause to read some library files and to construct the table from the data in the files, see chapter 53 for the details.

Admissible names for the **ordinary character table** *tbl* of the group *grp* are

- the ATLAS name if *tbl* is an ATLAS table (see 53.3), e.g., `M22` for the table of the Mathieu group M_{22}, `L2(13)` for $L_2(13)$ and `12_1.U4(3).2_1` for $12_1.U_4(3).2_1$,

- the names that were admissible for tables of *grp* in CAS if the CAS table library contained a table of *grp*, e.g., `sl42` for the table of the alternating group A_8 (but note that the table may be different from that in CAS, see 53.5) and

- some "relative" names:

  For *grp* the n–th maximal subgroup (in decreasing group order) of a sporadic simple group with admissible name *name*, *name*Mn is admissible for *tbl*, e.g., `J3M2` for the second maximal subgroup of the Janko group J_3 which has the name `J3`.

  For *grp* a nontrivial Sylow normalizer of a sporadic simple group with admissible name *name*, where nontrivial means that the group is not contained in p:$(p-1)$, *name*Np is an admissible name of *tbl*, e.g., $J4N11$ for the Sylow 11 normalizer of the Janko group J_4.

  In a few cases, the table of the Sylow p subgroup of *grp* is accessible by *name*Syl$p$ where *name* is an admissible name of the table of *grp*, e.g., `A11Syl2` for the Sylow 2 subgroup of the alternating group A_{11}.

In a few cases, the table of an element centralizer of *grp* is accessible by *name*C*cl* where *name* is an admissible name of the table of *grp*, e.g., `M11C2` for an involution centralizer in the Mathieu group M_{11}.

Admissible names for a **Brauer table** *tbl* (modulo the prime p) are all names *name*`mod`p where *name* is admissible for the corresponding ordinary table, e.g., `M12mod11` for the 11 modular table of M_{12}, and `L2(25).2_1mod3` for the 3 modular table of $L_2(25).2_1$. Brauer tables in the library can be got also from the underlying ordinary table using the `mod` operator, as in the following example.

```
gap> CharTable( "A5" ) mod 2;
CharTable( "A5mod2" )
```

Generic tables are accessible only by the name given by their `identifier` component (see below).

Case is not significant for table names, e.g., `suzm3` and `SuzM3` are both admissible names for the third maximal subgroup of the sporadic Suzuki group.

The admissible names reflect the structure of the libraries, see 53.1 and 53.6.

```
gap> CharTable( "A5.2" );;   # returns the character table of the
                             # symmetric group on five letters
                             # (in ATLAS format)
gap> CharTable( "Symmetric" );;   # returns the generic table of the
                                  # symmetric group
gap> CharTable( "J5" );
#E CharTableLibrary: no library table with name 'J5'
false
```

If `CharTable` is called with more than one parameter, the first must be a string specifying a series of groups which is implemented via a generic character table (see chapter 50), e.g. `"Symmetric"` for the symmetric groups; the following parameters specialise the required member of the series:

```
gap> CharTable( "Symmetric", 5 );;  # the table of the symmetric
                                    # group S_5 (got by specializing
                                    # the generic table)
```

These are the valid calls of `CharTable` with parameter *series*:

`CharTable( "Alternating",` n `)`
returns the table of the alternating group on n letters,

`CharTable( "Cyclic",` n `)`
returns the table of the cyclic group of order n,

`CharTable( "Dihedral",` $2n$ `)`
returns the table of the dihedral group of order $2n$,

`CharTable( "GL", 2,` q `)`
returns the table of the general linear group $\mathrm{GL}(2, q)$ for a prime power q,

`CharTable( "GU", 3,` q `)`
returns the table of the general unitary group $\mathrm{GU}(3, q)$ for a prime power q,

`CharTable( "P:Q", [` *p*`,` *q* `] )`
returns the table of the extension of the cyclic group of prime order p by a cyclic group of order q where q divides $p-1$,

`CharTable( "PSL", 2,` *q* `)`
returns the table of the projective special linear group $\mathrm{PSL}(2,q)$ for a prime power q,

`CharTable( "SL", 2,` *q* `)`
returns the table of the special linear group $\mathrm{SL}(2,q)$ for a prime power q,

`CharTable( "SU", 3,` *q* `)`
returns the table of the special unitary group $\mathrm{SU}(3,q)$ for a prime power q,

`CharTable( "Quaternionic",` *4n* `)`
returns the table of the quaternionic (dicyclic) group of order $4n$,

`CharTable( "Suzuki",` *q* `)`
returns the table of the Suzuki group $Sz(q) =^2 B_2(q)$ for q an odd power of 2,

`CharTable( "Symmetric",` *n* `)`
returns the table of the symmetric group on n letters.

`CharTable( "WeylB",` *n* `)`
returns the table of the Weyl group of type B_n.

`CharTable( "WeylD",` *n* `)`
returns the table of the Weyl group of type D_n.

49.13 Advanced Methods for Dixon Schneider Calculations

The computation of character tables of very large groups may take quite some time. On the other hand, for the expert only a few irreducible characters may be needed, since the other ones can be computed using character theoretic methods like tensoring, induction, and restriction. Thus GAP3 provides also step-by-step routines for doing the calculations, that will allow to compute some characters, and stop before all are calculated. Note that there is no 'safety net', i.e., the routines, being somehow internal, do no error checking, and assume the information given are correct.

When the global variable `InfoCharTable1` if set to `Print`, information about the progress of splitting is printed. The default value of `InfoCharTable1` is `Ignore`.

`DixonInit(` *G* `)`

does the setup for the computation of characters: It computes conjugacy classes, power maps and linear characters (in the case of AgGroups it also contains a call of `CharTablePGroup`). `DixonInit` returns a special record D (see below), which stores all informations needed for the further computations. The power maps are computed for all primes smaller than the exponent of G, thus allowing to induce the characters of all cyclic subgroups by `InducedCyclic` (see 51.23). For internal purposes, the algorithm uses a permuted arrangement of the classes and probably a different —but isomorphic— group. It is possible to obtain different informations about the progress of the splitting process as well as the partially computed character table from the record D.

`DixontinI(` *D* `)`

is the reverse function: It takes a Dixon record D and returns the old group G. It also does the cleanup of D. The returned group contains the component `charTable`, containing the character table as far as known. The classes are arranged in the same way, as the classes of G.

`DixonSplit(` *D* `)`

will do the main splitting task: It chooses a class and splits the character spaces using the corresponding class matrix. Characters are computed as far as possible.

`CombinatoricSplit(` *D* `)`

tries to split two-dimensional character spaces by combinatoric means. It is called automatically by `DixonSplit`. A separate call can be useful, when new characters have been found, that reduce the size of the character spaces.

`IncludeIrreducibles(` *D*`,` *list* `)`

If you have found irreducible characters by other means —like tensoring etc.— you must not include them in the character table yourself, but let them include, using this routine. Otherwise GAP3 would lose control of the characters yet known. The characters given in *list* must be according to the arrangement of classes in D. GAP3 will automatically take the closure of *list* under the galoisgroup and tensor products with one-dimensional characters.

`SplitCharacters(` *D*`,` *list* `)`

This routine decomposes the characters, given in *list* according to the character spaces found up to this point. By applying this routine to tensor products etc., it may result in characters with smaller norm, even irreducible ones. Since the recalculation of characters is only possible, if the degree is small enough, the splitting process is applied only to characters of sufficiently small degree.

Some notes on the record D returned by `DixonInit`:

This record stores several items of mainly internal interest. There are some entries, however, that may be useful to know about when using the advanced methods described above. The computation need not to take place in the original group, but in an isomorphic image W. This may be the same group as the group given, but — depending on the group — also a new one. Additionally the initialisation process will create a new list of the conjugacy classes with possibly different arrangement. For access to these informations, the following record components of the "Dixon Record" D might be of interest:

`group`
: the group W,

`oldG`
: the group G, of which the character table is to be computed,

`conjugacyClasses`
: classes of W; this list contains the **same** classes as W`.conjugacyClasses`, only the arrangement is different,

`charTable`
: contains the partially computed character table. The classes are arranged according to D`.conjugacyClasses`,

`classPermutation`
: permutation to apply to the classes to obtain the old arrangement.

49.14 An Example of Advanced Dixon Schneider Calculations

First, we set

```
gap> InfoCharTable1 := Print;;
```

for printout of some internal results. We now define our group, which is isomorphic to $PSL_4(3)$ (we use a permutation representation of $PSL_4(3)$ instead of matrices since this will speed up the computations).

```
gap> g := PrimitiveGroup(40,5);
PSL(4,3)
gap> Size(g);
6065280
gap> d := DixonInit(g);;
#I  29 classes
gap> c := d.charTable;;
```

After the initialisation, one structure matrix is evaluated, yielding smaller spaces and several irreducible characters.

```
gap> DixonSplit(d);
#I  Matrix 2, Representative of Order 3, Centralizer: 5832
#I  Dimensions: [ 1, 12, 2, 2, 4, 2, 1, 1, 1, 1, 1 ]
#I  Two-dim space split
#I  Two-dim space split
#I  Two-dim space split
```

In this case spaces of the listed dimensions are a result of the splitting process. The three two dimensional spaces are split successfully by combinatoric means.

We obtain several characters by tensor products and notify them to the program. The tensor products of the nonlinear characters are reduced with the irreducible characters. The result is split according to the spaces found, which yields characters of smaller norms, but no new irreducibles.

```
gap> asp:= AntiSymmetricParts( c, c.irreducibles, 2 );;
gap> ro:= ReducedOrdinary( c, c.irreducibles, asp );;
gap> Length( ro.irreducibles );
3
gap> IncludeIrreducibles( d, ro.irreducibles );
gap> nlc:= Filtered( c.irreducibles, i -> i[1] > 1 );;
gap> t:= Tensored( nlc, nlc );;
gap> ro:= ReducedOrdinary( c, c.irreducibles, t );; ro.irreducibles;
[  ]
gap> List( ro.remainders, i -> ScalarProduct( c, i, i) );
[ 2, 2, 4, 4, 4, 4, 13, 13, 18, 18, 19, 21, 21, 36, 36, 29, 34, 34,
  42, 34, 48, 54, 62, 68, 68, 78, 84, 84, 88, 90, 159, 169, 169, 172,
  172, 266, 271, 271, 268, 274, 274, 280, 328, 373, 373, 456, 532,
  576, 679, 683, 683, 754, 768, 768, 890, 912, 962, 1453, 1453, 1601,
  1601, 1728, 1739, 1739, 1802, 2058, 2379, 2414, 2543, 2744, 2744,
```

```
  2920, 3078, 3078, 4275, 4275, 4494, 4760, 5112, 5115, 5115, 5414,
  6080, 6318, 7100, 7369, 7369, 7798, 8644, 10392, 12373, 12922,
  14122, 14122, 18948, 21886, 24641, 24641, 25056, 38942, 44950,
  78778 ]
gap> t := SplitCharacters( d, ro.remainders );;
gap> List( t, i -> ScalarProduct( c, i, i ) );
[ 2, 2, 4, 2, 2, 4, 4, 3, 6, 5, 5, 9, 9, 4, 12, 13, 18, 18, 18, 26,
  32, 32, 16, 42, 36, 84, 84, 88, 90, 159, 169, 169, 172, 172, 266,
  271, 271, 268, 274, 274, 280, 328, 373, 373, 456, 532, 576, 679,
  683, 683, 754, 768, 768, 890, 912, 962, 1453, 1453, 1601, 1601,
  1728, 1739, 1739, 1802, 2058, 2379, 2414, 2543, 2744, 2744, 2920,
  3078, 3078, 4275, 4275, 4494, 4760, 5112, 5115, 5115, 5414, 6080,
  6318, 7100, 7369, 7369, 7798, 8644, 10392, 12373, 12922, 14122,
  14122, 18948, 21886, 24641, 24641, 25056, 38942, 44950, 78778 ]
```

Finally we calculate the characters induced from all cyclic subgroups and obtain the missing irreducibles by applying the LLL-algorithm to them.

```
gap> ic:= InducedCyclic( c, "all" );;
gap> ro:= ReducedOrdinary( c, c.irreducibles, ic );;
gap> Length( ro.irreducibles );
0
gap> l:= LLL( c, ro.remainders );;
gap> Length( l.irreducibles );
13
gap> IncludeIrreducibles( d, l.irreducibles );
gap> Length( c.irreducibles );
29
gap> Length( c.classes );
29
```

As the last step, we return to our original group.

```
gap> g:= DixontinI( d );
#I  Total:1 matrices, [ 2 ]
PSL(4,3)
gap> c:= g.charTable;;
gap> List( c.irreducibles, i -> i[1] );
[ 1, 26, 26, 39, 52, 65, 65, 90, 234, 234, 260, 260, 260, 351, 390,
  416, 416, 416, 416, 468, 585, 585, 640, 640, 640, 640, 729, 780,
  1040 ]
gap> Sum( last, i -> i^2 );
6065280
```

49.15 CharTableFactorGroup

`CharTableFactorGroup(` *tbl*`,` *classes_of_normal_subgroup* `)`

returns the table of the factor group of *tbl* with respect to a particular normal subgroup: If the list of irreducibles stored in *tbl*`.irreducibles` is complete, this normal subgroup is the normal closure of *classes_of_normal_subgroup*; otherwise it is the intersection of kernels of those irreducibles stored on *tbl* which contain *classes_of_normal_subgroups* in their kernel –that may cause strange results.

```
gap> s4:= CharTable( "Symmetric", 4 );;
gap> PrintCharTable( CharTableFactorGroup( s4, [ 3 ] ) );
rec( size := 6, identifier := "S4/[ 3 ]", order :=
6, name := "S4/[ 3 ]", centralizers := [ 6, 2, 3 ], powermap :=
[ , [ 1, 1, 3 ], [ 1, 2, 1 ] ], fusions := [  ], fusionsource :=
[ "S4" ], irreducibles := [ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ]
 ], orders := [ 1, 2, 3 ], classes :=
[ 1, 3, 2 ], operations := CharTableOps )
gap> s4.fusions;
[ rec(
      map := [ 1, 2, 1, 3, 2 ],
      type := "factor",
      name := "S4/[ 3 ]" ) ]
```

49.16 CharTableNormalSubgroup

`CharTableNormalSubgroup(` *tbl*, *normal_subgroup* `)`

returns the restriction of the character table *tbl* to the classes in the list *normal_subgroup*. This table is an approximation of the character table of this normal subgroup. It has components `order`, `identifier`, `centralizers`, `orders`, `classes`, `powermap`, `irreducibles` (contains the set of those restrictions of irreducibles of *tbl* which are irreducible), and `fusions` (contains the fusion in *tbl*).

In most cases, some classes of the normal subgroup must be split, see 49.21.

```
gap> s5:= CharTable( "A5.2" );;
gap> s3:= CharTable( "Symmetric", 3 );;
gap> SortCharactersCharTable( s3 );;
gap> s5xs3:= CharTableDirectProduct( s5, s3 );;
gap> nsg:= [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ];;
gap> sub:= CharTableNormalSubgroup( s5xs3, nsg );;
#I CharTableNormalSubgroup: classes in [ 8 ] necessarily split
gap> PrintCharTable( sub );
rec( identifier := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 2\
0 ])", size :=
360, name := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ])",\
 order := 360, centralizers := [ 360, 180, 24, 12, 18, 9, 15, 15/2,
  12, 4, 6 ], orders := [ 1, 3, 2, 6, 3, 3, 5, 15, 2, 4, 6
 ], powermap := [ , [ 1, 2, 1, 2, 5, 6, 7, 8, 1, 3, 5 ],
  [ 1, 1, 3, 3, 1, 1, 7, 7, 9, 10, 9 ],,
  [ 1, 2, 3, 4, 5, 6, 1, 2, 9, 10, 11 ] ], classes :=
[ 1, 2, 15, 30, 20, 40, 24, 48, 30, 90, 60
 ], operations := CharTableOps, irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 2, -1, 2, -1, 2, -1, 2, -1, 0, 0, 0 ],
  [ 6, 6, -2, -2, 0, 0, 1, 1, 0, 0, 0 ],
```

```
    [ 4, 4, 0, 0, 1, 1, -1, -1, 2, 0, -1 ],
    [ 4, 4, 0, 0, 1, 1, -1, -1, -2, 0, 1 ],
    [ 8, -4, 0, 0, 2, -1, -2, 1, 0, 0, 0 ],
    [ 5, 5, 1, 1, -1, -1, 0, 0, 1, -1, 1 ],
    [ 5, 5, 1, 1, -1, -1, 0, 0, -1, 1, -1 ],
    [ 10, -5, 2, -1, -2, 1, 0, 0, 0, 0, 0 ] ], fusions := [ rec(
        name := [ 'A', '5', '.', '2', 'x', 'S', '3' ],
        map := [ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ] ) ] )
```

49.17 CharTableDirectProduct

`CharTableDirectProduct(` *tbl1* `,` *tbl2* `)`

returns the character table of the direct product of the groups given by the character tables *tbl1* and *tbl2*.

The matrix of irreducibles is the Kronecker product (see 34.6) of *tbl1*`.irreducibles` with *tbl2*`.irreducibles`.

```
gap> c2:= CharTable( "Cyclic", 2 );; s2:= CharTable( "Symmetric", 2 );;
gap> SortCharactersCharTable( s2 );;
gap> v4:= CharTableDirectProduct( c2, s2 );;
gap> PrintCharTable( v4 );
rec( size := 4, identifier := "C2xS2", centralizers :=
[ 4, 4, 4, 4 ], order := 4, name := "C2xS2", classparam :=
[ [ [ 1, 0 ], [ 1, [ 1, 1 ] ] ], [ [ 1, 0 ], [ 1, [ 2 ] ] ],
  [ [ 1, 1 ], [ 1, [ 1, 1 ] ] ], [ [ 1, 1 ], [ 1, [ 2 ] ] ]
 ], orders := [ 1, 2, 2, 2 ], powermap := [ , [ 1, 1, 1, 1 ]
 ], irreducibles := [ [ 1, 1, 1, 1 ], [ 1, -1, 1, -1 ],
  [ 1, 1, -1, -1 ], [ 1, -1, -1, 1 ] ], irredinfo := [ rec(
      charparam := [ [ 1, 0 ], [ 1, [ 2 ] ] ] ), rec(
      charparam := [ [ 1, 0 ], [ 1, [ 1, 1 ] ] ] ), rec(
      charparam := [ [ 1, 1 ], [ 1, [ 2 ] ] ] ), rec(
      charparam := [ [ 1, 1 ], [ 1, [ 1, 1 ] ] ] ) ], charparam :=
[  ], fusionsource := [ [ 'C', '2' ], "S2" ], fusions := [ rec(
      name := [ 'C', '2' ],
      map := [ 1, 1, 2, 2 ],
      type := "factor" ), rec(
      name := "S2",
      map := [ 1, 2, 1, 2 ],
      type := "factor" ) ], classes :=
[ 1, 1, 1, 1 ], operations := CharTableOps )
gap> c2.fusions;
[ rec(
      map := [ 1, 3 ],
      type := "normal",
      name := "C2xS2" ) ]
```

Note: The result will contain those p-th powermaps for primes p where both *tbl1* and *tbl2* contain the p-th powermap. Additionally, if one of the tables contains it, and p does not divide the order of the other table, and the p-th powermap is uniquely determined (see 52.12), it will be computed; then the table of the direct product will contain the p-th powermap, too.

49.18 CharTableWreathSymmetric

`CharTableWreathSymmetric(` *tbl*, *n* `)`

returns the character table of the wreath product of an arbitrary group G with the full symmetric group S_n, where *tbl* is the character table of G.

```
gap> c3:= CharTable("Cyclic", 3);;
gap> wr:= CharTableWreathSymmetric(c3, 2);;
gap> PrintCharTable( wr );
rec( size := 18, identifier := "C3wrS2", centralizers :=
[ 18, 9, 9, 18, 9, 18, 6, 6, 6 ], classes :=
[ 1, 2, 2, 1, 2, 1, 3, 3, 3 ], orders := [ 1, 3, 3, 3, 3, 3, 2, 6, 6
 ], irredinfo := [ rec(
      charparam := [ [ 1, 1 ], [  ], [  ] ] ), rec(
      charparam := [ [ 1 ], [ 1 ], [  ] ] ), rec(
      charparam := [ [ 1 ], [  ], [ 1 ] ] ), rec(
      charparam := [ [  ], [ 1, 1 ], [  ] ] ), rec(
      charparam := [ [  ], [ 1 ], [ 1 ] ] ), rec(
      charparam := [ [  ], [  ], [ 1, 1 ] ] ), rec(
      charparam := [ [ 2 ], [  ], [  ] ] ), rec(
      charparam := [ [  ], [ 2 ], [  ] ] ), rec(
      charparam := [ [  ], [  ], [ 2 ] ] )
 ], name := "C3wrS2", order := 18, classparam :=
[ [ [ 1, 1 ], [  ], [  ] ], [ [ 1 ], [ 1 ], [  ] ],
  [ [ 1 ], [  ], [ 1 ] ], [ [  ], [ 1, 1 ], [  ] ],
  [ [  ], [ 1 ], [ 1 ] ], [ [  ], [  ], [ 1, 1 ] ],
  [ [ 2 ], [  ], [  ] ], [ [  ], [ 2 ], [  ] ], [ [  ], [  ], [ 2 ] ]
 ], powermap := [ , [ 1, 3, 2, 6, 5, 4, 1, 4, 6 ],
  [ 1, 1, 1, 1, 1, 1, 7, 7, 7 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 2, -E(3)^2, -E(3), 2*E(3), -1, 2*E(3)^2, 0, 0, 0 ],
  [ 2, -E(3), -E(3)^2, 2*E(3)^2, -1, 2*E(3), 0, 0, 0 ],
  [ 1, E(3), E(3)^2, E(3)^2, 1, E(3), -1, -E(3), -E(3)^2 ],
  [ 2, -1, -1, 2, -1, 2, 0, 0, 0 ],
  [ 1, E(3)^2, E(3), E(3), 1, E(3)^2, -1, -E(3)^2, -E(3) ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, E(3), E(3)^2, E(3)^2, 1, E(3), 1, E(3), E(3)^2 ],
  [ 1, E(3)^2, E(3), E(3), 1, E(3)^2, 1, E(3)^2, E(3) ]
 ], operations := CharTableOps )

gap> DisplayCharTable( wr );
C3wrS2

     2  1   .   .   1  .   1  1   1   1
     3  2   2   2   2  2   2  1   1   1
```

```
        1a  3a  3b  3c 3d  3e 2a  6a  6b
     2P 1a  3b  3a  3e 3d  3c 1a  3c  3e
     3P 1a  1a  1a  1a 1a  1a 2a  2a  2a

X.1      1   1   1   1  1   1 -1  -1  -1
X.2      2   A  /A   B -1  /B  .   .   .
X.3      2  /A   A  /B -1   B  .   .   .
X.4      1 -/A  -A  -A  1 -/A -1  /A   A
X.5      2  -1  -1   2 -1   2  .   .   .
X.6      1  -A -/A -/A  1  -A -1   A  /A
X.7      1   1   1   1  1   1  1   1   1
X.8      1 -/A  -A  -A  1 -/A  1 -/A  -A
X.9      1  -A -/A -/A  1  -A  1  -A -/A

A = -E(3)^2
  = (1+ER(-3))/2 = 1+b3
B = 2*E(3)
  = -1+ER(-3) = 2b3
```

The record component `classparam` contains the sequences of partitions that parametrize the classes as well as the characters of the wreath product. Note that this parametrization prevents the principal character from being the first one in the list `irreducibles`.

49.19 CharTableRegular

`CharTableRegular(` *tbl*, *prime* `)`

returns the character table consisting of the *prime*-regular classes of the character table *tbl*.

```
gap> a5:= CharTable( "Alternating", 5 );;
gap> PrintCharTable( CharTableRegular( a5, 2 ) );
rec( identifier := "Regular(A5,2)", prime := 2, size := 60, orders :=
[ 1, 3, 5, 5 ], centralizers := [ 60, 3, 5, 5 ], powermap :=
[ , [ 1, 2, 4, 3 ], [ 1, 1, 4, 3 ],, [ 1, 2, 1, 1 ] ], fusions :=
[ rec(
      map := [ 1, 3, 4, 5 ],
      type := "choice",
      name := "A5" )
 ], ordinary := CharTable( "A5" ), operations := CharTableOps, order :\
= 60, name := "Regular(A5,2)", classes := [ 1, 20, 12, 12 ] )
gap> a5.fusionsource;
[ "Regular(A5,2)" ]
```

49.20 CharTableIsoclinic

`CharTableIsoclinic(` *tbl* `)`
`CharTableIsoclinic(` *tbl*, *classes_of_normal_subgroup* `)`

If *tbl* is a character table of a group with structure $2.G.2$ with a unique central subgroup of order 2 and a unique subgroup of index 2, `CharTableIsoclinic( tbl )` returns the table of the isoclinic group (see [CCN+85, Chapter 6, Section 7]); if the subgroup of index 2 is not unique, it must be specified by enumeration of its classes in *classes_of_normal_subgroup*.

```
gap> d8:= CharTable( "Dihedral", 8 );;
gap> PrintCharTable( CharTableIsoclinic( d8, [ 1, 2, 3 ] ) );
rec( identifier := "Isoclinic(D8)", size := 8, centralizers :=
[ 8, 4, 8, 4, 4 ], classes := [ 1, 2, 1, 2, 2 ], orders :=
[ 1, 4, 2, 4, 4 ], fusions := [  ], fusionsource := [  ], powermap :=
[ , [ 1, 3, 1, 3, 3 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, -1, -1 ], [ 1, -1, 1, 1, -1 ],
  [ 1, -1, 1, -1, 1 ], [ 2, 0, -2, 0, 0 ]
 ], operations := CharTableOps, order := 8, name := "Isoclinic(D8)" )
```

49.21 CharTableSplitClasses

`CharTableSplitClasses( tbl, fusionmap )`
`CharTableSplitClasses( tbl, fusionmap, exponent )`

returns a character table where the classes of the character table *tbl* are split according to the fusion map *fusionmap*.

The two forms correspond to the two different situations to split classes:

`CharTableSplitClasses( tbl, fusionmap )`

If one constructs a normal subgroup (see 49.16), the order remains unchanged, powermaps, classlengths and centralizer orders are changed with respect to the fusion, representative orders and irreducibles are simply split. The "factor fusion" *fusionmap* to *tbl* is stored on the result.

```
#  see example in 49.16
gap> split:= CharTableSplitClasses(sub,[1,2,3,4,5,6,7,8,8,9,10,11]);;
gap> PrintCharTable( split );
rec( identifier := "Split(Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14,\
 17, 20 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ])", size :=
360, order :=
360, name := "Split(Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 2\
0 ]),[ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ])", centralizers :=
[ 360, 180, 24, 12, 18, 9, 15, 15, 15, 12, 4, 6 ], classes :=
[ 1, 2, 15, 30, 20, 40, 24, 24, 24, 30, 90, 60 ], orders :=
[ 1, 3, 2, 6, 3, 3, 5, 15, 15, 2, 4, 6 ], powermap :=
[ , [ 1, 2, 1, 2, 5, 6, 7, [ 8, 9 ], [ 8, 9 ], 1, 3, 5 ],
  [ 1, 1, 3, 3, 1, 1, 7, 7, 7, 10, 11, 10 ],,
  [ 1, 2, 3, 4, 5, 6, 1, 2, 2, 10, 11, 12 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 2, -1, 2, -1, 2, -1, 2, -1, -1, 0, 0, 0 ],
  [ 6, 6, -2, -2, 0, 0, 1, 1, 1, 0, 0, 0 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, -1, 2, 0, -1 ],
  [ 4, 4, 0, 0, 1, 1, -1, -1, -1, -2, 0, 1 ],
```

```
    [ 8, -4, 0, 0, 2, -1, -2, 1, 1, 0, 0, 0 ],
    [ 5, 5, 1, 1, -1, -1, 0, 0, 0, 1, -1, 1 ],
    [ 5, 5, 1, 1, -1, -1, 0, 0, 0, -1, 1, -1 ],
    [ 10, -5, 2, -1, -2, 1, 0, 0, 0, 0, 0, 0 ] ], fusions := [ rec(
        name := "Rest(A5.2xS3,[ 1, 3, 4, 6, 7, 9, 10, 12, 14, 17, 20 ])",
        map := [ 1, 2, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11 ] )
   ], operations := CharTableOps )
  gap> #  the table of (3 x A5):2 (incomplete)
```

`CharTableSplitClasses(` *tbl*, *fusionmap*, *exponent* `)`

To construct a downward extension is somewhat more complicated, since the new order, representative orders, centralizer orders and classlengths are not known at the moment when the classes are split. So the order remains unchanged, centralizer orders will just be split, classlengths are divided by the number of image classes, and the representative orders become parametrized with respect to the exponent *exponent* of the normal subgroup. Power maps and irreducibles are computed from *tbl* and *fusionmap*, and the factor fusion *fusionmap* to *tbl* is stored on the result.

```
gap> a5:= CharTable( "Alternating", 5 );;
gap> CharTableSplitClasses( a5, [ 1, 1, 2, 3, 3, 4, 4, 5, 5 ], 2 );;
gap> PrintCharTable( last );
rec( identifier := "Split(A5,[ 1, 1, 2, 3, 3, 4, 4, 5, 5 ])", size :=
60, order :=
60, name := "Split(A5,[ 1, 1, 2, 3, 3, 4, 4, 5, 5 ])", centralizers :=\
 [ 60, 60, 4, 3, 3, 5, 5, 5, 5 ], classes :=
[ 1/2, 1/2, 15, 10, 10, 6, 6, 6, 6 ], orders :=
[ 1, 2, [ 2, 4 ], [ 3, 6 ], [ 3, 6 ], [ 5, 10 ], [ 5, 10 ],
  [ 5, 10 ], [ 5, 10 ] ], powermap :=
[ , [ 1, 1, [ 1, 2 ], [ 4, 5 ], [ 4, 5 ], [ 8, 9 ], [ 8, 9 ],
      [ 6, 7 ], [ 6, 7 ] ],
  [ 1, 2, 3, [ 1, 2 ], [ 1, 2 ], [ 8, 9 ], [ 8, 9 ], [ 6, 7 ],
      [ 6, 7 ] ],,
  [ 1, 2, 3, [ 4, 5 ], [ 4, 5 ], [ 1, 2 ], [ 1, 2 ], [ 1, 2 ],
      [ 1, 2 ] ] ], irreducibles := [ [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 4, 4, 0, 1, 1, -1, -1, -1, -1 ], [ 5, 5, 1, -1, -1, 0, 0, 0, 0 ],
  [ 3, 3, -1, 0, 0, -E(5)-E(5)^4, -E(5)-E(5)^4, -E(5)^2-E(5)^3,
      -E(5)^2-E(5)^3 ],
  [ 3, 3, -1, 0, 0, -E(5)^2-E(5)^3, -E(5)^2-E(5)^3, -E(5)-E(5)^4,
      -E(5)-E(5)^4 ] ], fusions := [ rec(
      name := "A5",
      map := [ 1, 1, 2, 3, 3, 4, 4, 5, 5 ] )
 ], operations := CharTableOps )
```

Note that powermaps (and in the second case also the representative orders) may become parametrized maps (see Chapter 52).

The inverse process of splitting is the fusion of classes, see 49.22.

49.22 CharTableCollapsedClasses

`CharTableCollapsedClasses(` *tbl*, *fusionmap* `)`

returns a character table where all classes of the character table *tbl* with equal images under the map *fusionmap* are collapsed; the fields `orders`, `classes`, and the characters in `irreducibles` are the images under *fusionmap*, the powermaps are obtained on conjugation (see 52.9) with *fusionmap*, `order` remains unchanged, and `centralizers` arise from `classes` and `order`.

The fusion to the returned table is stored on *tbl*.

```
gap> c3:= CharTable( "Cyclic", 3 );;
gap> t:= CharTableSplitClasses( c3, [ 1, 2, 2, 3, 3 ] );;
gap> PrintCharTable( t );
rec( identifier := "Split(C3,[ 1, 2, 2, 3, 3 ])", size := 3, order :=
3, name := "Split(C3,[ 1, 2, 2, 3, 3 ])", centralizers :=
[ 3, 6, 6, 6, 6 ], classes := [ 1, 1/2, 1/2, 1/2, 1/2 ], orders :=
[ 1, 3, 3, 3, 3 ], powermap := [ ,, [ 1, 1, 1, 1, 1 ]
 ], irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, E(3), E(3), E(3)^2, E(3)^2 ],
  [ 1, E(3)^2, E(3)^2, E(3), E(3) ] ], fusions := [ rec(
      name := [ 'C', '3' ],
      map := [ 1, 2, 2, 3, 3 ] ) ], operations := CharTableOps )
gap> c:= CharTableCollapsedClasses( t, [ 1, 2, 2, 3, 3 ] );;
gap> PrintCharTable( c );
rec( identifier := "Collapsed(Split(C3,[ 1, 2, 2, 3, 3 ]),[ 1, 2, 2, 3\
, 3 ])", size := 3, order :=
3, name := "Collapsed(Split(C3,[ 1, 2, 2, 3, 3 ]),[ 1, 2, 2, 3, 3 ])",\
 centralizers := [ 3, 3, 3 ], orders := [ 1, 3, 3 ], powermap :=
[ ,, [ 1, 1, 1 ] ], fusionsource := [ "Split(C3,[ 1, 2, 2, 3, 3 ])"
 ], irreducibles := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ],
  [ 1, E(3)^2, E(3) ] ], classes :=
[ 1, 1, 1 ], operations := CharTableOps )
```

The inverse process of fusion is the splitting of classes, see 49.21.

49.23 CharDegAgGroup

`CharDegAgGroup(` G `[,` q `] )`

`CharDegAgGroup` computes the degrees of irreducible characters of the finite polycyclic group G over the algebraic closed field of characteristic q. The default value for q is zero. The degrees are returned as a list of pairs, the first entry denoting a degree, and the second denoting its multiplicity.

```
gap> g:= SolvableGroup( 24, 15 );
S4
gap> CharDegAgGroup( g );
[ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ] ]     # two linear characters, one of
                                     # degree 2, two of degree 3
gap> CharDegAgGroup( g, 3 );
```

```
[ [ 1, 2 ], [ 3, 2 ] ]
```

The algorithm bases on [Con90b]. It works for all solvable groups.

49.24 CharTableSSGroup

`CharTableSSGroup( G )`

`CharTableSSGroup` returns the character table of the supersolvable ag-group G and stores it in G`.charTable`. If G is not supersolvable not all irreducible characters migth be calculated and a warning will be printed out. The algorithm bases on [Con90a] and [Con90b].

All the characters calculated are monomial, so they are the induced of a linear character of some subgroup of G. For every character the subgroup it is induced from and the kernel the linear character has are written down in t`.irredinfo[i].inducedFrom.subgroup` and t`.irredinfo[i].inducedFrom.kernel`.

```
gap> t:= CharTableSSGroup( SolvableGroup( 8 , 5 ) );;
gap> PrintCharTable( t );
rec( size := 8, classes := [ 1, 1, 2, 2, 2 ], powermap :=
[ , [ 1, 1, 2, 2, 2 ]
 ], operations := CharTableOps, group := Q8, irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, -1, -1 ], [ 1, 1, -1, 1, -1 ],
  [ 1, 1, -1, -1, 1 ], [ 2, -2, 0, 0, 0 ] ], orders :=
[ 1, 2, 4, 4, 4 ], irredinfo := [ rec(
      inducedFrom := rec(
          subgroup := Q8,
          kernel := Q8 ) ), rec(
      inducedFrom := rec(
          subgroup := Q8,
          kernel := Subgroup( Q8, [ b, c ] ) ) ), rec(
      inducedFrom := rec(
          subgroup := Q8,
          kernel := Subgroup( Q8, [ a, c ] ) ) ), rec(
      inducedFrom := rec(
          subgroup := Q8,
          kernel := Subgroup( Q8, [ a*b, c ] ) ) ), rec(
      inducedFrom := rec(
          subgroup := Subgroup( Q8, [ b, c ] ),
          kernel := Subgroup( Q8, [  ] ) ) ) ], order :=
8, centralizers := [ 8, 8, 4, 4, 4
 ], identifier := "Q8", name := "Q8" )
```

49.25 MatRepresentationsPGroup

`MatRepresentationsPGroup( G )`
`MatRepresentationsPGroup( G [, int ] )`

`MatRepresentationsPGroup( G )` returns a list of homomorphisms from the finite polycyclic group G to irreducible complex matrix groups. These matrix groups form a system of representatives of the complex irreducible representations of G.

`MatRepresentationsPGroup(` *G*`,` *int* `)` returns only the *int*-th representation.

Let G be a finite polycyclic group with an abelian normal subgroup N such that the factorgroup G/N is supersolvable. `MatRepresentationsPGroup` uses the algorithm described in [Bau91]. Note that for such groups all such representations are equivalent to monomial ones, and in fact `MatRepresentationsPGroup` only returns monomial representations.

If G has not the property stated above, a system of representatives of irreducible representations and characters only for the factor group G/M can be computed using this algorithm, where M is the derived subgroup of the supersolvable residuum of G. In this case first a warning is printed. `MatRepresentationsPGroup` returns the irreducible representations of G with kernel containing M then.

```
gap> g:= SolvableGroup( 6, 2 );
S3
gap> MatRepresentationsPGroup( g );
[ GroupHomomorphismByImages( S3, Group( [ [ 1 ] ] ), [ a, b ],
    [ [ [ 1 ] ], [ [ 1 ] ] ] ), GroupHomomorphismByImages( S3, Group(
    [ [ -1 ] ] ), [ a, b ], [ [ [ -1 ] ], [ [ 1 ] ] ] ),
  GroupHomomorphismByImages( S3, Group( [ [ 0, 1 ], [ 1, 0 ] ],
    [ [ E(3), 0 ], [ 0, E(3)^2 ] ] ), [ a, b ],
    [ [ [ 0, 1 ], [ 1, 0 ] ], [ [ E(3), 0 ], [ 0, E(3)^2 ] ] ] ) ]
```

`CharTablePGroup` can be used to compute the character table of a group with the above properties (see 49.26).

49.26 CharTablePGroup

`CharTablePGroup(` *G* `)`

`CharTablePGroup` returns the character table of the finite polycyclic group G, and stores it in G`.charTable`. Do not change the order of G`.conjugacyClasses` after having called `CharTablePGroup`.

Let G be a finite polycyclic group with an abelian normal subgroup N such that the factorgroup G/N is supersolvable. `CharTablePGroup` uses the algorithm described in [Bau91].

If G has not the property stated above, a system of representatives of irreducible representations and characters only for the factor group G/M can be computed using this algorithm, where M is the derived subgroup of the supersolvable residuum of G. In this case first a warning is printed. `CharTablePGroup` returns an incomplete table containing exactly those irreducibles with kernel containing M.

```
gap> t:= CharTablePGroup( SolvableGroup( 8, 4 ) );;
gap> PrintCharTable( t );
rec( size := 8, centralizers := [ 8, 8, 4, 4, 4 ], classes :=
[ 1, 1, 2, 2, 2 ], orders := [ 1, 2, 2, 2, 4 ], irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, -1, 1, -1 ], [ 1, 1, 1, -1, -1 ],
  [ 1, 1, -1, -1, 1 ], [ 2, -2, 0, 0, 0 ]
 ], operations := CharTableOps, order := 8, powermap :=
[ , [ 1, 1, 1, 1, 2 ]
 ], identifier := "D8", name := "D8", group := D8 )
```

`MatRepresentationsPGroup` can be used to compute representatives of the complex irreducible representations (see 49.25).

49.27 InitClassesCharTable

`InitClassesCharTable(` *tbl* `)`

returns the list of conjugacy class lengths of the character table *tbl*, and assigns it to the field *tbl*`.classes`; the classlengths are computed from the centralizer orders of *tbl*.

`InitClassesCharTable` is called automatically for tables that are read from the library (see 49.12) or constructed as generic character tables (see 50).

```
gap> t:= rec( centralizers:= [ 2, 2 ], identifier:= "C2" );;
gap> InitClassesCharTable( t ); t;
[ 1, 1 ]
rec(
  centralizers := [ 2, 2 ],
  identifier := "C2",
  classes := [ 1, 1 ] )
```

49.28 InverseClassesCharTable

`InverseClassesCharTable(` *tbl* `)`

returns the list mapping each class of the character table *tbl* to its inverse class. This list can be regarded as (-1)-st powermap; it is computed using *tbl*`.irreducibles`.

```
gap> InverseClassesCharTable( CharTable( "L3(2)" ) );
[ 1, 2, 3, 4, 6, 5 ]
```

49.29 ClassNamesCharTable

`ClassNamesCharTable(` *tbl* `)`

`ClassNamesCharTable` computes names for the classes of the character table *tbl* as strings consisting of the order of an element of the class and and least one distinguishing letter.

The list of these names at the same time is returned by this function and stored in the table *tbl* as record component `classnames`.

Moreover for each class a component with its name is constructed, containing the position of this name in the list `classnames` as its value.

```
gap> c3:= CharTable( "Cyclic", 3 );;
gap> ClassNamesCharTable( c3 );
[ "1a", "3a", "3b" ]
gap> PrintCharTable( c3 );
rec( identifier := "C3", name := "C3", size := 3, order :=
3, centralizers := [ 3, 3, 3 ], orders := [ 1, 3, 3 ], powermap :=
[ ,, [ 1, 1, 1 ] ], irreducibles :=
[ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ]
 ], classparam := [ [ 1, 0 ], [ 1, 1 ], [ 1, 2 ] ], irredinfo :=
[ rec(
      charparam := [ 1, 0 ] ), rec(
      charparam := [ 1, 1 ] ), rec(
      charparam := [ 1, 2 ] )
```

```
 ], text := "computed using generic character table for cyclic groups"\
, classes := [ 1, 1, 1 ], operations := CharTableOps, fusions :=
[  ], fusionsource := [  ], projections := [  ], projectionsource :=
[  ], classnames := [ "1a", "3a", "3b" ], 1a := 1, 3a := 2, 3b := 3 )
```

If the record component `classnames` of *tbl* is unbound, `ClassNamesCharTable` is automatically called by `DisplayCharTable` (see 49.37).

Note that once the class names are computed the resulting record fields are stored on *tbl*. They are not deleted by another call of `ClassNamesCharTable`.

49.30 ClassMultCoeffCharTable

`ClassMultCoeffCharTable(` *tbl*`,` *c1* `,` *c2*`,` *c3* `)`

returns the class multiplication coefficient of the classes *c1*, *c2* and *c3* of the group G with character table *tbl*.

```
gap> t:= CharTable( "L3(2)" );;
gap> ClassMultCoeffCharTable( t, 2, 2, 4 );
4
```

The class multiplication coefficient $c_{1,2,3}$ of the classes *c1*, *c2* and *c3* equals the number of pairs (x, y) of elements $x, y \in G$ such that x lies in class *c1*, y lies in class *c2* and their product xy is a fixed element of class *c3*.

Also in the center of the group algebra these numbers are found as coefficients of the decomposition of the product of two class sums K_i and K_j into class sums,

$$K_iK_j = \sum_k c_{ijk}K_k.$$

Given the character table of a finite group G, whose classes are $C_1, \ldots, C_r$ with representatives $g_i \in C_i$, the class multiplication coefficients c_{ijk} can be computed by the following formula.

$$c_{ijk} = \frac{|C_i||C_j|}{|G|} \sum_{\chi \in \mathrm{Irr}(G)} \frac{\chi(g_i)\chi(g_j)\overline{\chi(g_k)}}{\chi(1)}$$

On the other hand the knowledge of the class multiplication coefficients enables the computation of the character table (see 49.12).

49.31 MatClassMultCoeffsCharTable

`MatClassMultCoeffsCharTable(` *tbl*`,` *class* `)`

returns the matrix $C_i = [a_{ijk}]_{j,k}$ of structure constants (see 49.30).

```
gap> L3_2:= CharTable( "L3(2)" );;
gap> MatClassMultCoeffsCharTable( t, 2 );
[ [ 0, 1, 0, 0, 0, 0 ], [ 21, 4, 3, 4, 0, 0 ], [ 0, 8, 6, 8, 7, 7 ],
  [ 0, 8, 6, 1, 7, 7 ], [ 0, 0, 3, 4, 0, 7 ], [ 0, 0, 3, 4, 7, 0 ] ]
```

49.32 ClassStructureCharTable

`ClassStructureCharTable(` *tbl*`,` *classes* `)`

returns the so–called class structure of the classes in the list *classes*, for the character table *tbl* of the group G. The length of *classes* must be at least 2.

```
gap> t:= CharTable( "M12" );;
gap> ClassStructureCharTable( t, [2,6,9,13] );
916185600
gap> ClassStructureCharTable( t, [2,9,13] ); # equals the group order
95040
```

Let $C_1, \ldots, C_n$ denote the conjugacy classes of G that are indexed by *classes*. The class structure $n(C_1, C_2, \ldots, C_n)$ equals the number of tuples $(g_1, g_2, \ldots, g_n)$ of elements $g_i \in C_i$ with $g_1 g_2 \cdots g_n = 1$. Note the difference to the definition of the class multiplication coefficients in 49.30 `ClassMultCoeffCharTable`.

$n(C_1, C_2, \ldots, C_n)$ is computed using the formula

$$n(C_1, C_2, \ldots, C_n) = \frac{|C_1||C_2|\cdots|C_n|}{|G|} \sum_{\chi \in \mathrm{Irr}(G)} \frac{\chi(g_1)\chi(g_2)\cdots\chi(g_n)}{\chi(1)^{n-2}}.$$

49.33 RealClassesCharTable

`RealClassesCharTable(` *tbl* `)`

returns a list of the real classes of the group G with character table *tbl*.

```
gap> RealClassesCharTable(L3_2);
[ 1, 2, 3, 4 ]
```

An element $x \in G$ is called **real**, if it is conjugate with its inverse. And as $x^{-1} = x^{o(x)-1}$, this fact is tested by looking at the powermap of *tbl*.

Real elements take only real character values.

49.34 ClassOrbitCharTable

`ClassOrbitCharTable(` *tbl*`,` *class* `)`

returns a list of classes containing elements of the cyclic subgroup generated by an element x of class *class*.

```
gap> ClassOrbitCharTable(L3_2, 5);
[ 5, 6 ]
```

Being all powers of x this data is recovered from the powermap of *tbl*.

49.35 ClassRootsCharTable

`ClassRootsCharTable(` *tbl* `)`

returns a list of the classes of all nontrivial p–th roots of the classes of *tbl* where for each class, p runs over the prime divisors of the representative order.

```
gap> ClassRootsCharTable(L3_2);
[ [ 2, 3, 5, 6 ], [ 4 ], [  ], [  ], [  ], [  ] ]
```

This information is found by looking at the powermap of *tbl*, too.

49.36 NrPolyhedralSubgroups

`NrPolyhedralSubgroups(` *tbl*, *c1*, *c2*, *c3* `)`

returns the number and isomorphism type of polyhedral subgroups of the group with character table *tbl* which are generated by an element g of class *c1* and an element h of class *c2* with the property that the product gh lies in class *c3*.

```
gap> NrPolyhedralSubgroups(L3_2, 2, 2, 4);
rec(
  number := 21,
  type := "D8" )
```

According to [NPP84, p. 233] the number of polyhedral subgroups of isomorphism type V_4, D_{2n}, A_4, S_4 and A_5 can be derived from the class multiplication coefficient (see 49.30) and the number of Galois conjugates of a class (see 49.34).

Note that the classes *c1*, *c2* and *c3* in the parameter list must be ordered according to the order of the elements in these classes.

49.37 DisplayCharTable

`DisplayCharTable(` *tbl* `)`
`DisplayCharTable(` *tbl*, *arec* `)`

`DisplayCharTable` prepares the data contained in the character table *tbl* for a pretty columnwise output.

In the first form `DisplayCharTable` prints all irreducible characters of the table *tbl*, together with the orders of the centralizers in factorized form and the available powermaps.

Thus it can be used to echo character tables in interactive use, being the value of the record field `Print` of a record field `operations` of *tbl* (see 49.2, 49.7).

Each displayed character is given a name `X.`n.

The number of columns printed at one time depends on the actual linelength, which is restored by the function `SizeScreen` (see 3.19).

The first lines of the output describe the order of the centralizer of an element of the class factorized into its prime divisor.

The next line gives the name of the class. If the record field `classnames` of the table *tbl* is not bound, `DisplayCharTable` calls the function `ClassNamesCharTable` to determine the class names and to store them on the table *tbl* (see 49.29).

Preceded by a name `P`n the next lines show the nth powermaps of *tbl* in terms of the former shown class names.

Every ambiguous or unknown (see 17.1) value of the table is displayed as a question mark `?`.

Irrational character values are not printed explicitly because the lengths of their printed representation might disturb the view. Instead of that every irrational value is indicated by a name, which is a string of a least one capital letter.

Once a name for an irrational number is found, it is used all over the printed table. Moreover the complex conjugate and the star of an irrationality are represented by that very name preceded by a `/` resp. a `*`.

The printed character table is then followed by a legend, a list identifying the occurred symbols with their actual irrational value. Occasionally this identity is supplemented by a quadratic representation of the irrationality together with the corresponding ATLAS–notation.

```
gap> a5:= CharTable("A5");;
gap> DisplayCharTable(a5);
A5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1

       1a 2a 3a 5a 5b
    2P 1a 1a 3a 5b 5a
    3P 1a 2a 1a 5b 5a
    5P 1a 2a 3a 1a 1a

X.1     1  1  1  1  1
X.2     3 -1  .  A *A
X.3     3 -1  . *A  A
X.4     4  .  1 -1 -1
X.5     5  1 -1  .  .

A = -E(5)-E(5)^4
  = (1-ER(5))/2 = -b5
```

In the second form `DisplayCharTable` takes an argument record *arec* as an additional argument. This record can be used to change the default style for displaying a character as shown above. Its relevant fields are

`chars`
: an integer or a list of integers to select a sublist of the irreducible characters of *tbl*, or a list of characters of *tbl* (in this case the letter `"X"` is replaced by `"Y"`),

`classes`
: an integer or a list of integers to select a sublist of the classes of *tbl*,

`centralizers`
: suppresses the printing of the orders of the centralizers if `false`,

`powermap`
: an integer or a list of integers to select a subset of the available powermaps, or `false` to suppress the powermaps,

`letter`
: a single capital letter (e. g. `"P"` for permutation characters) to replace `"X"`,

`indicator`
: `true` enables the printing of the second Schur indicator, a list of integers enables the printing of the corresponding indicators.

```
gap> arec:= rec( chars:= 4, classes:= [a5.3a..a5.5a],
> centralizers:= false, indicator:= true, powermap:= [2] );;
```

```
gap> Indicator( a5, 2 );;
gap> DisplayCharTable( a5, arec );
A5

          3a 5a
       2P 3a 5b
       2
X.4    +   1 -1
```

49.38 SortCharactersCharTable

`SortCharactersCharTable(` *tbl* `)`
`SortCharactersCharTable(` *tbl*`,` *permutation* `)`
`SortCharactersCharTable(` *tbl*`,` *chars* `)`
`SortCharactersCharTable(` *tbl*`,` *chars*`,` *permutation* `)`
`SortCharactersCharTable(` *tbl*`,` *chars*`, "norm")`
`SortCharactersCharTable(` *tbl*`,` *chars*`, "degree")`

If no list *chars* of characters of the character table *tbl* is entered, `SortCharactersCharTable` sorts *tbl*`.irreducibles`; then additionally the list *tbl*`.irredinfo` is permuted by the same permutation. Otherwise `SortCharactersCharTable` sorts the list *chars*.

There are four possibilities to sort characters: Besides the application of an explicitly given permutation (see 27.41), they can be sorted according to ascending norms (parameter `"norm"`), to ascending degree (parameter `"degree"`) or both (no third parameter), i.e., characters with same norm are sorted according to ascending degree, and characters with smaller norm precede those with bigger norm.

If the trivial character is contained in the sorted list, it will be sorted to the first position. Rational characters always will precede other ones with same norm resp. same degree afterwards.

`SortCharactersCharTable` returns the permutation that was applied to *chars*.

```
gap> t:= CharTable( "Symmetric", 5 );;
gap> PrintCharTable( t );
rec( identifier := "S5", name := "S5", size := 120, order :=
120, centralizers := [ 120, 12, 8, 6, 6, 4, 5 ], orders :=
[ 1, 2, 2, 3, 6, 4, 5 ], powermap :=
[ , [ 1, 1, 1, 4, 4, 3, 7 ], [ 1, 2, 3, 1, 2, 6, 7 ],,
  [ 1, 2, 3, 4, 5, 6, 1 ] ], irreducibles :=
[ [ 1, -1, 1, 1, -1, -1, 1 ], [ 4, -2, 0, 1, 1, 0, -1 ],
  [ 5, -1, 1, -1, -1, 1, 0 ], [ 6, 0, -2, 0, 0, 0, 1 ],
  [ 5, 1, 1, -1, 1, -1, 0 ], [ 4, 2, 0, 1, -1, 0, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1 ] ], classparam :=
[ [ 1, [ 1, 1, 1, 1, 1 ] ], [ 1, [ 2, 1, 1, 1 ] ], [ 1, [ 2, 2, 1 ] ],
  [ 1, [ 3, 1, 1 ] ], [ 1, [ 3, 2 ] ], [ 1, [ 4, 1 ] ], [ 1, [ 5 ] ]
 ], irredinfo := [ rec(
      charparam := [ 1, [ 1, 1, 1, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 2, 1, 1, 1 ] ] ), rec(
```

```
      charparam := [ 1, [ 2, 2, 1 ] ] ), rec(
      charparam := [ 1, [ 3, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 3, 2 ] ] ), rec(
      charparam := [ 1, [ 4, 1 ] ] ), rec(
      charparam := [ 1, [ 5 ] ] )
 ], text := "computed using generic character table for symmetric grou\
ps", classes := [ 1, 10, 15, 20, 20, 30, 24
 ], operations := CharTableOps, fusions := [  ], fusionsource :=
[  ], projections := [  ], projectionsource := [  ] )
gap> SortCharactersCharTable( t, t.irreducibles, "norm" );
(1,2,3,4,5,6,7)       # sort the trivial character to the first position
gap> SortCharactersCharTable( t );
(4,5,7)
gap> t.irreducibles;
[ [ 1, 1, 1, 1, 1, 1, 1 ], [ 1, -1, 1, 1, -1, -1, 1 ],
  [ 4, -2, 0, 1, 1, 0, -1 ], [ 4, 2, 0, 1, -1, 0, -1 ],
  [ 5, -1, 1, -1, -1, 1, 0 ], [ 5, 1, 1, -1, 1, -1, 0 ],
  [ 6, 0, -2, 0, 0, 0, 1 ] ]
```

49.39 SortClassesCharTable

`SortClassesCharTable(` *tbl* `)`
`SortClassesCharTable(` *tbl*`, "centralizers")`
`SortClassesCharTable(` *tbl*`, "representatives")`
`SortClassesCharTable(` *tbl*`,` *permutation* `)`
`SortClassesCharTable(` *chars*`,` *permutation* `)`

The last form simply permutes the classes of all elements of *chars* with *permutation*. All other forms take a character table *tbl* as parameter, and `SortClassesCharTable` permutes the classes of *tbl*:

`SortClassesCharTable(` *tbl*`, "centralizers")`
: sorts the classes according to descending centralizer orders,

`SortClassesCharTable(` *tbl*`, "representatives")`
: sorts the classes according to ascending representative orders,

`SortClassesCharTable(` *tbl* `)`
: sorts the classes according to ascending representative orders, and classes with equal representative orders according to descending centralizer orders,

`SortClassesCharTable(` *tbl*`,` *permutation* `)`
: sorts the classes by application of *permutation*

After having calculated the permutation, `SortClassesCharTable` will adjust the following fields of *tbl*:

by application of the permutation: `orders`, `centralizers`, `classes`, `print`, all entries of `irreducibles`, `classtext`, `classparam`, `classnames`, all fusion maps, all entries of the `chars` lists in the records of `projectives`

by conjugation with the permutation: all powermaps, `automorphisms`,

by multiplication with the permutation: `permutation`, and the fields corresponding to *tbl*.`classnames` (see 49.29).

The applied permutation is returned by `SortClassesCharTable`.

Note that many programs expect the class `1A` to be the first one (see 49.10).

```
gap> t:= CharTable( "Symmetric", 5 );;
gap> PrintCharTable( t );
rec( identifier := "S5", name := "S5", size := 120, order :=
120, centralizers := [ 120, 12, 8, 6, 6, 4, 5 ], orders :=
[ 1, 2, 2, 3, 6, 4, 5 ], powermap :=
[ , [ 1, 1, 1, 4, 4, 3, 7 ], [ 1, 2, 3, 1, 2, 6, 7 ],,
  [ 1, 2, 3, 4, 5, 6, 1 ] ], irreducibles :=
[ [ 1, -1, 1, 1, -1, -1, 1 ], [ 4, -2, 0, 1, 1, 0, -1 ],
  [ 5, -1, 1, -1, -1, 1, 0 ], [ 6, 0, -2, 0, 0, 0, 1 ],
  [ 5, 1, 1, -1, 1, -1, 0 ], [ 4, 2, 0, 1, -1, 0, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1 ] ], classparam :=
[ [ 1, [ 1, 1, 1, 1, 1 ] ], [ 1, [ 2, 1, 1, 1 ] ], [ 1, [ 2, 2, 1 ] ],
  [ 1, [ 3, 1, 1 ] ], [ 1, [ 3, 2 ] ], [ 1, [ 4, 1 ] ], [ 1, [ 5 ] ]
 ], irredinfo := [ rec(
      charparam := [ 1, [ 1, 1, 1, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 2, 1, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 2, 2, 1 ] ] ), rec(
      charparam := [ 1, [ 3, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 3, 2 ] ] ), rec(
      charparam := [ 1, [ 4, 1 ] ] ), rec(
      charparam := [ 1, [ 5 ] ] )
 ], text := "computed using generic character table for symmetric grou\
ps", classes := [ 1, 10, 15, 20, 20, 30, 24
 ], operations := CharTableOps, fusions := [  ], fusionsource :=
[  ], projections := [  ], projectionsource := [  ] )
gap> SortClassesCharTable( t, "centralizers" );
(6,7)
gap> SortClassesCharTable( t, "representatives" );
(5,7)
gap> t.centralizers; t.orders;
[ 120, 12, 8, 6, 4, 5, 6 ]
[ 1, 2, 2, 3, 4, 5, 6 ]
```

49.40 SortCharTable

`SortCharTable(` *tbl*, *kernel* `)`
`SortCharTable(` *tbl*, *normalseries* `)`
`SortCharTable(` *tbl*, *facttbl*, *kernel* `)`

sorts classes and `irreducibles` of the character table *tbl*, and returns a record with components `columns` and `rows`, which are the permutations applied to classes and characters.

The first form sorts the classes at positions contained in the list *kernel* to the beginning, and sorts all characters in *tbl*.`irreducibles` such that the first characters are those that contain *kernel* in their kernel.

The second form does the same successively for all kernels k_i in the list `normalseries` $= [k_1, k_2, \ldots, k_n]$ where k_i must be a sublist of k_{i+1} for $1 \leq i \leq n-1$.

The third form computes the table F of the factor group of *tbl* modulo the normal subgroup formed by the classes whose positions are contained in the list *kernel*; F must be permutation equivalent to the table *facttbl* (in the sense of 49.44), else `false` is returned. The classes of *tbl* are sorted such that the preimages of a class of F are consecutive, and that the succession of preimages is that of *facttbl*. *tbl*`.irreducibles` is sorted as by `SortCharTable(` *tbl*`,` *kernel* `)`. (**Note** that the transformation is only unique up to table automorphisms of F, and this need not be unique up to table automorphisms of *tbl*.)

All rearrangements of classes and characters are stable, i.e., the relative positions of classes and characters that are not distinguished by any relevant property is not changed.

`SortCharTable` uses 49.39 `SortClassesCharTable` and 49.38 `SortCharactersCharTable`.

```
gap> t:= CharTable("Symmetric",4);;
gap> Set( List( t.irreducibles, KernelChar ) );
[ [ 1 ], [ 1, 2, 3, 4, 5 ], [ 1, 3 ], [ 1, 3, 4 ] ]
gap> SortCharTable( t, Permuted( last, (2,4,3) ) );
rec(
  columns := (2,4,3),
  rows := (1,2,4,5) )
gap> DisplayCharTable( t );
S4

     2  3  3  .  2  2
     3  1  .  1  .  .

       1a 2a 3a 2b 4a
    2P 1a 1a 3a 1a 2a
    3P 1a 2a 1a 2b 4a

X.1     1  1  1  1  1
X.2     1  1  1 -1 -1
X.3     2  2 -1  .  .
X.4     3 -1  . -1  1
X.5     3 -1  .  1 -1
```

49.41 MatAutomorphisms

`MatAutomorphisms(` *mat*`,` *maps*`,` *subgroup* `)`

returns the permutation group record representing the matrix automorphisms of the matrix *mat* that respect all lists in the list *maps*, i.e. representing the group of those permutations of columns of *mat* which acts on the set of rows of *mat* and additionally fixes all lists in *maps*.

subgroup is a list of permutation generators of a subgroup of this group. E.g. generators of the Galois automorphisms of a matrix of ordinary characters may be entered here.

```
gap> t:= CharTable( "Dihedral", 8 );;
```

```
gap> PrintCharTable( t );
rec( identifier := "D8", name := "D8", size := 8, order :=
8, centralizers := [ 8, 4, 8, 4, 4 ], orders := [ 1, 4, 2, 2, 2
 ], powermap := [ , [ 1, 3, 1, 1, 1 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, -1, -1 ], [ 1, -1, 1, 1, -1 ],
  [ 1, -1, 1, -1, 1 ], [ 2, 0, -2, 0, 0 ] ], classparam :=
[ [ 1, 0 ], [ 1, 1 ], [ 1, 2 ], [ 2, 0 ], [ 2, 1 ] ], irredinfo :=
[ rec(
      charparam := [ 1, 0 ] ), rec(
      charparam := [ 1, 1 ] ), rec(
      charparam := [ 1, 2 ] ), rec(
      charparam := [ 1, 3 ] ), rec(
      charparam := [ 2, 1 ] )
 ], text := "computed using generic character table for dihedral group\
s", classes := [ 1, 2, 1, 2, 2
 ], operations := CharTableOps, fusions := [  ], fusionsource :=
[  ], projections := [  ], projectionsource := [  ] )
gap> MatAutomorphisms( t.irreducibles, [], Group( () ) );
Group( (4,5), (2,4) )
gap> MatAutomorphisms( t.irreducibles, [ t.orders ], Group( () ) );
Group( (4,5) )
```

49.42 TableAutomorphisms

`TableAutomorphisms(` *tbl*`,` *chars* `)`
`TableAutomorphisms(` *tbl*`,` *chars*`, "closed")`

returns a permutation group record for the group of those matrix automorphisms of *chars* (see 49.41) which are admissible by (i.e. which fix) the fields `orders` and all uniquely determined (i.e. not parametrized) maps in `powermap` of the character table *tbl*; the action on `orders` is the natural permutation, that on the powermaps is conjugation.

If *chars* is closed under galois conjugacy –this is always satisfied for ordinary character tables– the parameter "closed" may be entered. In that case the subgroup of Galois automorphisms is computed by 13.16 `GaloisMat`.

```
gap> t:= CharTable( "Dihedral", 8 );; # as in 49.41
gap> TableAutomorphisms( t, t.irreducibles );
Group( (4,5) )
```

49.43 TransformingPermutations

`TransformingPermutations(` *mat1*`,` *mat2* `)`

tries to construct a permutation π that transforms the set of rows of the matrix *mat1* to the set of rows of the matrix *mat2* by permutation of columns. If such a permutation exists, a record with fields `columns`, `rows` and `group` is returned, otherwise `false`: If `TransformingPermutations`$(mat1, mat2) = r \neq$ `false` then

`Permuted( List(`*mat1*`,x->Permuted(x,`*r*`.columns)),`*r*`.rows ) =` *mat2*, and *r*.`group` is the group of matrix automorphisms of *mat2*; this group stabilizes the transformation, i.e. for g in that group and π the value of the `columns` field, also πg would be a valid permutation of columns.

```
gap> mat1:= CharTable( "Alternating", 5 ).irreducibles;
[ [ 1, 1, 1, 1, 1 ], [ 4, 0, 1, -1, -1 ], [ 5, 1, -1, 0, 0 ],
  [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ]
gap> mat2:= CharTable( "A5" ).irreducibles;
[ [ 1, 1, 1, 1, 1 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> TransformingPermutations( mat1, mat2 );
rec(
  columns := (),
  rows := (2,4)(3,5),
  group := Group( (4,5) ) )
```

49.44 TransformingPermutationsCharTables

`TransformingPermutationsCharTables(` *tbl1* `,` *tbl2* `)`

tries to construct a permutation π that transforms the set of rows of *tbl1*`.irreducibles` to the set of rows of *tbl2*`.irreducibles` by permutation of columns (see 49.43) and that also transforms the powermaps and the `orders` field. If such a permutation exists, it returns a record with components `columns` (a valid permutation of columns), `rows` (the permutation of *tbl*`.irreducibles` corresponding to that permutation), and `group` (the permutation group record of table automorphisms of *tbl2*, see 49.42). If no such permutation exists, it returns `false`.

```
gap> t1:= CharTable("Dihedral",8);;t2:= CharTable("Quaternionic",8);;
gap> TransformingPermutations( t1.irreducibles, t2.irreducibles );
rec(
  columns := (),
  rows := (),
  group := Group( (4,5), (2,4) ) )
gap> TransformingPermutationsCharTables( t1, t2 );
false
gap> t1:= CharTable( "Dihedral", 6 );; t2:= CharTable("Symmetric",3);;
gap> TransformingPermutationsCharTables( t1, t2 );
rec(
  columns := (2,3),
  rows := (1,3,2),
  group := Group( () ) )
```

49.45 GetFusionMap

`GetFusionMap(` *source* `,` *destination* `)`
`GetFusionMap(` *source* `,` *destination* `,` *specification* `)`

For character tables *source* and *destination*, `GetFusionMap(` *source* `,` *destination* `)` returns the `map` field of the fusion stored on the character table *source* that has the `identifier` component *destination*`.name`;

`GetFusionMap(` *source* `,` *destination* `,` *specification* `)` gets that fusion that additionally has the `specification` field *specification*.

Both versions adjust the ordering of classes of *destination* using *destination*`.permutation` (see 49.39, 49.10). That is the reason why *destination* cannot be simply the identifier of the destination table.

If both *source* and *destination* are Brauer tables, `GetFusionMap` returns the fusion corresponding to that between the ordinary tables; for that, this fusion must be stored on *source*`.ordinary`.

If no appropriate fusion is found, `false` is returned.

```
gap> s:= CharTable( "L2(11)" );;
gap> t:= CharTable( "J1" );;
gap> SortClassesCharTable( t, ( 3, 4, 5, 6 ) );;
gap> t.permutation;
(3,4,5,6)
gap> GetFusionMap( s, t );
[ 1, 2, 4, 6, 5, 3, 10, 10 ]
gap> s.fusions[5];
rec(
  name := "J1",
  map := [ 1, 2, 3, 5, 4, 6, 10, 10 ],
  text := [ 'f', 'u', 's', 'i', 'o', 'n', ' ', 'i', 's', ' ', 'u',
      'n', 'i', 'q', 'u', 'e', ' ', 'u', 'p', ' ', 't', 'o', ' ',
      't', 'a', 'b', 'l', 'e', ' ', 'a', 'u', 't', 'o', 'm', 'o',
      'r', 'p', 'h', 'i', 's', 'm', 's', ',', '\n', 't', 'h', 'e',
      ' ', 'r', 'e', 'p', 'r', 'e', 's', 'e', 'n', 't', 'a', 't',
      'i', 'v', 'e', ' ', 'i', 's', ' ', 'e', 'q', 'u', 'a', 'l',
      ' ', 't', 'o', ' ', 't', 'h', 'e', ' ', 'f', 'u', 's', 'i',
      'o', 'n', ' ', 'm', 'a', 'p', ' ', 'o', 'n', ' ', 't', 'h',
      'e', ' ', 'C', 'A', 'S', ' ', 't', 'a', 'b', 'l', 'e' ] )
```

49.46 StoreFusion

`StoreFusion(` *source* `,` *destination* `,` *fusion* `)`
`StoreFusion(` *source* `,` *destination* `,` *fusionmap* `)`

For character tables *source* and *destination*, *fusion* must be a record containing at least the field `map` which is regarded as a fusion from *source* to *destination*. *fusion* is stored on *source* if no ambiguity arises, i.e. if there is not yet a fusion into *destination* stored on *source* or if any fusion into destination stored on *source* has a `specification` field different from that of *fusion*. The `map` field of *fusion* is adjusted by *destination*`.permutation`. (Thus the map will remain correct even if the classes of a concerned table are sorted, see 49.39 and 49.10; the correct fusion can be got using 49.45, so be careful!). Additionally, *source*`.identifier` is added to *destination*`.fusionsource`.

The second form works like the first, with *fusion* `= rec( map:=` *fusionmap* `)`.

```
gap> s:= CharTable( "A6.2_1" );; t:= CharTable( "A7.2" );;
gap> fus:= RepresentativesFusions( s, SubgroupFusions( s, t ), t );
[ [ 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13 ] ]
gap> s.fusions; t.fusionsource;
[  ]
[ "2.A7.2", "3.A7.2", "6.A7.2", "A7" ]
gap> StoreFusion( s, t, fus[1] );
gap> s.fusions; t.fusionsource;
[ rec(
      name := "A7.2",
      map := [ 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13 ] ) ]
[ "2.A7.2", "3.A7.2", "6.A7.2", "A6.2_1", "A7" ]
```

49.47 FusionConjugacyClasses

`FusionConjugacyClasses(` *subgroup*, *group* `)`
`FusionConjugacyClasses(` *group* , *factorgroup* `)`

`FusionConjugacyClasses` returns a list denoting the fusion of conjugacy classes from the first group to the second one. If both groups have components `charTable` this list is written to the character tables, too.

```
gap> g := SolvableGroup( 24, 14 );
Sl(2,3)
gap> FusionConjugacyClasses( g, g / Subgroup( g, [ g.4 ] ) );
[ 1, 1, 2, 3, 3, 4, 4 ]
gap> FusionConjugacyClasses( Subgroup( g, [ g.2, g.3, g.4 ] ), g );
[ 1, 2, 3, 3, 3 ]
```

49.48 MAKElb11

`MAKElb11(` *listofns* `)`

prints field information for fields with conductor Q_n where n is in the list *listofns*;

`MAKElb11( [ 3 .. 189 ] )` will print something very similar to Richard Parker's file `lb11`.

```
gap> MAKElb11( [ 3, 4 ] );
   3   2   0   1   0
   4   2   0   1   0
```

49.49 ScanMOC

`ScanMOC(` *list* `)`

returns a record containing the information encoded in the list *list*, the components of the result are the labels in *list*. If *list* is in MOC2 format (10000–format) the names of components are 30000–numbers, if it is in MOC3 format the names of components have yABC–format.

```
gap> ScanMOC( "y100y105ay110t28t22z" );
```

```
rec(
  y105 := [ 0 ],
  y110 := [ 28, 22 ] )
```

49.50 MOCChars

`MOCChars(` *tbl*`,` *gapchars* `)`

returns translations of GAP3 format characters *gapchars* to MOC format. *tbl* must be a GAP3 format table or a MOC format table.

49.51 GAPChars

`GAPChars(` *tbl*`,` *mocchars* `)`

returns translations of MOC format characters *mocchars* to GAP3 format. *tbl* must be a GAP3 format table or a MOC format table.

mocchars may also be a list of integers, e.g., a component containing characters in a record produced by 49.49.

49.52 MOCTable

`MOCTable(` *gaptbl* `)`
`MOCTable(` *gaptbl*`,` *basicset* `)`

return the MOC format table record of the GAP3 table *gaptbl*, and stores it in the component `MOCtbl` of *gaptbl*.

The first form can be used for ordinary ($G.0$) tables only, for modular ($G.p$) tables one has to specify a basic set *basicset* of ordinary irreducibles which must be the list of positions of these characters in the `irreducibles` component of the corresponding ordinary table (which is stored in *gaptbl*.ordinary).

The result contains the information of *gaptbl* in a format similar to the MOC 3 format, the table itself can e.g. easily be printed out or be used to print out characters using 49.53.

The components of the result are `identifier` the string `MOCTable(` *name* `)` where *name* is the `identifier` component of *gaptbl*,

`isMOCformat` has value `true`,

`GAPtbl` the record *gaptbl*,

`operations` equal to `MOCTableOps`, containing just an appropriate `Print` function,

`prime` the characteristic of the field (label 30105 in MOC),

`centralizers` centralizer orders for cyclic subgroups (label 30130)

`orders` element orders for cyclic subgroups (label 30140)

`fields` at position `i` the number field generated by the character values of the `i`–th cyclic subgroup; the `base` component of each field is a Parker base, (the length of `fields` is equal to the value of label 30110 in MOC).

`cycsubgps` `cycsubgps[i] = j` means that class `i` of the GAP3 table belongs to the `j`–th cyclic subgroup of the GAP3 table, `repcycsub` `repcycsub[j] = i` means that class `i` of the GAP3 table is the representative of the `j`–th cyclic subgroup of the GAP3 table. **Note** that the representatives of GAP3 table and MOC table need not agree!

`galconjinfo` a list $[r_1, c_1, r_2, c_2, \ldots, r_n, c_n]$ which means that the i–th class of the GAP table is the c_i–th conjugate of the representative of the r_i–th cyclic subgroup on the MOC table. (This is used to translate back to GAP format, stored under label 30160)

`30170` (power maps) for each cyclic subgroup (except the trivial one) and each prime divisor of the representative order store four values, the number of the subgroup, the power, the number of the cyclic subgroup containing the image, and the power to which the representative must be raised to give the image class. (This is used only to construct the `30230` power map/embedding information.) In `result.30170` only a list of lists (one for each cyclic subgroup) of all these values is stored, it will not be used by GAP3.

`tensinfo` tensor product information, used to compute the coefficients of the Parker base for tensor products of characters (label 30210 in MOC). For a field with vector space base $(v_1, v_2, \ldots, v_n)$ the tensor product information of a cyclic subgroup in MOC (as computed by `fct`) is either 1 (for rational classes) or a sequence

$$n x_{1,1} y_{1,1} z_{1,1} x_{1,2} y_{1,2} z_{1,2} \ldots x_{1,m_1} y_{1,m_1} z_{1,m_1} 0 x_{2,1} \ldots z_{2,m_2} 0 \ldots x_{n,m_n} y_{n,m_n} z_{n,m_n} 0$$

which means that the coefficient of v_k in the product

$$(\sum_{i=1}^{n} a_i v_i)(\sum_{j=1}^{n} b_j v_j)$$

is equal to

$$\sum_{i=1}^{m_k} x_{k,i} a_{y_{k,i}} b_{z_{k,i}} \ .$$

On a MOC table in GAP3 the `tensinfo` component is a list of lists, each containing exactly the sequence

`invmap` inverse map to compute complex conjugate characters, label 30220 in MOC.

`powerinfo` field embeddings for p–th symmetrizations, p prime in `[ 2 .. 19 ]`; note that the necessary power maps must be stored on *gaptbl* to compute this component. (label 30230 in MOC)

`30900` basic set of restricted ordinary irreducibles in the case of nonzero characteristic, all ordinary irreducibles else.

49.53 PrintToMOC

`PrintToMOC(` *moctbl* `)`
`PrintToMOC(` *moctbl*`,` *chars* `)`

The first form prints the MOC3 format of the character table *moctbl* which must be an character table in MOC format (as produced by 49.52). The second form prints a table in MOC3 format that contains the MOC format characters *chars* (as produced by 49.50) under label `y900`.

```
gap> t:= CharTable( "A5mod3" );;
gap> moct:= MOCTable( t, [ 1, 2, 3, 4 ] );;
gap> PrintTo( "a5mod3", PrintToMOC( moct ), "\n" );
```

produces a file `a5mod3` whose first characters are

```
y100y105dy110edy130t60efy140bcfy150bbfcabbey160bbcbdbdcy170ccbbefbb
```

49.54 PrintToCAS

`PrintToCAS(` *filename*`,` *tbl* `)`
`PrintToCAS(` *tbl*`,` *filename* `)`

produces a file with name *filename* which contains a CAS library table of the GAP3 character table *tbl*; this file can be read into CAS using the `get`-command (see [NPP84]).

The line length in the file is at most the current value `SizeScreen()[1]` (see 3.19).

Only the components `identifier`, `text`, `order`, `centralizers`, `orders`, `print`, `powermap`, `classtext` (for partitions only), `fusions`, `irredinfo`, `characters`, `irreducibles` of *tbl* are considered.

If *tbl*`.characters` is bound, this list is taken as `characters` entry of the CAS table, otherwise *tbl*`.irreducibles` (if exists) will form the list `characters` of the CAS table.

```
gap> PrintToCAS( "c2", CharTable( "Cyclic", 2 ) );
```

produces a file with name `c2` containing the following data:

```
'C2'
00/00/00. 00.00.00.
(2,2,0,2,-1,0)
text:
(#computed using generic character table for cyclic groups#),
order=2,
centralizers:(2,2),
reps:(1,2),
powermap:2(1,1),
characters:
(1,1)
(1,-1);
/// converted from GAP
```

Chapter 50

Generic Character Tables

This chapter informs about the conception of generic character tables (see 50.1), it gives some examples of generic tables (see 50.2), and introduces the specialization function (see 50.3).

The generic tables that are actually available in the GAP3 group collection are listed in 49.12, see also 53.1.

50.1 More about Generic Character Tables

Generic character tables provide a means for writing down the character tables of all groups in a (usually infinite) series of similar groups, e.g. the cyclic groups, the symmetric groups or the general linear groups $\mathrm{GL}(2, q)$.

Let $\{G_q | q \in I\}$, where I is an index set, be such a series. The table of a member G_q could be computed using a program for this series which takes q as parameter, and constructs the table. It is, however, desirable to compute not only the whole table but to get a single character or just one character value without computation the table. E.g. both conjugacy classes and irreducible characters of the symmetric group S_n are in bijection with the partitions of n. Thus for given n, it makes sense to ask for the character corresponding to a particular partition, and its value at a partition:

```
gap> t:= CharTable( "Symmetric" );;
gap> t.irreducibles[1][1]( 5, [ 3, 2 ], [ 2, 2, 1 ] );
1  #  a character value of S5
gap> t.orders[1]( 5, [ 2, 1, 1, 1 ] );
2  #  a representative order in S5
```

Generic table in GAP3 means that such local evaluation is possible, so GAP3 can also deal with tables that are too big to be computed as a whole. In some cases there are methods to compute the complete table of small members G_q faster than local evaluation. If such an algorithm is part of the generic table, it will be used when the generic table is used to compute the whole table (see 50.3).

While the numbers of conjugacy classes for the series are usually not bounded, there is a fixed finite number of **types** (equivalence classes) of conjugacy classes; very often the equivalence relation is isomorphism of the centralizer of the representatives.

For each type t of classes and a fixed $q \in I$, a **parametrisation** of the classes in t is a function that assigns to each conjugacy class of G_q in t a **parameter** by which it is uniquely determined. Thus the classes are indexed by pairs (t, p_t) for a type t and a parameter p_t for that type.

There has to be a fixed number of types of irreducibles characters of G_q, too. Like the classes, the characters of each type are parametrised.

In GAP3, the parametrisations of classes and characters of the generic table is given by the record fields `classparam` and `charparam`; they are both lists of functions, each function representing the parametrisation of a type. In the specialized table, the field `classparam` contains the lists of class parameters, the character parameters are stored in the field `charparam` of the `irredinfo` records (see 49.2).

The centralizer orders, representative orders and all powermaps of the generic character table can be represented by functions in q, t and p_t; in GAP3, however, they are represented by lists of functions in q and a class parameter where each function represents a type of classes. The value of a powermap at a particular class is a pair consisting of type and parameter that specifies the image class.

The values of the irreducible characters of G_q can be represented by functions in q, class type and parameter, character type and parameter; in GAP3, they are represented by lists of lists of functions, each list of functions representing the characters of a type, the function (in q, character parameters and class parameters) representing the classes of a type in these characters.

Any generic table is a record like an ordinary character table (see 49.2). There are some fields which are used for generic tables only:

`isGenericTable`
: always `true`

`specializedname`
: function that maps q to the name of the table of G_q

`domain`
: function that returns `true` if its argument is a valid q for G_q in the series

`wholetable`
: function to construct the whole table, more efficient than the local evaluation for this purpose

The table of G_q can be constructed by specializing q and evaluating the functions in the generic table (see 50.3 and the examples given in 50.2).

The available generic tables are listed in 53.1 and 49.12.

50.2 Examples of Generic Character Tables

1. The generic table of the cyclic group:

For the cyclic group $C_q = \langle x \rangle$ of order q, there is one type of classes. The class parameters are integers $k \in \{0, \ldots, q-1\}$, the class of the parameter k consists of the group element x^k. Group order and centralizer orders are the identity function $q \mapsto q$, independent of the parameter k.

The representative order function maps (q, k) to $\frac{q}{\gcd(q,k)}$, the order of x^k in C_q; the p-th powermap is the function $(q, k, p) \mapsto [1, (kp \bmod q)]$.

There is one type of characters with parameters $l \in \{0, \ldots, q-1\}$; for e_q a primitive complex q-th root of unity, the character values are $\chi_l(x^k) = e_q^{kl}$.

The library file contains the following generic table:

```
rec(name:="Cyclic",
specializedname:=(q->ConcatenationString("C",String(q))),
order:=(n->n),
text:="generic character table for cyclic groups",
centralizers:=[function(n,k) return n;end],
classparam:=[(n->[0..n-1])],
charparam:=[(n->[0..n-1])],
powermap:=[function(n,k,pow) return [1,k*pow mod n];end],
orders:=[function(n,k) return n/Gcd(n,k);end],
irreducibles:=[[function(n,k,l) return E(n)^(k*l);end]],
domain:=(n->IsInt(n) and n>0),
libinfo:=rec(firstname:="Cyclic",othernames:=[]),
isGenericTable:=true);
```

2. The generic table of the general linear group $\mathrm{GL}(2, \mathrm{q})$:

We have four types t_1, t_2, t_3, t_4 of classes according to the rational canonical form of the elements:
t_1 scalar matrices,
t_2 nonscalar diagonal matrices,
t_3 companion matrices of $(X - \rho)^2$ for elements $\rho \in F_q^*$ and
t_4 companion matrices of irreducible polynomials of degree 2 over F_q.

The sets of class parameters of the types are in bijection with
F_q^* for t_1 and t_3, $\{\{\rho, \tau\}; \rho, \tau \in F_q^*, \rho \neq \tau\}$ for t_2 and $\{\{\epsilon, \epsilon^q\}; \epsilon \in F_{q^2} \setminus F_q\}$ for t_4.

The centralizer order functions are $q \mapsto (q^2 - 1)(q^2 - q)$ for type t_1, $q \mapsto (q-1)^2$ for type t_2, $q \mapsto q(q-1)$ for type t_3 and $q \mapsto q^2 - 1$ for type t_4.

The representative order function of t_1 maps (q, ρ) to the order of ρ in F_q, that of t_2 maps $(q, \{\rho, \tau\})$ to the least common multiple of the orders of ρ and τ.

The file contains something similar to this table:

```
rec(name:="GL2",
    specializedname:=(q->ConcatenationString("GL(2,",String(q),")")),
    order:= ( q -> (q^2-1)*(q^2-q) ),
    text:= "generic character table of GL(2,q),\
 see Robert Steinberg: The Representations of Gl(3,q), Gl(4,q),\
 PGL(3,q) and PGL(4,q), Canad. J. Math. 3 (1951)",
    classparam:= [ ( q -> [0..q-2] ), ( q -> [0..q-2] ),
                 ( q -> Combinations( [0..q-2], 2 ) ),
                 ( q -> Filtered( [1..q^2-2], x -> not (x mod (q+1) = 0)
                            and (x mod (q^2-1)) < (x*q mod (q^2-1)) ))],
    charparam:= [ ( q -> [0..q-2] ), ( q -> [0..q-2] ),
                ( q -> Combinations( [0..q-2], 2 ) ),
```

```
            ( q -> Filtered( [1..q^2-2], x -> not (x mod (q+1) = 0)
                          and (x mod (q^2-1)) < (x*q mod (q^2-1)) ))],
    centralizers := [ function( q, k ) return (q^2-1) * (q^2-q); end,
                      function( q, k ) return q^2-q; end,
                      function( q, k ) return (q-1)^2; end,
                      function( q, k ) return q^2-1; end],
    orders:= [ function( q, k ) return (q-1)/Gcd( q-1, k ); end,
               ..., ..., ... ],
    classtext:= [ ..., ..., ..., ... ],
    powermap:=
           [ function( q, k, pow ) return [1, (k*pow) mod (q-1)]; end,
             ..., ..., ... ],
    irreducibles := [[ function( q, k, l ) return E(q-1)^(2*k*l); end,
                       function( q, k, l ) return E(q-1)^(2*k*l); end,
                       ...,
                       function( q, k, l ) return E(q-1)^(k*l); end    ],
                         [ ..., ..., ..., ... ],
                         [ ..., ..., ..., ... ],
                         [ ..., ..., ..., ... ]],
    domain := ( q->IsInt(q) and q>1 and Length(Set(FactorsInt(q)))=1 ),
    isGenericTable := true )
```

50.3 CharTableSpecialized

`CharTableSpecialized(` *generic_table*, *q* `)`

returns a character table which is computed by evaluating the generic character table *generic_table* at the parameter *q*.

```
gap> t:= CharTableSpecialized( CharTable( "Cyclic" ), 5 );;
gap> PrintCharTable( t );
rec( identifier := "C5", name := "C5", size := 5, order :=
5, centralizers := [ 5, 5, 5, 5, 5 ], orders := [ 1, 5, 5, 5, 5
 ], powermap := [ ,,,, [ 1, 1, 1, 1, 1 ] ], irreducibles :=
[ [ 1, 1, 1, 1, 1 ], [ 1, E(5), E(5)^2, E(5)^3, E(5)^4 ],
  [ 1, E(5)^2, E(5)^4, E(5), E(5)^3 ],
  [ 1, E(5)^3, E(5), E(5)^4, E(5)^2 ],
  [ 1, E(5)^4, E(5)^3, E(5)^2, E(5) ] ], classparam :=
[ [ 1, 0 ], [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ], irredinfo :=
[ rec(
      charparam := [ 1, 0 ] ), rec(
      charparam := [ 1, 1 ] ), rec(
      charparam := [ 1, 2 ] ), rec(
      charparam := [ 1, 3 ] ), rec(
      charparam := [ 1, 4 ] )
 ], text := "computed using generic character table for cyclic groups"\
, classes := [ 1, 1, 1, 1, 1
 ], operations := CharTableOps, fusions := [  ], fusionsource :=
[  ], projections := [  ], projectionsource := [  ] )
```

Chapter 51

Characters

The functions described in this chapter are used to handle characters (see Chapter 49). For this, in many cases one needs maps (see Chapter 52).

There are four kinds of functions:

First, those functions which get informations about characters; they compute the scalar product of characters (see 51.1, 51.2), decomposition matrices (see 51.3, 51.4), the kernel of a character (see 51.5), p-blocks (see 51.6), Frobenius-Schur indicators (see 51.7), eigenvalues of the corresponding representations (see 51.8), and Molien series of characters (see 51.9), and decide if a character might be a permutation character candidate (see 51.26).

Second, those functions which construct characters or virtual characters, that is, differences of characters; these functions compute reduced characters (see 51.10, 51.11), tensor products (see 51.12), symmetrisations (see 51.13, 51.14, 51.15, 51.16, 51.17, 51.18), and irreducibles differences of virtual characters (see 51.19). Further, one can compute restricted characters (see 51.20), inflated characters (see 51.21), induced characters (see 51.22, 51.23), and permutation character candidates (see 51.26, 51.31).

Third, those functions which use general methods for lattices. These are the LLL algorithm to compute a lattice base consisting of short vectors (see 51.33, 51.34, 51.35), functions to compute all orthogonal embeddings of a lattice (see 51.36), and one for the special case of D_n lattices (see 51.40). A backtrack search for irreducible characters in the span of proper characters is performed by 51.38.

Finally, those functions which find special elements of parametrized characters (see 52.1); they compute the set of contained virtual characters (see 51.41) or characters (see 51.42), the set of contained vectors which possibly are virtual characters (see 51.43, 51.45) or characters (see 51.44).

51.1 ScalarProduct

`ScalarProduct(` *tbl*`,` *character1*`,` *character2* `)`

returns the scalar product of *character1* and *character2*, regarded as characters of the character table *tbl*.

```
gap> t:= CharTable( "A5" );;
```

```
gap> ScalarProduct( t, t.irreducibles[1], [ 5, 1, 2, 0, 0 ] );
1
gap> ScalarProduct( t, [ 4, 0, 1, -1, -1 ], [ 5, -1, 1, 0, 0 ] );
2/3
```

51.2 MatScalarProducts

`MatScalarProducts(` *tbl*`,` *chars1*`,` *chars2* `)`
`MatScalarProducts(` *tbl*`,` *chars* `)`

For a character table *tbl* and two lists *chars1*, *chars2* of characters, the first version returns the matrix of scalar products (see 51.1); we have

$\texttt{MatScalarProducts}(tbl, chars1, chars2)[\texttt{i}][\texttt{j}] = \texttt{ScalarProduct}(tbl, chars1[\texttt{j}], chars2[\texttt{i}])$,

i.e., row `i` contains the scalar products of *chars2*`[i]` with all characters in *chars1*.

The second form returns a lower triangular matrix of scalar products:

$\texttt{MatScalarProducts}(tbl, chars)[\texttt{i}][\texttt{j}] = \texttt{ScalarProduct}(tbl, chars[\texttt{j}], chars[\texttt{i}])$

for $\texttt{j} \leq \texttt{i}$.

```
gap> t:= CharTable( "A5" );;
gap> chars:= Sublist( t.irreducibles, [ 2 .. 4 ] );;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> MatScalarProducts( t, chars );
[ [ 2 ], [ 1, 3 ], [ 1, 2, 3 ], [ 2, 2, 1, 3 ], [ 2, 1, 2, 2, 3 ],
  [ 2, 3, 3, 3, 3, 5 ] ]
```

51.3 Decomposition

`Decomposition(` A`,` B`,` *depth* `)`
`Decomposition(` A`,` B`, "nonnegative")`

For a $m \times n$ matrix A of cyclotomics that has rank $m \leq n$, and a list B of cyclotomic vectors, each of dimension n, `Decomposition` tries to find integral solutions x of the linear equation systems x `*`A `=` B`[i]` by computing the p–adic series of hypothetical solutions.

`Decomposition(` A`,` B`,` *depth* `)`, where *depth* is a nonnegative integer, computes for every vector B`[i]` the initial part $\sum_{k=0}^{depth} x_k p^k$ (all x_k integer vectors with entries bounded by $\pm\frac{p-1}{2}$). The prime p is 83 first; if the reduction of A modulo p is singular, the next prime is chosen automatically.

A list X is returned. If the computed initial part really **is** a solution of x `*`A `=` B`[i]`, we have X`[i] =` x, otherwise X`[i] = false`.

`Decomposition(` A`,` B`, "nonnegative")` assumes that the solutions have only nonnegative entries, and that the first column of A consists of positive integers. In this case the necessary number *depth* of iterations is computed; the `i`-th entry of the returned list is `false` if there **exists** no nonnegative integral solution of the system x `*`A `=` B`[i]`, and it is the solution otherwise.

If A is singular, an error is signalled.

```
gap> a5:= CharTable( "A5" );; a5m3:= CharTable( "A5mod3" );;
gap> a5m3.irreducibles;
[ [ 1, 1, 1, 1 ], [ 3, -1, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, -1, -1 ] ]
gap> reg:= CharTableRegular( a5, 3 );;
gap> chars:= Restricted( a5, reg, a5.irreducibles );
[ [ 1, 1, 1, 1 ], [ 3, -1, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, -1, -1 ],
  [ 5, 1, 0, 0 ] ]
gap> Decomposition( a5m3.irreducibles, chars, "nonnegative" );
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ],
  [ 1, 0, 0, 1 ] ]
gap> last * a5m3.irreducibles = chars;
true
```

For the subroutines of `Decomposition`, see 51.4.

51.4 Subroutines of Decomposition

Let A be a square integral matrix, p an odd prime. The reduction of A modulo p is $\overline{A}$, its entries are chosen in the interval $[-\frac{p-1}{2}, \frac{p-1}{2}]$. If $\overline{A}$ is regular over the field with p elements, we can form $A' = \overline{A}^{-1}$. Now consider the integral linear equation system $xA = b$, i.e., we look for an integral solution x. Define $b_0 = b$, and then iteratively compute

$$x_i = (b_i A') \bmod p, \quad b_{i+1} = \frac{1}{p}(b_i - x_i A) \text{ for} i = 0, 1, 2, \ldots \ .$$

By induction, we get

$$p^{i+1} b_{i+1} + (\sum_{j=0}^{i} p^j x_j) A = b.$$

If there is an integral solution x, it is unique, and there is an index l such that b_{l+1} is zero and $x = \sum_{j=0}^{l} p^j x_j$.

There are two useful generalizations. First, A need not be square; it is only necessary that there is a square regular matrix formed by a subset of columns. Second, A does not need to be integral; the entries may be cyclotomic integers as well, in this case one has to replace each column of A by the columns formed by the coefficients (which are integers). Note that this preprocessing must be performed compatibly for A and b.

And these are the subroutines called by `Decomposition`:

`LinearIndependentColumns(` *A* `)`

returns for a matrix A a maximal list *lic* of positions such that the rank of `List(` *A*`, x -> Sublist( x,` *lic* `) )` is the same as the rank of A.

`InverseMatMod(` *A*`,` *p* `)`

returns for a square integral matrix A and a prime p a matrix A' with the property that $A'A$ is congruent to the identity matrix modulo p; if A is singular modulo p, `false` is returned.

`PadicCoefficients(` *A*`,` *Amodpinv*`,` *b*`,` *p*`,` *depth* `)`

returns the list $[x_0, x_1, \ldots, x_l, b_{l+1}]$ where $l =$ *depth* or l is minimal with the property that $b_{l+1} = 0$.

`IntegralizedMat(` *A* `)`
`IntegralizedMat(` *A*`,` *inforec* `)`

return for a matrix A of cyclotomics a record *intmat* with components `mat` and `inforec`. Each family of galois conjugate columns of A is encoded in a set of columns of the rational matrix *intmat*`.mat` by replacing cyclotomics by their coefficients. *intmat*`.inforec` is a record containing the information how to encode the columns.

If the only argument is A, the component `inforec` is computed that can be entered as second argument *inforec* in a later call of `IntegralizedMat` with a matrix B that shall be encoded compatible with A.

`DecompositionInt(` *A*`,` *B*`,` *depth* `)`

does the same as `Decomposition` (see 51.3), but only for integral matrices A, B, and non-negative integers *depth*.

51.5 KernelChar

`KernelChar(` *char* `)`

returns the set of classes which form the kernel of the character *char*, i.e. the set of positions i with *char*$[i] =$ *char*$[1]$.

For a factor fusion map *fus*, `KernelChar(` *fus* `)` is the kernel of the epimorphism.

```
gap> s4:= CharTable( "Symmetric", 4 );;
gap> s4.irreducibles;
[ [ 1, -1, 1, 1, -1 ], [ 3, -1, -1, 0, 1 ], [ 2, 0, 2, -1, 0 ],
  [ 3, 1, -1, 0, -1 ], [ 1, 1, 1, 1, 1 ] ]
gap> List( last, KernelChar );
[ [ 1, 3, 4 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]
```

51.6 PrimeBlocks

`PrimeBlocks(` *tbl*`,` *prime* `)`
`PrimeBlocks(` *tbl*`,` *chars*`,` *prime* `)`

For a character table *tbl* and a prime *prime*, `PrimeBlocks(` *tbl*`,` *chars*`,` *prime* `)` returns a record with fields

`block`
: a list of positive integers which has the same length as the list *chars* of characters, the entry n at position i in that list means that *chars*`[`*i*`]` belongs to the n-th *prime*-block

`defect`
: a list of nonnegative integers, the value at position i is the defect of the i–th block.

`PrimeBlocks(` *tbl*`,` *prime* `)` does the same for *chars* `=` *tbl*`.irreducibles`, and additionally the result is stored in the `irredinfo` field of *tbl*.

```
gap> t:= CharTable( "A5" );;
gap> PrimeBlocks( t, 2 ); PrimeBlocks( t, 3 ); PrimeBlocks( t, 5 );
rec(
  block := [ 1, 1, 1, 2, 1 ],
  defect := [ 2, 0 ] )
rec(
  block := [ 1, 2, 3, 1, 1 ],
  defect := [ 1, 0, 0 ] )
rec(
  block := [ 1, 1, 1, 1, 2 ],
  defect := [ 1, 0 ] )
gap> InverseMap( last.block ); # distribution of characters to blocks
[ [ 1, 2, 3, 4 ], 5 ]
```

If `InfoCharTable2 = Print`, the defects of the blocks and the heights of the contained characters are printed.

51.7 Indicator

`Indicator(` *tbl*`,` *n* `)`
`Indicator(` *tbl*`,` *chars*`,` *n* `)`
`Indicator(` *modtbl*`, 2 )`

For a character table *tbl*, `Indicator(` *tbl*`,` *chars*`,` *n* `)` returns the list of n-th Frobenius Schur indicators for the list *chars* of characters.

`Indicator(` *tbl*`,` *n* `)` does the same for *chars* `=` *tbl*`.irreducibles`, and additionally the result is stored in the field `irredinfo` of *tbl*.

`Indicator(` *modtbl*`, 2 )` returns the list of 2nd indicators for the irreducible characters of the Brauer character table *modtbl* and stores the indicators in the `irredinfo` component of *modtbl*; this does not work for tables in characteristic 2.

```
gap> t:= CharTable( "M11" );; Indicator( t, t.irreducibles, 2 );
[ 1, 1, 0, 0, 1, 0, 0, 1, 1, 1 ]
```

51.8 Eigenvalues

`Eigenvalues(` *tbl*`,` *char*`,` *class* `)`

Let M be a matrix of a representation with character *char* which is a character of the table *tbl*, for an element in the conjugacy class *class*. `Eigenvalues(` *tbl*`,` *char*`,` *class* `)` returns a list of length n `=` *tbl*`.orders[` *class* `]` where at position `i` the multiplicity of `E(n)^i =` $e^{\frac{2\pi i}{n}}$ as eigenvalue of M is stored.

```
gap> t:= CharTable( "A5" );;
gap> chi:= t.irreducibles[2];
[ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ]
```

```
gap> List( [ 1 .. 5 ], i -> Eigenvalues( t, chi, i ) );
[ [ 3 ], [ 2, 1 ], [ 1, 1, 1 ], [ 0, 1, 1, 0, 1 ], [ 1, 0, 0, 1, 1 ] ]
```

`List( [1..`*n*`], i -> E(n)^i) * Eigenvalues(`*tbl*`,`*char*`,`*class*`) )` is equal to *char*`[` *class* `]`.

51.9 MolienSeries

`MolienSeries(` *psi* `)`
`MolienSeries(` *psi*`,` *chi* `)`
`MolienSeries(` *tbl*`,` *psi* `)`
`MolienSeries(` *tbl*`,` *psi*`,` *chi* `)`

returns a record that describes the series

$$M_{\psi,\chi}(z) = \sum_{d=0}^{\infty} (\chi, \psi^{[d]}) z^d$$

where $\psi^{[d]}$ denotes the symmetrization of ψ with the trivial character of the symmetric group S_d (see 51.14).

psi and *chi* must be characters of the table *tbl*, the default for χ is the trivial character. If no character table is given, *psi* and *chi* must be class function recods.

`ValueMolienSeries(` *series*`,` *i* `)`

returns the *i*-th coefficient of the Molien series *series*.

```
gap> psi:= Irr( CharTable( "A5" ) )[3];
Character( CharTable( "A5" ),
[ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] )
gap> mol:= MolienSeries( psi );;
gap> List( [ 1 .. 10 ], i -> ValueMolienSeries( mol, i ) );
[ 0, 1, 0, 1, 0, 2, 0, 2, 0, 3 ]
```

The record returned by `MolienSeries` has components

`summands`
: a list of records with components `numer`, `r`, and `k`, describing the summand `numer`$/(1-z^r)^k$,

`size`
: the `size` component of the character table,

`degree`
: the degree of *psi*.

51.10 Reduced

`Reduced(` *tbl*`,` *constituents*`,` *reducibles* `)`
`Reduced(` *tbl*`,` *reducibles* `)`

returns a record with fields `remainders` and `irreducibles`, both lists: Let `rems` be the set of nonzero characters obtained from *reducibles* by subtraction of

$$\sum_{\chi \in \texttt{constituents}} \frac{\texttt{ScalarProduct}(tbl, \chi, reducibles[\texttt{i}])}{\texttt{ScalarProduct}(tbl, \chi, constituents[\texttt{j}])} \cdot \chi$$

from *reducibles*`[i]` in the first case or subtraction of

$$\sum_{j \leq i} \frac{\texttt{ScalarProduct}(tbl, reducibles[\texttt{j}], reducibles[\texttt{i}])}{\texttt{ScalarProduct}(tbl, reducibles[\texttt{j}], reducibles[\texttt{j}])} \cdot reducibles[j]$$

in the second case.

Let `irrs` be the list of irreducible characters in `rems`. `rems` is reduced with `irrs` and all found irreducibles until no new irreducibles are found. Then `irreducibles` is the set of all found irreducible characters, `remainders` is the set of all nonzero remainders.

If one knows that *reducibles* are ordinary characters of *tbl* and *constituents* are irreducible ones, 51.11 `ReducedOrdinary` may be faster.

Note that elements of `remainders` may be only virtual characters even if *reducibles* are ordinary characters.

```
gap> t:= CharTable( "A5" );;
gap> chars:= Sublist( t.irreducibles, [ 2 .. 4 ] );;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> Reduced( t, chars );
rec(
  remainders := [  ],
  irreducibles :=
   [ [ 1, 1, 1, 1, 1 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
      [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, 1, -1, -1 ],
      [ 5, 1, -1, 0, 0 ] ] )
```

51.11 ReducedOrdinary

`ReducedOrdinary(` *tbl*`,` *constituents*`,` *reducibles* `)`

works like 51.10 `Reduced`, but assumes that the elements of *constituents* and *reducibles* are ordinary characters of the character table *tbl*. So scalar products are calculated only for those pairs of characters where the degree of the constituent is smaller than the degree of the reducible.

51.12 Tensored

`Tensored(` *chars1*`,` *chars2* `)`

returns the list of tensor products (i.e. pointwise products) of all characters in the list *chars1* with all characters in the list *chars2*.

```
gap> t:= CharTable( "A5" );;
gap> chars1:= Sublist( t.irreducibles, [ 1 .. 3 ] );;
gap> chars2:= Sublist( t.irreducibles, [ 2 .. 3 ] );;
```

```
gap> Tensored( chars1, chars2 );
[ [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ],
  [ 9, 1, 0, -2*E(5)-E(5)^2-E(5)^3-2*E(5)^4,
      -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4 ], [ 9, 1, 0, -1, -1 ],
  [ 9, 1, 0, -1, -1 ],
  [ 9, 1, 0, -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4, -2*E(5)-E(5)^2-E(5)^3
          -2*E(5)^4 ] ]
```

Note that duplicate tensor products are not deleted; the tensor product of *chars1* `[`*i*`]` with *chars2* `[`*j*`]` is stored at position $(i-1)$`Length`(*chars1*) $+ j$.

51.13 Symmetrisations

`Symmetrisations(` *tbl*`,` *chars*`,` *Sn* `)`
`Symmetrisations(` *tbl*`,` *chars*`,` *n* `)`

returns the list of nonzero symmetrisations of the characters *chars*, regarded as characters of the character table *tbl*, with the ordinary characters of the symmetric group of degree n; alternatively, the table of the symmetric group can be entered as *Sn*.

The symmetrisation $\chi^{[\lambda]}$ of the character χ of *tbl* with the character λ of the symmetric group S_n of degree n is defined by

$$\chi^{[\lambda]}(g) = \frac{1}{n!} \sum_{\rho \in S_n} \lambda(\rho) \prod_{k=1}^{n} \chi(g^k)^{a_k(\rho)},$$

where $a_k(\rho)$ is the number of cycles of length k in ρ.

For special symmetrisations, see 51.14, 51.15, 51.16 and 51.17, 51.18.

```
gap> t:= CharTable( "A5" );;
gap> chars:= Sublist( t.irreducibles, [ 1 .. 3 ] );;
gap> Symmetrisations( t, chars, 3 );
[ [ 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1 ], [ 8, 0, -1, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 10, -2, 1, 0, 0 ], [ 1, 1, 1, 1, 1 ],
  [ 8, 0, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 10, -2, 1, 0, 0 ] ]
```

Note that the returned list may contain zero characters, and duplicate characters are not deleted.

51.14 SymmetricParts

`SymmetricParts(` *tbl*`,` *chars*`,` *n* `)`

returns the list of symmetrisations of the characters *chars*, regarded as characters of the character table *tbl*, with the trivial character of the symmetric group of degree n (see 51.13).

```
gap> t:= CharTable( "A5" );;
gap> SymmetricParts( t, t.irreducibles, 3 );
[ [ 1, 1, 1, 1, 1 ], [ 10, -2, 1, 0, 0 ], [ 10, -2, 1, 0, 0 ],
  [ 20, 0, 2, 0, 0 ], [ 35, 3, 2, 0, 0 ] ]
```

51.15 AntiSymmetricParts

`AntiSymmetricParts(` *tbl*, *chars*, *n* `)`

returns the list of symmetrisations of the characters *chars*, regarded as characters of the character table *tbl*, with the alternating character of the symmetric group of degree n (see 51.13).

```
gap> t:= CharTable( "A5" );;
gap> AntiSymmetricParts( t, t.irreducibles, 3 );
[ [ 0, 0, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ],
  [ 4, 0, 1, -1, -1 ], [ 10, -2, 1, 0, 0 ] ]
```

51.16 MinusCharacter

`MinusCharacter(` *char*, *prime_powermap*, *prime* `)`

returns the (possibly parametrized, see chapter 52) character χ^{p-} for the character $\chi =$ *char* and a prime $p =$ *prime*, where χ^{p-} is defined by $\chi^{p-}(g) = (\chi(g)^p - \chi(g^p))/p$, and *prime_powermap* is the (possibly parametrized) p-th powermap.

```
gap> t:= CharTable( "S7" );; pow:= InitPowermap( t, 2 );;
gap> Congruences( t, t.irreducibles, pow, 2 );; pow;
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, 1, [ 2, 9, 10 ], 3, 4, 6,
  [ 7, 12 ] ]
gap> chars:= Sublist( t.irreducibles, [ 2 .. 5 ] );;
gap> List( chars, x-> MinusCharacter( x, pow, 2 ) );
[ [ 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, [ 0, 1 ] ],
  [ 15, -1, 3, 0, [ -2, -1, 0 ], 0, -1, 1, 5, -3, [ 0, 1, 2 ], -1, 0,
      0, [ 0, 1 ] ],
  [ 15, -1, 3, 0, [ -1, 0, 2 ], 0, -1, 1, 5, -3, [ 1, 2, 4 ], -1, 0,
      0, 1 ],
  [ 190, -2, 1, 1, [ 0, 2 ], 0, 1, 1, -10, -10, [ 0, 2 ], -1, -1, 0,
      [ -1, 0 ] ] ]
```

51.17 OrthogonalComponents

`OrthogonalComponents(` *tbl*, *chars*, *m* `)`

If χ is a (nonlinear) character with indicator $+1$, a splitting of the tensor power χ^m is given by the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible constituents than the symmetrizations with the symmetric group of degree m (see 51.13).

`OrthogonalComponents` returns the set of orthogonal symmetrisations of the characters of the character table *tbl* in the list *chars*, up to the power m, where the integer m is one of $\{2, 3, 4, 5, 6\}$.

Note: It is not checked if all characters in *chars* do really have indicator $+1$; if there are characters with indicator 0 or -1, the result might contain virtual characters, see also 51.18.

The Murnaghan functions are implemented as in [Fra82].

```
gap> t:= CharTable( "A8" );; chi:= t.irreducibles[2];
```

```
[ 7, -1, 3, 4, 1, -1, 1, 2, 0, -1, 0, 0, -1, -1 ]
gap> OrthogonalComponents( t, [ chi ], 4 );
[ [ 21, -3, 1, 6, 0, 1, -1, 1, -2, 0, 0, 0, 1, 1 ],
  [ 27, 3, 7, 9, 0, -1, 1, 2, 1, 0, -1, -1, -1, -1 ],
  [ 105, 1, 5, 15, -3, 1, -1, 0, -1, 1, 0, 0, 0, 0 ],
  [ 35, 3, -5, 5, 2, -1, -1, 0, 1, 0, 0, 0, 0, 0 ],
  [ 77, -3, 13, 17, 2, 1, 1, 2, 1, 0, 0, 0, 2, 2 ],
  [ 189, -3, -11, 9, 0, 1, 1, -1, 1, 0, 0, 0, -1, -1 ],
  [ 330, -6, 10, 30, 0, -2, -2, 0, -2, 0, 1, 1, 0, 0 ],
  [ 168, 8, 8, 6, -3, 0, 0, -2, 2, -1, 0, 0, 1, 1 ],
  [ 35, 3, -5, 5, 2, -1, -1, 0, 1, 0, 0, 0, 0, 0 ],
  [ 182, 6, 22, 29, 2, 2, 2, 2, 1, 0, 0, 0, -1, -1 ] ]
```

51.18 SymplecticComponents

`SymplecticComponents(` *tbl*`,` *chars*`,` *m* `)`

If χ is a (nonlinear) character with indicator -1, a splitting of the tensor power χ^m is given in terms of the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible constituents than the symmetrizations with the symmetric group of degree m (see 51.13).

`SymplecticComponents` returns the set of symplectic symmetrisations of the characters of the character table *tbl* in the list *chars*, up to the power m, where the integer m is one of $\{2, 3, 4, 5\}$.

Note: It is not checked if all characters in *chars* do really have indicator -1; if there are characters with indicator 0 or $+1$, the result might contain virtual characters, see also 51.17.

```
gap> t:= CharTable( "U3(3)" );; chi:= t.irreducibles[2];
[ 6, -2, -3, 0, -2, -2, 2, 1, -1, -1, 0, 0, 1, 1 ]
gap> SymplecticComponents( t, [ chi ], 4 );
[ [ 14, -2, 5, -1, 2, 2, 2, 1, 0, 0, 0, 0, -1, -1 ],
  [ 21, 5, 3, 0, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1 ],
  [ 64, 0, -8, -2, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 ],
  [ 14, 6, -4, 2, -2, -2, 2, 0, 0, 0, 0, 0, -2, -2 ],
  [ 56, -8, 2, 2, 0, 0, 0, -2, 0, 0, 0, 0, 0, 0 ],
  [ 70, -10, 7, 1, 2, 2, 2, -1, 0, 0, 0, 0, -1, -1 ],
  [ 189, -3, 0, 0, -3, -3, -3, 0, 0, 0, 1, 1, 0, 0 ],
  [ 90, 10, 9, 0, -2, -2, -2, 1, -1, -1, 0, 0, 1, 1 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 126, 14, -9, 0, 2, 2, 2, -1, 0, 0, 0, 0, -1, -1 ] ]
```

51.19 IrreducibleDifferences

`IrreducibleDifferences(` *tbl*`,` *chars1*`,` *chars2* `)`
`IrreducibleDifferences(` *tbl*`,` *chars1*`,` *chars2*`,` *scprmat* `)`
`IrreducibleDifferences(` *tbl*`,` *chars*`, "triangle")`
`IrreducibleDifferences(` *tbl*`,` *chars*`, "triangle",` *scprmat* `)`

returns the list of irreducible characters which occur as difference of two elements of *chars* (if `"triangle"` is specified) or of an element of *chars1* and an element of *chars2*; if *scprmat* is not specified it will be computed (see 51.2), otherwise we must have

$$scprmat[\mathtt{i}][\mathtt{j}] = \mathtt{ScalarProduct}(tbl, chars[\mathtt{i}], chars[\mathtt{j}])$$

resp.

$$scprmat[\mathtt{i}][\mathtt{j}] = \mathtt{ScalarProduct}(tbl, chars1[\mathtt{i}], chars2[\mathtt{j}])$$

.

```
gap> t:= CharTable( "A5" );;
gap> chars:= Sublist( t.irreducibles, [ 2 .. 4 ] );;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> IrreducibleDifferences( t, chars, "triangle" );
[ [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ]
```

51.20 Restricted

`Restricted(` *tbl*, *subtbl*, *chars* `)`
`Restricted(` *tbl*, *subtbl*, *chars*, *specification* `)`
`Restricted(` *chars*, *fusionmap* `)`

returns the restrictions, i.e. the indirections, of the characters in the list *chars* by a subgroup fusion map. This map can either be entered directly as *fusionmap*, or it must be stored on the character table *subtbl* and must have destination *tbl*; in the latter case the value of the `specification` field of the desired fusion may be specified as *specification* (see 49.45). If no such fusion is stored, `false` is returned.

The fusion map may be a parametrized map (see 52.1); any value that is not uniquely determined in a restricted character is set to an unknown (see 17.1); for parametrized indirection of characters, see 52.2.

Restriction and inflation are the same procedures, so `Restricted` and `Inflated` are identical, see 51.21.

```
gap> s5:= CharTable( "A5.2" );; a5:= CharTable( "A5" );;
gap> Restricted( s5, a5, s5.irreducibles );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ], [ 6, -2, 0, 1, 1 ],
  [ 4, 0, 1, -1, -1 ], [ 4, 0, 1, -1, -1 ], [ 5, 1, -1, 0, 0 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> Restricted( s5.irreducibles, [ 1, 6, 2, 6 ] );
                          # restrictions to the cyclic group of order 4
[ [ 1, 1, 1, 1 ], [ 1, -1, 1, -1 ], [ 6, 0, -2, 0 ], [ 4, 0, 0, 0 ],
  [ 4, 0, 0, 0 ], [ 5, -1, 1, -1 ], [ 5, 1, 1, 1 ] ]
```

51.21 Inflated

`Inflated(` *factortbl*, *tbl*, *chars* `)`
`Inflated(` *factortbl*, *tbl*, *chars*, *specification* `)`
`Inflated(` *chars*, *fusionmap* `)`

returns the inflations, i.e. the indirections of *chars* by a factor fusion map. This map can either be entered directly as *fusionmap*, or it must be stored on the character table *tbl* and must have destination *factortbl*; in the latter case the value of the `specification` field of the desired fusion may be specified as *specification* (see 49.45). If no such fusion is stored, `false` is returned.

The fusion map may be a parametrized map (see 52.1); any value that is not uniquely determined in an inflated character is set to an unknown (see 17.1); for parametrized indirection of characters, see 52.2.

Restriction and inflation are the same procedures, so `Restricted` and `Inflated` are identical, see 51.20.

```
gap> s4:= CharTable( "Symmetric", 4 );;
gap> s3:= CharTableFactorGroup( s4, [3] );;
gap> s3.irreducibles;
[ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ] ]
gap> s4.fusions;
[ rec(
      map := [ 1, 2, 1, 3, 2 ],
      type := "factor",
      name := [ 'S', '4', '/', '[', ' ', '3', ' ', ']' ] ) ]
gap> Inflated( s3, s4, s3.irreducibles );
[ [ 1, -1, 1, 1, -1 ], [ 2, 0, 2, -1, 0 ], [ 1, 1, 1, 1, 1 ] ]
```

51.22 Induced

`Induced(` *subtbl*, *tbl*, *chars* `)`
`Induced(` *subtbl*, *tbl*, *chars*, *specification* `)`
`Induced(` *subtbl*, *tbl*, *chars*, *fusionmap* `)`

returns a set of characters induced from *subtbl* to *tbl*; the elements of the list *chars* will be induced. The subgroup fusion map can either be entered directly as *fusionmap*, or it must be stored on the table *subtbl* and must have destination *tbl*; in the latter case the value of the `specification` field may be specified by *specification* (see 49.45). If no such fusion is stored, `false` is returned.

The fusion map may be a parametrized map (see 52.1); any value that is not uniquely determined in an induced character is set to an unknown (see 17.1).

```
gap> Induced( a5, s5, a5.irreducibles );
[ [ 2, 2, 2, 2, 0, 0, 0 ], [ 6, -2, 0, 1, 0, 0, 0 ],
  [ 6, -2, 0, 1, 0, 0, 0 ], [ 8, 0, 2, -2, 0, 0, 0 ],
  [ 10, 2, -2, 0, 0, 0, 0 ] ]
```

51.23 InducedCyclic

`InducedCyclic(` *tbl* `)`
`InducedCyclic(` *tbl*, `"all")`
`InducedCyclic(` *tbl*, *classes* `)`
`InducedCyclic(` *tbl*, *classes*, `"all")`

returns a set of characters of the character table *tbl*. They are characters induced from cyclic subgroups of *tbl*. If `"all"` is specified, all irreducible characters of those subgroups are induced, otherwise only the permutation characters are computed. If a list *classes* is specified, only those cyclic subgroups generated by these classes are considered, otherwise all classes of *tbl* are considered.

Note that the powermaps for primes dividing *tbl*`.order` must be stored on *tbl*; if any powermap for a prime not dividing *tbl*`.order` that is smaller than the maximal representative order is not stored, this map will be computed (see 52.12) and stored afterwards.

The powermaps may be parametrized maps (see 52.1); any value that is not uniquely determined in an induced character is set to an unknown (see 17.1). The representative orders of the classes to induce from must not be parametrized (see 52.1).

```
gap> t:= CharTable( "A5" );; InducedCyclic( t, "all" );
[ [ 12, 0, 0, 2, 2 ], [ 12, 0, 0, E(5)^2+E(5)^3, E(5)+E(5)^4 ],
  [ 12, 0, 0, E(5)+E(5)^4, E(5)^2+E(5)^3 ], [ 20, 0, -1, 0, 0 ],
  [ 20, 0, 2, 0, 0 ], [ 30, -2, 0, 0, 0 ], [ 30, 2, 0, 0, 0 ],
  [ 60, 0, 0, 0, 0 ] ]
```

51.24 CollapsedMat

`CollapsedMat(` *mat*`,` *maps* `)`

returns a record with fields `mat` and `fusion`: The `fusion` field contains the fusion that collapses the columns of *mat* that are identical also for all maps in the list *maps*, the `mat` field contains the image of *mat* under that fusion.

```
gap> t.irreducibles;
[ [ 1, 1, 1, 1, 1 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> t:= CharTable( "A5" );; RationalizedMat( t.irreducibles );
[ [ 1, 1, 1, 1, 1 ], [ 6, -2, 0, 1, 1 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
gap> CollapsedMat( last, [] );
rec(
  mat := [ [ 1, 1, 1, 1 ], [ 6, -2, 0, 1 ], [ 4, 0, 1, -1 ],
      [ 5, 1, -1, 0 ] ],
  fusion := [ 1, 2, 3, 4, 4 ] )
gap> Restricted( last.mat, last.fusion );
[ [ 1, 1, 1, 1, 1 ], [ 6, -2, 0, 1, 1 ], [ 4, 0, 1, -1, -1 ],
  [ 5, 1, -1, 0, 0 ] ]
```

51.25 Power

`Power(` *powermap*`,` *chars*`,` *n* `)`

returns the list of indirections of the characters *chars* by the n-th powermap; for a character χ in *chars*, this indirection is often called $\chi^{(n)}$. The powermap is calculated from the (necessarily stored) powermaps of the prime divisors of n if it is not stored in *powermap* (see 52.30).

Note that $\chi^{(n)}$ is in general only a virtual characters.

```
gap> t:= CharTable( "A5" );; Power( t.powermap, t.irreducibles, 2 );
[ [ 1, 1, 1, 1, 1 ], [ 3, 3, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ],
  [ 3, 3, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ], [ 4, 4, 1, -1, -1 ],
  [ 5, 5, -1, 0, 0 ] ]
gap> MatScalarProducts( t, t.irreducibles, last );
[ [ 1, 0, 0, 0, 0 ], [ 1, -1, 0, 0, 1 ], [ 1, 0, -1, 0, 1 ],
  [ 1, -1, -1, 1, 1 ], [ 1, -1, -1, 0, 2 ] ]
```

51.26 Permutation Character Candidates

For groups H, G with $H \leq G$, the induced character $(1_G)^H$ is called the **permutation character** of the operation of G on the right cosets of H. If only the character table of G is known, one can try to get informations about possible subgroups of G by inspection of those characters π which might be permutation characters, using that such a character must have at least the following properties:

$\pi(1)$ divides $|G|$,

$[\pi, \psi] \leq \psi(1)$ for each character ψ of G,

$[\pi, 1_G] = 1$,

$\pi(g)$ is a nonnegative integer for $g \in G$,

$\pi(g)$ is smaller than the centralizer order of g for $1 \neq g \in G$,

$\pi(g) \leq \pi(g^m)$ for $g \in G$ and any integer m,

$\pi(g) = 0$ for every $|g|$ not diving $\frac{|G|}{\pi(1)}$,

$\pi(1)|N_G(g)|$ divides $|G|\pi(g)$, where $|N_G(g)|$ denotes the normalizer order of $\langle g \rangle$.

Any character with these properties will be called a **permutation character candidate** from now on.

GAP3 provides some algorithms to compute permutation character candidates, see 51.31. Some information about the subgroup can computed from a permutation character using `PermCharInfo` (see 51.28).

51.27 IsPermChar

`IsPermChar(` *tbl*`,` *pi* `)`

missing, like tests `TestPerm1`, `TestPerm2`, `TestPerm3`

51.28 PermCharInfo

`PermCharInfo(` *tbl*`,` *permchars* `)`

Let *tbl* be the character table of the group G, and *permchars* the permutation character $(1_U)^G$ for a subgroup U of G, or a list of such characters. `PermCharInfo` returns a record with components

`contained`

a list containing for each character in *permchars* a list containing at position *i* the number of elements of U that are contained in class i of *tbl*, this is equal to `permchar`$[i]|U|/$`tbl.centralizers`$[i]$,

`bound`
: Let *permchars*`[k]` be the permutation character $(1_U)^G$. Then the class length in U of an element in class i of *tbl* must be a multiple of the value `bound`$[k][i] = |U|/\gcd(|U|,$ *tbl*`.centralizers`$[i])$,

`display`
: a record that can be used as second argument of `DisplayCharTable` to display the permutation characters and the corresponding components `contained` and `bound`, for the classes where at least one character of *permchars* is nonzero,

`ATLAS`
: list of strings containing for each character in *permchars* the decomposition into *tbl*`.irreducibles` in ATLAS notation.

```
gap> t:= CharTable("A6");;
gap> PermCharInfo( t, [ 15, 3, 0, 3, 1, 0, 0 ] );
rec(
  contained := [ [ 1, 9, 0, 8, 6, 0, 0 ] ],
  bound := [ [ 1, 3, 8, 8, 6, 24, 24 ] ],
  display := rec(
      classes := [ 1, 2, 4, 5 ],
      chars := [ [ 15, 3, 0, 3, 1, 0, 0 ], [ 1, 9, 0, 8, 6, 0, 0 ],
          [ 1, 3, 8, 8, 6, 24, 24 ] ],
      letter := "I" ),
  ATLAS := [ "1a+5b+9a" ] )
gap> DisplayCharTable( t, last.display );
A6

     2  3  3  .  2
     3  2  .  2  .
     5  1  .  .  .

       1a 2a 3b 4a
    2P 1a 1a 3b 2a
    3P 1a 2a 1a 4a
    5P 1a 2a 3b 4a

I.1    15  3  3  1
I.2     1  9  8  6
I.3     1  3  8  6
```

51.29 Inequalities

`Inequalities(` *tbl* `)`

The condition $\pi(g) \geq 0$ for every permutation character candidate π places restrictions on the multiplicities a_i of the irreducible constituents χ_i of $\pi = \sum_{i=1}^{r} a_i\chi_i$. For every group element g holds $\sum_{i=1}^{r} a_i\chi_i(g) \geq 0$. The power map provides even stronger conditions.

This system of inequalities is kind of diagonalized, resulting in a system of inequalities restricting a_i in terms of $a_j, j < i$. These inequalities are used to construct characters with nonnegative values (see 51.31). `PermChars` either calls `Inequalities` or takes this information from the record field `ineq` of its argument record.

The number of inequalities arising in the process of diagonalization may grow very strong.

There are two strategies to perform this diagonalization. The default is to simply eliminate one unknown a_i after the other with decreasing i. In some cases it turns out to be better first to look which choice for the next unknown will yield the fewest new inequalities.

51.30 PermBounds

`PermBounds(` *tbl*`,` *d* `)`

All characters π satisfying $\pi(g) > 0$ and $\pi(1) = d$ for a given degree d lie in a simplex described by these conditions. `PermBounds` computes the boundary points of this simplex for $d = 0$, from which the boundary points for any other d are easily derived. Some conditions from the powermap are also involved.

For this purpose a matrix similar to the rationalized character table has to be inverted.

These boundary points are used by `PermChars` (see 51.31) to construct all permutation character candidates of a given degree. `PermChars` either calls `PermBounds` or takes this information from the record field `bounds` of its argument record.

51.31 PermChars

`PermChars(` *tbl* `)`
`PermChars(` *tbl*`,` *degree* `)`
`PermChars(` *tbl*`,` *arec* `)`

GAP3 provides several algorithms to determine permutation character candidates from a given character table. The algorithm is selected from the choice of the record fields of the optional argument record *arec*. The user is encouraged to try different approaches especially if one choice fails to come to an end.

Regardless of the algorithm used in a special case, `PermChars` returns a list of **all** permutation character candidates with the properties given in *arec*. There is no guarantee that a character of this list is in fact a permutation character. But an empty list always means there is no permutation character with these properties (e.g. of a certain degree).

In the first form `PermChars(` *tbl* `)` returns the list of all permutation characters of the group with character table *tbl*. This list might be rather long for big groups, and it might take much time. The algorithm depends on a preprocessing step, where the inequalities arising from the condition $\pi(g) \leq 0$ are transformed into a system of inequalities that guides the search (see 51.29).

```
gap> m11:= CharTable("M11");;
gap> PermChars(m11);;      # will return the list of 39 permutation
                           # character candidates of M11.
```

There are two different search strategies for this algorithm. One simply constructs all characters with nonnegative values and then tests for each such character whether its degree is a divisor of the order of the group. This is the default. The other strategy uses the inequalities to predict if it is possible to find a character of a certain degree in the currently searched part of the search tree. To choose this strategy set the field `mode` of *arec* to `"preview"` and the field `degree` to the degree (or a list of degrees which might be all divisors of the order of the group) you want to look for. The record field `ineq` can take the inequalities from `Inequalities` if they are needed more than once.

In the second form `PermChars(` *tbl*`,` *degree* `)` returns the list of all permutation characters of degree *degree*. For that purpose a preprocessing step is performed where essentially the rationalized character table is inverted in order to determine boundary points for the simplex in which the permutation character candidates of a given degree must lie (see 51.30). Note that inverting big integer matrices needs a lot of time and space. So this preprocessing is restricted to groups with less than 100 classes, say.

```
gap> PermChars(m11, 220);
[ [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ],
  [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ],
  [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ]
```

In the third form `PermChars(` *tbl*`,` *arec* `)` returns the list of all permutation characters which have the properties given in the argument record *arec*. If *arec* contains a degree in the record field `degree` then `PermChars` will behave exactly as in the second form.

```
gap> PermChars(m11, rec(degree:= 220));
[ [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ],
  [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ],
  [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ]
```

Alternatively *arec* may have the record fields `chars` and `torso`. *arec*`.chars` is a list of (in most cases all) **rational** irreducible characters of *tbl* which might be constituents of the required characters, and *arec*`.torso` is a list that contains some known values of the required characters at the right positions.

Note: At least the degree *arec*`.torso[1]` must be an integer. If *arec*`.chars` does not contain all rational irreducible characters of G, it may happen that any scalar product of π with an omitted character is negative; there should be nontrivial reasons for excluding a character that is known to be not a constituent of π.

```
gap> rat:= RationalizedMat(m11.irreducibles);;
gap> PermChars(m11, rec(torso:= [220], chars:= rat));
[ [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ],
  [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ],
  [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ] ]
gap> PermChars(m11, rec(torso:= [220,,,,,2], chars:= rat));
[ [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ]
```

51.32 Faithful Permutation Characters

`PermChars(` *tbl*`,` *arec* `)`

`PermChars` may as well determine faithful candidates for permutation characters. In that case *arec* requires the fields `normalsubgrp`, `nonfaithful`, `chars`, `lower`, `upper`, and `torso`.

Let *tbl* be the character table of the group G, *arec*.`normalsubgrp` a list of classes forming a normal subgroup N of G. *arec*.`nonfaithful` is a permutation character candidate (see 51.26) of G with kernel N. *arec*.`chars` is a list of (in most cases all) rational irreducible characters of *tbl*.

`PermChars` computes all those permutation character candidates π having following properties:

> *arec*.`chars` contains every rational irreducible constituent of π.
>
> $\pi[i] \geq$ *arec*.`lower`$[i]$ for all integer values of the list *arec*.`lower`.
>
> $\pi[i] \leq$ *arec*.`upper`$[i]$ for all integer values of the list *arec*.`upper`.
>
> $\pi[i] =$ *arec*.`torso`$[i]$ for all integer values of the list *arec*.`torso`.
>
> No irreducible constituent of $\pi -$ *arec*.`nonfaithful` has N in its kernel.

If there exists a subgroup V of G, $V \geq N$, with *nonfaithful* $= (1_V)^G$, the last condition means that the candidates for those possible subgroups U with $V = UN$ are constructed.

Note:At least the degree *torso*[1] must be an integer. If *chars* does not contain all rational irreducible characters of G, it may happen that any scalar product of π with an omitted character is negative; there should be nontrivial reasons for excluding a character that is known to be not a constituent of π.

51.33 LLLReducedBasis

`LLLReducedBasis([`*L*`],`*vectors*`[,`*y*`][,"linearcomb"])`

`LLLReducedBasis` provides an implementation of the LLL lattice reduction algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

`LLLReducedBasis` returns a record whose component `basis` is a list of LLL reduced linearly independent vectors spanning the same lattice as the list *vectors*.

L must be a lattice record whose scalar product function is stored in the component `operations.NoMessageScalarProduct` or `operations.ScalarProduct`. It must be a function of three arguments, namely the lattice and the two vectors. If no lattice L is given the standard scalar product is taken.

In the case of option `"linearcomb"`, the record contains also the components `relations` and `transformation`, which have the following meaning. `relations` is a basis of the relation space of *vectors*, i.e., of vectors x such that x `*`*vectors* is zero. `transformation` gives the expression of the new lattice basis in terms of the old, i.e., `transformation *`*vectors* equals the `basis` component of the result.

Another optional argument is y, the "sensitivity"of the algorithm, a rational number between $\frac{1}{4}$ and 1 (the default value is $\frac{3}{4}$).

(The function 51.34 computes an LLL reduced Gram matrix.)

```
gap> vectors:= [ [ 9, 1, 0, -1, -1 ], [ 15, -1, 0, 0, 0 ],
>                [ 16, 0, 1, 1, 1 ], [ 20, 0, -1, 0, 0 ],
>                [ 25, 1, 1, 0, 0 ] ];;
gap> LLLReducedBasis( vectors, "linearcomb" );
rec(
```

```
basis :=
 [ [ 1, 1, 1, 1, 1 ], [ 1, 1, -2, 1, 1 ], [ -1, 3, -1, -1, -1 ],
    [ -3, 1, 0, 2, 2 ] ],
relations := [ [ -1, 0, -1, 0, 1 ] ],
transformation :=
 [ [ 0, -1, 1, 0, 0 ], [ -1, -2, 0, 2, 0 ], [ 1, -2, 0, 1, 0 ],
    [ -1, -2, 1, 1, 0 ] ] )
```

51.34 LLLReducedGramMat

`LLLReducedGramMat(` G `[,`y`] )`

`LLLReducedGramMat` provides an implementation of the LLL lattice reduction algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

Let G the Gram matrix of the vectors $(b_1, b_2, \ldots, b_n)$; this means G is either a square symmetric matrix or lower triangular matrix (only the entries in the lower triangular half are used by the program).

`LLLReducedGramMat` returns a record whose component `remainder` is the Gram matrix of the LLL reduced basis corresponding to $(b_1, b_2, \ldots, b_n)$. If G was a lower triangular matrix then also the `remainder` component is a lower triangular matrix.

The result record contains also the components `relations` and `transformation`, which have the following meaning.

`relations` is a basis of the space of vectors $(x_1, x_2, \ldots, x_n)$ such that $\sum_{i=1}^{n} x_i b_i$ is zero, and `transformation` gives the expression of the new lattice basis in terms of the old, i.e., `transformation` is the matrix T such that $T \cdot G \cdot T^{tr}$ is the `remainder` component of the result.

The optional argument y denotes the "sensitivity of the algorithm, it must be a rational number between $\frac{1}{4}$ and 1; the default value is $y = \frac{3}{4}$.

(The function 51.33 computes an LLL reduced basis.)

```
gap> g:= [ [ 4, 6, 5, 2, 2 ], [ 6, 13, 7, 4, 4 ],
>    [ 5, 7, 11, 2, 0 ], [ 2, 4, 2, 8, 4 ], [ 2, 4, 0, 4, 8 ] ];;
gap> LLLReducedGramMat( g );
rec(
  remainder :=
   [ [ 4, 2, 1, 2, -1 ], [ 2, 5, 0, 2, 0 ], [ 1, 0, 5, 0, 2 ],
      [ 2, 2, 0, 8, 2 ], [ -1, 0, 2, 2, 7 ] ],
  relation := [  ],
  transformation :=
   [ [ 1, 0, 0, 0, 0 ], [ -1, 1, 0, 0, 0 ], [ -1, 0, 1, 0, 0 ],
      [ 0, 0, 0, 1, 0 ], [ -2, 0, 1, 0, 1 ] ],
  scalarproducts := [ , [ 1/2 ], [ 1/4, -1/8 ], [ 1/2, 1/4, -2/25 ],
      [ -1/4, 1/8, 37/75, 8/21 ] ],
  bsnorms := [ 4, 4, 75/16, 168/25, 32/7 ] )
```

51.35 LLL

`LLL(` *tbl*`,` *characters* `[,` *y*`] [, "sort"] [, "linearcomb"] )`

calls the LLL algorithm (see 51.33) in the case of lattices spanned by (virtual) characters *characters* of the character table *tbl* (see 51.1). By finding shorter vectors in the lattice spanned by *characters*, i.e. virtual characters of smaller norm, in some cases `LLL` is able to find irreducible characters.

`LLL` returns a record with at least components `irreducibles` (the list of found irreducible characters), `remainders` (a list of reducible virtual characters), and `norms` (the list of norms of `remainders`). `irreducibles` together with `remainders` span the same lattice as *characters*.

There are some optional parameters:

y
: controls the sensitivity of the algorithm; the value of *y* must be between 1/4 and 1, the default value is 3/4.

`"sort"`
: `LLL` sorts *characters* and the `remainders` component of the result according to the degrees.

`"linearcomb"`
: The returned record contains components `irreddecomp` and `reddecomp` which are decomposition matrices of `irreducibles` and `remainders`, with respect to *characters*.

```
gap> s4:= CharTable( "Symmetric", 4 );;
gap> chars:= [ [ 8, 0, 0, -1, 0 ], [ 6, 0, 2, 0, 2 ],
>     [ 12, 0, -4, 0, 0 ], [ 6, 0, -2, 0, 0 ], [ 24, 0, 0, 0, 0 ],
>     [ 12, 0, 4, 0, 0 ], [ 6, 0, 2, 0, -2 ], [ 12, -2, 0, 0, 0 ],
>     [ 8, 0, 0, 2, 0 ], [ 12, 2, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ] ];;
gap> LLL( s4, chars );
rec(
  irreducibles :=
   [ [ 2, 0, 2, -1, 0 ], [ 1, 1, 1, 1, 1 ], [ 3, 1, -1, 0, -1 ],
      [ 3, -1, -1, 0, 1 ], [ 1, -1, 1, 1, -1 ] ],
  remainders := [  ],
  norms := [  ] )
```

51.36 OrthogonalEmbeddings

`OrthogonalEmbeddings(` *G* `[, "positive"] [,` *maxdim* `] )`

computes all possible orthogonal embeddings of a lattice given by its Gram matrix G which must be a regular matrix (see 51.34). In other words, all solutions X of the problem

$$X^{tr}X = G$$

are calculated (see [Ple90]). Usually there are many solutions X but all their rows are chosen from a small set of vectors, so `OrthogonalEmbeddings` returns the solutions in an encoded form, namely as a record with components

`vectors`
: the list $[x_1, x_2, \ldots, x_n]$ of vectors that may be rows of a solution; these are exactly those vectors that fulfill the condition $x_i G^{-1} x_i^{tr} \leq 1$ (see 51.37), and we have $G = \sum_{i=1}^{n} x_i^{tr} x_i$,

`norms`
: the list of values $x_i G^{-1} x_i^{tr}$, and

`solutions`
: a list S of lists; the i–th solution matrix is `Sublist(` L`,` S`[`i`] )`, so the dimension of the i–th solution is the length of S`[`i`]`.

The optional argument `"positive"` will cause `OrthogonalEmbeddings` to compute only vectors x_i with nonnegative entries. In the context of characters this is allowed (and useful) if G is the matrix of scalar products of ordinary characters.

When `OrthogonalEmbeddings` is called with the optional argument *maxdim* (a positive integer), it computes only solutions up to dimension *maxdim*; this will accelerate the algorithm in some cases.

G may be the matrix of scalar products of some virtual characters. From the characters and the embedding given by the matrix X, `Decreased` (see 51.39) may be able to compute irreducibles.

```
gap> b := [ [  3, -1, -1 ], [ -1,  3, -1 ], [ -1, -1,  3 ] ];;
gap> c:=OrthogonalEmbeddings(b);
rec(
  vectors :=
   [ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ], [ -1, 1, 0 ],
       [ -1, 0, 1 ], [ 1, 0, 0 ], [ 0, -1, 1 ], [ 0, 1, 0 ],
       [ 0, 0, 1 ] ],
  norms := [ 1, 1, 1, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2 ],
  solutions := [ [ 1, 2, 3 ], [ 1, 6, 6, 7, 7 ], [ 2, 5, 5, 8, 8 ],
       [ 3, 4, 4, 9, 9 ], [ 4, 5, 6, 7, 8, 9 ] ] )
gap> Sublist( c.vectors, c.solutions[1] );
[ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ] ]
```

`OrthogonalEmbeddingsSpecialDimension`
`(` *tbl*`,` *reducibles*`,` *grammat* `[, "positive"],` *dim* `)`

This form can be used if you want to find irreducible characters of the table *tbl*, where *reducibles* is a list of virtual characters, *grammat* is the matrix of their scalar products, and *dim* is the maximal dimension of an embedding. First all solutions up to *dim* are compute, and then 51.39 `Decreased` is called in order to find irreducible characters of *tab*.

If *reducibles* consists of ordinary characters only, you should enter the optional argument `"positive"`; this imposes some conditions on the possible embeddings (see the description of `OrthogonalEmbeddings`).

`OrthogonalEmbeddingsSpecialDimension` returns a record with components

`irreducibles`
: a list of found irreducibles, the intersection of all lists of irreducibles found by `Decreased`, for all possible embeddings, and

`remainders`
: a list of remaining reducible virtual characters

```
gap> s6:= CharTable( "Symmetric", 6 );;
gap> b:= InducedCyclic( s6, "all" );;
gap> Add( b, [1,1,1,1,1,1,1,1,1,1,1] );
gap> c:= LLL( s6, b ).remainders;;
gap> g:= MatScalarProducts( s6, c, c );;
gap> d:= OrthogonalEmbeddingsSpecialDimension( s6, c, g, 8 );
rec(
  irreducibles :=
   [ [ 5, -3, 1, 1, 2, 0, -1, -1, -1, 0, 1 ], [ 5, 1, 1, -3, -1, 1,
          2, -1, -1, 0, 0 ], [ 10, -2, -2, 2, 1, 1, 1, 0, 0, 0, -1 ],
      [ 10, 2, -2, -2, 1, -1, 1, 0, 0, 0, 1 ] ],
  remainders :=
   [ [ 0, 4, 0, -4, 3, 1, -3, 0, 0, 0, -1 ], [ 4, 0, 0, 4, -2, 0, 1,
          -2, 2, -1, 1 ], [ 6, 2, 2, -2, 3, -1, 0, 0, 0, 1, -2 ],
      [ 14, 6, 2, 2, 2, 0, -1, 0, 0, -1, -1 ] ] )
```

51.37 ShortestVectors

`ShortestVectors(` *G*`,` *m* `)`
`ShortestVectors(` *G*`,` *m*`, "positive")`

computes all vectors x with $xGx^{tr} \leq m$, where G is a matrix of a symmetric bilinear form, and m is a nonnegative integer. If the optional argument `"positive"` is entered, only those vectors x with nonnegative entries are computed.

`ShortestVectors` returns a record with components

`vectors` the list of vectors x, and

`norms` the list of their norms according to the Gram matrix G.

```
gap> g:= [ [ 2, 1, 1 ], [ 1, 2, 1 ], [ 1, 1, 2 ] ];;
gap> ShortestVectors(g,4);
rec(
  vectors := [ [ -1, 1, 1 ], [ 0, 0, 1 ], [ -1, 0, 1 ], [ 1, -1, 1 ],
      [ 0, -1, 1 ], [ -1, -1, 1 ], [ 0, 1, 0 ], [ -1, 1, 0 ],
      [ 1, 0, 0 ] ],
  norms := [ 4, 2, 2, 4, 2, 4, 2, 2, 2 ] )
```

This algorithm is used in 51.36 `OrthogonalEmbeddings`.

51.38 Extract

`Extract(` *tbl*`,` *reducibles*`,` *grammat* `)`
`Extract(` *tbl*`,` *reducibles*`,` *grammat*`,` *missing* `)`

tries to find irreducible characters by drawing conclusions out of a given matrix *grammat* of scalar products of the reducible characters in the list *reducibles*, which are characters of the table *tbl*. `Extract` uses combinatorial and backtrack means.

Note: `Extract` works only with ordinary characters!

missing number of characters missing to complete the *tbl* perhaps `Extract` may be accelerated by the specification of *missing*.

`Extract` returns a record *extr* with components `solution` and `choice` where `solution` is a list $[M_1, \ldots, M_n]$ of decomposition matrices that satisfy the equation

$$M_i^{tr} \cdot X = \texttt{Sublist}(\mathit{reducibles}, \mathit{extr}.\texttt{choice}[\texttt{i}]) \ ,$$

for a matrix X of irreducible characters, and `choice` is a list of length n whose entries are lists of indices.

So each column stands for one of the reducible input characters, and each row stands for an irreducible character. You can use 51.39 `Decreased` to examine the solution for computable irreducibles.

```
gap> s4 := CharTable( "Symmetric", 4 );;
gap> y := [ [ 5, 1, 5, 2, 1 ], [ 2, 0, 2, 2, 0 ], [ 3, -1, 3, 0, -1 ],
>  [ 6, 0, -2, 0, 0 ], [ 4, 0, 0, 1, 2 ] ];;
gap> g := MatScalarProducts( s4, y, y );
[ [ 6, 3, 2, 0, 2 ], [ 3, 2, 1, 0, 1 ], [ 2, 1, 2, 0, 0 ],
  [ 0, 0, 0, 2, 1 ], [ 2, 1, 0, 1, 2 ] ]
gap> e:= Extract( s4, y, g, 5 );
rec(
  solution :=
   [ [ [ 1, 1, 0, 0, 2 ], [ 1, 0, 1, 0, 1 ], [ 0, 1, 0, 1, 0 ],
          [ 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0 ] ] ],
  choice := [ [ 2, 5, 3, 4, 1 ] ] )
#  continued in Decreased ( see 51.39 )
```

51.39 Decreased

`Decreased(` *tbl*`,` *reducibles*`,` *mat* `)`
`Decreased(` *tbl*`,` *reducibles*`,` *mat*`,` *choice* `)`

tries to solve the output of 51.36 `OrthogonalEmbeddings` or 51.38 `Extract` in order to find irreducible characters. *tbl* must be a character table, *reducibles* the list of characters used for the call of `OrtgogonalEmbeddings` or `Extract`, *mat* one solution, and in the case of a solution returned by `Extract`, `choice` must be the corresponding `choice` component.

`Decreased` returns a record with components

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the remaining reducible characters, and

`matrix`
: the decomposition matrix of the characters in the `remainders` component, which could not be solved.

```
#  see example in 51.38 Extract
gap> d := Decreased( s4, y, e.solution[1], e.choice[1] );
rec(
  irreducibles :=
   [ [ 1, 1, 1, 1, 1 ], [ 3, -1, -1, 0, 1 ], [ 1, -1, 1, 1, -1 ],
      [ 3, 1, -1, 0, -1 ], [ 2, 0, 2, -1, 0 ] ],
  remainders := [  ],
  matrix := [  ] )
```

51.40 DnLattice

`DnLattice(` *tbl*, *grammat*, *reducibles* `)`

tries to find sublattices isomorphic to root lattices of type D_n (for $n \geq 5$ or $n = 4$) in a lattice that is generated by the norm 2 characters *reducibles*, which must be characters of the table *tbl*. *grammat* must be the matrix of scalar products of *reducibles*, i.e., the Gram matrix of the lattice.

`DnLattice` is able to find irreducible characters if there is a lattice with $n > 4$. In the case $n = 4$ `DnLattice` only in some cases finds irreducibles.

`DnLattice` returns a record with components

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the list of remaining reducible characters, and

`gram`
: the Gram matrix of the characters in `remainders`.

The remaining reducible characters are transformed into a normalized form, so that the lattice-structure is cleared up for further treatment. So `DnLattice` might be useful even if it fails to find irreducible characters.

```
gap> tbl:= CharTable( "Symmetric", 4 );;
gap> y1:=[ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ], [ 5, -1, 1, -1, 1 ],
>          [ -1, 1, 3, -1, -1 ] ];;
gap> g1:= MatScalarProducts( tbl, y1, y1 );
[ [ 2, 1, 0, 0 ], [ 1, 2, 1, -1 ], [ 0, 1, 2, 0 ], [ 0, -1, 0, 2 ] ]
gap> e:= DnLattice( tbl, g1, y1 );
rec(
  gram := [  ],
  remainders := [  ],
  irreducibles :=
   [ [ 2, 0, 2, -1, 0 ], [ 1, -1, 1, 1, -1 ], [ 1, 1, 1, 1, 1 ],
      [ 3, -1, -1, 0, 1 ] ] )
```

`DnLatticeIterative(` *tbl*, *arec* `)`

was made for iterative use of `DnLattice`. *arec* must be either a list of characters of the table *tbl*, or a record with components

`remainders`
: a list of characters of the character table *tbl*, and

`norms`
: the norms of the characters in `remainders`,

e.g., a record returned by 51.35 `LLL`. `DnLatticeIterative` will select the characters of norm 2, call `DnLattice`, reduce the characters with found irreducibles, call `DnLattice` for the remaining characters, and so on, until no new irreducibles are found.

`DnLatticeIterative` returns (like 51.35 `LLL`) a record with components

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the list of remaining reducible characters, and

`norms`
: the list of norms of the characters in `remainders`.

```
gap> tbl:= CharTable( "Symmetric", 4 );;
gap> y1:= [ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ],
>   [ 5, -1, 1, -1, 1 ], [ -1, 1, 3, -1, -1 ], [ 6, -2, 2, 0, 0 ] ];;
gap> DnLatticeIterative( tbl, y1);
rec(
  irreducibles :=
   [ [ 2, 0, 2, -1, 0 ], [ 1, -1, 1, 1, -1 ], [ 1, 1, 1, 1, 1 ],
      [ 3, -1, -1, 0, 1 ] ],
  remainders := [  ],
  norms := [  ] )
```

51.41 ContainedDecomposables

`ContainedDecomposables(` *constituents*`,` *moduls*`,` *parachar*`,` *func* `)`

For a list of **rational** characters *constituents* and a parametrized rational character *parachar* (see 52.1), the set of all elements χ of *parachar* is returned that satisfy *func*(χ) (i.e., for that `true` is returned) and that "modulo *moduls* lie in the lattice spanned by *constituents*". This means they lie in the lattice spanned by *constituents* and the set $\{moduls[i] \cdot e_i; 1 \leq i \leq n\}$, where n is the length of *parachar* and e_i is the i-th vector of the standard base.

```
gap> hs:= CharTable("HS");; s:= CharTable("HSM12");; s.identifier;
"5:4xa5"
gap> rat:= RationalizedMat(s.irreducibles);;
gap> fus:= InitFusion( s, hs );
[ 1, [ 2, 3 ], [ 2, 3 ], [ 2, 3 ], 4, 5, 5, [ 5, 6, 7 ], [ 5, 6, 7 ],
  9, [ 8, 9 ], [ 8, 9 ], [ 8, 9, 10 ], [ 8, 9, 10 ], [ 11, 12 ],
  [ 17, 18 ], [ 17, 18 ], [ 17, 18 ], 21, 21, 22, [ 23, 24 ],
  [ 23, 24 ], [ 23, 24 ], [ 23, 24 ] ]
# restrict a rational character of hs by fus,
# see chapter 52:
gap> rest:= CompositionMaps( hs.irreducibles[8], fus );
[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ],
  [ -1, 15 ], 1, [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ],
  [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]
# all vectors in the lattice:
gap> ContainedDecomposables( rat, s.centralizers, rest, x -> true );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
```

```
        0, 1, 0, 0, 0, 0 ],
    [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
        0, 1, 0, 0, 0, 0 ] ]
#  better filter, only characters (see 51.42):
gap> ContainedDecomposables( rat, s.centralizers, rest,
>                  x->NonnegIntScalarProducts(s,s.irreducibles,x) );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
        0, 1, 0, 0, 0, 0 ],
    [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
        0, 1, 0, 0, 0, 0 ] ]
```

An application of `ContainedDecomposables` is 51.42 `ContainedCharacters`.

For another strategy that works also for irrational characters, see 51.43.

51.42 ContainedCharacters

`ContainedCharacters(` *tbl*`,` *constituents*`,` *parachar* `)`

returns the set of all characters contained in the parametrized rational character *parachar* (see 52.1), that modulo centralizer orders lie in the linear span of the **rational** characters *constituents* of the character table *tbl* and that have nonnegative integral scalar products with all elements of *constituents*.

Note: This does not imply that an element of the returned list is necessary a linear combination of *constituents*.

```
gap> s:= CharTable( "HSM12" );; hs:= CharTable( "HS" );;
gap> rat:= RationalizedMat( s.irreducibles );;
gap> fus:= InitFusion( s, hs );;
gap> rest:= CompositionMaps( hs.irreducibles[8], fus );;
gap> ContainedCharacters( s, rat, rest );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
        0, 1, 0, 0, 0, 0 ],
    [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
        0, 1, 0, 0, 0, 0 ] ]
```

`ContainedCharacters` calls 51.41 `ContainedDecomposables`.

51.43 ContainedSpecialVectors

`ContainedSpecialVectors(` *tbl*`,` *chars*`,` *parachar*`,` *func* `)`

returns the list of all elements *vec* of the parametrized character *parachar* (see 52.1), that have integral norm and integral scalar product with the principal character of the character table *tbl* and that satisfy *func*`(` *tbl*`,` *chars*`,` *vec* `)`, i.e., for that `true` is returned.

```
gap> s:= CharTable( "HSM12" );; hs:= CharTable( "HS" );;
gap> fus:= InitFusion( s, hs );;
gap> rest:= CompositionMaps( hs.irreducibles[8], fus );;
#  no further condition:
gap> ContainedSpecialVectors( s, s.irreducibles, rest,
>                       function(tbl,chars,vec) return true; end );;
```

```
gap> Length( last );
24
# better filter: those with integral scalar products
gap> ContainedSpecialVectors( s, s.irreducibles, rest,
>                             IntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
# better filter: the scalar products must be nonnegative
gap> ContainedSpecialVectors( s, s.irreducibles, rest,
>                             NonnegIntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
```

Special cases of `ContainedSpecialVectors` are 51.44 `ContainedPossibleCharacters` and 51.45 `ContainedPossibleVirtualCharacters`.

`ContainedSpecialVectors` successively examines all vectors contained in *parachar*, thus it might not be useful if the indeterminateness exceeds 10^6. For another strategy that works for rational characters only, see 51.41.

51.44 ContainedPossibleCharacters

`ContainedPossibleCharacters(` *tbl*, *chars*, *parachar* `)`

returns the list of all elements *vec* of the parametrized character *parachar* (see 52.1), which have integral norm and integral scalar product with the principal character of the character table *tbl* and nonnegative integral scalar product with all elements of the list *chars* of characters of *tbl*.

```
# see example in 51.43
gap> ContainedPossibleCharacters( s, s.irreducibles, rest );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
```

`ContainedPossibleCharacters` calls 51.43 `ContainedSpecialVectors`.

`ContainedPossibleCharacters` successively examines all vectors contained in *parachar*, thus it might not be useful if the indeterminateness exceeds 10^6. For another strategy that works for rational characters only, see 51.41.

51.45 ContainedPossibleVirtualCharacters

`ContainedPossibleVirtualCharacters(` *tbl*, *chars*, *parachar* `)` returns the list of all elements *vec* of the parametrized character *parachar* (see 52.1), which have integral norm and integral scalar product with the principal character of the character table *tbl* and integral scalar product with all elements of the list *chars* of characters of *tbl*.

```
#  see example in 51.43
gap> ContainedPossibleVirtualCharacters( s, s.irreducibles, rest );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0,
      0, 1, 0, 0, 0, 0 ] ]
```

`ContainedPossibleVirtualCharacters` calls 51.43 `ContainedSpecialVectors`.

`ContainedPossibleVirtualCharacters` successively examines all vectors that are contained in *parachar*, thus it might not be useful if the indeterminateness exceeds 10^6. For another strategy that works for rational characters only, see 51.41.

Chapter 52

Maps and Parametrized Maps

In this chapter, first the data structure of (parametrized) maps is introduced (see 52.1). Then a description of several functions which mainly deal with parametrized maps follows; these are

basic operations with paramaps (see 52.2, 52.3, 52.29, 52.4, 52.5, 52.6, 52.7, 52.8, 52.9),

functions which inform about ambiguity with respect to a paramap (see 52.10, 52.11),

functions used for the construction of powermaps and subgroup fusions (see 52.12, 52.13 and their subroutines 52.14, 52.15, 52.16, 52.17, 52.23, 52.18, 52.19, 52.20, 52.21, 52.22, 52.24, 52.26, 52.25, 52.28, 52.27, 52.30) and

the function 52.31.

52.1 More about Maps and Parametrized Maps

Besides the characters, **powermaps** are an important part of a character table. Often their computation is not easy, and in general they cannot be obtained from the matrix of irreducible characters, so it is useful to store them on the table.

If not only a single table is considered but different tables of groups and subgroups are used, also **subgroup fusion maps** must be known to get informations about the embedding or simply to induce or restrict characters.

These are examples of class functions which are called **maps** for short; in GAP3, maps are lists: Characters are maps, the lists of element orders, centralizer orders, classlengths are maps, and for a permutation *perm* of classes, `ListPerm(` *perm* `)` is a map.

When maps are constructed, in most cases one only knows that the image of any class is contained in a set of possible images, e.g. that the image of a class under a subgroup fusion is in the set of all classes with the same element order. Using further informations, like centralizer orders, powermaps and the restriction of characters, the sets of possible images can be diminished. In many cases, at the end the images are uniquely determined.

For this, many functions do not only work with maps but with **parametrized maps** (or short paramaps): These are lists whose entries are either the images themselves (i.e. integers for fusion maps, powermaps, element orders etc. and cyclotomics for characters) or lists of possible images. Thus maps are special paramaps. A paramap *paramap* can be identified with the set of all maps *map* with *map*`[i] =` *paramap*`[i]` or *map*`[i]` contained in *paramap*`[i]`; we say that *map* is contained in *paramap* then.

The composition of two paramaps is defined as the paramap that contains all compositions of elements of the paramaps. For example, the indirection of a character by a parametrized subgroup fusion map is the parametrized character that contains all possible restrictions of that character.

52.2 CompositionMaps

`CompositionMaps(` *paramap2* `,` *paramap1* `)`
`CompositionMaps(` *paramap2* `,` *paramap1* `,` *class* `)`

For parametrized maps *paramap1* and *paramap2* where *paramap*`[i]` is a bound position or a set of bound positions in *paramap2*, `CompositionMaps(` *paramap2* `,` *paramap1* `)` is a parametrized map with image `CompositionMaps(` *paramap2* `,` *paramap1* `,` *class* `)` at position *class*.

If *paramap1* `[` *class* `]` is unique, we have

$$\texttt{CompositionMaps}(\mathit{paramap2}, \mathit{paramap1}, \mathit{class}) = \mathit{paramap2}[\mathit{paramap1}[\mathit{class}]],$$

otherwise it is the union of *paramap2*`[i]` for `i` in *paramap1* `[` *class* `]`.

```
gap> map1:= [ 1, [ 2, 3, 4 ], [ 4, 5 ], 1 ];;
gap> map2:= [ [ 1, 2 ], 2, 2, 3, 3 ];;
gap> CompositionMaps( map2, map1 ); CompositionMaps( map1, map2 );
[ [ 1, 2 ], [ 2, 3 ], 3, [ 1, 2 ] ]
[ [ 1, 2, 3, 4 ], [ 2, 3, 4 ], [ 2, 3, 4 ], [ 4, 5 ], [ 4, 5 ] ]
```

Note: If you want to get indirections of characters which contain unknowns (see chapter 17) instead of sets of possible values, use 52.29 `Indirected`.

52.3 InverseMap

`InverseMap(` *paramap* `)`

`InverseMap(` *paramap* `)[i]` is the unique preimage or the set of all preimages of `i` under *paramap*, if there are any; otherwise it is unbound.

(We have `CompositionMaps(` *paramap* `, InverseMap(` *paramap* `) )` the identity map.)

```
gap> t:= CharTable( "2.A5" );; f:= CharTable( "A5" );;
gap> fus:= GetFusionMap( t, f );      # the factor fusion map
[ 1, 1, 2, 3, 3, 4, 4, 5, 5 ]
gap> inverse:= InverseMap( fus );
[ [ 1, 2 ], 3, [ 4, 5 ], [ 6, 7 ], [ 8, 9 ] ]
gap> CompositionMaps( fus, inverse );
[ 1, 2, 3, 4, 5 ]
gap> t.powermap[2];
[ 1, 1, 2, 4, 4, 8, 8, 6, 6 ]
# transfer a powermap up to the factor group:
```

```
gap> CompositionMaps( fus, CompositionMaps( last, inverse ) );
[ 1, 1, 3, 5, 4 ]                    # is equal to f.powermap[2]
# transfer a powermap down to the group:
gap> CompositionMaps( inverse, CompositionMaps( last, fus ) );
[ [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], [ 4, 5 ], [ 4, 5 ], [ 8, 9 ],
  [ 8, 9 ], [ 6, 7 ], [ 6, 7 ] ]     # contains t.powermap[2]
```

52.4 ProjectionMap

`ProjectionMap(` *map* `)`

For each image `i` under the (necessarily **not** parametrized) map *map*, `ProjectionMap(` *map* `)[`*i*`]` is the smallest preimage of *i*.

(We have `CompositionMaps(` *map*`, ProjectionMap(` *map* `) )` the identity map.)

```
gap> ProjectionMap( [1,1,1,2,2,2,3,4,5,5,5,6,6,6,7,7,7] );
[ 1, 4, 7, 8, 9, 12, 15 ]
```

52.5 Parametrized

`Parametrized(` *list* `)`

returns the parametrized cover of *list*, i.e. the parametrized map with smallest indeterminateness that contains all maps in *list*. `Parametrized` is the inverse function of 52.6 in the sense that `Parametrized( ContainedMaps(` *paramap* `) ) =` *paramap*.

```
gap> Parametrized( [ [ 1, 3, 4, 6, 8, 10, 11, 11, 15, 14 ],
>                    [ 1, 3, 4, 6, 8, 10, 11, 11, 14, 15 ],
>                    [ 1, 3, 4, 7, 8, 10, 12, 12, 15, 14 ],
>                    [ 1, 3, 4, 7, 8, 10, 12, 12, 14, 15 ] ] );
[ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ], [ 14, 15 ],
  [ 14, 15 ] ]
```

52.6 ContainedMaps

`ContainedMaps(` *paramap* `)`

returns the set of all maps contained in the parametrized map *paramap*. `ContainedMaps` is the inverse function of 52.5 in the sense that `Parametrized( ContainedMaps(` *paramap* `) ) =` *paramap*.

```
gap> ContainedMaps( [ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ],
> 14, 15 ] );
[ [ 1, 3, 4, 6, 8, 10, 11, 11, 14, 15 ],
  [ 1, 3, 4, 6, 8, 10, 11, 12, 14, 15 ],
  [ 1, 3, 4, 6, 8, 10, 12, 11, 14, 15 ],
  [ 1, 3, 4, 6, 8, 10, 12, 12, 14, 15 ],
  [ 1, 3, 4, 7, 8, 10, 11, 11, 14, 15 ],
  [ 1, 3, 4, 7, 8, 10, 11, 12, 14, 15 ],
  [ 1, 3, 4, 7, 8, 10, 12, 11, 14, 15 ],
  [ 1, 3, 4, 7, 8, 10, 12, 12, 14, 15 ] ]
```

52.7 UpdateMap

`UpdateMap(` *char*, *paramap*, *indirected* `)`

improves the paramap *paramap* using that *indirected* is the (possibly parametrized) indirection of the character *char* by *paramap*.

```
gap> s:= CharTable( "S4(4).2" );; he:= CharTable( "He" );;
gap> fus:= InitFusion( s, he );
[ 1, 2, 2, [ 2, 3 ], 4, 4, [ 7, 8 ], [ 7, 8 ], 9, 9, 9, [ 10, 11 ],
  [ 10, 11 ], 18, 18, 25, 25, [ 26, 27 ], [ 26, 27 ], 2, [ 6, 7 ],
  [ 6, 7 ], [ 6, 7, 8 ], 10, 10, 17, 17, 18, [ 19, 20 ], [ 19, 20 ] ]
gap> Filtered( s.irreducibles, x -> x[1] = 50 );
[ [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0, 0, 1, 1, 0, 0, 0, 0, -1, -1,
      10, 2, 2, 2, 1, 1, 0, 0, 0, -1, -1 ],
  [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0, 0, 1, 1, 0, 0, 0, 0, -1, -1,
      -10, -2, -2, -2, -1, -1, 0, 0, 0, 1, 1 ] ]
gap> UpdateMap( he.irreducibles[2], fus, last[1] + s.irreducibles[1] );
gap> fus;
[ 1, 2, 2, 3, 4, 4, 8, 7, 9, 9, 9, 10, 10, 18, 18, 25, 25,
  [ 26, 27 ], [ 26, 27 ], 2, [ 6, 7 ], [ 6, 7 ], [ 6, 7 ], 10, 10,
  17, 17, 18, [ 19, 20 ], [ 19, 20 ] ]
```

52.8 CommutativeDiagram

`CommutativeDiagram(` *paramap1*, *paramap2*, *paramap3*, *paramap4* `)`
`CommutativeDiagram(` *paramap1*, *paramap2*, *paramap3*, *paramap4*, *improvements* `)`

If

$$\texttt{CompositionMaps}(\mathit{paramap2}, \mathit{paramap1}) = \texttt{CompositionMaps}(\mathit{paramap4}, \mathit{paramap3})$$

shall hold, the consistency is checked and the four maps will be improved according to this condition.

If a record *improvements* with fields `imp1`, `imp2`, `imp3`, `imp4` (all lists) is entered as parameter, only diagrams containing elements of `imp`i as positions in the i-th paramap are considered.

`CommutativeDiagram` returns `false` if an inconsistency was found, otherwise a record is returned that contains four lists `imp1`, ..., `imp4`, where `imp`i is the list of classes where the i-th paramap was improved.

```
gap> map1:= [ [ 1, 2, 3 ], [ 1, 3 ] ];;
gap> map2:= [ [ 1, 2 ], 1, [ 1, 3 ] ];;
gap> map3:= [ [ 2, 3 ], 3 ];; map4:= [ , 1, 2, [ 1, 2 ] ];;
gap> CommutativeDiagram( map1, map2, map3, map4 );
rec(
  imp1 := [ 2 ],
  imp2 := [ 1 ],
  imp3 := [  ],
  imp4 := [  ] )
```

```
gap> imp:= last;; map1; map2; map3; map4;
[ [ 1, 2, 3 ], 1 ]
[ 2, 1, [ 1, 3 ] ]
[ [ 2, 3 ], 3 ]
[ , 1, 2, [ 1, 2 ] ]
gap> CommutativeDiagram( map1, map2, map3, map4, imp );
rec(
  imp1 := [  ],
  imp2 := [  ],
  imp3 := [  ],
  imp4 := [  ] )
```

52.9 TransferDiagram

`TransferDiagram(` *inside1* `,` *between* `,` *inside2* `)`
`TransferDiagram(` *inside1* `,` *between* `,` *inside2* `,` *improvements* `)`

Like in 52.8, it is checked that

$$\texttt{CompositionMaps}(\mathit{between}, \mathit{inside1}) = \texttt{CompositionMaps}(\mathit{inside2}, \mathit{between})$$

holds for the paramaps *inside1*, *between* and *inside2*, that means the paramap *between* occurs twice in each commutative diagram.

Additionally, 52.20 `CheckFixedPoints` is called.

If a record *improvements* with fields `impinside1`, `impbetween` and `impinside2` is specified, only those diagrams with elements of `impinside1` as positions in *inside1*, elements of `impbetween` as positions in *between* or elements of `impinside2` as positions in *inside2* are considered.

When an inconsistency occurs, the program immediately returns `false`; else it returns a record with lists `impinside1`, `impbetween` and `impinside2` of found improvements.

```
gap> s:= CharTable( "2F4(2)" );; ru:= CharTable( "Ru" );;
gap> fus:= InitFusion( s, ru );;
gap> permchar:= Sum( Sublist( ru.irreducibles, [ 1, 5, 6 ] ) );;
gap> CheckPermChar( s, ru, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> TransferDiagram( s.powermap[2], fus, ru.powermap[2] );
rec(
  impinside1 := [  ],
  impbetween := [ 12, 23 ],
  impinside2 := [  ] )
gap> TransferDiagram( s.powermap[3], fus, ru.powermap[3] );
rec(
  impinside1 := [  ],
  impbetween := [ 14, 24, 25 ],
  impinside2 := [  ] )
gap> TransferDiagram( s.powermap[2], fus, ru.powermap[2], last );
```

```
rec(
  impinside1 := [  ],
  impbetween := [  ],
  impinside2 := [  ] )
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ],
  [ 25, 26 ], 5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27,
  27 ]
```

52.10 Indeterminateness

`Indeterminateness(` *paramap* `)`

returns the indeterminateness of *paramap*, i.e. the number of maps contained in the parametrized map *paramap*

```
gap> m11:= CharTable( "M11" );; m12:= CharTable( "M12" );;
gap> fus:= InitFusion( m11, m12 );
[ 1, [ 2, 3 ], [ 4, 5 ], [ 6, 7 ], 8, [ 9, 10 ], [ 11, 12 ],
  [ 11, 12 ], [ 14, 15 ], [ 14, 15 ] ]
gap> Indeterminateness( fus );
256
gap> TestConsistencyMaps( m11.powermap, fus, m12.powermap );; fus;
[ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ], [ 14, 15 ],
  [ 14, 15 ] ]
gap> Indeterminateness( fus );
32
```

52.11 PrintAmbiguity

`PrintAmbiguity(` *list*, *paramap* `)`

prints for each character in *list* its position, its indeterminateness with respect to *paramap* and the list of ambiguous classes

```
gap> s:= CharTable( "2F4(2)" );; ru:= CharTable( "Ru" );;
gap> fus:= InitFusion( s, ru );;
gap> permchar:= Sum( Sublist( ru.irreducibles, [ 1, 5, 6 ] ) );;
gap> CheckPermChar( s, ru, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> PrintAmbiguity( Sublist( ru.irreducibles, [ 1 .. 8 ] ), fus );
1 1 [  ]
2 16 [ 16, 17, 26, 27 ]
3 16 [ 16, 17, 26, 27 ]
4 32 [ 12, 14, 23, 24, 25 ]
5 4 [ 12, 23 ]
6 1 [  ]
7 32 [ 12, 14, 23, 24, 25 ]
```

```
8 1 [  ]
gap> Indeterminateness( fus );
512
```

52.12 Powermap

`Powermap(` *tbl*`,` *prime* `)`
`Powermap(` *tbl*`,` *prime*`,` *parameters* `)`

returns a list of possibilities for the *prime*-th powermap of the character table *tbl*.

The optional record *parameters* may have the following fields:

`chars`
: a list of characters which are used for the check of kernels (see 52.16), the test of congruences (see 52.15) and the test of scalar products of symmetrisations (see 52.23); the default is *tbl*`.irreducibles`

`powermap`
: a (parametrized) map which is an approximation of the desired map

`decompose`
: a boolean; if `true`, the symmetrisations of `chars` must have all constituents in `chars`, that will be used in the algorithm; if `chars` is not bound and *tbl*`.irreducibles` is complete, the default value of `decompose` is `true`, otherwise `false`

`quick`
: a boolean; if `true`, the subroutines are called with the option `"quick"`; especially, a unique map will be returned immediately without checking all symmetrisations; the default value is `false`

`parameters`
: a record with fields `maxamb`, `minamb` and `maxlen` which control the subroutine 52.23: It only uses characters with actual indeterminateness up to `maxamb`, tests decomposability only for characters with actual indeterminateness at least `minamb` and admits a branch only according to a character if there is one with at most `maxlen` possible minus-characters.

```
#  cf. example in 52.14
gap> t:= CharTable( "U4(3).4" );;
gap> pow:= Powermap( t, 2 );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4,
      5, 6, 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18,
      18, 20, 20, 20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
```

52.13 SubgroupFusions

`SubgroupFusions(` *subtbl*`,` *tbl* `)`
`SubgroupFusions(` *subtbl*`,` *tbl*`,` *parameters* `)`

returns the list of all subgroup fusion maps from *subtbl* into *tbl*.

The optional record *parameters* may have the following fields:

`chars`
: a list of characters of *tbl* which will be restricted to *subtbl*, (see 52.24); the default is *tbl.irreducibles*

`subchars`
: a list of characters of *subtbl* which are constituents of the retrictions of `chars`, the default is *subtbl*`.irreducibles`

`fusionmap`
: a (parametrized) map which is an approximation of the desired map

`decompose`
: a boolean; if `true`, the restrictions of `chars` must have all constituents in `subchars`, that will be used in the algorithm; if `subchars` is not bound and *subtbl*`.irreducibles` is complete, the default value of `decompose` is `true`, otherwise `false`

`permchar`
: a permutation character; only those fusions are computed which afford that permutation character (see 52.19)

`quick`
: a boolean; if `true`, the subroutines are called with the option `"quick"`; especially, a unique map will be returned immediately without checking all symmetrisations; the default value is `false`

`parameters`
: a record with fields `maxamb`, `minamb` and `maxlen` which control the subroutine 52.24: It only uses characters with actual indeterminateness up to `maxamb`, tests decomposability only for characters with actual indeterminateness at least `minamb` and admits a branch only according to a character if there is one with at most `maxlen` possible restrictions.

```
# cf. example in 52.24
gap> s:= CharTable( "U3(3)" );; t:= CharTable( "J4" );;
gap> SubgroupFusions( s, t, rec( quick:= true ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 16, 16, 22, 22 ] ]
```

52.14 InitPowermap

`InitPowermap(` *tbl*`,` *prime* `)`

computes a (probably parametrized, see 52.1) first approximation of of the *prime*-th powermap of the character table *tbl*, using that for any class `i` of *tbl*, the following properties hold:

The centralizer order of the image is a multiple of the centralizer order of `i`. If the element order of `i` is relative prime to *prime*, the centralizer orders of `i` and its image must be equal.

If *prime* divides the element order x of the class `i`, the element order of its image must be $x/prime$; otherwise the element orders of `i` and its image must be equal. Of course, this is used only if the element orders are stored on the table.

If no *prime*-th powermap is possible because of these properties, `false` is returned. Otherwise `InitPowermap` returns the parametrized map.

```
# cf. example in 52.12
gap> t:= CharTable( "U4(3).4" );;
gap> pow:= InitPowermap( t, 2 );
[ 1, 1, 3, 4, 5, [ 2, 16 ], [ 2, 16, 17 ], 8, 3, [ 3, 4 ],
  [ 11, 12 ], [ 11, 12 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ], 14,
  [ 9, 20 ], 1, 1, 2, 2, 3, [ 3, 4, 5 ], [ 3, 4, 5 ],
  [ 6, 7, 18, 19, 30, 31, 32, 33 ], 8, 9, 9, [ 9, 10, 20, 21, 22 ],
  [ 11, 12 ], [ 11, 12 ], 16, 16, [ 2, 16 ], [ 2, 16 ], 17, 17,
  [ 6, 18, 30, 31, 32, 33 ], [ 6, 18, 30, 31, 32, 33 ],
  [ 6, 7, 18, 19, 30, 31, 32, 33 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ],
  20, 20, [ 9, 20 ], [ 9, 20 ], [ 9, 10, 20, 21, 22 ],
  [ 9, 10, 20, 21, 22 ], 24, 24, [ 15, 25, 26, 40, 41, 42, 43 ],
  [ 15, 25, 26, 40, 41, 42, 43 ], [ 28, 29 ], [ 28, 29 ], [ 28, 29 ],
  [ 28, 29 ] ]
# continued in 52.15
```

`InitPowermap` is used by 52.12 `Powermap`.

52.15 Congruences

`Congruences(` *tbl*`,` *chars*`,` *prime_powermap*`,` *prime* `)`
`Congruences(` *tbl*`,` *chars*`,` *prime_powermap*`,` *prime*`, "quick")`

improves the parametrized map *prime_powermap* (see 52.1) that is an approximation of the *prime*-th powermap of the character table *tbl*:

For G a group with character table *tbl*, $g \in G$ and a character χ of *tbl*, the congruence

$$\texttt{GaloisCyc}(\chi(g), prime) \equiv \chi(g^{prime}) \pmod{prime}$$

holds; if the representative order of g is relative prime to *prime*, we have

$$\texttt{GaloisCyc}(\chi(\mathrm{g}), prime) = \chi(\mathrm{g}^{prime}).$$

`Congruences` checks these congruences for the (virtual) characters in the list *chars*.

If `"quick"` is specified, only those classes are considered for which *prime_powermap* is ambiguous.

If there are classes for which no image is possible, `false` is returned, otherwise `Congruences` returns `true`.

```
# see example in 52.14
gap> Congruences( t, t.irreducibles, pow, 2 ); pow;
true
[ 1, 1, 3, 4, 5, [ 2, 16 ], [ 2, 16, 17 ], 8, 3, 4, 11, 12,
  [ 6, 7, 18, 19 ], 14, [ 9, 20 ], 1, 1, 2, 2, 3, 4, 5,
  [ 6, 7, 18, 19 ], 8, 9, 9, [ 10, 21 ], 11, 12, 16, 16, [ 2, 16 ],
  [ 2, 16 ], 17, 17, [ 6, 18 ], [ 6, 18 ], [ 6, 7, 18, 19 ],
  [ 6, 7, 18, 19 ], 20, 20, [ 9, 20 ], [ 9, 20 ], 22, 22, 24, 24,
  [ 15, 25, 26 ], [ 15, 25, 26 ], 28, 28, 29, 29 ]
# continued in 52.16
```

`Congruences` is used by 52.12 `Powermap`.

52.16 ConsiderKernels

`ConsiderKernels(` *tbl*`,` *chars*`,` *prime_powermap*`,` *prime* `)`
`ConsiderKernels(` *tbl*`,` *chars*`,` *prime_powermap*`,` *prime*`, "quick")`

improves the parametrized map *prime_powermap* (see 52.1) that is an approximation of the *prime*-th powermap of the character table *tbl*:

For G a group with character table *tbl*, the kernel of each character in the list *chars* is a normal subgroup of G, so for every $g \in$ `Kernel`(*chi*) we have $g^{prime} \in$ `Kernel`(*chi*).

Depending on the order of the factor group modulo `Kernel(` *chi* `)`, there are two further properties:If the order is relative prime to *prime*, for each $g \notin$ `Kernel`(*chi*) the *prime*-th power is not contained in `Kernel(` *chi* `)`; if the order is equal to *prime*, the *prime*-th powers of all elements lie in `Kernel(` *chi* `)`.

If `"quick"` is specified, only those classes are considered for which *prime_powermap* is ambiguous.

If `Kernel(` *chi* `)` has an order not dividing *tbl*`.order` for an element *chi* of *chars*, or if no image is possible for a class, `false` is returned; otherwise `ConsiderKernels` returns `true`.

Note that *chars* must consist of ordinary characters, since the kernel of a virtual character is not defined.

```
# see example in 52.15
gap> ConsiderKernels( t, t.irreducibles, pow, 2 ); pow;
true
[ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, [ 6, 7 ], 14, 9, 1, 1, 2, 2,
  3, 4, 5, [ 6, 7, 18, 19 ], 8, 9, 9, [ 10, 21 ], 11, 12, 16, 16,
  [ 2, 16 ], [ 2, 16 ], 17, 17, [ 6, 18 ], [ 6, 18 ],
  [ 6, 7, 18, 19 ], [ 6, 7, 18, 19 ], 20, 20, [ 9, 20 ], [ 9, 20 ],
  22, 22, 24, 24, [ 15, 25, 26 ], [ 15, 25, 26 ], 28, 28, 29, 29 ]
# continued in 52.23
```

`ConsiderKernels` is used by 52.12 `Powermap`.

52.17 ConsiderSmallerPowermaps

`ConsiderSmallerPowermaps(` *tbl*`,` *prime_powermap*`,` *prime* `)`
`ConsiderSmallerPowermaps(` *tbl*`,` *prime_powermap*`,` *prime*`, "quick")`

improves the parametrized map *prime_powermap* (see chapter 52) that is an approximation of the *prime*-th powermap of the character table *tbl*:

If *prime* $>$ *tbl*`.orders[i]` for a class `i`, try to improve *prime_powermap* at class `i` using that for g in class `i`, $g_i^{prime} = g_i^{prime \bmod tbl.\mathtt{orders[i]}}$ holds;
so if the (*prime* `mod` *tbl*`.orders[i]`)-th powermap at class `i` is determined by the maps stored in *tbl*`.powermap`, this information is used.

If `"quick"` is specified, only those classes are considered for which *prime_powermap* is ambiguous.

If there are classes for which no image is possible, `false` is returned, otherwise `true`.

Note: If *tbl*`.orders` is unbound, `true` is returned without tests.

```
gap> t:= CharTable( "3.A6" );; init:= InitPowermap( t, 5 );;
gap> Indeterminateness( init );
4096
gap> ConsiderSmallerPowermaps( t, init, 5 );;
gap> Indeterminateness( init );
256
```

`ConsiderSmallerPowermaps` is used by 52.12 `Powermap`.

52.18 InitFusion

`InitFusion(` *subtbl*, *tbl* `)`

computes a (probably parametrized, see 52.1) first approximation of of the subgroup fusion from the character table *subtbl* into the character table *tbl*, using that for any class `i` of *subtbl*, the centralizer order of the image is a multiple of the centralizer order of `i` and the element order of `i` is equal to the element order of its image (used only if element orders are stored on the tables).

If no fusion map is possible because of these properties, `false` is returned. Otherwise `InitFusion` returns the parametrized map.

```
gap> s:= CharTable( "2F4(2)" );; ru:= CharTable( "Ru" );;
gap> fus:= InitFusion( s, ru );
[ 1, 2, 2, 4, [ 5, 6 ], [ 5, 6, 7, 8 ], [ 5, 6, 7, 8 ], [ 9, 10 ],
  11, 14, 14, [ 13, 14, 15 ], [ 16, 17 ], [ 18, 19 ], 20, [ 25, 26 ],
  [ 25, 26 ], [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], [ 5, 6, 7, 8 ],
  [ 13, 14, 15 ], [ 13, 14, 15 ], [ 18, 19 ], [ 18, 19 ], [ 25, 26 ],
  [ 25, 26 ], [ 27, 28, 29 ], [ 27, 28, 29 ] ]
```

`InitFusion` is used by 52.13 `SubgroupFusions`.

52.19 CheckPermChar

`CheckPermChar(` *subtbl*, *tbl*, *fusionmap*, *permchar* `)`

tries to improve the parametrized fusion *fusionmap* (see Chapter 52) from the character table *subtbl* into the character table *tbl* using the permutation character *permchar* that belongs to the required fusion: A possible image `x` of class `i` is excluded if class `i` is too large, and a possible image `y` of class `i` is the right image if `y` must be the image of *all* classes where `y` is a possible image.

`CheckPermChar` returns `true` if no inconsistency occurred, and `false` otherwise.

```
gap> fus:= InitFusion( s, ru );; # cf. example in 52.18
gap> permchar:= Sum( Sublist( ru.irreducibles, [ 1, 5, 6 ] ) );;
gap> CheckPermChar( s, ru, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

`CheckPermChar` is used by 52.13 `SubgroupFusions`.

52.20 CheckFixedPoints

`CheckFixedPoints(` *inside1* `,` *between* `,` *inside2* `)`

If the parametrized map (see 52.1) *between* transfers the parametrized map *inside1* to *inside2*, i.e. $inside2 \circ between = between \circ inside1$, *between* must map fixed points of *inside1* to fixed points of *inside2*. Using this property, `CheckFixedPoints` tries to improve *between* and *inside2*.

If an inconsistency occurs, `false` is returned. Otherwise, `CheckFixedPoints` returns the list of classes where improvements were found.

```
gap> s:= CharTable( "L4(3).2_2" );; o7:= CharTable( "O7(3)" );;
gap> fus:= InitFusion( s, o7 );;
gap> CheckFixedPoints( s.powermap[5], fus, o7.powermap[5] );
[ 48, 49 ]
gap> fus:= InitFusion( s, o7 );; Sublist( fus, [ 48, 49 ] );
[ [ 54, 55, 56, 57 ], [ 54, 55, 56, 57 ] ]
gap> CheckFixedPoints( s.powermap[5], fus, o7.powermap[5] );
[ 48, 49 ]
gap> Sublist( fus, [ 48, 49 ] );
[ [ 56, 57 ], [ 56, 57 ] ]
```

`CheckFixedPoints` is used by 52.13 `SubgroupFusions`.

52.21 TestConsistencyMaps

`TestConsistencyMaps(` *powmap1* `,` *fusmap* `,` *powmap2* `)`
`TestConsistencyMaps(` *powmap1* `,` *fusmap* `,` *powmap2* `,` *fus_imp* `)`

Like in 52.9, it is checked that parametrized maps (see chapter 52) commute:

For all positions `i` where both *powmap1*`[i]` and *powmap2*`[i]` are bound,

$$\texttt{CompositionMaps}(\mathit{fusmap}, \mathit{powmap1}[\mathtt{i}]) = \texttt{CompositionMaps}(\mathit{powmap2}[\mathtt{i}], \mathit{fusmap})$$

shall hold, so *fusmap* occurs in diagrams for all considered elements of *powmap1* resp. *powmap2*, and it occurs twice in each diagram.

If a set *fus_imp* is specified, only those diagrams with elements of *fus_imp* as preimages of *fusmap* are considered.

`TestConsistencyMaps` stores all found improvements in *fusmap* and elements of *powmap1* and *powmap2*. When an inconsistency occurs, the program immediately returns `false`; otherwise `true` is returned.

`TestConsistencyMaps` stops if no more improvements of *fusmap* are possible. E.g. if *fusmap* was unique from the beginning, the powermaps will not be improved. To transfer powermaps by fusions, use 52.9 `TransferDiagram`.

```
gap> s:= CharTable( "2F4(2)" );; ru:= CharTable( "Ru" );;
gap> fus:= InitFusion( s, ru );;
gap> permchar:= Sum( Sublist( ru.irreducibles, [ 1, 5, 6 ] ) );;
gap> CheckPermChar( s, ru, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
```

```
    [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
    [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> TestConsistencyMaps( s.powermap, fus, ru.powermap );
true
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ],
  [ 25, 26 ], 5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27,
  27 ]
gap> Indeterminateness( fus );
16
```

`TestConsistencyMaps` is used by 52.13 `SubgroupFusions`.

52.22 ConsiderTableAutomorphisms

`ConsiderTableAutomorphisms(` *parafus*, *tableautomorphisms* `)`

improves the parametrized subgroup fusion map *parafus* (see 52.1): Let T be the permutation group that has the list *tableautomorphisms* as generators, let T_0 be the subgroup of T that is maximal with the property that T_0 operates on the set of fusions contained in *parafus* by permutation of images.

`ConsiderTableAutomorphisms` replaces orbits by representatives at suitable positions so that afterwards exactly one representative of fusion maps (that is contained in *parafus*) in every orbit under the operation of T_0 is contained in *parafus*.

The list of positions where improvements were found is returned.

```
gap> fus:= InitFusion( s, ru );;
gap> permchar:= Sum( Sublist( ru.irreducibles, [ 1, 5, 6 ] ) );;
gap> CheckPermChar( s, ru, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ],
  [ 18, 19 ], [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> ConsiderTableAutomorphisms( fus, ru.automorphisms );
[ 16 ]
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  25, [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ],
  [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

`ConsiderTableAutomorphisms` is used by `SubgroupFusions` (see 52.13). Note that the function `SubgroupFusions` forms orbits of fusion maps under table automorphisms, but it returns all possible fusions. If you want to get only orbit representatives, use the function `RepresentativesFusions` (see 52.27).

52.23 PowermapsAllowedBySymmetrisations

`PowermapsAllowedBySymmetrisations(` *tbl*, *subchars*, *chars*, *pow*, *prime*, *parameters* `)`

returns a list of (possibly parametrized, see 52.1) maps *map* which are contained in the parametrized map *pow* and which have the property that for all χ in the list *chars* of characters of the character table *tbl*, the symmetrizations

$$\chi^{p-} = \texttt{MinusCharacter}(\chi, map, prime)$$

(see 51.16) have nonnegative integral scalar products with all characters in the list *subchars*.

parameters must be a record with fields

`maxlen`
: an integer that controls the position where branches take place

`contained`
: a function, usually 51.42 or 51.44; for a symmetrization *minus*, it returns the list `contained(` *tbl*`,` *subchars*`,` *minus* `)`

`minamb`, `maxamb`
: two arbitrary objects; *contained* is called only for symmetrizations *minus* with

$$\texttt{minamb} < \texttt{Indeterminateness}(minus) < \texttt{maxamb}$$

`quick`
: a boolean; if it is true, the scalar products of uniquely determined symmetrizations are not checked.

pow will be improved, i.e. is changed by the algorithm.

If there is no character left which allows an immediate improvement but there are characters in *chars* with indeterminateness of the symmetrizations bigger than *parameters*`.minamb`, a branch is necessary. Two kinds of branches may occur: If *parameters*`.contained(` *tbl*`,` *subchars*`,` *minus* `)` has length at most *parameters*`.maxlen`, the union of maps allowed by the characters in *minus* is computed; otherwise a suitable class `c` is taken which is significant for some character, and the union of all admissible maps with image `x` on `c` is computed, where `x` runs over *pow*`[c]`.

```
# see example in 52.16
gap> t := CharTable( "U4(3).4" );;
gap> PowermapsAllowedBySymmetrisations(t,t.irreducibles,t.irreducibles,
>      pow, 2, rec( maxlen:=10, contained:=ContainedPossibleCharacters,
>      minamb:= 2, maxamb:= "infinity", quick:= false ) );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4,
      5, 6, 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18,
      18, 20, 20, 20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
gap> t.powermap[2] = last[1];
true
```

52.24 FusionsAllowedByRestrictions

`FusionsAllowedByRestrictions(` *subtbl*`,` *tbl*`,` *subchars*`,` *chars*`,` *fus*`,` *parameters* `)`

returns a list of (possibly parametrized, see 52.1) maps *map* which are contained in the parametrized map *fus* and which have the property that for all χ in the list *chars* of characters of the character table *tbl*, the restrictions

$$\chi_{subtbl} = \texttt{CompositionMaps}(\chi, fus)$$

(see 52.2) have nonnegative integral scalar products with all characters in the list *subchars*.

parameters must be a record with fields

`maxlen`
: an integer that controls the position where branches take place

`contained`
: a function, usually 51.42 or 51.44; for a restriction *rest*, it returns the list `contained(` *subtbl*`,` *subchars*`,` *rest* `);`

`minamb`, `maxamb`
: two arbitrary objects; *contained* is called only for restrictions *rest* with `minamb <Indeterminateness(` *rest* `) <maxamb`;

`quick`
: a boolean value; if it is true, the scalar products of uniquely determined restrictions are not checked.

fus will be improved, i.e. is changed by the algorithm.

If there is no character left which allows an immediate improvement but there are characters in *chars* with indeterminateness of the restrictions bigger than *parameters*`.minamb`, a branch is necessary. Two kinds of branches may occur: If *parameters*`.contained(` *tbl*`,` *subchars*`,` *rest* `)` has length at most *parameters*`.maxlen`, the union of maps allowed by the characters in *rest* is computed; otherwise a suitable class `c` is taken which is significant for some character, and the union of all admissible maps with image `x` on `c` is computed, where `x` runs over *fus*`[c]`.

```
gap> s:= CharTable( "U3(3)" );; t:= CharTable( "J4" );;
gap> fus:= InitFusion( s, t );;
gap> TestConsistencyMaps( s.powermap, fus, t.powermap );;
gap> ConsiderTableAutomorphisms( fus, t.automorphisms );; fus;
[ 1, 2, 4, 4, [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], 10, 12, [ 12, 13 ],
  [ 14, 15, 16 ], [ 14, 15, 16 ], [ 21, 22 ], [ 21, 22 ] ]
gap> FusionsAllowedByRestrictions( s, t, s.irreducibles,
>            t.irreducibles, fus, rec( maxlen:= 10,
>            contained:= ContainedPossibleCharacters,
>            minamb:= 2, maxamb:= "infinity", quick:= false ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ] ]
# cf. example in 52.13
```

`FusionsAllowedByRestrictions` is used by 52.13 `SubgroupFusions`.

52.25 OrbitFusions

`OrbitFusions(` *subtblautomorphisms*`,` *fusionmap*`,` *tblautomorphisms* `)`

returns the orbit of the subgroup fusion map *fusionmap* under the operations of maximal admissible subgroups of the table automorphism groups of the character tables. *subtblautomorphisms* is a list of generators of the automorphisms of the subgroup table, *tblautomorphisms* is a list of generators of the automorphisms of the supergroup table.

```
gap> s:= CharTable( "U3(3)" );; t:= CharTable( "J4" );;
gap> GetFusionMap( s, t );
[ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ]
gap> OrbitFusions( s.automorphisms, last, t.automorphisms );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ] ]
```

52.26 OrbitPowermaps

`OrbitPowermaps(` *powermap*, *matautomorphisms* `)`

returns the orbit of the powermap *powermap* under the operation of the subgroup *matautomorphisms* of the maximal admissible subgroup of the matrix automorphisms of the corresponding character table.

```
gap> t:= CharTable( "3.McL" );;
gap> maut:= MatAutomorphisms( t.irreducibles, [], Group( () ) );
Group( (55,58)(56,59)(57,60)(61,64)(62,65)(63,66), (35,36), (26,29)
(27,30)(28,31)(49,52)(50,53)(51,54), (40,43)(41,44)(42,45), ( 2, 3)
( 5, 6)( 8, 9)(12,13)(15,16)(18,19)(21,22)(24,25)(27,28)(30,31)(33,34)
(38,39)(41,42)(44,45)(47,48)(50,51)(53,54)(56,57)(59,60)(62,63)
(65,66) )
gap> OrbitPowermaps( t.powermap[3], maut );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
```

52.27 RepresentativesFusions

`RepresentativesFusions(` *subtblautomorphisms*, *listoffusionmaps*, *tblautomorphisms* `)`
`RepresentativesFusions(` *subtbl*, *listoffusionmaps*, *tbl* `)`

returns a list of representatives of the list *listoffusionmaps* of subgroup fusion maps under the operations of maximal admissible subgroups of the table automorphism groups of the character tables. *subtblautomorphisms* is a list of generators of the automorphisms of the subgroup table, *tblautomorphisms* is a list of generators of the automorphisms of the supergroup table. if the parameters *subtbl* and *tbl* (character tables) are used, the values of *subtbl*`.automorphisms` and *subtbl*`.automorphisms` will be taken.

```
gap> s:= CharTable( "2F4(2)" );; ru:= CharTable( "Ru" );;
gap> SubgroupFusions( s, ru );
[ [ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, 25, 26, 5, 5,
      6, 8, 14, 13, 19, 19, 26, 25, 27, 27 ],
  [ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, 26, 25, 5, 5,
      6, 8, 14, 13, 19, 19, 25, 26, 27, 27 ] ]
```

```
gap> RepresentativesFusions( s, last, ru );
[ [ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, 25, 26, 5, 5,
      6, 8, 14, 13, 19, 19, 26, 25, 27, 27 ] ]
```

52.28 RepresentativesPowermaps

`RepresentativesPowermaps(` *listofpowermaps*, *matautomorphisms* `)`

returns a list of representatives of the list *listofpowermaps* of powermaps under the operation of a subgroup *matautomorphisms* of the maximal admissible subgroup of matrix automorphisms of irreducible characters of the corresponding character table.

```
gap> t:= CharTable( "3.McL" );;
gap> maut:= MatAutomorphisms( t.irreducibles, [], Group( () ) );
Group( (55,58)(56,59)(57,60)(61,64)(62,65)(63,66), (35,36), (26,29)
(27,30)(28,31)(49,52)(50,53)(51,54), (40,43)(41,44)(42,45), ( 2, 3)
( 5, 6)( 8, 9)(12,13)(15,16)(18,19)(21,22)(24,25)(27,28)(30,31)(33,34)
(38,39)(41,42)(44,45)(47,48)(50,51)(53,54)(56,57)(59,60)(62,63)
(65,66) )
gap> Powermap( t, 3 );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> RepresentativesPowermaps( last, maut );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
      4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37,
      37, 37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49,
      49, 14, 14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ] ]
```

52.29 Indirected

`Indirected(` *char*, *paramap* `)`

We have

$$\texttt{Indirected}(char, paramap)[\texttt{i}] = char[paramap[\texttt{i}]],$$

if this value is unique; otherwise it is set unknown (see chapter 17). (For a parametrized indirection, see 52.2.)

```
gap> m12:= CharTable( "M12" );;
gap> fus:= [ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ],
>            [ 14, 15 ], [ 14, 15 ] ];;   # parametrized subgroup fusion
                                          # from M11
gap> chars:= Sublist( m12.irreducibles, [ 1 .. 6 ] );;
gap> List( chars, x -> Indirected( x, fus ) );
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
  [ 11, 3, 2, Unknown(1), 1, 0, Unknown(2), Unknown(3), 0, 0 ],
  [ 11, 3, 2, Unknown(4), 1, 0, Unknown(5), Unknown(6), 0, 0 ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(7), Unknown(8) ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(9), Unknown(10) ],
  [ 45, -3, 0, 1, 0, 0, -1, -1, 1, 1 ] ]
```

52.30 Powmap

`Powmap(` *powermap*`,` *n* `)`
`Powmap(` *powermap*`,` *n*`,` *class* `)`

The first form returns the n-th powermap where *powermap* is the powermap of a character table (see 49.2). If the n-th position in *powermap* is bound, this map is returned, otherwise it is computed from the (necessarily stored) powermaps of the prime divisors of n.

The second form returns the image of *class* under the n-th powermap; for any valid class *class*, we have `Powmap(` *powermap*`,` *n* `)[` *class* `] = Powmap(` *powermap*`,` *n*`,` *class* `)`.

The entries of *powermap* may be parametrized maps (see 52.1).

```
gap> t:= CharTable( "3.McL" );;
gap> Powmap( t.powermap, 3 );
[ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
  4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 37,
  37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14,
  14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ]
gap> Powmap( t.powermap, 27 );
[ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17,
  4, 4, 4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 1, 1, 37, 37,
  37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14,
  14, 14, 14, 14, 14, 37, 37, 37, 37, 37, 37 ]
gap> Lcm( t.orders ); Powmap( t.powermap, last );
27720
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

52.31 ElementOrdersPowermap

`ElementOrdersPowermap(` *powermap* `)`

returns the list of element orders given by the maps in the powermap *powermap*. The entries at positions where the powermaps do not uniquely determine the element order are set to unknowns (see chapter 17).

```
gap> t:= CharTable( "3.J3.2" );; t.powermap;
[ , [ 1, 2, 1, 2, 5, 6, 7, 3, 4, 10, 11, 12, 5, 6, 8, 9, 18, 19, 17,
      10, 11, 12, 13, 14, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 1,
      3, 7, 8, 8, 13, 18, 19, 17, 23, 23, 28, 30 ],
  [ 1, 1, 3, 3, 1, 1, 1, 8, 8, 10, 10, 10, 3, 3, 15, 15, 7, 7, 7, 20,
      20, 20, 8, 8, 10, 10, 10, 30, 30, 28, 28, 32, 32, 32, 35, 36,
      35, 38, 39, 36, 37, 37, 37, 38, 38, 47, 46 ],,
```

```
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 2, 13, 14, 15, 16, 19, 17, 18,
        3, 4, 4, 23, 24, 5, 6, 6, 30, 31, 28, 29, 32, 34, 33, 35, 36,
        37, 38, 39, 40, 43, 41, 42, 44, 45, 47, 46 ],,,,,,,,,,,,
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24, 25, 26, 27, 1, 2, 1, 2, 32, 34, 33, 35,
        36, 37, 38, 39, 40, 41, 42, 43, 45, 44, 35, 35 ],,
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 1, 2, 2,
        35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47 ] ]
  gap> ElementOrdersPowermap( last );
  [ 1, 3, 2, 6, 3, 3, 3, 4, 12, 5, 15, 15, 6, 6, 8, 24, 9, 9, 9, 10,
    30, 30, 12, 12, 15, 15, 15, 17, 51, 17, 51, 19, 57, 57, 2, 4, 6, 8,
    8, 12, 18, 18, 18, 24, 24, 34, 34 ]
  gap> Unbind( t.powermap[17] ); ElementOrdersPowermap( t.powermap );
  [ 1, 3, 2, 6, 3, 3, 3, 4, 12, 5, 15, 15, 6, 6, 8, 24, 9, 9, 9, 10,
    30, 30, 12, 12, 15, 15, 15, Unknown(11), Unknown(12), Unknown(13),
    Unknown(14), 19, 57, 57, 2, 4, 6, 8, 8, 12, 18, 18, 18, 24, 24,
    Unknown(15), Unknown(16) ]
```

Chapter 53

Character Table Libraries

The utility of GAP3 for character theoretical tasks depends on the availability of many known character tables, so there is a lot of tables in the GAP3 group collection.

There are three different libraries of character tables, namely **ordinary character tables**, **Brauer tables** and **generic character tables**.

Of course, these libraries are "open" in the sense that they shall be extended. So we would be grateful for any further tables of interest sent to us for inclusion into our libraries.

This chapter mainly explains properties not of single tables but of the libraries and their structure; for the format of character tables, see 49.2, 49.3 and chapter 50.

The chapter informs about

- the actually available tables (see 53.1),
- the sublibraries of ATLAS tables (see 53.3) and CAS tables (see 53.5),
- the organization of the libraries (see 53.6),
- and how to extend a library (see 53.7).

53.1 Contents of the Table Libraries

As stated at the beginning of the chapter, there are three libraries of character tables: ordinary character tables, Brauer tables, and generic character tables.

Ordinary Character Tables

Two different aspects are useful to list up the ordinary character tables available to GAP3 the aspect of **source** of the tables and that of **connections** between the tables.

As for the source, there are two big sources, the ATLAS (see 53.3) and the CAS library of character tables. Many ATLAS tables are contained in the CAS library, and difficulties may arise because the succession of characters or classes in CAS tables and ATLAS tables are different, so see 53.5 and 49.2 for the relations between the (at least) two forms of the same table. A large subset of the CAS tables is the set of tables of Sylow normalizers of sporadic simple groups as published in [Ost86], so this may be viewed as another source.

To avoid confusions about the actual format of a table, authorship and so on, the `text` component of the table contains the information

`origin: ATLAS of finite groups`
: for ATLAS tables (see 53.3)

`origin: Ostermann`
: for tables of [Ost86] and

`origin: CAS library`
: for any table of the CAS table library that is contained neither in the ATLAS nor in [Ost86].

If one is interested in the aspect of connections between the tables, i.e., the internal structure of the library of ordinary tables (which corresponds to the access to character tables, as described in 49.12), the contents can be listed up the following way:

We have

- all ATLAS tables (see 53.3), i.e. the tables of the simple groups which are contained in the ATLAS, and the tables of cyclic and bicyclic extensions of these groups;
- most tables of maximal subgroups of sporadic simple groups (**not all** for HN, F3+, B, M);
- some tables of maximal subgroups of other ATLAS tables (**which?**)
- most nontrivial Sylow normalizers of sporadic simple groups as printed in [Ost86], where nontrivial means that the group is not contained in $p{:}(p-1)$ (**not** J_4N2, Co_1N2, Co_1N5, all of Fi_{23}, Fi_{24}', B, M, HN, and $Fi_{22}N2$)
- some tables of element centralizers
- some tables of Sylow subgroups
- a few other tables, e.g. `W(F4)`
  namely which?

Brauer Tables

This library contains the tables of the modular ATLAS which are yet known. Some of them still contain unknowns (see 17.1). Since there is ongoing work in computing new tables, this library is changed nearly every day.

These Brauer tables contain the information

```
origin: modular ATLAS of finite groups
```

in their text component.

Generic Character Tables

At the moment, generic tables of the following groups are available in GAP3 (see 49.12):

- alternating groups

- cyclic groups,
- dihedral groups,
- some linear groups,
- quaternionic (dicyclic) groups
- Suzuki groups,
- symmetric groups,
- wreath products of a group with a symmetric group (see 49.18),
- Weyl groups of types B_n and D_n

53.2 Selecting Library Tables

Single library tables can be selected by their name (see 49.12 for admissible names of library tables, and 53.1 for the organization of the library).

In general it does not make sense to select tables with respect to certain properties, as is useful for group libraries (see 38). But it may be useful to get an overview of **all** library tables, or all library tables of **simple** groups, or all library tables of **sporadic simple** groups. It is sufficient to know an admissible name of these tables, so they need not be loaded. A table can then be read using 49.12 `CharTable`.

The mechanism is similar to that for group libraries.

`AllCharTableNames()`
: returns a list with an admissible name for every library table,

`AllCharTableNames( IsSimple )`
: returns a list with an admissible name for every library table of a simple group,

`AllCharTableNames( IsSporadicSimple )`
: returns a list with an admissible name for every library table of a sporadic simple group.

Admissible names of **maximal subgroups** of sporadic simple groups are stored in the component `maxes` of the tables of the sporadic simple groups. Thus

```
gap> maxes:= CharTable( "M11" ).maxes;
[ "A6.2_3", "L2(11)", "3^2:Q8.2", "A5.2", "2.S4" ]
```

returns the list containing these names for the Mathieu group M_{11}, and

```
gap> List( maxes, CharTable );
[ CharTable( "A6.2_3" ), CharTable( "L2(11)" ),
  CharTable( "3^2:Q8.2" ), CharTable( "A5.2" ), CharTable( "2.S4" ) ]
```

will read them from the library files.

53.3 ATLAS Tables

The GAP3 group collection contains all character tables that are included in the Atlas of finite groups ([CCN+85], from now on called ATLAS) and the Brauer tables contained in the modular ATLAS ([JLPW95]). Although the Brauer tables form a library of their own, they are described here since all conventions for ATLAS tables stated here hold for Brauer tables, too.

Additionally some conventions are necessary about follower characters!

These tables have the information

```
origin: ATLAS of finite groups
```

resp.

```
origin: modular ATLAS of finite groups
```

in their `text` component, further on they are simply called ATLAS tables.

In addition to the information given in Chapters 6–8 of the ATLAS which tell how to read the printed tables, there are some rules relating these to the corresponding GAP3 tables.

Improvements

Note that for the GAP3 library not the printed ATLAS is relevant but the revised version given by the list of **Improvements to the ATLAS** which can be got from Cambridge.

Also some tables are regarded as ATLAS tables which are not printed in the ATLAS but available in ATLAS format from Cambridge; at the moment, these are the tables related to $L_2(49)$, $L_2(81)$, $L_6(2)$, $O_8^-(3)$, $O_8^+(3)$ and $S_{10}(2)$.

Powermaps

In a few cases (namely the tables of $3.McL$, $3_2.U_4(3)$ and its covers, $3_2.U_4(3).2_3$ and its covers) the powermaps are not uniquely determined by the given information but determined up to matrix automorphisms (see 49.41) of the characters; then the first possible map according to lexicographical ordering was chosen, and the automorphisms are listed in the `text` component of the concerned table.

Projective Characters

For any nontrivial multiplier of a simple group or of an automorphic extension of a simple group, there is a component `projectives` in the table of G that is a list of records with the names of the covering group (e.g. `"12_1.U4(3)"`) and the list of those faithful characters which are printed in the ATLAS(so–called *proxy characters*).

Projections

ATLAS tables contain the component `projections`: For any covering group of G for which the character table is available in ATLAS format a record is stored there containing components `name` (the name of the cover table) and `map` (the projection map); the projection maps any class of G to that preimage in the cover for that the column is printed in the ATLAS; it is called g_0 in Chapter 7, Section 14 there.

(In a sense, a projection map is an inverse of the factor fusion from the cover table to the actual table (see 52.4).)

Tables of Isoclinic Groups

As described in Chapter 6, Section 7 and Chapter 7, Section 18 of the ATLAS, there exist two different groups of structure $2.G.2$ for a simple group G which are isoclinic. The ATLAS table in the library is that which is printed in the ATLAS, the isoclinic variant can be got using 49.20 `CharTableIsoclinic`.

Succession of characters and classes

(Throughout this paragraph, G always means the involved simple group.)

1. For G itself, the succession of classes and characters in the GAP3 table is as printed in the ATLAS.

2. For an automorphic extension $G.a$, there are three types of characters:

   - If a character χ of G extends to $G.a$, the different extensions $\chi^0, \chi^1, \ldots, \chi^{a-1}$ are consecutive (see ATLAS, Chapter 7, Section 16).
   - If some characters of G fuse to give a single character of $G.a$, the position of that character is the position of the first involved character of G.
   - If both, extension and fusion, occur, the result characters are consecutive, and each replaces the first involved character.

3. Similarly, there are different types of classes for an automorphic extension $G.a$:

   - If some classes collapse, the result class replaces the first involved class.
   - For $a > 2$, any proxy class and its followers are consecutive; if there are more than one followers for a proxy class (the only case that occurs is for $a = 5$), the succession of followers is the natural one of corresponding galois automorphisms (see ATLAS, Chapter 7, Section 19).

   The classes of $G.a_1$ always precede the outer classes of $G.a_2$ for a_1, a_2 dividing a and $a_1 < a_2$. This succession is like in the ATLAS, with the only exception $U_3(8).6$.

4. For a central extension $M.G$, there are different types of characters:

   - Every character can be regarded as a faithful character of the factor group $m.G$, where m divides M. Characters faithful for the same factor group are consecutive like in the ATLAS, the succession of these sets of characters is given by the order of precedence $1, 2, 4, 3, 6, 12$ for the different values of m.
   - If $m > 2$, a faithful character of $m.G$ that is printed in the ATLAS (a so-called *proxy*) represents one or more *followers*, this means galois conjugates of the proxy; in any GAP3 table, the proxy precedes its followers; the case $m = 12$ is the only one that occurs with more than one follower for a proxy, then the three followers are ordered according to the corresponding galois automorphisms 5, 7, 11 (in that succession).

5. For the classes of a central extension we have:

   - The preimages of a G-class in $M.G$ are subsequent, the succession is the same as that of the lifting order rows in the ATLAS.
   - The primitive roots of unity chosen to represent the generating central element (class 2) are `E(3)`, `E(4)`, `E(6)^5` (`= E(2) *E(3)`) and `E(12)^7` (`= E(3) *E(4)`) for $m = 3, 4, 6$ and 12, respectively.

6. For tables of bicyclic extensions $m.G.a$, both the rules for automorphic and central extensions hold; additionally we have:

   - Whenever classes of the subgroup $m.G$ collapse or characters fuse, the result class resp. character replaces the first involved class resp. character.
   - Extensions of a character are subsequent, and the extensions of a proxy character precede the extensions of its followers.
   - Preimages of a class are subsequent, and the preimages of a proxy class precede the preimages of its followers.

53.4 Examples of the ATLAS format for GAP tables

We give three little examples for the conventions stated in 53.3, listing up the ATLAS format and the table displayed by GAP3.

First, let G be the trivial group. The cyclic group C_6 of order 6 can be viewed in several ways:

1. As a downward extension of the factor group C_2 which contains G as a subgroup; equivalently, as an upward extension of the subgroup C_3 which has a factor group G:

G	G.2
3.G	3.G.2

```
       ;   @   ;   ;   @
           1           1
   p power             A
   p' part             A
   ind  1A fus ind    2A
χ1   +   1   :  ++     1
   ind   1 fus ind     2
         3             6
         3             6
χ2 o2    1   : oo2     1
```

```
     2   1   1   1   1   1   1
     3   1   1   1   1   1   1
        1a  3a  3b  2a  6a  6b
    2P  1a  3b  3a  1a  3b  3a
    3P  1a  1a  1a  2a  2a  2a
X.1      1   1   1   1   1   1
X.2      1   1   1  -1  -1  -1
X.3      1   A  /A   1   A  /A
X.4      1   A  /A  -1  -A -/A
X.5      1  /A   A   1  /A   A
X.6      1  /A   A  -1 -/A  -A
A = E(3)
  = (-1+ER(-3))/2 = b3
```

X.1, X.2 extend χ_1. X.3, X.4 extend the proxy character χ_2. X.5, X.6 extend its follower. 1a, 3a, 3b are preimages of 1A, and 2a, 6a, 6b are preimages of 2A.

2. As a downward extension of the factor group C_3 which contains G as a subgroup; equivalently, as an upward extension of the subgroup C_2 which has a factor group G:

G	G.3
2.G	2.G.3

```
       ;   @   ;   ;   @
           1           1
   p power             A
   p' part             A
   ind  1A fus ind    3A
χ1   +   1   : +oo     1
   ind   1 fus ind     3
         2             6
χ2   +   1   : +oo     1
```

```
     2   1   1   1   1   1   1
     3   1   1   1   1   1   1
        1a  2a  3a  6a  3b  6b
    2P  1a  1a  3b  3b  3a  3a
    3P  1a  2a  1a  2a  1a  2a
X.1      1   1   1   1   1   1
X.2      1   1   A   A  /A  /A
X.3      1   1  /A  /A   A   A
X.4      1  -1   1  -1   1  -1
X.5      1  -1   A  -A  /A -/A
X.6      1  -1  /A -/A   A  -A
A = E(3)
  = (-1+ER(-3))/2 = b3
```

X.1-X.3 extend χ_1, X.4-X.6 extend χ_2. 1a, 2a are preimages of 1A. 3a, 6a are preimages of the proxy class 3A, and 3b, 6b are preimages of its follower class.

3. As a downward extension of the factor groups C_3 and C_2 which have G as a factor group:

G
2.G
3.G
6.G

```
            ;   @

                1
       p power
       p' part
       ind  1A

χ1      +   1

       ind  1
            2

χ2      +   1

       ind  1
            3
            3

χ3     o2   1

       ind  1
            6
            3
            2
            3
            6

χ4     o2   1
```

```
      2  1  1  1  1  1  1
      3  1  1  1  1  1  1
        1a 6a 3a 2a 3b 6b
     2P 1a 3a 3b 1a 3a 3b
     3P 1a 2a 1a 2a 1a 2a
X.1      1  1  1  1  1  1
X.2      1 -1  1 -1  1 -1
X.3      1  A /A  1  A /A
X.4      1 /A  A  1 /A  A
X.5      1 -A /A -1  A -/A
X.6      1 -/A A -1 /A -A
A = E(3)
  = (-1+ER(-3))/2 = b3
```

`X.1`, `X.2` correspond to χ_1, χ_2, respectively; `X.3`, `X.5` correspond to the proxies χ_3, χ_4, and `X.4`, `X.6` to their followers. The factor fusion onto $3.G$ is `[ 1, 2, 3, 1, 2, 3 ]`, that onto $G.2$ is `[ 1, 2, 1, 2, 1, 2 ]`.

4. As an upward extension of the subgroups C_3 or C_2 which both contain a subgroup G:

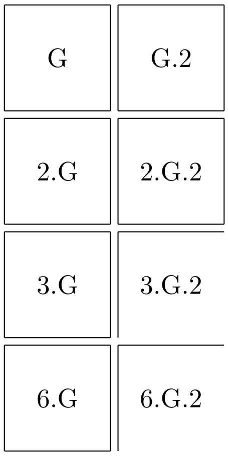
G	G.2	G.3	G.6

```
   ;  @   ;   ;  @   ;   ;  @   ;     ;  @

      1          1          1          1
 p power         A          A         AA
 p' part         A          A         AA
 ind  1A fus ind  2A fus ind  3A fus   ind  6A

χ1  +  1   :  ++   1   : +oo   1   :+oo+oo   1
```

```
      2  1  1  1  1  1  1
      3  1  1  1  1  1  1
        1a 2a 3a 3b 6a 6b
     2P 1a 1a 3b 3a 3b 3a
     3P 1a 2a 1a 1a 2a 2a
X.1      1  1  1  1  1  1
X.2      1 -1  A /A -A -/A
X.3      1  1 /A  A /A  A
X.4      1 -1  1  1 -1 -1
X.5      1  1  A /A  A /A
X.6      1 -1 /A  A -/A -A
A = E(3)
  = (-1+ER(-3))/2 = b3
```

`1a`, `2a` correspond to $1A, 2A$, respectively; `3a`, `6a` correspond to the proxies $3A, 6A$, and `3b`, `6b` to their followers.

The second example explains the fusion case; again, G is the trivial group.




```
        ;    @    ;    ;   @
             1             1
   p power                 A
   p' part                 A
   ind    1A  fus ind  2A

χ1   +    1    :   ++   1

   ind    1  fus ind   2
          2            2

χ2   +    1    :   ++   1

   ind    1  fus ind   2
          3
          3

χ3  o2    1    *    +

   ind    1  fus ind   2
          6            2
          3
          2
          3
          6

χ4  o2    1    *    +
```

```
3.G.2
   2   1   .   1
   3   1   1   .
      1a  3a  2a
  2P  1a  3a  1a
  3P  1a  1a  2a
X.1    1   1   1
X.2    1   1  -1
X.3    2  -1   .

6.G.2
   2   2   1   1   2   2   2
   3   1   1   1   1   .   .
      1a  6a  3a  2a  2b  2c
  2P  1a  3a  3a  1a  1a  1a
  3P  1a  2a  1a  2a  2b  2c
Y.1    1   1   1   1   1   1
Y.2    1   1   1   1  -1  -1
Y.3    1  -1   1  -1   1  -1
Y.4    1  -1   1  -1  -1   1
Y.5    2  -1  -1   2   .   .
Y.6    2   1  -1  -2   .   .
```

The tables of $G, 2.G, 3.G, 6.G$ and $G.2$ are known from the first example, that of $2.G.2 \cong V_4$ will be given in the next one. So here we only print the GAP3 tables of $3.G.2 \cong D_6$ and $6.G.2 \cong D_{12}$:

In $3.G.2$, `X.1`, `X.2` extend χ_1; χ_3 and its follower fuse to give `X.3`, and two of the preimages of `1A` collapse.

In $6.G.2$, `Y.1`-`Y.4` are extensions of χ_1, χ_2, so these characters are the inflated characters from $2.G.2$ (with respect to the factor fusion `[ 1, 2, 1, 2, 3, 4 ]`). `Y.5` is inflated from $3.G.2$ (with respect to the factor fusion `[ 1, 2, 2, 1, 3, 3 ]`), and `Y.6` is the result of the fusion of χ_4 and its follower.

For the last example, let G be the group 2^2. Consider the following tables:

G	G.3
2.G	2.G.3

```
          ;    @    @    @    @    ;    ;    @

               4    4    4    4              1
  p power           A    A    A              A
  p' part           A    A    A              A
  ind      1A      2A   2B   2C  fus  ind   3A

χ1   +      1       1    1    1    :  +oo    1
χ2   +      1       1   -1   -1    |   +     0
χ3   +      1      -1    1   -1    |
χ4   +      1      -1   -1    1    |

  ind       1       4    4    4  fus  ind    3
            2                                6
χ5   -      2       0    0    0    :  -oo    1
```

$G.3$

```
     2   2   2   .   .
     3   1   .   1   1

        1a  2a  3a  3b
    2P  1a  1a  3b  3a
    3P  1a  2a  1a  1a

X.1      1   1   1   1
X.2      1   1   A  /A
X.3      1   1  /A   A
X.4      3  -1   .   .

A = E(3)
  = (-1+ER(-3))/2 = b3
```

$2.G$

```
     2   3   3   2   2   2

        1a  2a  4a  4b  4c
    2P  1a  1a  2a  1a  1a
    3P  1a  2a  4a  4b  4c

X.1      1   1   1   1   1
X.2      1   1   1  -1  -1
X.3      1   1  -1   1  -1
X.4      1   1  -1  -1   1
X.5      2  -2   .   .   .
```

$2.G.3$

```
     2   3   3   2   1   1   1   1
     3   1   1   .   1   1   1   1

        1a  2a  4a  3a  6a  3b  6b
    2P  1a  1a  2a  3b  3b  3a  3a
    3P  1a  2a  4a  1a  2a  1a  2a

X.1      1   1   1   1   1   1   1
X.2      1   1   1   A   A  /A  /A
X.3      1   1   1  /A  /A   A   A
X.4      3   3  -1   .   .   .   .
X.5      2  -2   .   1   1   1   1
X.6      2  -2   .   A  -A  /A -/A
X.7      2  -2   .  /A -/A   A  -A

A = E(3)
  = (-1+ER(-3))/2 = b3
```

In the table of $G.3 \cong A_4$, the characters χ_2, χ_3 and χ_4 fuse, and the classes `2A`, `2B` and `2C` collapse. To get the table of $2.G \cong Q_8$ one just has to split the class `2A` and adjust the representative orders. Finally, the table of $2.G.3 \cong SL_2(3)$ is given; the subgroup fusion corresponding to the injection $2.G \hookrightarrow 2.G.3$ is `[ 1, 2, 3, 3, 3 ]`, and the factor fusion corresponding to the epimorphism $2.G.3 \to G.3$ is `[ 1, 1, 2, 3, 3, 4, 4 ]`.

53.5 CAS Tables

All tables of the CAS table library are available in GAP3, too. This sublibrary has been completely revised, i.e., errors have been corrected and powermaps have been completed.

Any CAS table is accessible by each of its CAS names, that is, the table name or the filename (see 49.12):

```
gap> t:= CharTable( "m10" );; t.name;
"A6.2_3"
```

One does, however, not always get the original CAS table: In many cases (mostly ATLAS tables, see 53.3) not only the name but also the succession of classes and characters has changed; the records in the component `CAS` of the table (see 49.2) contain the permutations which must be applied to classes and characters to get the original CAS table:

```
gap> t.CAS;
[ rec(
      name := "m10",
      permchars := (3,5)(4,8,7,6),
      permclasses := (),
      text := [ 'n', 'a', 'm', 'e', 's', ':', ' ', ' ', ' ', ' ',
          ' ', 'm', '1', '0', '\n', 'o', 'r', 'd', 'e', 'r', ':',
          ' ', ' ', ' ', ' ', ' ', '2', '^', '4', '.', '3', '^', '2',
          '.', '5', ' ', '=', ' ', '7', '2', '0', '\n', 'n', 'u',
          'm', 'b', 'e', 'r', ' ', 'o', 'f', ' ', 'c', 'l', 'a', 's',
          's', 'e', 's', ':', ' ', '8', '\n', 's', 'o', 'u', 'r',
          'c', 'e', ':', ' ', ' ', ' ', ' ', 'c', 'a', 'm', 'b', 'r',
          'i', 'd', 'g', 'e', ' ', 'a', 't', 'l', 'a', 's', '\n',
          'c', 'o', 'm', 'm', 'e', 'n', 't', 's', ':', ' ', ' ', 'p',
          'o', 'i', 'n', 't', ' ', 's', 't', 'a', 'b', 'i', 'l', 'i',
          'z', 'e', 'r', ' ', 'o', 'f', ' ', 'm', 'a', 't', 'h', 'i',
          'e', 'u', '-', 'g', 'r', 'o', 'u', 'p', ' ', 'm', '1', '1',
          '\n', 't', 'e', 's', 't', ':', ' ', ' ', ' ', ' ', ' ',
          ' ', 'o', 'r', 't', 'h', ',', ' ', 'm', 'i', 'n', ',', ' ',
          's', 'y', 'm', '[', '3', ']', ' ', ' ', ' ', ' ', ' ', ' ',
          ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
          ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '\n' ] ) ]
```

The subgroup fusions were computed anew; their record component `text` tells if the fusion is equal to that in the CAS library –of course modulo the permutation of classes.

Note that the fusions are neither tested to be consistent for any two subgroups of a group and their intersection, nor tested to be consistent with respect to composition of maps.

53.6 Organization of the Table Libraries

The **primary files** are `TBLNAME/ctadmin.tbl` and `TBLNAME/ctprimar.tbl`. The former contains the evaluation function `CharTableLibrary` (see 49.12) and some utilities, the latter contains the global variable `LIBLIST` which encodes all information where to find library tables; the file `TBLNAME/ctprimar.tbl` can be constructed from the data files of the table libraries using the `awk` script `maketbl` in the `etc` directory of the GAP3 distribution.

Also the **secondary files** are all stored in the directory `TBLNAME`; they are

`clmelab.tbl`	`clmexsp.tbl`	`ctadmin.tbl`	`ctbalter.tbl`	`ctbatres.tbl`
`ctbconja.tbl`	`ctbfisc1.tbl`	`ctbfisc2.tbl`	`ctbline1.tbl`	`ctbline2.tbl`
`ctbline3.tbl`	`ctbline4.tbl`	`ctbline5.tbl`	`ctbmathi.tbl`	`ctbmonst.tbl`
`ctborth1.tbl`	`ctborth2.tbl`	`ctborth3.tbl`	`ctbspora.tbl`	`ctbsympl.tbl`
`ctbtwis1.tbl`	`ctbtwis2.tbl`	`ctbunit1.tbl`	`ctbunit2.tbl`	`ctbunit3.tbl`
`ctbunit4.tbl`	`ctgeneri.tbl`	`ctoalter.tbl`	`ctoatres.tbl`	`ctocliff.tbl`
`ctoconja.tbl`	`ctofisc1.tbl`	`ctofisc2.tbl`	`ctoholpl.tbl`	`ctoinert.tbl`
`ctoline1.tbl`	`ctoline2.tbl`	`ctoline3.tbl`	`ctoline4.tbl`	`ctoline5.tbl`
`ctoline6.tbl`	`ctomathi.tbl`	`ctomaxi1.tbl`	`ctomaxi2.tbl`	`ctomaxi3.tbl`
`ctomaxi4.tbl`	`ctomaxi5.tbl`	`ctomaxi6.tbl`	`ctomisc1.tbl`	`ctomisc2.tbl`
`ctomisc3.tbl`	`ctomisc4.tbl`	`ctomisc5.tbl`	`ctomisc6.tbl`	`ctomonst.tbl`
`ctonews.tbl`	`ctoorth1.tbl`	`ctoorth2.tbl`	`ctoorth3.tbl`	`ctoorth4.tbl`
`ctoorth5.tbl`	`ctospora.tbl`	`ctosylno.tbl`	`ctosympl.tbl`	`ctotwis1.tbl`
`ctotwis2.tbl`	`ctounit1.tbl`	`ctounit2.tbl`	`ctounit3.tbl`	`ctounit4.tbl`

The names start with `ct` for "character table", followed by `o` for "ordinary", `b` for "Brauer" or `g` for "generic", then an up to 5 letter description of the contents, e.g., `alter` for the alternating groups, and the extension `.tbl`.

The file `ctb`*descr*`.tbl` contains (at most) the Brauer tables corresponding to the ordinary tables in `cto`*descr*`.tbl`.

The **format of library tables** is always like this:

```
MOT(tblname,
     ...
          # here the data components are stored
     ... );
```

Here *tblname* is the value of the `identifier` component of the table, e.g. `"A5"`.

For the contents of the table record, there are three different ways how tables are stored:

Full tables (like that of A_5) are stored similar to the internal format (see 49.2). Lists of characters, however, will be abbreviated in the following way:

For each subset of characters which differ just by multiplication with a linear character or by Galois conjugacy, only one is given by its values, the others are replaced by `[TENSOR,[`i`,`j`]]` (which means that the character is the tensor product of the i-th and the j-th character) or `[GALOIS,[`i`,`j`]]` (which means that the character is the j-th Galois conjugate of the i-th character.

Brauer tables (like that of A_5 mod 2) are stored relative to the corresponding ordinary table; instead of irreducible characters the files contain decomposition matrices or Brauer trees for the blocks of nonzero defect (see 49.3), and components which can be got by restriction to p–regular classes are not stored at all.

Construction tables (like that of $O_8^-(3)M7$) have a component `construction` that is a function of one variable. This function is called by `CharTable` (see 49.12) when the table is constructed, i.e. **not** when the file containing the table is read.

The aim of this rather complicated way to store a character table is that big tables with a simple structure (e.g. direct products) can be stored in a very compact way.

Another special case where construction tables are useful is that of projective tables:

In their component `irreducibles` they do not contain irreducible characters but a list with information about the factor groups: Any entry is a list of length 2 that contains at position 1 the name of the table of the factor group, at the second position a list of integers representing the Galois automorphisms to get follower characters. E.g., for $12.M_{22}$, the value of `irreducibles` is

```
[["M22",[]],["2.M22",[]],
 ["3.M22",[-1,-13,-13,-1,23,23,-1,-1,-1,-1,-1]],
 ["4.M22",[-1,-1,15,15,23,23,-1,-1]],,
 ["6.M22",[-13,-13,-1,23,23,-1,-7,-7,-1,-1]],,,,,,
 ["12.M22",[[17,-17,-1],[17,-17,-1],[-55,-377,-433],[-55,-377,-433],
 [89,991,1079],[89,991,1079],[-7,7,-1]]]]
```

Using this and the `projectives` component of the table of the smallest nontrivial factor group, 49.12 `CharTable` constructs the irreducible characters. The table head, however, need not be constructed.

53.7 How to Extend a Table Library

If you have some ordinary character tables which are not (or not yet) in a GAP3 table library, but which you want to treat as library tables, e.g., assign them to variables using 49.12 `CharTable`, you can include these tables. For that, two things must be done:

First you must notify each table, i.e., tell GAP3 on which file it can be found, and which names are admissible; this can be done using

`NotifyCharTable(` *firstname*`,` *filename*`,` *othernames* `)`,

with strings *firstname* (the `identifier` component of the table) and *filename* (the name of the file containing the table, relative to `TBLNAME`, and without extension `.tbl`), and a list *othernames* of strings which are other admissible names of the table (see 49.12).

`NotifyCharTable` will add the necessary information to `LIBLIST`. A warning is printed for each table *libtbl* that was already accessible by some of the names, and these names are ignored for the new tables. Of course this affects only the value of `LIBLIST` in the current GAP3 session, not that on the file.

Note that an error is raised if you want to notify a table with *firstname* or name in *othernames* which is already the `identifier` component of a library table.

```
gap> Append( TBLNAME, ";tables/" );
# tells GAP3 that the directory tables is a place to look for
# library tables
gap> NotifyCharTable( "Private", "mytables", [ "My" ] );
# tells GAP3 that the table with names "Private" and "My"
# is stored on file mytables.tbl
gap> FirstNameCharTable( "My" );
"Private"
gap> FileNameCharTable( "My" );
"mytables"
```

The second condition is that each file must contain tables in library format as described in 53.6; in the example, the contents of the file `tables/mytables.tbl` may be this:

```
SET_TABLEFILENAME("mytables");
```

```
ALN:= Ignore;
MOT("Private",
[
"my private character table"
],
[2,2],
[],
[[1,1],[1,-1]],
[]);
ALN("Private",["my"]);
LIBTABLE.LOADSTATUS.("mytables"):="loaded";
```

We simulate reading this file by explicitly assigning some of the components.

```
gap> LIBTABLE.("mytables"):= rec(
> Private:= rec( identifier:= "Private",
>                centralizers:= [2,2],
>                irreducibles:= [[1,1],[1,-1]] ) );;
gap> LIBTABLE.LOADSTATUS.("mytables"):="loaded";;
```

Now the private table is a library table:

```
gap> CharTable( "My" );
CharTable( "Private" )
```

To append the table *tbl* in library format to the file with name *file*, use

`PrintToLib(` *file*, *tbl* `)`.

Note that here *file* is the absolute name of the file, not the name relative to `TBLNAME`. Thus the filename in the row with the assignment to `LIBTABLE` must be adjusted to make the file a library file.

53.8 FirstNameCharTable

`FirstNameCharTable(` *name* `)`

returns the value of the `identifier` component of the character table with admissible name *name*, if exists; otherwise `false` is returned.

For each admissible name, also the lowercase string is admissible.

```
gap> FirstNameCharTable( "m22mod3" );
"M22mod3"
gap> FirstNameCharTable( "s5" );
"A5.2"
gap> FirstNameCharTable( "J5" );
false
```

53.9 FileNameCharTable

`FileNameCharTable(` *tblname* `)`

returns the value of the `filename` component of the information record in `LIBLIST` for the table with admissible name *tblname*, if exists; otherwise `false` is returned.

```
gap> FileNameCharTable( "M22mod3" );
"ctbmathi"
gap> FileNameCharTable( "J5" );
false
```

Chapter 54

Class Functions

This chapter introduces class functions and group characters in GAP3.

First section 54.1 tells about the ideas why to use these structures besides the characters and character tables described in chapters 49 and 51.

The subsequent section 54.2 tells details about the implementation of group characters and class functions in GAP3.

Sections 54.3 and 54.4 tell about the operators and functions for class functions and (virtual) characters.

Sections 54.5, 54.6, 54.7, and 54.11 describe how to construct such class functions and group characters.

Sections 54.8, 54.9, and 54.10 describe the characteristic functions of class functions and virtual characters.

Then sections 54.12 and 54.13 describe other functions for characters.

Then sections 54.14, 54.15, 54.16, and 54.17 tell about some functions and record components to access and store frequently used (normal) subgroups.

The final section 54.18 describes the records that implement class functions.

In this chapter, all examples use irreducible characters of the symmetric group S_4. For running the examples, you must first define the group and its characters as follows.

```
gap> S4:= SolvableGroup( "S4" );;
gap> irr:= Irr( S4 );;
```

54.1 Why Group Characters

When one says "χ is a character of a group G" then this object χ carries a lot of information. χ has certain properties such as being irreducible or not. Several subgroups of G are related to χ, such as the kernel and the centre of χ. And one can apply operators to χ, such as forming the conjugate character under the action of an automorphism of G, or computing the determinant of χ.

In GAP3, the characters known from chapters 49 and 51 are just lists of character values. This has several disadvantages. Firstly one cannot store knowledge about a character directly in the character, and secondly for every computation that requires more than just the character values one has to regard this list explicitly as a character belonging to a character table. In practice this means that the user has the task to put the objects into the right context, or –more concrete– the user has to supply lots of arguments.

This works nicely for characters that are used without groups, like characters of library tables. And if one deals with incomplete character tables often it is necessary to specify the arguments explicitly, for example one has to choose a fusion map or power map from a set of possibilities.

But for dealing with a group and its characters, and maybe also subgroups and their characters, it is desirable that GAP3 keeps track of the interpretation of characters.

Because of this it seems to be useful to introduce an alternative concept where a group character in GAP3 is represented as a record that contains the character values, the underlying group or character table, an appropriate operations record, and all the knowledge about the group character.

Together with characters, also the more general class functions and virtual characters are implemented.

Here is an **example** that shows both approaches. First we define the groups.

```
gap> S4:= SolvableGroup( "S4" );;
gap> D8:= SylowSubgroup( S4, 2 );; D8.name:= "D8";;
```

We do some computations using the functions described in chapters 51 and 49.

```
gap> t   := CharTable( S4 );;
gap> tD8 := CharTable( D8 );;
gap> FusionConjugacyClasses( D8, S4 );;
gap> chi:= tD8.irreducibles[2];
[ 1, -1, 1, 1, -1 ]
gap> Tensored( [ chi ], [ chi ] )[1];
[ 1, 1, 1, 1, 1 ]
gap> ind:= Induced( tD8, t, [ chi ] )[1];
[ 3, -1, 0, 1, -1 ]
gap> List( t.irreducibles, x -> ScalarProduct( t, x, ind ) );
[ 0, 0, 0, 1, 0 ]
gap> det:= DeterminantChar( t, ind );
[ 1, 1, 1, -1, -1 ]
gap> cent:= CentralChar( t, ind );
[ 1, -1, 0, 2, -2 ]
gap> rest:= Restricted( t, tD8, [ cent ] )[1];
[ 1, -1, -1, 2, -2 ]
```

And now we do the same calculations with the class function records.

```
gap> irr   := Irr( S4 );;
gap> irrD8 := Irr( D8 );;
gap> chi:= irrD8[2];
Character( D8, [ 1, -1, 1, 1, -1 ] )
```

```
gap> chi * chi;
Character( D8, [ 1, 1, 1, 1, 1 ] )
gap> ind:= chi ^ S4;
Character( S4, [ 3, -1, 0, 1, -1 ] )
gap> List( irr, x -> ScalarProduct( x, ind ) );
[ 0, 0, 0, 1, 0 ]
gap> det:= Determinant( ind );
Character( S4, [ 1, 1, 1, -1, -1 ] )
gap> cent:= Omega( ind );
ClassFunction( S4, [ 1, -1, 0, 2, -2 ] )
gap> rest:= Character( D8, cent );
Character( D8, [ 1, -1, -1, 2, -2 ] )
```

Of course we could have used the `Induce` and `Restricted` function also for lists of class functions.

```
gap> Induced( tD8, t, tD8.irreducibles{ [ 1, 3 ] } );
[ [ 3, 3, 0, 1, 1 ], [ 3, 3, 0, -1, -1 ] ]
gap> Induced( irrD8{ [ 1, 3 ] }, S4 );
[ Character( S4, [ 3, 3, 0, 1, 1 ] ),
  Character( S4, [ 3, 3, 0, -1, -1 ] ) ]
```

If one deals with complete character tables then often the table provides enough information, so it is possible to use the table instead of the group.

```
gap> s5 := CharTable( "A5.2" );; irrs5 := Irr( s5  );;
gap> m11:= CharTable( "M11"  );; irrm11:= Irr( m11 );;
gap> irrs5[2];
Character( CharTable( "A5.2" ), [ 1, 1, 1, 1, -1, -1, -1 ] )
gap> irrs5[2] ^ m11;
Character( CharTable( "M11" ), [ 66, 2, 3, -2, 1, -1, 0, 0, 0, 0 ] )
gap> Determinant( irrs5[4] );
Character( CharTable( "A5.2" ), [ 1, 1, 1, 1, -1, -1, -1 ] )
```

In this case functions that compute **normal** subgroups related to characters will return the list of class positions corresponding to that normal subgroup.

```
gap> Kernel( irrs5[2] );
[ 1, 2, 3, 4 ]
```

But if we ask for non-normal subgroups of course there is no chance to get an answer without the group, for example inertia subgroups cannot be computed from character tables.

54.2 More about Class Functions

Let G be a finite group. A **class function** of G is a function from G into the complex numbers (or a subfield of the complex numbers) that is constant on conjugacy classes of G. Addition, multiplication, and scalar multiplication of class functions are defined pointwise. Thus the set of all class functions of G is an algebra (or ring, or vector space).

Class functions and (virtual) group characters

Every mapping with source G that is constant on conjugacy classes of G is called a **class function** of G. Differences of characters of G are called **virtual characters** of G.

Class functions occur in a natural way when one deals with characters. For example, the central character of a group character is only a class function.

Every character is a virtual character, and every virtual character is a class function. Any function or operator that is applicable to a class function can of course be applied to a (virtual) group character. There are functions only for (virtual) group characters, like `IsIrreducible`, which doesn't make sense for a general class function, and there are also functions that do not make sense for virtual characters but only for characters, like `Determinant`.

Class functions as mappings

In GAP3, class functions of a group G are mappings (see chapter 43) with source G and range `Cyclotomics` (or a subfield). All operators and functions for mappings (like 43.8 `Image`, 43.12 `PreImages`) can be applied to class functions.

Note, however, that the operators `*` and `^` allow also other arguments than mappings do (see 54.3).

54.3 Operators for Class Functions

chi `=` *psi*
chi `<` *psi*

Equality and comparison of class functions are defined as for mappings (see 43.6); in case of equal source and range the `values` components are used to compute the result.

```
gap> irr[1]; irr[2];
Character( S4, [ 1, 1, 1, 1, 1 ] )
Character( S4, [ 1, 1, 1, -1, -1 ] )
gap> irr[1] < irr[2];
false
gap> irr[1] > irr[2];
true
gap> irr[1] = Irr( SolvableGroup( "S4" ) )[1];
false     # The groups are different.
```

chi `+` *psi*
chi `-` *psi*

`+` and `-` denote the addition and subtraction of class functions.

n `*` *chi*
chi `*` *psi*

`*` denotes (besides the composition of mappings, see 43.7) the multiplication of a class function *chi* with a scalar *n* and the tensor product of two class functions.

chi `/` *n*

`/` denotes the division of the class function *chi* by a scalar *n*.

```
gap> psi:= irr[3] * irr[4];
```

```
Character( S4, [ 6, -2, 0, 0, 0 ] )
gap> psi:= irr[3] - irr[1];
VirtualCharacter( S4, [ 1, 1, -2, -1, -1 ] )
gap> phi:= psi * irr[4];
VirtualCharacter( S4, [ 3, -1, 0, -1, 1 ] )
gap> IsCharacter( phi ); phi;
true
Character( S4, [ 3, -1, 0, -1, 1 ] )
gap> psi:= ( 3 * irr[2] - irr[3] ) * irr[4];
VirtualCharacter( S4, [ 3, -1, 0, -3, 3 ] )
gap> 2 * psi ;
VirtualCharacter( S4, [ 6, -2, 0, -6, 6 ] )
gap> last / 3;
ClassFunction( S4, [ 2, -2/3, 0, -2, 2 ] )
```

chi ^ *n*
g ^ *chi*

denote the tensor power by a nonnegative integer n and the image of the group element g, like for all mappings (see 43.7).

chi ^ *g*

is the conjugate class function by the group element g, that must be an element of the parent of the source of *chi* or something else that acts on the source via `^`. If *chi*`.source` is not a permutation group then g may also be a permutation that is interpreted as acting by permuting the classes (This maybe useful for table characters.).

chi ^ *G*

is the induced class function.

```
gap> V4:= Subgroup( S4, S4.generators{ [ 3, 4 ] } );
Subgroup( S4, [ c, d ] )
gap> V4.name:= "V4";;
gap> V4irr:= Irr( V4 );;
gap> chi:= V4irr[3];
Character( V4, [ 1, -1, 1, -1 ] )
gap> chi ^ S4;
Character( S4, [ 6, -2, 0, 0, 0 ] )
gap> chi ^ S4.2;
Character( V4, [ 1, -1, -1, 1 ] )
gap> chi ^ ( S4.2 ^ 2 );
Character( V4, [ 1, 1, -1, -1 ] )
gap> S4.3 ^ chi; S4.4 ^ chi;
1
-1
gap> chi ^ 2;
Character( V4, [ 1, 1, 1, 1 ] )
```

54.4 Functions for Class Functions

Besides the usual **mapping functions** (see chapter 43 for the details.), the following polymorphic functions are overlaid in the `operations` records of class functions and (virtual) characters. They are listed in alphabetical order.

`Centre(` *chi* `)`
: centre of a class function

`Constituents(` *chi* `)`
: set of irreducible characters of a virtual character

`Degree(` *chi* `)`
: degree of a class function

`Determinant(` *chi* `)`
: determinant of a character

`Display(` *chi* `)`
: displays the class function with the table head

`Induced(` *list*`,` *G* `)`
: induced class functions corresp. to class functions in the list *list* from subgroup H to group G

`IsFaithful(` *chi* `)`
: property check (virtual characters only)

`IsIrreducible(` *chi* `)`
: property check (characters only)

`Kernel(` *chi* `)`
: kernel of a class function

`Norm(` *chi* `)`
: norm of class function

`Omega(` *chi* `)`
: central character

`Print(` *chi* `)`
: prints a class function

`Restricted(` *list*`,` *H* `)`
: restrictions of class functions in the list *list* to subgroup H

`ScalarProduct(` *chi*`,` *psi* `)`
: scalar product of two class functions

54.5 ClassFunction

`ClassFunction(` *G*`,` *values* `)`

returns the class function of the group G with values list *values*.

`ClassFunction(` *G*`,` *chi* `)`

returns the class function of G corresponding to the class function *chi* of H. The group H can be a factor group of G, or G can be a subgroup or factor group of H.

```
gap> phi:= ClassFunction( S4, [ 1, -1, 0, 2, -2 ] );
ClassFunction( S4, [ 1, -1, 0, 2, -2 ] )
gap> coeff:= List( irr, x -> ScalarProduct( x, phi ) );
```

```
[ -1/12, -1/12, -1/6, 5/4, -3/4 ]
gap> ClassFunction( S4, coeff );
ClassFunction( S4, [ -1/12, -1/12, -1/6, 5/4, -3/4 ] )
gap> syl2:= SylowSubgroup( S4, 2 );;
gap> ClassFunction( syl2, phi );
ClassFunction( D8, [ 1, -1, -1, 2, -2 ] )
```

54.6 VirtualCharacter

`VirtualCharacter(` G, *values* `)`

returns the virtual character of the group G with values list *values*.

`VirtualCharacter(` G, *chi* `)`

returns the virtual character of G corresponding to the virtual character *chi* of H. The group H can be a factor group of G, or G can be a subgroup or factor group of H.

```
gap> syl2:= SylowSubgroup( S4, 2 );;
gap> psi:= VirtualCharacter( S4, [ 0, 0, 3, 0, 0 ] );
VirtualCharacter( S4, [ 0, 0, 3, 0, 0 ] )
gap> VirtualCharacter( syl2, psi );
VirtualCharacter( D8, [ 0, 0, 0, 0, 0 ] )
gap> S3:= S4 / V4;
Group( a, b )
gap> VirtualCharacter( S3, irr[3] );
VirtualCharacter( Group( a, b ), [ 2, -1, 0 ] )
```

Note that it is not checked whether the result is really a virtual character.

54.7 Character

`Character(` *repres* `)`

returns the character of the group representation *repres*.

`Character(` G, *values* `)`

returns the character of the group G with values list *values*.

`Character(` G, *chi* `)`

returns the character of G corresponding to the character *chi* with source H. The group H can be a factor group of G, or G can be a subgroup or factor group of H.

```
gap> syl2:= SylowSubgroup( S4, 2 );;
gap> Character( syl2, irr[3] );
Character( D8, [ 2, 2, 2, 0, 0 ] )
gap> S3:= S4 / V4;
Group( a, b )
gap> Character( S3, irr[3] );
Character( Group( a, b ), [ 2, -1, 0 ] )
gap> reg:= Character( S4, [ 24, 0, 0, 0, 0 ] );
Character( S4, [ 24, 0, 0, 0, 0 ] )
```

Note that it is not checked whether the result is really a character.

54.8 IsClassFunction

`IsClassFunction(` *obj* `)`

returns `true` if *obj* is a class function, and `false` otherwise.

```
gap> chi:= S4.charTable.irreducibles[3];
[ 2, 2, -1, 0, 0 ]
gap> IsClassFunction( chi );
false
gap> irr[3];
Character( S4, [ 2, 2, -1, 0, 0 ] )
gap> IsClassFunction( irr[3] );
true
```

54.9 IsVirtualCharacter

`IsVirtualCharacter(` *obj* `)`

returns `true` if *obj* is a virtual character, and `false` otherwise. For a class function *obj* that does not know whether it is a virtual character, the scalar products with all irreducible characters of the source of *obj* are computed. If they are all integral then *obj* is turned into a virtual character record.

```
gap> psi:= irr[3] - irr[1];
VirtualCharacter( S4, [ 1, 1, -2, -1, -1 ] )
gap> cf:= ClassFunction( S4, [ 1, 1, -2, -1, -1 ] );
ClassFunction( S4, [ 1, 1, -2, -1, -1 ] )
gap> IsVirtualCharacter( cf );
true
gap> IsCharacter( cf );
false
gap> cf;
VirtualCharacter( S4, [ 1, 1, -2, -1, -1 ] )
```

54.10 IsCharacter

`IsCharacter(` *obj* `)`

returns `true` if *obj* is a character, and `false` otherwise. For a class function *obj* that does not know whether it is a character, the scalar products with all irreducible characters of the source of *obj* are computed. If they are all integral and nonegative then *obj* is turned into a character record.

```
gap> psi:= ClassFunction( S4, S4.charTable.centralizers );
ClassFunction( S4, [ 24, 8, 3, 4, 4 ] )
gap> IsCharacter( psi ); psi;
true
Character( S4, [ 24, 8, 3, 4, 4 ] )
gap> cf:= ClassFunction( S4, irr[3] - irr[1] );
ClassFunction( S4, [ 1, 1, -2, -1, -1 ] )
gap> IsCharacter( cf ); cf;
```

```
false
VirtualCharacter( S4, [ 1, 1, -2, -1, -1 ] )
```

54.11 Irr

`Irr(` *G* `)`

returns the list of irreducible characters of the group G. If necessary the character table of G is computed. The succession of characters is the same as in `CharTable(` *G* `)`.

```
gap> Irr( SolvableGroup( "S4" ) );
[ Character( S4, [ 1, 1, 1, 1, 1 ] ),
  Character( S4, [ 1, 1, 1, -1, -1 ] ),
  Character( S4, [ 2, 2, -1, 0, 0 ] ),
  Character( S4, [ 3, -1, 0, 1, -1 ] ),
  Character( S4, [ 3, -1, 0, -1, 1 ] ) ]
```

54.12 InertiaSubgroup

`InertiaSubgroup(` *G*`,` *chi* `)`

For a class function *chi* of a normal subgroup N of the group G, `InertiaSubgroup(` *G*`,` *chi* `)` returns the inertia subgroup $I_G(chi)$, that is, the subgroup of all those elements $g \in G$ that satisfy $chi\hat{\ }g = chi$.

```
gap> V4:= Subgroup( S4, S4.generators{ [ 3, 4 ] } );
Subgroup( S4, [ c, d ] )
gap> irrsub:= Irr( V4 );
#W  Warning: Group has no name
[ Character( Subgroup( S4, [ c, d ] ), [ 1, 1, 1, 1 ] ),
  Character( Subgroup( S4, [ c, d ] ), [ 1, 1, -1, -1 ] ),
  Character( Subgroup( S4, [ c, d ] ), [ 1, -1, 1, -1 ] ),
  Character( Subgroup( S4, [ c, d ] ), [ 1, -1, -1, 1 ] ) ]
gap> List( irrsub, x -> InertiaSubgroup( S4, x ) );
[ Subgroup( S4, [ a, b, c, d ] ), Subgroup( S4, [ a*b^2, c, d ] ),
  Subgroup( S4, [ a*b, c, d ] ), Subgroup( S4, [ a, c, d ] ) ]
```

54.13 OrbitsCharacters

`OrbitsCharacters(` *irr* `)`

returns a list of orbits of the characters *irr* under the action of Galois automorphisms and multiplication with linear characters in *irr*. This is used for functions that need to consider only representatives under the operation of this group, like 55.9.

`OrbitsCharacters` works also for *irr* a list of character value lists. In this case the result contains orbits of these lists.

Note that `OrbitsCharacters` does not require that *irr* is closed under the described action, so the function may also be used to **complete** the orbits.

```
gap> irr:= Irr( SolvableGroup( "S4" ) );;
gap> OrbitsCharacters( irr );
```

```
[ [ Character( S4, [ 1, 1, 1, -1, -1 ] ),
      Character( S4, [ 1, 1, 1, 1, 1 ] ) ],
  [ Character( S4, [ 2, 2, -1, 0, 0 ] ) ],
  [ Character( S4, [ 3, -1, 0, -1, 1 ] ),
      Character( S4, [ 3, -1, 0, 1, -1 ] ) ] ]
gap> OrbitsCharacters( List( irr{ [1,2,4] }, x -> x.values ) );
[ [ [ 1, 1, 1, -1, -1 ], [ 1, 1, 1, 1, 1 ] ],
  [ [ 3, -1, 0, 1, -1 ], [ 3, -1, 0, -1, 1 ] ] ]
```

54.14 Storing Subgroup Information

Many computations for a group character χ of a group G, such as that of kernel or centre of χ, involve computations in (normal) subgroups or factor groups of G.

There are two aspects that make it reasonable to store relevant information used in these computations.

First it is possible to use the character table of a group for computations with the group. For example, suppose we know for every normal subgroup N the list of positions of conjugacy classes that form N. Then we can compute the intersection of normal subgroups efficiently by intersecting the corresponding lists.

Second one should try to reuse (expensive) information one has computed. Suppose you need the character table of a certain subgroup U that was constructed for example as inertia subgroup of a character. Then it may be probable that this group has been constructed already. So one should look whether U occurs in a list of interesting subgroups for that the tables are already known.

This section lists several data structures that support storing and using information about subgroups.

Storing Normal Subgroup Information

In some cases a question about a normal subgroup N can be answered efficiently if one knows the character table of G and the G-conjugacy classes that form N, e.g., the question whether a character of G restricts irreducibly to N. But other questions require the computation of the group N or even more information, e.g., if we want to know whether a character restricts homogeneously to N this will in general require the computation of the character table of N.

In order to do such computations only once, we introduce three components in the group record of G to store normal subgroups, the corresponding lists of conjugacy classes, and (if known) the factor groups, namely

`nsg`
: a list of (not necessarily all) normal subgroups of G,

`nsgclasses`
: at position i the list of positions of conjugacy classes forming the i-th entry of the `nsg` component,

`nsgfactors`
: at position i (if bound) the factor group modulo the i-th entry of the `nsg` component.

The functions

`NormalSubgroupClasses`,
`FactorGroupNormalSubgroupClasses`,
`ClassesNormalSubgroup`

initialize these components and update them. They are the only functions that do this.

So if you need information about a normal subgroup of G for that you know the G-conjugacy classes, you should get it using `NormalSubgroupClasses`. If the normal subgroup was already stored it is just returned, with all the knowledge it contains. Otherwise the normal subgroup is computed and added to the lists, and will be available for the next call.

Storing information for computing conjugate class functions

The computation of conjugate class functions requires the computation of permutatins of the list of conjugacy classes. In order to minimize the number of membership tests in conjugacy classes it is useful to store a partition of classes that is respected by every admissible permutation. This is stored in the component `globalPartitionClasses`.

If the normalizer N of H in its parent is stored in H, or if H is normal in its parent then the component `permClassesHomomorphism` is used. It holds the group homomorphism mapping every element of N to the induced permutation of classes.

Both components are generated automatically when they are needed.

Storing inertia subgroup information

Let N be the normalizer of H in its parent, and χ a character of H. The inertia subgroup $I_N(\chi)$ is the stabilizer in N of χ under conjugation of class functions. Characters with same value distribution, like Galois conjugate characters, have the same inertia subgroup. It seems to be useful to store this information. For that, the `inertiaInfo` component of H is initialized when needed, a record with components `partitions` and `stabilizers`, both lists. The `stabilizers` component contains the stabilizer in N of the corresponding partition.

54.15 NormalSubgroupClasses

`NormalSubgroupClasses(` *G*`,` *classes* `)`

returns the normal subgroup of the group G that consists of the conjugacy classes whose positions are in the list *classes*.

If G`.nsg` does not contain the required normal subgroup, and if G contains the component G`.normalSubgroups` then the result and the group in G`.normalSubgroups` will be identical.

```
gap> ccl:= ConjugacyClasses( S4 );
[ ConjugacyClass( S4, IdAgWord ), ConjugacyClass( S4, d ),
  ConjugacyClass( S4, b ), ConjugacyClass( S4, a ),
  ConjugacyClass( S4, a*d ) ]
gap> NormalSubgroupClasses( S4, [ 1, 2 ] );
Subgroup( S4, [ c, d ] )
```

The list of classes corresponding to a normal subgroup is returned by 54.16.

54.16 ClassesNormalSubgroup

`ClassesNormalSubgroup(` G, N `)`

returns the list of positions of conjugacy classes of the group G that are contained in the normal subgroup N of G.

```
gap> ccl:= ConjugacyClasses( S4 );
[ ConjugacyClass( S4, IdAgWord ), ConjugacyClass( S4, d ),
  ConjugacyClass( S4, b ), ConjugacyClass( S4, a ),
  ConjugacyClass( S4, a*d ) ]
gap> V4:= NormalClosure( S4, Subgroup( S4, [ S4.4 ] ) );
Subgroup( S4, [ c, d ] )
gap> ClassesNormalSubgroup( S4, V4 );
[ 1, 2 ]
```

The normal subgroup corresponding to a list of classes is returned by 54.15.

54.17 FactorGroupNormalSubgroupClasses

`FactorGroupNormalSubgroupClasses(` G, *classes* `)`

returns the factor group of the group G modulo the normal subgroup of G that consists of the conjugacy classes whose positions are in the list *classes*.

```
gap> ccl:= ConjugacyClasses( S4 );
[ ConjugacyClass( S4, IdAgWord ), ConjugacyClass( S4, d ),
  ConjugacyClass( S4, b ), ConjugacyClass( S4, a ),
  ConjugacyClass( S4, a*d ) ]
gap> S3:= FactorGroupNormalSubgroupClasses( S4, [ 1, 2 ] );
Group( a, b )
```

54.18 Class Function Records

Every class function has the components

`isClassFunction`
: always `true`,

`source`
: the underlying group (or character table),

`values`
: the list of values, corresponding to the `conjugacyClasses` component of `source`,

`operations`
: the operations record which is one of `ClassFunctionOps`, `VirtualCharacterOps`, `CharacterOps`.

Optional components are

`isVirtualCharacter`
: The class function knows to be a virtual character.

`isCharacter`
: The class function knows to be a character.

Chapter 55

Monomiality Questions

This chapter describes functions dealing with monomiality questions.

Section 55.1 gives some hints how to use the functions in the package.

The next sections (see 55.2, 55.3, 55.4) describe functions that deal with character degrees and derived length.

The next sections describe tests for homogeneous restriction, quasiprimitivity, and induction from a normal subgroup of a group character (see 55.5, 55.6, 55.7, 55.8).

The next sections describe tests for subnormally monomiality, monomiality, and relatively subnormally monomiality of a group or group character (see 55.9, 55.10, 55.11, 55.12).

The final sections 55.13 and 55.14 describe functions that construct minimal nonmonomial groups, or check whether a group is minimal nonmonomial.

All examples in this chapter use the symmetric group S_4 and the special linear group $Sl(2,3)$. For running the examples, you must first define the groups.

```
gap> S4:= SolvableGroup( "S4" );;
gap> Sl23:= SolvableGroup( "Sl(2,3)" );;
```

55.1 More about Monomiality Questions

Group Characters

All the functions in this package assume **characters** to be character records as described in chapter 54.

Property Tests

When we ask whether a group character χ has a certain property, like quasiprimitivity, we usually want more information than yes or no. Often we are interested in the reason why a group character χ could be proved to have a certain property, e.g., whether monomiality of χ was proved by the observation that the underlying group is nilpotent, or if it was necessary to construct a linear character of a subgroup from that χ can be induced. In the latter case we also may be interested in this linear character.

Because of this the usual property checks of GAP3 that return either `true` or `false` are not sufficient for us. Instead there are test functions that return a record with the possibly useful information. For example, the record returned by the function `TestQuasiPrimitive` (see 55.6) contains the component `isQuasiPrimitive` which is the known boolean property flag, a component `comment` which is a string telling the reason for the value of the `isQuasiPrimitive` component, and in the case that the argument χ was a not quasiprimitive character the component `character` which is an irreducible constituent of a nonhomogeneous restriction of χ to a normal subgroup.

The results of these test functions are stored in the respective records, in our example χ will have a component `testQuasiPrimitive` after the call of `TestQuasiPrimitive`.

Besides these test functions there are also the known property checks, e.g., the function `IsQuasiPrimitive` which will call `TestQuasiPrimitive` and return the value of the `isQuasiPrimitive` component of the result.

Where one should be careful

Monomiality questions usually involve computations in a lot of subgroups and factor groups of a given group, and for these groups often expensive calculations like that of the character table are necessary. If it is probable that the character table of a group will occur at a later stage again, one should try to store the group (with the character table stored in the group record) and use this record later rather than a new record that describes the same group.

An example: Suppose you want to restrict a character to a normal subgroup N that was constructed as a normal closure of some group elements, and suppose that you have already computed normal subgroups (by calls to `NormalSubgroups` or `MaximalNormalSubgroups`) and their character tables. Then you should look in the lists of known normal subgroups whether N is contained, and if yes you can use the known character table.

A mechanism that supports this for normal subgroups is described in 54.14. The following hint may be useful in this context.

If you know that sooner or later you will compute the character table of a group G then it may be advisable to do this as soon as possible. For example if you need the normal subgroups of G then they can be computed more efficiently if the character table of G is known, and they can be stored compatibly to the contained G-conjugacy classes. This correspondence of classes list and normal subgroup can be used very often.

Package Information

Some of the functions print (perhaps useful) information if the function `InfoMonomial` is set to the value `Print`.

55.2 Alpha

`Alpha(` G `)`

returns for a solvable group G a list whose i-th entry is the maximal derived length of groups $G/\ker(\chi)$ for $\chi \in Irr(G)$ with $\chi(1)$ at most the i-th irreducible degree of G.

The result is stored in the group record as G`.alpha`.

Note that calling this function will cause the computation of factor groups of G, so it works efficiently only for AG groups.

```
gap> Alpha( Sl23 );
[ 1, 3, 3 ]
gap> Alpha( S4 );
[ 1, 2, 3 ]
```

55.3 Delta

`Delta(` G `)`

returns for a solvable group G the list `[ 1, alp[2]-alp[1], ..., alp[`n`]-alp[`n`-1] ]` where `alp = Alpha(` G `)` (see 55.2).

```
gap> Delta( Sl23 );
[ 1, 2, 0 ]
gap> Delta( S4 );
[ 1, 1, 1 ]
```

55.4 BergerCondition

`BergerCondition(` *chi* `)`
`BergerCondition(` G `)`

Called with an irreducible character *chi* of the group G of degree d, `BergerCondition` returns `true` if *chi* satisfies $M' \leq \ker(\chi)$ for every normal subgroup M of G with the property that $M \leq \ker(\psi)$ for all $\psi \in Irr(G)$ with $\psi(1) < \chi(1)$, and `false` otherwise.

Called with a group G, `BergerCondition` returns `true` if all irreducible characters of G satisfy the inequality above, and `false` otherwise; in the latter case `InfoMonomial` tells about the smallest degree for that the inequality is violated.

For groups of odd order the answer is always `true` by a theorem of T. R. Berger (see [Ber76], Thm. 2.2).

```
gap> BergerCondition( S4 );
true
gap> BergerCondition( Sl23 );
false
gap> List( Irr( Sl23 ), BergerCondition );
[ true, true, true, false, false, false, true ]
gap> List( Irr( Sl23 ), Degree );
[ 1, 1, 1, 2, 2, 2, 3 ]
```

55.5 TestHomogeneous

`TestHomogeneous(` *chi*`,` N `)`

returns a record with information whether the restriction of the character *chi* of the group G to the normal subgroup N of G is homogeneous, i.e., is a multiple of an irreducible character of N.

N may be given also as list of conjugacy class positions w.r. to G.

The components of the result are

`isHomogeneous`
: `true` or `false`,

`comment`
: a string telling a reason for the value of the `isHomogeneous` component,

`character`
: irreducible constituent of the restriction, only bound if the restriction had to be checked,

`multiplicity`
: multiplicity of the `character` component in the restriction of *chi*.

```
gap> chi:= Irr( Sl23 )[4];
Character( Sl(2,3), [ 2, -2, 0, -1, 1, -1, 1 ] )
gap> n:= NormalSubgroupClasses( Sl23, [ 1, 2, 3 ] );
Subgroup( Sl(2,3), [ b, c, d ] )
gap> TestHomogeneous( chi, [ 1, 2, 3 ] );
rec(
  isHomogeneous := true,
  comment := "restricts irreducibly" )
gap> chi:= Irr( Sl23 )[7];
Character( Sl(2,3), [ 3, 3, -1, 0, 0, 0, 0 ] )
gap> TestHomogeneous( chi, n );
#W  Warning: Group has no name
rec(
  isHomogeneous := false,
  comment := "restriction checked",
  character := Character( Subgroup( Sl(2,3), [ b, c, d ] ),
    [ 1, 1, -1, 1, -1 ] ),
  multiplicity := 1 )
```

55.6 TestQuasiPrimitive

`TestQuasiPrimitive(` *chi* `)`

returns a record with information about quasiprimitivity of the character *chi* of the group G (i.e., whether *chi* restricts homogeneously to every normal subgroup of G).

The record contains the components

`isQuasiPrimitive`
: `true` or `false`,

`comment`
: a string telling a reason for the value of the `isQuasiPrimitive` component,

`character`
: an irreducible constituent of a nonhomogeneous restriction of *chi*, bound only if *chi* is not quasi-primitive.

`IsQuasiPrimitive(` *chi* `)`

returns `true` or `false`, depending on whether the character *chi* of the group G is quasiprimitive.

```
gap> chi:= Irr( Sl23 )[4];
Character( Sl(2,3), [ 2, -2, 0, -1, 1, -1, 1 ] )
gap> TestQuasiPrimitive( chi );
#W  Warning: Group has no name
rec(
  isQuasiPrimitive := true,
  comment := "all restrictions checked" )
gap> chi:= Irr( Sl23 )[7];
Character( Sl(2,3), [ 3, 3, -1, 0, 0, 0, 0 ] )
gap> TestQuasiPrimitive( chi );
rec(
  isQuasiPrimitive := false,
  comment := "restriction checked",
  character := Character( Subgroup( Sl(2,3), [ b, c, d ] ),
    [ 1, 1, -1, 1, -1 ] ) )
```

55.7 IsPrimitive for Characters

`IsPrimitive(` *chi* `)`

returns `true` if the irreducible character *chi* of the solvable group G is not induced from any proper subgroup of G, and `false` otherwise.

Note that an irreducible character of a solvable group is primitive if and only if it is quasiprimitive (see 55.6).

```
gap> IsPrimitive( Irr( Sl23 )[4] );
true
gap> IsPrimitive( Irr( Sl23 )[7] );
false
```

55.8 TestInducedFromNormalSubgroup

`TestInducedFromNormalSubgroup(` *chi*, *N* `)`
`TestInducedFromNormalSubgroup(` *chi* `)`

returns a record with information about whether the irreducible character *chi* of the group G is induced from a **proper** normal subgroup of G.

If *chi* is the only argument then it is checked whether there is a maximal normal subgroup of G from that *chi* is induced. If there is a second argument N, a normal subgroup of G, then it is checked whether *chi* is induced from N. N may also be given as the list of positions of conjugacy classes contained in the normal subgroup in question.

The result contains the components

`isInduced`
: `true` or `false`,

`comment`
: a string telling a reason for the value of the `isInduced` component,

`character`
: if bound, a character of a maximal normal subgroup of G or of the argument N from that *chi* is induced.

`IsInducedFromNormalSubgroup(` *chi* `)`

returns `true` if the group character *chi* is induced from a **proper** normal subgroup of the group of *chi*, and `false` otherwise.

```
gap> List( Irr( Sl23 ), IsInducedFromNormalSubgroup );
[ false, false, false, false, false, false, true ]
gap> List( Irr( S4 ){ [ 1, 3, 4 ] },
>          TestInducedFromNormalSubgroup );
#W  Warning: Group has no name
[ rec(
      isInduced := false,
      comment := "linear character" ), rec(
      isInduced := true,
      comment := "induced from component '.character'",
      character := Character( Subgroup( S4, [ b, c, d ] ),
        [ 1, 1, E(3), E(3)^2 ] ) ), rec(
      isInduced := false,
      comment := "all maximal normal subgroups checked" ) ]
```

55.9 TestSubnormallyMonomial

`TestSubnormallyMonomial(` *G* `)`
`TestSubnormallyMonomial(` *chi* `)`

returns a record with information whether the group G or the irreducible group character *chi* of the group G is subnormally monomial.

The result contains the components

`isSubnormallyMonomial`
: `true` or `false`,

`comment`
: a string telling a reason for the value of the `isSubnormallyMonomial` component,

`character`
: if bound, a character of G that is not subnormally monomial.

`IsSubnormallyMonomial(` *G* `)`
`IsSubnormallyMonomial(` *chi* `)`

returns `true` if the group G or the group character *chi* is subnormally monomial, and `false` otherwise.

```
gap> TestSubnormallyMonomial( S4 );
rec(
  isSubnormallyMonomial := false,
  character := Character( S4, [ 3, -1, 0, -1, 1 ] ),
```

```
  comment := "found not SM character" )
gap> TestSubnormallyMonomial( Irr( S4 )[4] );
rec(
  isSubnormallyMonomial := false,
  comment := "all subnormal subgroups checked" )
gap> TestSubnormallyMonomial( SolvableGroup( "A4" ) );
#W  Warning: Group has no name
rec(
  isSubnormallyMonomial := true,
  comment := "all irreducibles checked" )
```

55.10 TestMonomialQuick

`TestMonomialQuick( `*chi*` )`
`TestMonomialQuick( `*G*` )`

does some easy checks whether the irreducible character *chi* or the group *G* are monomial. `TestMonomialQuick` returns a record with components

`isMonomial`
: either `true` or `false` or the string `"?"`, depending on whether (non)monomiality could be proved, and

`comment`
: a string telling the reason for the value of the `isMonomial` component.

A group *G* is proved to be monomial by `TestMonomialQuick` if its order is not divisible by the third power of a prime, or if *G* is nilpotent or Sylow abelian by supersolvable. Nonsolvable groups are proved to me nonmonomial by `TestMonomialQuick`.

An irreducible character is proved to be monomial if it is linear, or if its codegree is a prime power, or if its group knows to be monomial, or if the factor group modulo the kernel can be proved to be monomial by `TestMonomialQuick`.

```
gap> TestMonomialQuick( Irr( S4 )[3] );
rec(
  isMonomial := true,
  comment := "kernel factor group is supersolvable" )
gap> TestMonomialQuick( S4 );
rec(
  isMonomial := true,
  comment := "abelian by supersolvable group" )
gap> TestMonomialQuick( Sl23 );
rec(
  isMonomial := "?",
  comment := "no decision by cheap tests" )
```

55.11 TestMonomial

`TestMonomial( `*chi*` )`
`TestMonomial( `*G*` )`

returns a record containing information about monomiality of the group G or the group character *chi* of a solvable group, respectively.

If a character *chi* is proved to be monomial the result contains components `isMonomial` (then `true`), `comment` (a string telling a reason for monomiality), and if it was necessary to compute a linear character from that *chi* is induced, also a component `character`.

If *chi* or G is proved to be nonmonomial the component `isMonomial` is `false`, and in the case of G a nonmonomial character is contained in the component `character` if it had been necessary to compute it.

If the program cannot prove or disprove monomiality then the result record contains the component `isMonomial` with value `"?"`.

This case occurs in the call for a character *chi* if and only if *chi* is not induced from the inertia subgroup of a component of any reducible restriction to a normal subgroup. It can happen that *chi* is monomial in this situation.

For a group this case occurs if no irreducible character can be proved to be nonmonomial, and if no decision is possible for at least one irreducible character.

`IsMonomial(` G `)`
`IsMonomial(` *chi* `)`

returns `true` if the group G or the character *chi* of a solvable group can be proved to be monomial, `false` if it can be proved to be nonmonomial, and the string `"?"` otherwise.

```
gap> TestMonomial( S4 );
rec(
  isMonomial := true,
  comment := "abelian by supersolvable group" )
gap> TestMonomial( Sl23 );
rec(
  isMonomial := false,
  comment := "list Delta( G ) contains entry > 1" )
```

`IsMonomial(` n `)`

for a positive integer n returns `true` if every solvable group of order n is monomial, and `false` otherwise.

```
gap> Filtered( [ 1 .. 111 ], x -> not IsMonomial( x ) );
[ 24, 48, 72, 96, 108 ]
```

55.12 TestRelativelySM

`TestRelativelySM(` G `)`
`TestRelativelySM(` *chi*, N `)`

If the only argument is a SM group G then `TestRelativelySM` returns a record with information about whether G is relatively subnormally monomial (relatively SM) with respect to every normal subgroup.

If there are two arguments, an irreducible character *chi* of a SM group G and a normal subgroup N of G, then `TestRelativelySM` returns a record with information whether *chi* is relatively SM with respect to N, i.e, whether there is a subnormal subgroup H of G that contains N such that *chi* is induced from a character ψ of H where the restriction of ψ to N is irreducible.

The component `isRelativelySM` is `true` or `false`, the component `comment` contains a string that describes the reason. If the argument is G, and G is not relatively SM with respect to a normal subgroup then the component `character` contains a not relatively SM character of such a normal subgroup.

Note: It is not checked whether G is SM.

```
gap> IsSubnormallyMonomial( SolvableGroup( "A4" ) );
#W  Warning: Group has no name
true
gap> TestRelativelySM( SolvableGroup( "A4" ) );
rec(
  isRelativelySM := true,
  comment :=
   "normal subgroups are abelian or have nilpotent factor group" )
```

55.13 IsMinimalNonmonomial

`IsMinimalNonmonomial(` *G* `)`

returns `true` if the solvable group G is a minimal nonmonomial group, and `false` otherwise. A group is called **minimal nonmonomial** if it is nonmonomial, and all proper subgroups and factor groups are monomial.

The solvable minimal nonmonomial groups were classified by van der Waall (see [vdW76]).

```
gap> IsMinimalNonmonomial( Sl23 );
true
gap> IsMinimalNonmonomial( S4 );
false
```

55.14 MinimalNonmonomialGroup

`MinimalNonmonomialGroup(` *p*`,` *factsize* `)`

returns a minimal nonmonomial group described by the parameters *factsize* and p if such a group exists, and `false` otherwise.

Suppose that a required group K exists. *factsize* is the size of the Fitting factor $K/F(K)$; this value must be 4, 8, an odd prime, twice an odd prime, or four times an odd prime.

In the case that *factsize* is twice an odd prime the centre $Z(K)$ iscyclic of order 2^{p+1}. In all other cases p denotes the (unique) prime that divides the order of $F(K)$.

The solvable minimal nonmonomial groups were classified by van der Waall (see [vdW76], the construction follows this article).

```
gap> MinimalNonmonomialGroup(  2,  3 ); # SL_2(3)
2^(1+2):3
gap> MinimalNonmonomialGroup(  3,  4 );
3^(1+2):4
```

```
gap> MinimalNonmonomialGroup(  5,  8 );
5^(1+2):Q8
gap> MinimalNonmonomialGroup( 13, 12 );
13^(1+2):2.D6
gap> MinimalNonmonomialGroup(  1, 14 );
2^(1+6):D14
gap> MinimalNonmonomialGroup(  2, 14 );
(2^(1+6)Y4):D14
```

Chapter 56

Getting and Installing GAP

GAP3 runs on several different operating systems. It behaves slightly different on each of those. This chapter describes the behaviour of GAP3, the installation, and the options on some of those operating systems.

Currently it contains sections for **UNIX** (see 56.2), **WINDOWS** (see 56.5), and **Mac/OSX** (see 56.9).

For other systems the section 56.13 gives hints how to approach such a port.

56.1 Getting GAP

GAP3 is distributed **free of charge**. You can give it away to your colleagues. GAP3 is **not** in the public domain, however. In particular you are not allowed to incorporate GAP3 or parts thereof into a commercial product.

This distribution of GAP3 is maintained by Jean Michel, `jean.michel@imj-prg.fr`. I would appreciate if let me know, e.g., by sending a short e-mail message, if you are using it. I also take bug reports.

If you publish some result that was partly obtained using GAP3, we would appreciate it if you would cite GAP3, just as you would cite another paper that you used. Specifically, please refer to

```
[S+ 97] Martin Sch"onert et.al.  GAP -- Groups, Algorithms, and Programming.
        Lehrstuhl D f"ur Mathematik, Rheinisch Westf"alische Technische
        Hochschule, Aachen, Germany, sixth edition, 1997.
```

Again we would appreciate if you could inform us about such a paper.

This distribultion contains **full source** for everything, the C code for the kernel, the GAP3 code for the library, and the LaTeX code for the manual, which has at present about 1900 pages. So it should be no problem to get GAP3, even if you have a rather uncommon system. Of course, ports to non UNIX systems may require some work. We already have ports for MS-DOS/Windows, and Apple Mac. Note that about 50 MByte of main memory and about 100MB of disk space are required to run GAP3. A full GAP3 installation, including all share packages and data libraries uses up to 100MB of disk space.

The easiest way to get this GAP3 distribution is to download it from `http://webusers.imj-prg.fr/~jmichel/ga`

The original site for the distribution is `http://www-gap.dcs.st-and.ac.uk/~gap`, but it now distributes GAP34.

At `http://webusers.imj-prg.fr/~jmichel/gap3` you can browse this manual and download the system.

56.2 GAP for UNIX

GAP3 runs best under UNIX. In fact it has being developed under UNIX. GAP3 running on any UNIX machine should behave exactly as described in the manual.

The section 56.3 describes how you install GAP3 on a UNIX machine, and the section 56.4 describe the options that GAP3 accepts under UNIX.

56.3 Installation of GAP for UNIX

Installation of GAP3 for UNIX is fairly easy.

First go to the directory where you want to install GAP3. If you will be the only user using GAP3, you probably should install it in you homedirectory. If other users will be using GAP3 also, you should install it in a public place, such as `/usr/local/lib/`. GAP3 will be installed in a subdirectory `gap3-jm` of this directory. You can later move GAP3 to a different location. For example you can first install it in your homedirectory and when it works move it to `/usr/local/lib/`. In the following example we will assume that you want to install GAP3 for your own use in your homedirectory. Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

Get the distribution `gap3-jmxxx.tar.gz` where `xxx` is the date of the version you are downloading (like `gap3-jm19feb18.tar.gz` for the version released on 30 november 2016). Unpack this archive in the chosen directory with the command

```
tar -xvzf gap3-jm19feb18.tar.gz
```

This will create a `gap3-jm` directory containing the GAP3 distribution. Then edit the shell script `gap.sh` in the `gap3-jm/bin` directory according to the instructions in this file (the main thing to do is to set the variable `GAP_DIR` to the directory where you installed GAP3). Then copy this script to a directory in your search path, i.e., `/bin/`. This script will start GAP3.

If there is no executable in the `bin` directory matching your system, it means you are attempting a new port. Change into the source directory `gap3-jm/src/` and execute `make` to see which compilation targets are predefined. Choose the best matching target. There is a good chance that `linux` or `linux32` will do the job, otherwise create a new target.

Now start GAP3 and try a few things. The `-b` option suppresses the banner. Note that GAP3 has to read most of the library for the fourth statement below.

```
you@ernie:~ > gap -b
gap> 2 * 3 + 4;
10
gap> Factorial( 30 );
265252859812191058636308480000000
gap> Factors( 10^42 + 1 );
```

```
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;
gap> Size( m11 );
7920
gap> Factors( 7920 );
[ 2, 2, 2, 2, 3, 3, 5, 11 ]
gap> Number( ConjugacyClasses( m11 ) );
10
```

Especially try the command line editing and history facilities, because the are probably the most machine dependent feature of GAP3. Enter a few commands and then make sure that *ctr*-`P` redisplays the last command, that *ctr*-`E` moves the cursor to the end of the line, that *ctr*-`B` moves the cursor back one character, and that *ctr*-`D` deletes single characters. So, after entering the above commands, typing
ctr-`P` *ctr*-`P` *ctr*-`E` *ctr*-`B` *ctr*-`B` *ctr*-`B` *ctr*-`B` *ctr*-`D` `1`
should give the following line.

```
gap> Factors( 7921 );
[ 89, 89 ]
```

If command line editing does not work, remove the file `system.o` and try to compile with a different target, i.e., `bsd` instead of `linux` or vice versa. If neither works, we suggest that you disable command line editing by calling GAP3 always with the `-n` option. In any case we would like to hear about such problems.

If your operating system has job control, make sure that you can still stop GAP3, which is usually done by pressing *ctr*-`Z`.

If you want to redo the manual after some changes, you need to have LaTeX installed, and `ruby` to make the `html` version. Go to the `doc` subdirectory of `gap3-jm` and do `make doc`. This will make files `gap3-jm/doc/manual.dvi`, `gap3-jm/manual.pdf` and `gap3-jm/htm/chap*.htm`. To remove the unneeded `.aux` files, you can excute `make clean` in the `doc` directory. The full manual is, to put it mildly, now rather long (1900 pages).

Thats all, finally you are done. We hope that you will enjoy using GAP3. If you have problems, do not hesitate to contact us.

56.4 Features of GAP for UNIX

When you start GAP3 for UNIX, you may specify a number of options on the command-line to change the default behaviour of GAP3. All these options start with a hyphen `-`, followed by a single letter. Options must not be grouped, e.g., `gap -gq` is illegal, use `gap -g -q` instead. Some options require an argument, this must follow the option and must be separated by a *space*, e.g., `gap -m 256k`, it is not correct to say `gap -m256k` instead.

GAP3 for UNIX will only accept lower case options.

As is described in the previous section (see 56.3) usually you will not execute GAP3 directly. Instead you will call a shell script, with the name `gap`, which in turn executes GAP3. This shell script sets some options as necessary to make GAP3 work on your system. This means that the default settings mentioned below may not be what you experience when you execute GAP3 on your system.

`-g`

The option `-g` tells GAP3 to print an information message every time a garbage collection is performed.

G collect garbage, 1567022 used, 412991 dead, 84.80MB free, 512MB total

For example, this tells you that there are 1567022 live objects that survived a garbage collection, that 412991 unused objects were reclaimed by it, and that 84 MBytes of totally allocated 512 MBytes are available afterwards.

`-l` *libname* The option `-l` tells GAP3 that the library of GAP3 functions is in the directory *libname*. Per default *libname* is `lib/`, i.e., the library is normally expected in the subdirectory `lib/` of the current directory. GAP3 searches for the library files, whose filenames end in `.g`, and which contain the functions initially known to GAP3, in this directory. *libname* should end with a pathname separator, i.e., `/`, but GAP3 will silently add one if it is missing. GAP3 will read the file *libname*`/init.g` during startup. If GAP3 cannot find this file it will print the following warning

```
gap: hmm, I cannot find 'lib/init.g', maybe use option '-l <lib>'?
```

If you want a bare bones GAP3, i.e., if you do not need any library functions, you may ignore this warning, otherwise you should leave GAP3 and start it again, specifying the correct library path using the `-l` option.

It is also possible to specify several alternative library paths by separating them with semicolons `;`. Note that in this case all path names must end with the pathname separator `/`. GAP3 will then search for its library files in all those directories in turn, reading the first it finds. E.g., if you specify `-l "lib/;/usr/local/lib/gap3-jm/lib/"` GAP3 will search for a library file first in the subdirectory `lib/` of the current directory, and if it does not find it there in the directory `/usr/local/lib/gap3-jm/lib/`. This way you can built your own directory of GAP3 library files that override the standard ones.

GAP3 searches for the group files, whose filenames end in `.grp`, and which contain the groups initially known to GAP3, in the directory one gets by replacing the string `lib` in *libname* with the string `grp`. If you do not want to put the group directory `grp/` in the same directory as the `lib/` directory, for example if you want to put the groups onto another hard disk partition, you have to edit the assignment in *libname*`/init.g` that reads

```
GRPNAME := ReplacedLib( "grp" );
```

This path can also consist of several alternative paths, just as the library path. If the library path consists of several alternative paths the default value for this path will consist of the same paths, where in each component the last occurrence of `lib/` is replaced by `grp/`.

Similar considerations apply to the character table files. Those filenames end in `.tbl`. GAP3 looks for those files in the directory given by `TBLNAME`. The default value for `TBLNAME` is obtained by replacing `lib` in *libname* with `tbl`.

`-h` *docname*

The option `-h` tells GAP3 that the on-line documentation for GAP3 is in the directory *docname*. Per default *docname* is obtained by replacing `lib` in *libname* with `doc`. *docname* should end with a pathname separator, i.e., `/`, but GAP3 will silently add one if it is missing. GAP3 will read the file *docname*`/manual.toc` when you first use the help system. If GAP3 cannot find this file it will print the following warning

```
help: hmm, I cannot open the table of contents file 'doc/manual.toc'
```

```
maybe you should use the option '-h <docname>'?
```

`-m` *memory*

The option `-m` tells GAP3 to allocate *memory* bytes at startup time. If the last character of *memory* is `k` or `K` it is taken in KBytes and if the last character is `m` or `M` *memory* is taken in MBytes.

Under UNIX the default amount of memory allocated by GAP3 is 4 MByte. The amount of memory should be large enough so that computations do not require too many garbage collections. On the other hand if GAP3 allocates more virtual memory than is physically available it will spend most of the time paging.

`-n`

The option `-n` tells GAP3 to disable the line editing and history (see 3.4).

You may want to do this if the command line editing is incompatible with another program that is used to run GAP3. For example if GAP3 is run from inside a GNU Emacs shell window, `-n` should be used since otherwise every input line will be echoed twice, once by Emacs and once by GAP3.

`-b`

The option `-b` tells GAP3 to suppress the banner. That means that GAP3 immediately prints the prompt. This is useful when you get tired of the banner after a while.

`-q`

The option `-q` tells GAP3 to be quiet. This means that GAP3 does not display the banner and the prompts `gap>`. This is useful if you want to run GAP3 as a filter with input and output redirection and want to avoid the the banner and the prompts clobbering the output file.

`-e`

The option `-e` tells GAP3 not to act on `ctrl-D`. This means that you have to type explicitely `quit;` to exit error loops or GAP3 at the prompt. This may be useful if you find `ctrl-D` too easy to type by accident.

`-x` *length*

With this option you can tell GAP3 how long lines are. GAP3 uses this value to decide when to split long lines. The default value is 80.

`-y` *length*

With this option you can tell GAP3 how many lines your screen has. GAP3 uses this value to decide after how many lines of on-line help it should display `-- <space> for more --`. The default value is 24.

GAP3 does not read the variables specifying the screen size automatically. On most shells you can tell GAP3 by giving the options

`-x $COLUMNS -y $LINES`

Further arguments are taken as filenames of files that are read by GAP3 during startup, after *libname*`/init.g` is read, but before the first prompt is printed. The files are read in the order in that they appear on the command line. GAP3 only accepts 14 filenames on the command line. If a file cannot be opened GAP3 will print an error message and will abort.

When you start GAP3, it looks for the file with the name `.gaprc` in your homedirectory. If such a file is found it is read after *libname*/`init.g`, but before any of the files mentioned on the command line are read. You can use this file for your private customizations. For example, if you have a file containing functions or data that you always need, you could read this from `.gaprc`. Or if you find some of the names in the library too long, you could define abbreviations for those names in `.gaprc`. The following sample `.gaprc` file does both.

```
Read("/usr/you/dat/mygroups.grp");
Op := Operation;
OpHom := OperationHomomorphism;
RepOp := RepresentativeOperation;
RepsOp := RepresentativesOperation;
```

`-r`

The option `-r` tells GAP3 to ignore a `.gaprc` file. This may be useful to see if a problem may be caused by the contents of your `.gaprc` file.

56.5 GAP for Windows

This sections contain information about GAP3 that is specific to the port of GAP3 for IBM PC compatibles under Windows (simply called GAP3 for Windows below).

To run GAP3 for Windows you need an IBM PC compatible with an Intel 80386, Intel 80486, or Intel Pentium processor, it will not run on IBM PC compatibles with an Intel 80186 or Intel 80286 processor. The system must have at least 4 MByte of main memory and a harddisk. The operating system must be Windows version 5.0 or later or Windows 3.1 or later (earlier versions may work, but this has not been tested).

The section 56.6 describes the copyright as it applies to the executable version that we distribute. The section 56.7 describes how you install GAP3 for Windows, and the section 56.8 describes the special features of GAP3 for Windows.

56.6 Copyright of GAP for Windows

In addition to the general copyright for GAP3 set forth in the Copyright the following terms apply to GAP3 for Windows.

The system dependent part for GAP3 for Windows was written by Steve Linton. He assigns the copyright to the Lehrstuhl D fuer Mathematik. Many thanks to Steve Linton for his work.

The executable of GAP3 for Windows that we distribute was compiled with DJ Delorie's port of the Free Software Foundation's GNU C compiler version 2.7.2. The compiler can be obtained by anonymous `ftp` from a variety of general public FTP archives. Many thanks to the Free Software Foundation and DJ Delorie for this amazing piece of work.

The GNU C compiler is

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

under the terms of the GNU General Public License (GPL). Note that the GNU GPL states that the mere act of compiling does not affect the copyright status of GAP3.

The modifications to the compiler to make it operating under Windows, the functions from the standard library `libpc.a`, the modifications of the functions from the standard library `libc.a` to make them operate under Windows, and the DOS extender `go32` (which is prepended to `gapexe.386`) are

also under the terms of the GNU GPL. The terms of the GPL require that we make the source code for `libpc.a` available. They can be obtained by writing to Steve Linton (however, it may be easier for you to `ftp` them from `grape.ecs.clarkson.edu` yourself). They also require that GAP3 falls under the GPL too, i.e., is distributed freely, which it basically does anyhow.

The functions in `libc.a` that GAP3 for the 386 uses are

under the following terms

56.7 Installation of GAP for Windows

Installation of GAP3 on WIndows is similar to Unix.

First go to a directory where you want to install GAP3, e.g., `c:\`. GAP3 will be installed in a subdirectory `gap3-jm\` of this directory. You can later move GAP3 to another location, for example you can first install it in `d:\tmp\` and once it works move it to `c:\`. In the following example we assume that you want to install GAP3 in `c:\`. Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

Get the GAP3 distribution onto your IBM PC compatible — see the Unix instructions how to get it from the web.

Change into the directory `gap-jm\bin\` and edit the script `gap.cmd`, which starts GAP3, according to the instructions in this file. Then copy this script to a directory in your search path, e.g., `c:\bin\`, with the commands

```
C: > chdir gap-jm\bin
C:\GAPR4P4\BIN > edit gap.cmd
# edit the script gap.cmd
```

```
C:\GAPR4P4\BIN > copy gap.cmd c:\bin\gap.cmd
C:\GAPR4P4\BIN > chdir ..\..
C: >
```

When you later move GAP3 to another location you must only edit this script.

An alternative possibility is to compile a version of GAP3 for use under Windows, on a UNIX system, using a **cross-compiler**. Cross-compiling versions of `gcc` can be found on some FTP archives, or compiled according to the instructions supplied with the `gcc` source distribution.

Start GAP3 and try a few things. Note that GAP3 has to read most of the library for the fourth statement below, so this takes quite a while. Subsequent definitions of groups will be much faster.

```
C: > gap -b
gap> 2 * 3 + 4;
10
gap> Factorial( 30 );
265252859812191058636308480000000
gap> Factors( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;
gap> Size( m11 );
7920
gap> Factors( 7920 );
[ 2, 2, 2, 2, 3, 3, 5, 11 ]
gap> Number( ConjugacyClasses( m11 ) );
10
```

Especially try the command line editing and history facilities, because the are probably the most machine dependent feature of GAP3. Enter a few commands and then make sure that *ctr*-`P` redisplays the last command, that *ctr*-`E` moves the cursor to the end of the line, that *ctr*-`B` moves the cursor back one character, and that *ctr*-`D` deletes single characters. So after entering the above three commands typing
ctr-`P` *ctr*-`P` *ctr*-`E` *ctr*-`B` *ctr*-`B` *ctr*-`B` *ctr*-`B` *ctr*-`D` `1`
should give the following line.

```
gap> Factors( 7921 );
[ 89, 89 ]
```

If you have a big version of LaTeX available you may now want to make a printed copy of the manual. Change into the directory `gap3-jm\doc\` and run LaTeX **twice** on the source. The first pass with LaTeX produces the `.aux` files, which resolve all the cross references. The second pass produces the final formatted **dvi** file `manual.dvi`. This will take quite a while, since the manual is large. Then print the **dvi** file. How you actually print the `dvi` file produced by LaTeX depends on the printer you have, the version of LaTeX you have, and whether you use a TeX-shell or not, so we will not attempt to describe it here.

```
C: > chdir gap3-jm\doc
C:\GAPR4P4\DOC > latex manual
# about 2000 messages about undefined references
C:\GAPR4P4\DOC > latex manual
```

```
# there should be no warnings this time
C:\GAPR4P4\DOC > dir manual.dvi
-a---   4591132 Nov 13 23:29 manual.dvi
C:\GAPR4P4\DOC > chdir ..\..
C: >
```

Note that because of the large number of cross references in the manual you need a **big** LaTeX to format the GAP3 manual. If you see the error message `TeX capacity exceeded`, you do not have a big LaTeX. In this case you may also obtain the already formatted **dvi** file `manual.dvi` from the same place where you obtained the rest of the GAP3 distribution.

Note that, apart from the `*.tex` files and the file `manual.bib` (bibliography database), which you absolutely need, we supply also the files `manual.toc` (table of contents), `manual.ind` (unsorted index), `manual.idx` (sorted index), and `manual.bbl` (bibliography). If those files are missing, or if you prefer to do everything yourself, here is what you will have to do. After the first pass with LaTeX, you will have preliminary `manual.toc` and `manual.ind` files. All the page numbers are still incorrect, because the do not account for the pages used by the table of contents itself. Now `bibtex manual` will create `manual.bbl` from `manual.bib`. After the second pass with LaTeX you will have a correct `manual.toc` and `manual.ind`. `makeindex` now produces the sorted index `manual.idx` from `manual.ind`. The third pass with LaTeX incorporates this index into the manual.

```
C: > chdir gap3-jm\doc
# about 2000 messages about undefined references
C:\GAPR4P4\DOC > bibtex manual
# bibtex prints the name of each file it is scanning
C:\GAPR4P4\DOC > latex manual
# still some messages about undefined citations
C:\GAPR4P4\DOC > makeindex manual
# makeindex prints some diagnostic output
C:\GAPR4P4\DOC > latex manual
# there should be no warnings this time
C:\GAPR4P4\DOC > chdir ..\..
C: >
```

The full manual is, to put it mildly, now rather long (almost 1600 pages). For this reason, it may be more convenient just to **print selected chapters**. This can be done using the `\includeonly` LaTeX command, which is present in `manual.tex` (around line 240), but commented out. To use this, you must first **LaTeX the whole manual** as normal, to obtain the complete set of `.aux` files and determine the pages and numbers of all the chapters and sections. After that, you can edit `manual.tex` to uncomment the `\includeonly` command and select the chapters you want. A good start can be to include only the first chapter, from the file `aboutgap.tex`, by editing the line to read `\includeonly{aboutgap}`. The next step is to LaTeX the manual again. This time only the selected chapter(s) and the table of contents and indices will be processed, producing a shorter `dvi` file that you can print by whatever means applies locally.

```
C:\GAPR4P4\DOC > latex manual
# many messages about undefined references, 1600 pages output
C:\GAPR4P4\DOC > edit manual.tex
# edit line 241 to include only aboutgap
```

```
C:\GAPR4P4\DOC > latex manual
# pages 0-196 and 1503-1553 only output no warnings
C:\GAPR4P4\DOC > dir manual.dvi
-a---   1291132 Nov 13 23:29 manual.dvi
C:\GAPR4P4\DOC >
# now print the DVI file in whatever way is appropriate
```

Thats all, finally you are done. We hope that you will enjoy using GAP3. If you have problems, do not hesitate to contact us.

56.8 Features of GAP for Windows

Note that GAP3 for Windows will use up to 128 MByte of extended memory (using XMS, VDISK memory allocation strategies) or up to 128 MByte of expanded memory (using VCPI programs, such as QEMM and 386MAX) and up to 128 MByte of disk space for swapping.

If you hit *ctr*-C the DOS extender (go32) catches it and aborts GAP3 immediately. The keys *ctr*-Z and *alt*-C can be used instead to interrupt GAP3.

The arrow keys *left*, *right*, *up*, *down*, *home*, *end*, and *delete* can be used for command line editing with their intuitive meaning.

Pathnames may be given inside GAP3 using either slash (/) or backslash (\) as a separator (though \ must be escaped in strings of course).

When you start GAP3 you may specify a number of options on the command-line to change the default behaviour of GAP3. All these options start with a hyphen -, followed by a single letter. Options must not be grouped, e.g., `gap -gq` is illegal, use `gap -g -q` instead. Some options require an argument, this must follow the option and must be separated by a *space*, e.g., `gap -m 256k`, it is not correct to say `gap -m256k` instead.

GAP3 for Windows accepts the following (lowercase) options.

-g

The options -g tells GAP3 to print a information message every time a garbage collection is performed.

    # G collect garbage, 1931 used, 5012 dead, 912 KB free, 3072 KB total

For example, this tells you that there are 1931 live objects that survived a garbage collection, that 5012 unused objects were reclaimed by it, and that 912 KByte of totally allocated 3072 KBytes are available afterwards.

-l *libname*

The option -l tells GAP3 that the library of GAP3 functions is in the directory *libname*. Per default *libname* is `lib/`, i.e., the library is normally expected in the subdirectory `lib/` of the current directory. GAP3 searches for the library files, whose filenames end in `.g`, and which contain the functions initially known to GAP3, in this directory. *libname* should end with a pathname separator, i.e., \, but GAP3 will silently add one if it is missing. GAP3 will read the file *libname*`\init.g` during startup. If GAP3 cannot find this file it will print the following warning

```
gap: hmm, I cannot find 'lib\init.g', maybe use option '-l <lib>'?
```

If you want a bare bones GAP3, i.e., if you do not need any library functions, you may ignore this warning, otherwise you should leave GAP3 and start it again, specifying the correct library path using the `-l` option.

It is also possible to specify several alternative library paths by separating them with semicolons `;`. Note that in this case all path names must end with the pathname separator `\`. GAP3 will then search for its library files in all those directories in turn, reading the first it finds. E.g., if you specify `-l "lib\;\usr\local\lib\gap3-jm\lib\"` GAP3 will search for a library file first in the subdirectory `lib\` of the current directory, and if it does not find it there in the directory `\usr\local\lib\gap3-jm\lib\`. This way you can built your own directory of GAP3 library files that override the standard ones.

GAP3 searches for the group files, whose filenames end in `.grp`, and which contain the groups initially known to GAP3, in the directory one gets by replacing the string `lib` in *libname* by the string `grp`. If you do not want to put the group directory `grp\` in the same directory as the `lib\` directory, for example if you want to put the groups onto another hard disk partition, you have to edit the assignment in *libname*`\init.g` that reads

```
GRPNAME := ReplacedString( LIBNAME, "lib", "grp" );
```

This path can also consist of several alternative paths, just as the library path. If the library path consists of several alternative paths the default value for this path will consist of the same paths, where in each component the last occurrence of `lib\` is replaced by `grp\`.

Similar considerations apply to the character table files. Those filenames end in `.tbl`. GAP3 looks for those files in the directory given by `TBLNAME`. The default value for `TBLNAME` is obtained by replacing `lib` in *libname* with `tbl`.

`-h` *docname*

The option `-h` tells GAP3 that the on-line documentation for GAP3 is in the directory *docname*. Per default *docname* is obtained by replacing `lib` in *libname* with `doc`. *docname* should end with a pathname separator, i.e., `\`, but GAP3 will silently add one if it is missing. GAP3 will read the file *docname*`\manual.toc` when you first use the help system. If GAP3 cannot find this file it will print the following warning

```
help: hmm, I cannot open the table of contents file 'doc\manual.toc'
maybe you should use the option '-h <docname>'?
```

`-m` *memory*

The option `-m` tells GAP3 to allocate *memory* bytes at startup time. If the last character of *memory* is `k` or `K` it is taken in KBytes and if the last character is `m` or `M` *memory* is taken in MBytes.

GAP3 for Windows will by default allocate 4 MBytes of memory. If you specify `-m` *memory* GAP3 will only allocate that much memory. The amount of memory should be large enough so that computations do not require too many garbage collections. On the other hand if GAP3 allocates more virtual memory than is physically available it will spend most of the time paging.

`-n`

The options `-n` tells GAP3 to disable the line editing and history (see 3.4).

There does not seem to be a good reason to do this on IBM PC compatibles.

`-b`

The option `-b` tells GAP3 to suppress the banner. That means that GAP3 immediately prints the prompt. This is useful when you get tired of the banner after a while.

`-q`

The option `-q` tells GAP3 to be quiet. This means that GAP3 does not display the banner and the prompts `gap>`. This is useful if you want to run GAP3 as a filter with input and output redirection and want to avoid the the banner and the prompts clobber the output file.

`-x` *length*

With this option you can tell GAP3 how long lines are. GAP3 uses this value to decide when to split long lines.

The default value is 80, which is correct if you start GAP3 from the desktop or one of the usual shells. However, if you start GAP3 from a window shell such as `gemini`, you may want to decrease this value. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`-y` *length*

With this option you can tell GAP3 how many lines your screen has. GAP3 uses this value to decide after how many lines of on-line help it should display `-- <space> for more --`.

The default value is 24, which is the right value if you start GAP3 from the desktop or one of the usual shells. However, if you start GAP3 from a window shell such as `gemini`, you may want to decrease this value. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`-z` *freq*

GAP3 for Windows checks in regular intervals whether the user has entered *ctr*-`Z` or *alt*-`C` to interrupt an ongoing computation. Under Windows this requires reading the keyboard status (UNIX on the other hand will deliver a signal to GAP3 when the user entered *ctr*-`C`), which is rather expensive. Therefor GAP3 only reads the keyboard status every *freq*-th time. The default is 20. With the option `-z` this value can be changed. Lower values make GAP3 more responsive to interrupts, higher values make GAP3 a little bit faster.

Further arguments are taken as filenames of files that are read by GAP3 during startup, after *libname*`\init.g` is read, but before the first prompt is printed. The files are read in the order in that they appear on the command line. GAP3 only accepts 14 filenames on the command line. If a file cannot be opened GAP3 will print an error message and will abort.

When you start GAP3, it looks for the file with the name `gap.rc` in your homedirectory (i.e., the directory defined by the environment variable `HOME`). If such a file is found it is read after *libname*`\init.g`, but before any of the files mentioned on the command line are read. You can use this file for your private customizations. For example, if you have a file containing functions or data that you always need, you could read this from `gap.rc`. Or if you find some of the names in the library too long, you could define abbreviations for those names in `gap.rc`. The following sample `gap.rc` file does both.

```
Read("c:\\gap\\dat\\mygroups.grp");
Op := Operation;
OpHom := OperationHomomorphism;
RepOp := RepresentativeOperation;
RepsOp := RepresentativesOperation;
```

56.9 GAP for Mac/OSX

This sections contain information about GAP3 that is specific to the port of GAP3 for Apple Macintosh systems under Mac/OSX (simply called GAP3 for Mac/OSX below).

To run GAP3 for Mac/OSX you need *to be written*

The section 56.10 describes the copyright as it applies to the executable version that we distribute. The section 56.11 describes how you install GAP3 for Mac/OSX, and the section 56.12 describes the special features of GAP3 for Mac/OSX.

56.10 Copyright of GAP for Mac/OSX

to be written

56.11 Installation of GAP for Mac/OSX

to be written

56.12 Features of GAP for Mac/OSX

to be written

56.13 Porting GAP

Porting GAP3 to a new operating system should not be very difficult. However, GAP3 expects some features from the operating system and the compiler and porting GAP3 to a system or with a compiler that do not have those features may prove very difficult.

The design of GAP3 makes it quite portable. GAP3 consists of a small kernel written in the programming language C and a large library written in the programming language provided by the GAP3 kernel, which is also called GAP3.

Once the kernel has been ported, the library poses no additional problem, because all those functions only need the kernel to work, they need no additional support from the environment.

The kernel itself is separated into a large part that is largely operating system and compiler independent, and one file that contains all the operating system and compiler dependent functions. Usually only this file must be modified to port GAP3 to a new operating system.

Now lets take a look at the minimal support that GAP3 needs from the operating system and the machine.

First of all you need enough filespace. The kernel sources and the object files need between 3.5 MByte and 4 MByte, depending on the size of object files produced by your compiler. The library takes up an additional 4.8 MBytes, and the online documentation also needs 4 MByte. So you need about 13 MByte of available filespace, for example on a harddisk.

Next you need enough main memory in your computer. The size of the GAP3 kernel varies between different machine, with as little as 300 KByte (compiled with GNU C on an Atari ST) and as much as 600 KByte (compiled with UNIX cc on a HP 9000/800). Add to that the fact the library of functions that GAP3 loads takes up another 1.5 MByte. So it is clear that at least 4 MByte of main memory are required to do any serious work with GAP3.

Note that this implies that there is no point in trying to port GAP3 to plain Windows running on IBM PCs and compatibles. The version of GAP3 for IBM PC compatibles that we provide runs on machines with the Intel 80386, Intel 80486, Pentium or Pentium Pro processor under extended DOS in protected 32 bit mode. (This is also necessary, because, as mentioned below, GAP3 wants to view its memory as a large flat address space.)

Next lets turn to the requirements for the C compiler and its library.

As was already mentioned, the GAP3 kernel is written in the C language. We have tried to use as few features of the C language as possible. GAP3 has been compiled without problems with compilers that adhere to the old definition from Kernighan and Ritchie, and with compilers that adhere to the new definition from the ANSI-C standard.

GAP3 was wriiten for 32-bit compilers (`sizeof(int)==4`), but it has been ported by Jean Michel to 64 bits, allowing use of terabytes of memory. Since Jean Michel works on Linux machines, this 64-bit version works for now only on such machines. The versions distributed for MAC/OSX and Windows are still 32-bit.

Dependencies on the operating system or compiler are separated in one special file which is called the `system` file. When you port GAP3 to a new operating system, you probably have to create a new `system` file. You should however look through the `system.c` file that we supply and take as much code from them as possible. Currently `system.c` supports Linux with gcc, Windows with the DJGPP compiler, and OS/X with gcc.

The `system` file contains the following functions.

First file input and output. The functions used by the three system files mentioned above are `fopen`, `fclose`, `fgets`, and `fputs`. They are pretty standard, and in fact are in the ANSI C standard library. The only thing that may be necessary is to make sure that files are opened in text mode. However, the most important transformation applied in text mode seems to be to replace the end of line sequence *newline-return*, used in some operating systems, with a single *newline*, used in C. However, since GAP3 treats *newline* and *return* both as whitespaces even this is not absolutely necessary.

Second you need character oriented input from the keyboard and to the screen. This is not absolutely necessary, you can use the line oriented input and output described above. However, if you want the history and the command line editing, GAP3 must be able to read every single character as the user types it without echo, and also must be able to put single characters to the screen. Reading characters unblocked and without echo is very system dependent.

Third you need a way to detect if the user typed *ctr*-`C` to interrupt an ongoing computation in GAP3. Again this is not absolutely necessary, you can do without. However if you want to be able to interrupt computations, GAP3 must be able to receive the interrupt. This can be done in two ways. Under UNIX you can catch the signal that is generated if the user types *ctr*-`C` (`SIGINT`). Under other operating systems that do not support such signals you can poll the input stream at regular intervals and simply look for *ctr*-`C`.

Fourth you need a way to find out how long GAP3 has been running. Again this is not absolutely necessary. You can simply always return 0, fooling GAP3 into thinking that it is extremely fast. However if you want timing statistics, GAP3 must be able to find out how much CPU time it has spent.

The last and most important function is the function to allocate memory for GAP3. GAP3 assumes that it can allocate the initial workspace with the function `SyGetmem` and expand this workspace on demand with further calls to `SyGetmem`. The memory allocated by consecutive calls to `SyGetmem` must be adjacent, because GAP3 wants to manage a single large block of memory. Usually this can be done with the C library function `sbrk`. If this does not work, you can define a large static array in the `system` file and return this on the first call to `SyGetmem` and return 0 on subsequent calls to indicate that this array cannot dynamically be expanded.

Chapter 57

Share Libraries

Contributions from people working at Lehrstuhl D, RWTH Aachen, or any other place can become available in GAP3 in two different ways:

1. They can become parts of the main GAP3 library of functions. Their origin will then be rather carefully documented in the respective program files, but will not occur in the description of these functions in the manual. This is e.g. the case – to mention just one of many such contributions – with programs for finding composition factors of permutation groups, written by Akos Seress. The reason for this decision about keeping track of the origin of such contributions is that quite often such functions in the main GAP3 library have a complicated history with changes and contributions from various people.

2. On the other hand there are packages written by one or several persons for specific purposes either in the GAP3 language or even in C which are made available en block in GAP3. Such packages will constitute **share libraries**. A share library will stay under the full responsibility of its author(s), which will be named in the respective chapter in the manual, they will in particular keep the copyright for this package, and they will also have to provide the documentation for it. However provisions will be made to call the functions of such a package like any other GAP3 functions, and to call the documentation via help functions like any other part of the GAP3 documentation. Also these packages will automatically be made available with the main body of GAP3 through ftp and will be sent together with the main body of GAP3 in case we have to fulfill a request to send GAP3 to institutions that cannot obtain GAP3 via electronic networks.

The inclusion of packages as GAP3 share libraries should be negotiated with Lehrstuhl D für Mathematik, RWTH Aachen, for certain standards of the documentation and program organisation that should be met in order to facilitate the use of the packages in the context of GAP3 without problems. A necessary condition for any package to become a GAP3 share library is that it is made available under the conditions formulated in the GAP3 copyright statement, in particular free of any charge, except for refund of expenses for sending, if such occur.

The first section describes how to load a share library package (see 57.1).

The next sections describe the ANU pq package and how to install it (see 57.2 and 57.3).

The next sections describe the ANU Sq package and how to install it (see 57.4 and 57.5).

The next sections describe the GRAPE package and how to install it (see 57.6 and 57.7).

The next sections describe the MeatAxe package and how to install it (see 57.8 and 57.9).

The next sections describe the NQ package and how to install it (see 57.10 and 57.11).

The next sections describe the SISYPHOS package and how to install it (see 57.12 and 57.13).

The next sections describe the Vector Enumeration package and how to install it (see 57.14 and 57.15).

The last sections describe the experimental X-Windows interface (see 57.16).

57.1 RequirePackage

`RequirePackage( name )`

`RequirePackage` will try to initialize the share library *name*. If the package *name* is not installed at your site `RequirePackage` will signal an error. If the package *name* is already initialized `RequirePackage` simply returns without any further actions.

```
gap> CartanMat( "A", 4 );
Error, Variable: 'CartanMat' must have a value
gap> ?CartanMat
CartanMat ________________________ Root systems and finite Coxeter groups

'CartanMat( <type>, <n> )'

returns the Cartan matrix of Dynkin type  <type> and rank <n>. Admissible
types are the  strings '"A"',   '"B"',   '"C"',    '"D"',   '"E"',
'"F"',    '"G"',   '"H"',    '"I"'.

    gap> C := CartanMat( "F", 4 );;
    gap> PrintArray( C );
    [ [   2,  -1,   0,   0 ],
      [  -1,   2,  -1,   0 ],
      [   0,  -2,   2,  -1 ],
      [   0,   0,  -1,   2 ] ]

For type  I_2(m),  which is in fact an infinity of types depending on the
number  m,  a third argument is needed specifying  the integer  m  so the
syntax is in fact 'CartanMat(  "I",  2, <m> )':

    gap> CartanMat( "I", 2, 5 );
    [ [ 2, E(5)^2+E(5)^3 ], [ E(5)^2+E(5)^3, 2 ] ]

'CartanMat( <type1>, <n1>, ... , <typek>, <nk> )'

returns the   direct sum  of   'CartanMat( <type1>, <n1> )',   ldots,
'CartanMat( <typek>, <nk> )'. One can use as argument a computed list of
types by 'ApplyFunc( CartanMat, [ <type1>, <n1>, ... , <typek>, <nk> ] )'.
```

```
    This function requires the package  "chevie"  (see "RequirePackage").

    gap> RequirePackage( "Chevie" );
    Error, share library "Chevie" is not installed in
    LoadPackage( name ) called from
    RequirePackage( "Chevie" ) called from
    main loop
    brk> quit;
    gap> RequirePackage( "chevie" );
--- Loading package chevie ------- version 4 development of 29Feb2016--------
    If you use this package in your work please cite the authors as follows:
(C) [Jean Michel] The development version of the CHEVIE package of GAP3
     Journal of algebra 435 (2015) 308--336
(C) [Meinolf Geck, Gerhard Hiss, Frank Luebeck, Gunter Malle, Goetz Pfeiffer]
     CHEVIE -- a system for computing and processing generic character tables
     Applicable Algebra in Engineering Comm. and Computing 7 (1996) 175--210

    gap> CartanMat( "A", 4 );;
    gap> PrintArray( last );
    [ [   2,  -1,   0,   0 ],
      [  -1,   2,  -1,   0 ],
      [   0,  -1,   2,  -1 ],
      [   0,   0,  -1,   2 ] ]
```

57.2 ANU pq Package

The ANU pq provides access to implementations of the following algorithms:

1. A p-quotient algorithm to compute a power-commutator presentation for a group of prime power order. The algorithm implemented here is based on that described in Newman and O'Brien (1996), Havas and Newman (1980), and papers referred to there. Another description of the algorithm appears in Vaughan-Lee (1990). A FORTRAN implementation of this algorithm was programmed by Alford and Havas. The basic data structures of that implementation are retained.

2. A p-group generation algorithm to generate descriptions of groups of prime power order. The algorithm implemented here is based on the algorithms described in Newman (1977) and O'Brien (1990). A FORTRAN implementation of this algorithm was earlier developed by Newman and O'Brien.

3. A standard presentation algorithm used to compute a canonical power-commutator presentation of a p-group. The algorithm implemented here is described in O'Brien (1994).

4. An algorithm which can be used to compute the automorphism group of a p-group. The algorithm implemented here is described in O'Brien (1994).

The following section describes the installation of the ANU pq package, a description of the functions available in the ANU pq package is given in chapter 58.

A reader interested in details of the algorithms and explanations of terms used is referred to [NO96], [HN80], [O´Br90], [O´Br94], [O´Br95], [New77], [VL84], [VL90b], and [VL90a].

For details about the implementation and the standalone version see the README. This implementation was developed in C by

Eamonn O'Brien
Lehrstuhl D fuer Mathematik
RWTH Aachen

e-mail obrien@math.rwth-aachen.de

57.3 Installing the ANU pq Package

The ANU pq is written in C and the package can only be installed under UNIX. It has been tested on DECstation running Ultrix, a HP 9000/700 and HP 9000/800 running HP-UX, a MIPS running RISC/os Berkeley, a NeXTstation running NeXTSTEP 3.0, and SUNs running SunOS.

If you got a complete binary and source distribution for your machine, nothing has to be done if you want to use the ANU pq for a single architecture. If you want to use the ANU pq for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the ANU pq package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a NeXTstation, called `bjerun`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `anupq.zoo` (see 56.1). Then you must locate the GAP3 directory containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be replaced by the current patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap     1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap   360891 Dec 27 15:16 anupq.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x   2 gap     3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap     1024 Nov  8  1991 grp
drwxr-xr-x   2 gap     2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap     2048 Nov 26 09:42 pkg
drwxr-xr-x   2 gap     2048 Nov 26 09:42 src
drwxr-xr-x   2 gap     1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x anupq
gap@tiffy:~ > cd gap3r4p0/pkg/anupq
gap@tiffy:../anupq> ls -l
```

```
drwxr-xr-x  5 gap       512 Feb 24 11:17 MakeLibrary
-rw-r--r--  1 gap     28926 Jun  8 14:21 Makefile
-rw-r--r--  1 gap      8818 Jun  8 14:21 README
-rw-r--r--  1 gap       753 Jun 23 18:59 StandardPres
drwxr-xr-x  2 gap      1024 Jun  8 14:15 TEST
drwxr-xr-x  2 gap       512 Jun 16 16:03 bin
drwxr-xr-x  2 gap       512 May 16 06:58 cayley
drwxr-xr-x  2 gap       512 Jun  8 08:48 doc
drwxr-xr-x  2 gap      1024 Mar  5 04:01 examples
drwxr-xr-x  2 gap       512 Jun 23 16:37 gap
drwxr-xr-x  2 gap       512 Jun 24 10:51 include
-rw-rw-rw-  1 gap       867 Jun  9 16:12 init.g
drwxr-xr-x  2 gap      1024 May 21 02:28 isom
drwxr-xr-x  2 gap       512 May 16 07:58 magma
drwxr-xr-x  2 gap      6656 Jun 24 11:10 src
```

Typing `make` will produce a list of possible target.

```
gap@tiffy:../anupq > make
usage: 'make <target> EXT=<ext>'  where <target> is one of
'dec-mips-ultrix-gcc2-gmp' for DECstations under Ultrix with gcc/gmp
'dec-mips-ultrix-cc-gmp'   for DECstations under Ultrix with cc/gmp
'dec-mips-ultrix-gcc2'     for DECstations under Ultrix with gcc
'dec-mips-ultrix-cc'       for DECstations under Ultrix with cc
'hp-hppa1.1-hpux-cc-gmp'   for HP 9000/700 under HP-UX with cc/gmp
'hp-hppa1.1-hpux-cc'       for HP 9000/700 under HP-UX with cc
'hp-hppa1.0-hpux-cc-gmp'   for HP 9000/800 under HP-UX with cc/gmp
'hp-hppa1.0-hpux-cc'       for HP 9000/800 under HP-UX with cc
'ibm-i386-386bsd-gcc2-gmp' for IBM PCs under 386BSD with gcc/gmp
'ibm-i386-386bsd-cc-gmp'   for IBM PCs under 386BSD with cc/gmp
'ibm-i386-386bsd-gcc2'     for IBM PCs under 386BSD with gcc2
'ibm-i386-386bsd-cc'       for IBM PCs under 386BSD with cc
'mips-mips-bsd-cc-gmp'     for MIPS under RISC/os Berkeley with cc/gmp
'mips-mips-bsd-cc'         for MIPS under RISC/os Berkeley with cc
'next-m68k-mach-gcc2-gmp'  for NeXT under Mach with gcc/gmp
'next-m68k-mach-cc-gmp'    for NeXT under Mach with cc/gmp
'next-m68k-mach-gcc2'      for NeXT under Mach with gcc
'next-m68k-mach-cc'        for NeXT under Mach with cc
'sun-sparc-sunos-gcc2-gmp' for SUN 4 under SunOs with gcc/gmp
'sun-sparc-sunos-cc-gmp'   for SUN 4 under SunOs with cc/gmp
'sun-sparc-sunos-gcc2'     for SUN 4 under SunOs with gcc2
'sun-sparc-sunos-cc'       for SUN 4 under SunOs with cc
'unix-gmp'                 for a generic unix system with cc/gmp
'unix'                     for a generic unix system with cc
'clean'                    remove all created files

   where <ext> should be a sensible extension, i.e.,
   'EXT=-sun-sparc-sunos' for SUN 4 or 'EXT=' if the PQ only
   runs on a single architecture
```

```
targets are listed according to preference,
i.e., 'sun-sparc-sunos-gcc2' is better than 'sun-sparc-sunos-cc'.
additional C compiler and linker flags can be passed with
'make <target> COPTS=<compiler-opts> LOPTS=<linker-opts>',
i.e., 'make sun-sparc-sunos-cc COPTS=-g LOPTS=-g'.

set GAP if gap 3.4 is not started with the command 'gap',
i.e., 'make sun-sparc-sunos-cc GAP=/home/gap/bin/gap-3.4'.

in order to use the GNU multiple precision (gmp) set
'GNUINC' (default '/usr/local/include') and
'GNULIB' (default '/usr/local/lib')
```

Select the target you need. If you have the **GNU** multiple precision arithmetic (gmp) installed on your system, select the target ending in `-gmp`. Note that the gmp is **not required**. In our case we first compile the DECstation version. We assume that the command to start GAP3 is `/usr/local/bin/gap` for `tiffy` and `bjerun` and `/rem/tiffy/usr/local/bin/gap` for `bert`.

```
gap@tiffy:../anupq > make dec-mips-ultric-cc \
                          GAP=/usr/local/bin/gap \
   EXT=-dec-mips-ultrix
# you will see a lot of messages and a few warnings
```

Now repeat the compilation for the NeXTstation. **Do not** forget to clean up.

```
gap@tiffy:../anupq > rlogin bjerun
gap@bjerun:~ > cd gap3r4p0/pkg/anupq
gap@bjerun:../anupq > make clean
gap@bjerun:../src > make next-m68k-mach-cc \
                        GAP=/usr/local/bin/gap \
 EXT=-next-m68k-mach
# you will see a lot of messages and a few warnings
gap@bjerun:../anupq > exit
gap@tiffy:../anupq >
```

Switch into the subdirectory `bin/` and create a script which will call the correct binary for each machine. A skeleton shell script is provided in `bin/pq.sh`.

```
gap@tiffy:../anupq > cd bin
gap@tiffy:../bin > cat > pq
#!/bin/csh
switch ( `hostname` )
  case 'tiffy':
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'bert':
    setenv ANUPQ_GAP_EXEC /rem/tiffy/usr/local/bin/gap ;
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'bjerun':
```

```
      limit stacksize 2048 ;
      exec $0-next-m68k-mach $* ;
      breaksw ;
    default:
      echo "pq: sorry, no executable exists for this machine" ;
      breaksw ;
endsw
ctr-D
gap@tiffy:../bin > chmod 755 pq
gap@tiffy:../bin > cd ..
```

Note that the NeXTstation requires you to raise the stacksize. If your default limit on any other machine for the stack size is less than 1024 you might need to add the `limit stacksize 2048` line.

If the documentation is not already installed or an older version is installed, copy the file `gap/anupq.tex` into the `doc/` directory and run latex again (see 56.3). In general the documentation will already be installed so you can just skip the following step.

```
gap@tiffy:../anupq > cp gap/anupq.tex ../../doc
gap@tiffy:../anupq > cd ../../doc
gap@tiffy:../doc > latex manual
# a few messages about undefined references
gap@tiffy:../doc > latex manual
# a few messages about undefined references
gap@tiffy:../doc > makeindex manual
# makeindex prints some diagnostic output
gap@tiffy:../doc > latex manual
# there should be no warnings this time
gap@tiffy:../doc cd ../pkg/anupq
```

Now it is time to test the installation. The first test will only test the ANU pq.

```
gap@tiffy:../anupq > bin/pq < gap/test1.pga
# a lot of messages ending in
##################################################
Starting group: c3c3 #2;2 #4;3
Order: 3^7
Nuclear rank: 3
3-multiplicator rank: 4
# of immediate descendants of order 3^8 is 7
# of capable immediate descendants is 5

##################################################
34 capable groups saved on file c3c3_class4
Construction of descendants took 1.92 seconds

Select option: 0
Exiting from p-group generation

Select option: 0
```

```
Exiting from ANU p-Quotient Program
Total user time in seconds is 1.97
gap@tiffy:../anupq > ls -l c3c3*
total 89
-rw-r--r--    1 gap    3320 Jun 24 11:24 c3c3_class2
-rw-r--r--    1 gap    5912 Jun 24 11:24 c3c3_class3
-rw-r--r--    1 gap   56184 Jun 24 11:24 c3c3_class4
gap:../anupq > rm c3c3_class*
```

The second test will test the stacksize. If it is too small you will get a memory fault, try to raise the stacksize as described above.

```
gap@tiffy:../anupq > bin/pq < gap/test2.pga
#  a lot of messages ending in
**************************************************
Starting group: c2c2 #1;1 #1;1 #1;1
Order: 2^5
Nuclear rank: 1
2-multiplicator rank: 3
Group c2c2 #1;1 #1;1 #1;1 is an invalid starting group

**************************************************
Starting group: c2c2 #2;1 #1;1 #1;1
Order: 2^5
Nuclear rank: 1
2-multiplicator rank: 3
Group c2c2 #2;1 #1;1 #1;1 is an invalid starting group
Construction of descendants took 0.47 seconds

Select option: 0
Exiting from p-group generation

Select option: 0
Exiting from ANU p-Quotient Program
Total user time in seconds is 0.50
gap@tiffy:../anupq > ls -l c2c2*
total 45
-rw-r--r--    1 gap   6228 Jun 24 11:25 c2c2_class2
-rw-r--r--    1 gap  11156 Jun 24 11:25 c2c2_class3
-rw-r--r--    1 gap   2248 Jun 24 11:25 c2c2_class4
-rw-r--r--    1 gap      0 Jun 24 11:25 c2c2_class5
gap:../anupq > rm c2c2_class*
```

The third example tests the link between the ANU pq and GAP3. If there is a problem you will get a error message saying `Error in system call to GAP`; if this happens, check the environment variable `ANUPQ_GAP_EXEC`.

```
gap@tiffy:../anupq > bin/pq < gap/test3.pga
#  a lot of messages ending in
**************************************************
Starting group: c5c5 #1;1 #1;1
```

```
Order: 5^4
Nuclear rank: 1
5-multiplicator rank: 2
# of immediate descendants of order 5^5 is 2

**************************************************
Starting group: c5c5 #1;1 #2;2
Order: 5^5
Nuclear rank: 3
5-multiplicator rank: 3
# of immediate descendants of order 5^6 is 3
# of immediate descendants of order 5^7 is 3
# of capable immediate descendants is 1
# of immediate descendants of order 5^8 is 1
# of capable immediate descendants is 1

**************************************************
2 capable groups saved on file c5c5_class4

**************************************************
Starting group: c5c5 #1;1 #2;2 #4;2
Order: 5^7
Nuclear rank: 1
5-multiplicator rank: 2
# of immediate descendants of order 5^8 is 2
# of capable immediate descendants is 2

**************************************************
Starting group: c5c5 #1;1 #2;2 #7;3
Order: 5^8
Nuclear rank: 2
# of immediate descendants of order 5^9 is 1
# of capable immediate descendants is 1
# of immediate descendants of order 5^10 is 1
# of capable immediate descendants is 1

**************************************************
4 capable groups saved on file c5c5_class5
Construction of descendants took 0.62 seconds

Select option: 0
Exiting from p-group generation

Select option: 0
Exiting from ANU p-Quotient Program
Total user time in seconds is 0.68
gap@tiffy:../anupq > ls -l c5c5*
total 41
```

```
-rw-r--r--    1 gap     924 Jun 24 11:27 c5c5_class2
-rw-r--r--    1 gap    2220 Jun 24 11:28 c5c5_class3
-rw-r--r--    1 gap    3192 Jun 24 11:30 c5c5_class4
-rw-r--r--    1 gap    7476 Jun 24 11:32 c5c5_class5
gap:../anupq > rm c5c5_class*
```

The fourth test will test the standard presentation part of the pq.

```
gap@tiffy:../anupq > bin/pq -i -k < gap/test4.sp
#  a lot of messages ending in
Computing standard presentation for class 9 took 0.43 seconds
The largest 5-quotient of the group has class 9

Select option: 0
Exiting from ANU p-Quotient Program
Total user time in seconds is 2.17
gap@tiffy:../anupq > ls -l SPRES
-rw-r--r--  1 gap     768 Jun 24 11:33 SPRES
gap@tiffy:../anupq > diff SPRES gap/out4.sp
#  there should be no difference if compiled with -gmp
156250000
gap@tiffy:../anupq > rm SPRES
```

The last test will test the link between GAP3 and the ANU pq. If everything goes well you should not see any message.

```
gap@tiffy:../anupq > gap -b
gap> RequirePackage( "anupq" );
gap> ReadTest( "gap/anupga.tst" );
gap>
```

You may now repeat the tests for the other machines.

57.4 ANU Sq Package

`Sq(` G`,` L `)`

The function `Sq` is the interface to the Soluble Quotient standalone program.

Let G be a finitely presented group and let L be a list of lists. Each of these lists is a list of integer pairs $[p_i,c_i]$, where p_i is a prime and c_i is a non-negative integer and $p_i \neq p_{i+1}$ and c_i positive for $i < k$. `Sq` computes a consistent power conjugate presentation for a finite soluble group given as a quotient of the finitely presented group G which is described by L as follows.

Let H be a group and p a prime. The series

$$H = P_0^p(H) \geq P_1^p(H) \geq \cdots \qquad \text{with } P_i^p(H) = [P_{i-1}^p(H), H] \left(P_{i-1}^p(H)\right)^p$$

for $i \geq 1$ is the *lower exponent-p central series* of H.

For $1 \leq i \leq k$ and $0 \leq j \leq c_i$ define the list $\mathrm{L}_{i,j} = [(p_1, c_1), \ldots, (p_{i-1}, c_{i-1}), (p_i, j)]$. Define $\mathrm{L}_{1,0}(G) = G$. For $1 \leq i \leq k$ and $1 \leq j \leq c_i$ define the subgroups

$$\mathrm{L}_{i,j}(G) = \mathrm{P}_j^{p_i}(\mathrm{L}_{i,0}(G))$$

and for $1 \leq i < k$ define the subgroups

$$\mathrm{L}_{i+1,0}(G) = \mathrm{L}_{i,c_i}(G)$$

and $\mathrm{L}(G) = \mathrm{L}_{k,c_k}(G)$. Note that $\mathrm{L}_{i,j}(G) \geq \mathrm{L}_{i,j+1}(G)$ holds for $j < c_i$.

The chain of subgroups

$$G = \mathrm{L}_{1,0}(G) \geq \mathrm{L}_{1,1}(G) \geq \cdots \geq \mathrm{L}_{1,c_1}(G) = \mathrm{L}_{2,0}(G) \geq \cdots \geq \mathrm{L}_{k,c_k}(G) = \mathrm{L}(G)$$

is called the *soluble* L-*series* of G.

`Sq` computes a consistent power conjugate presentation for $G/\mathrm{L}(G)$, where the presentation exhibits a composition series of the quotient group which is a refinement of the soluble L-series. An epimorphism from G onto $G/\mathrm{L}(G)$ is listed in comments.

The algorithm proceeds by computing power conjugate presentations for the quotients $G/\mathrm{L}_{i,j}(G)$ in turn. Without loss of generality assume that a power conjugate presentation for $G/\mathrm{L}_{i,j}(G)$ has been computed for $j < c_i$. The basic step computes a power conjugate presentation for $G/\mathrm{L}_{i,j+1}(G)$. The group $\mathrm{L}_{i,j}(G)/\mathrm{L}_{i,j+1}(G)$ is a p_i-group. If during the basic step it is discovered that $\mathrm{L}_{i,j}(G) = \mathrm{L}_{i,j+1}(G)$, then $\mathrm{L}_{i+1,0}(G)$ is set to $\mathrm{L}_{i,j}(G)$.

Note that during the basic step the vector enumerator is called.

```
gap> RequirePackage("anusq");
gap> f := FreeGroup( "a", "b" );;
gap> f := f/[ (f.1*f.2)^2*f.2^-6, f.1^4*f.2^-1*f.1*f.2^-9*f.1^-1*f.2 ];
Group( a, b )
gap> g := Sq( f, [[2,1],[3,1],[2,2],[3,2]] );
rec(
  generators := [ a.1, a.2, a.3, a.4, a.5, a.6, a.7, a.8 ],
  relators :=
   [ a.1^2*a.3^-1, a.1^-1*a.2*a.1*a.4^-1*a.2^-2, a.2^3*a.5^-1,
      a.1^-1*a.3*a.1*a.3^-1,
      a.2^-1*a.3*a.2*a.6^-1*a.5^-1*a.4^-1*a.3^-1, a.3^2*a.7^-1*a.5^-1,
      a.1^-1*a.4*a.1*a.7^-1*a.4^-1*a.3^-1,
      a.2^-1*a.4*a.2*a.8^-1*a.7^-1*a.6^-2*a.3^-1,
      a.3^-1*a.4*a.3*a.8^-2*a.7^-2*a.5^-1*a.4^-1,
      a.4^2*a.8^-2*a.7^-2*a.6^-2*a.5^-1,
      a.1^-1*a.5*a.1*a.8^-1*a.7^-1*a.6^-1*a.5^-1,
      a.2^-1*a.5*a.2*a.5^-1, a.3^-1*a.5*a.3*a.8^-2*a.6^-1*a.5^-1,
      a.4^-1*a.5*a.4*a.7^-1*a.5^-1, a.5^2,
      a.1^-1*a.6*a.1*a.8^-1*a.7^-2*a.6^-1,
      a.2^-1*a.6*a.2*a.8^-2*a.6^-2,
      a.3^-1*a.6*a.3*a.8^-2*a.7^-2*a.6^-2,
      a.4^-1*a.6*a.4*a.8^-1*a.7^-2, a.5^-1*a.6*a.5*a.8^-2*a.6^-2,
      a.6^3, a.1^-1*a.7*a.1*a.6^-2, a.2^-1*a.7*a.2*a.7^-2*a.6^-2,
      a.3^-1*a.7*a.3*a.8^-1*a.7^-1*a.6^-2, a.4^-1*a.7*a.4*a.6^-1,
      a.5^-1*a.7*a.5*a.7^-2, a.6^-1*a.7*a.6*a.8^-1*a.7^-1, a.7^3,
      a.1^-1*a.8*a.1*a.8^-2, a.2^-1*a.8*a.2*a.8^-1,
      a.3^-1*a.8*a.3*a.8^-1, a.4^-1*a.8*a.4*a.8^-1,
      a.5^-1*a.8*a.5*a.8^-1, a.6^-1*a.8*a.6*a.8^-1,
```

```
        a.7^-1*a.8*a.7*a.8^-1, a.8^3 ] )
```

This implementation was developed in C by

Alice C. Niemeyer
Department of Mathematics
University of Western Australia
Nedlands, WA 6009
Australia

email alice@maths.uwa.edu.au

57.5 Installing the ANU Sq Package

The ANU Sq is written in C and the package can only be installed under UNIX. It has been tested on DECstation running Ultrix, a HP 9000/700 and HP 9000/800 running HP-UX, a MIPS running RISC/os Berkeley, a PC running NnetBSD 0.9, and SUNs running SunOS.

It requires Steve Linton's vector enumerator (either as standalone or as GAP share library). Make sure that it is installed before trying to install the ANU Sq.

If you have a complete binary and source distribution for your machine, nothing has to be done if you want to use the ANU Sq for a single architecture. If you want to use the ANU Sq for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you have a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the ANU Sq package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a Sun running SunOS 5.3, called `galois`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `anusq.zoo` (see 56.1). Then you must locate the GAP3 directory containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be replaced by the current patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap     1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap   360891 Dec 27 15:16 anusq.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x   2 gap     3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap     1024 Nov  8  1991 grp
drwxr-xr-x   2 gap     2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap     2048 Nov 26 09:42 pkg
drwxr-xr-x   2 gap     2048 Nov 26 09:42 src
drwxr-xr-x   2 gap     1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x anusq
gap@tiffy:~ > cd gap3r4p0/pkg/anusq
gap@tiffy:../anusq> ls -l
-rw-r--r--   1 gap     5232 Apr 10 12:40 Makefile
-rw-r--r--   1 gap    13626 Mar 28 16:31 README
drwxr-xr-x   2 gap      512 Apr 10 13:30 bin
drwxr-xr-x   2 gap      512 Apr  9 20:28 examples
drwxr-xr-x   2 gap      512 Apr 10 14:22 gap
-rw-r--r--   1 gap     5272 Apr 10 13:34 init.g
drwxr-xr-x   2 gap     1024 Apr 10 13:41 src
-rwxr-xr-x   1 gap      525 Mar 28 15:50 testSq
```

Typing `make` will produce a list of possible target.

```
gap@tiffy:../anusq > make
usage: 'make <target> EXT=<ext>'  where <target> is one of
'bsd-gcc'    for Berkeley UNIX with GNU cc 2
'bsd-cc'     for Berkeley UNIX with cc
'usg-gcc'    for System V UNIX with cc
'usg-cc'     for System V UNIX with cc
'clean'      remove all created files

   where <ext> should be a sensible extension, i.e.,
   'EXT=-sun-sparc-sunos' for SUN 4 or 'EXT=' if the SQ only
   runs on a single architecture

   additional C compiler and linker flags can be passed with
   'make <target> COPTS=<compiler-opts> LOPTS=<linker-opts>',
   i.e., 'make bsd-cc COPTS="-DTAILS -DCOLLECT"', see the
   README file for details on TAILS and COLLECT.

   set ME if the vector enumerator is not started with the
   command '`pwd`/../ve/bin/me',
   i.e., 'make bsd-cc ME=/home/ve/bin/me'.
```

Select the target you need. The DECstations are running Ultrix, so we chose `bsd-gcc`.

```
gap@tiffy:../anusq > make bsd-gcc EXT=-dec-mips-ultrix
# you will see a lot of messages
```

Now repeat the compilation for the Sun run SunOS 5.3. **Do not** forget to clean up.

```
gap@tiffy:../anusq > rlogin galois
gap@galois:~ > cd gap3r4p0/pkg/anusq
gap@galois:../anusq > make clean
gap@galois:../src > make usg-cc EXT=-sun-sparc-sunos
# you will see a lot of messages and a few warnings
gap@galois:../anusq > exit
gap@tiffy:../anusq >
```

Switch into the subdirectory `bin/` and create a script which will call the correct binary for each machine. A skeleton shell script is provided in `bin/Sq.sh`.

```
gap@tiffy:../anusq > cd bin
gap@tiffy:../bin > cat > sq
#!/bin/csh
switch ( `hostname` )
  case 'tiffy':
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'bert':
    setenv ANUSQ_ME_EXEC /rem/tiffy/usr/local/bin/me ;
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'galois':
    exec $0-sun-sparc-sunos $* ;
    breaksw ;
  default:
    echo "sq: sorry, no executable exists for this machine" ;
    breaksw ;
endsw
ctr-D
gap@tiffy:../bin > chmod 755 Sq
gap@tiffy:../bin > cd ..
```

Now it is time to test the installation. The first test will only test the ANU Sq.

```
gap@tiffy:../anusq > ./testSq
Testing examples/grp1.fp  . . . . . . . . succeeded
Testing examples/grp2.fp  . . . . . . . . succeeded
Testing examples/grp3.fp  . . . . . . . . succeeded
```

If there is a problem and you get an error message saying `me not found`, set the environment variable `ANUSQ_ME_EXEC` to the module enumerator executable and try again.

The second test will test the link between GAP3 and the ANU Sq. If everything goes well you should not see any message.

```
gap@tiffy:../anusq > gap -b
gap> RequirePackage( "anusq" );
gap> ReadTest( "gap/test1.tst" );
gap>
```

You may now repeat the tests for the other machines.

57.6 GRAPE Package

GRAPE (Version 2.2) is a system for computing with graphs, and is primarily designed for constructing and analysing graphs related to groups and finite geometries.

The vast majority of GRAPE functions are written entirely in the GAP3 language, except for the automorphism group and isomorphism testing functions, which use Brendan McKay's *nauty* (Version 1.7) package [McK90].

Except for the *nauty* 1.7 package included with GRAPE, the GRAPE system was designed and written by Leonard H. Soicher, School of Mathematical Sciences, Queen Mary and Westfield College, Mile End Road, London E1 4NS, U.K., email:L.H.Soicher@qmw.ac.uk.

Please tell Leonard Soicher if you install GRAPE. Also, if you use GRAPE to solve a problem then also tell him about it, and reference

L.H.Soicher, GRAPE: a system for computing with graphs and groups, in *Groups and Computation* (L. Finkelstein and W.M. Kantor, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science **11**, pp. 287–291.

If you use the automorphism group and graph isomorphism testing functions of GRAPE then you are also using Brendan McKay's *nauty* package, and should also reference

B.D.McKay, *nauty* users guide (version 1.5), Technical Report TR-CS-90-02, Computer Science Department, Australian National University, 1990.

This document is in `nauty17/nug.alw` in postscript form. There is also a readme for *nauty* in `nauty17/read.me`.

Warning A canonical labelling given by *nauty* can depend on the version of *nauty* (Version 1.7 in GRAPE 2.2), certain parameters of *nauty* (always set the same by GRAPE 2.2) and the compiler and computer used. If you use a canonical labelling (say by using the `IsIsomorphicGraph` function) of a graph stored on a file, then you must be sure that this field was created in the same environment in which you are presently computing. If in doubt, unbind the `canonicalLabelling` field of the graph.

The only incompatible changes from GRAPE 2.1 to GRAPE 2.2 are that `Components` is now called `ConnectedComponents`, and `Component` is now called `ConnectedComponent`, and only works for simple graphs.

GRAPE is provided "as is", with no warranty whatsoever. Please read the copyright notice in the file `COPYING`.

Please send comments on GRAPE, bug reports, etc. to L.H.Soicher@qmw.ac.uk.

57.7 Installing the GRAPE Package

GRAPE consists of two parts. The first part is a set of GAP3 functions for constructing and analysing graphs, which will run on any machine that supports GAP3. The second part is based on the *nauty* package written in C and computes automorphism groups of graphs, and tests for graph isomorphisms. This part of the package can only be installed under UNIX.

If you got a complete binary and source distribution for your machine, nothing has to be done if you want to use GRAPE for a single architecture. If you want to use GRAPE for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the GRAPE package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a PC running 386BSD, called `waldorf`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `grape.zoo` (see 56.1). Then you must locate the GAP3 directories containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be replaced by current the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap      gap          1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap      gap        342865 May 27 15:16 grape.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x   2 gap      gap          3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap      gap          1024 Nov  8  1991 grp
drwxr-xr-x   2 gap      gap          2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap      gap          2048 Nov 26 09:42 src
drwxr-xr-x   2 gap      gap          1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x grape.zoo
gap@tiffy:~ > ls -l gap3r4p0/pkg/grape
-rw-r--r--    1 gap   1063 May 22 14:40 COPYING
-rw-r--r--    1 gap   2636 May 28 09:58 Makefile
-rw-r--r--    1 gap   4100 May 24 14:57 README
drwxr-xr-x    2 gap    512 May 28 11:36 bin
drwxr-xr-x    2 gap    512 May 25 14:52 doc
drwxr-xr-x    2 gap    512 May 22 16:59 grh
-rw-r--r--    1 gap  82053 May 27 12:19 init.g
drwxr-xr-x    2 gap    512 May 27 14:18 lib
drwxr-xr-x    2 gap    512 May 28 11:36 nauty17
drwxr-xr-x    2 gap    512 May 22 12:32 prs
drwxr-xr-x    2 gap    512 May 28 11:36 src
```

You are now able to use the all functions described in chapter 64 except `AutGroupGraph` and `IsIsomorphicGraph` which use the *nauty* package.

```
gap> RequirePackage("grape");

Loading  GRAPE 2.2  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmw.ac.uk.

gap> gamma := JohnsonGraph( 4, 2 );
rec(
  isGraph := true,
  order := 6,
  group := Group( (1,5)(2,6), (1,3)(4,6), (2,3)(4,5) ),
  schreierVector := [ -1, 3, 2, 3, 1, 2 ],
  adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ],
      [ 3, 4 ] ],
  isSimple := true )
```

If the documentation is not already installed or an older version is installed, copy the file `doc/grape.tex` into the `doc/` directory and run latex again (see 56.3). In general the documentation will already be installed so you can just skip the following step.

```
gap@tiffy:~ > cd gap3r4p0/pkg/grape
gap@tiffy:../grape > cp doc/grape.tex ../../doc
gap@tiffy:../grape > cd ../../doc
gap@tiffy:../doc > latex manual
# a few messages about undefined references
gap@tiffy:../doc > latex manual
# a few messages about undefined references
gap@tiffy:../doc > makeindex manual
# makeindex prints some diagnostic output
gap@tiffy:../doc > latex manual
# there should be no warnings this time
gap@tiffy:../doc cd ../pkg/grape
```

In order to compile *nauty* and the filters used by GRAPE to interact with *nauty* type `make` to get a list of support machines.

```
gap@tiffy:../grape > make
usage: 'make <target>' EXT=<ext> where target is one of
'dec-mips-ultrix-cc'   for DECstations running Ultrix with cc
'hp-hppa1.1-hpux-cc'   for HP 9000/700 under HP-UX with cc
'hp-hppa1.0-hpux-cc'   for HP 9000/800 under HP-UX with cc
'ibm-i386-386bsd-gcc2' for IBM PCs under 386BSD with GNU cc 2
'ibm-i386-386bsd-cc'   for IBM PCs under 386BSD with cc (GNU)
'sun-sparc-sunos-cc'   for SUN 4 under SunOS with cc
'bsd'                  for others under Berkeley UNIX with cc
'usg'                  for others under System V UNIX with cc

where <ext> should be a sensible extension, i.e.,
'EXT=.sun' for SUN or 'EXT=' if GRAPE only runs
on a single architecture
```

Select the target you need. In your case we first compile the DECstation version. We use the extension `-dec-mips-ultrix`, which creates the binaries
`dreadnaut-dec-mips-ultrix`, `drcanon3-dec-mips-ultrix`,
`gap3todr-dec-mips-ultrix` and `drtogap3-dec-mips-ultrix`
in the `bin/` directory.

```
gap@tiffy:../grape > make dec-mips-ultric-cc EXT=-dec-mips-ultrix
# you will see a lot of messages
```

Now repeat the compilation for the PC. **Do not** forget to clean up.

```
gap@tiffy:../grape > rlogin waldorf
gap@waldorf:~ > cd gap3r4p0/pkg/grape
gap@waldorf:../grape > make clean
gap@waldorf:../grape > make ibm-i386-386bsd-gcc2 EXT=-ibm-i386-386bsd
# you will see a lot of messages
gap@waldorf:../grape > exit
gap@tiffy:../grape >
```

Switch into the subdirectory `bin/` and create four shell scripts which will call the correct binary for each machine. Skeleton shell scripts are provided in `bin/dreadnaut.sh`, `bin/drcanon3.sh`, etc.

```
gap@tiffy:../grape > cat > bin/dreadnaut
#!/bin/csh
switch ( `hostname` )
  case 'tiffy':
  case 'bert':
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'waldorf':
    exec $0-ibm-i386-386bsd $* ;
    breaksw ;
  default:
    echo "dreadnaut: sorry, no executable exists for this machine" ;
    breaksw ;
endsw
ctr-D
gap@tiffy:../grape > chmod 755 bin/dreadnaut
```

You must also create similar shell scripts for `drcanon3`, `drtogap3`, and `gap3todr`. Note that if you are using GRAPE only on a single architecture you can specify an empty extension using `EXT=` as a parameter to `make`. In this case **do not** create the above shell scripts. The following example will test the interface between GRAPE and *nauty*.

```
gap> IsIsomorphicGraph( JohnsonGraph(7,3), JohnsonGraph(7,4) );
true
gap> AutGroupGraph( JohnsonGraph(4,2) );
Group( (3,4), (2,3)(4,5), (1,2)(5,6) )
```

57.8 MeatAxe Package

The MeatAxe package provides algorithms for computing with finite field matrices, permutations, matrix groups, matrix algebras, and their modules.

Every such object exists outside GAP3 on a file, and GAP3 is only responsible for handling these files using the appropriate programs.

Details about the standalone can be found in [Rin93]. This implementation was developed in C by

Michael Ringe
Lehrstuhl D für Mathematik
RWTH Aachen
52062 Aachen, Germany

e-mail mringe@math.rwth-aachen.de

57.9 Installing the MeatAxe Package

The MeatAxe is written in C, and it is assumed that the package is installed under UNIX. Some other systems –currently MS-DOS and VM/CMS– are supported, but this applies only for the standalone and not for the use of the MeatAxe from within GAP3 (see the MeatAxe manual [Rin93] for details of the installation in these cases).

If you got a complete binary and source distribution, skip the extraction and compilation part of this section. All what you have to do in this case is to make the executables accessible via a pathname that contains the hostname of the machine; this is best done by creating suitable links, as is described at the end of this section.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the MeatAxe package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a NeXTstation, called `bjerun`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `meataxe.zoo` (see 56.1). Then you must locate the GAP3 directory containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be be replaced by the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap      gap          1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap      gap        359381 May 11 11:34 meataxe.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x    2 gap         3072 Nov 26 11:53 doc
drwxr-xr-x    2 gap         1024 Nov  8  1991 grp
drwxr-xr-x    2 gap         2048 Nov 26 09:42 lib
drwxr-xr-x    2 gap         2048 Nov 26 09:42 src
drwxr-xr-x    2 gap         1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x meataxe.zoo
gap@tiffy:~ > ls -l gap3r4p0/pkg/meataxe
-rw-r--r--    1 gap        17982 Aug  6  1993 COPYING
-rw-r--r--    1 gap         3086 Mar 15 15:07 README
drwxr-xr-x    3 gap          512 Mar 26 18:01 bin
drwxr-xr-x    2 gap          512 Feb 25 12:07 doc
drwxr-xr-x    2 gap          512 May 11 09:34 gap
-rw-r--r--    1 gap         1023 May 11 09:34 init.g
drwxr-xr-x    2 gap         1024 Mar 26 18:02 lib
drwxr-xr-x    2 gap         1536 Mar 26 18:02 src
drwxr-xr-x    2 gap          512 Mar 15 11:36 tests
```

Switch into the directory `bin/`, edit the `Makefile`, and follow the instructions given there. In most cases it will suffice to choose the right `COMPFLAGS`. Then type `make` to compile the MeatAxe. In your case we first compile the DECstation version.

```
gap@tiffy:~ > cd gap3r4p0/pkg/meataxe/bin
gap@tiffy:../bin > make
# you will see a lot of messages
```

The executables reside in a directory with the same name as the host, in this case this is `tiffy`. The programs will be called from GAP3 using the hostname, thus for every machine that shall run the MeatAxe under GAP3 such a directory is necessary. In your case there is a second DEC-station called `bert` which can use the same executables, we make them available via a link.

```
gap@tiffy:../bin > ln -s tiffy bert
```

Now repeat the compilation for the NeXTstation. If you want to save space you can clean up using `make clean` but this is not necessary. If the `make` run was interrupted you can return to the prior situation using `make delete`, and then call `make` again.

```
gap@tiffy:../bin > rlogin bjerun
gap@bjerun:~ > cd gap3r4p0/pkg/meataxe/bin
gap@bjerun:../bin > make clean
gap@bjerun:../bin > make
# you will see a lot of messages
gap@bjerun:../bin > exit
gap@tiffy:../bin >
```

Now it is time to test the package. Switch into the directory `../tests/` and type `./testmtx`. You should get no error messages, and end up with the message `all tests passed`.

```
gap@tiffy:../bin > cd ../tests
gap@tiffy:../tests > ./testmtx
# you will see a lot of messages
gap@tiffy:../tests >
```

57.10 NQ Package

`NilpotentQuotient(` *F* `)`
`NilpotentQuotient(` *F*`,` *c* `)`

`NilpotentQuotient` computes the quotient groups of the finitely presented group F successively modulo the terms of the lower central series of F. If it terminates, it returns a list L. The i-th entry of L contains the non-trivial abelian invariants of the i-th factor of the lower central series of F (the largest abelian quotient being the first factor).

`NilpotentQuotient` accepts a positive integer c as an optional second argument. If the second argument is present, the function computes the quotient group of F modulo the c-th term of the lower central series of F (the commutator subgroup is the first term).

```
gap> RequirePackage("nq");
gap> a := AbstractGenerator( "a" );;
gap> b := AbstractGenerator( "b" );;
gap>
gap> G := rec( generators := [a, b],
>     relators   := [ LeftNormedComm( b,a,a,a,a ),
>                     LeftNormedComm( b,a,b,b,b ),
>                     LeftNormedComm( b,a,a*b,a*b,a*b ),
>                     LeftNormedComm( b,a,a*b^2,a*b^2,a*b^2 ),
>                     LeftNormedComm( b,a,b,a,a,a ),
>                     LeftNormedComm( b,a,a,b,b,b ) ]
```

```
>    );;
gap>
gap> NilpotentQuotient( G, 6 );
[ [ 0, 0 ], [ 0 ], [ 0, 0 ], [ 0, 0, 0 ], [ 2, 0, 0 ], [ 2, 10, 0 ] ]
```

This implementation was developed in C by

Werner Nickel
School of Mathematical Sciences
Australian National University
Canberra, ACT 0200

e-mail werner@pell.anu.edu.au

57.11 Installing the NQ Package

The NQ is written in C and the package can only be installed under UNIX. It has been tested on DECstation running Ultrix, a NeXTstation running NeXT-Step 3.0, and SUNs running SunOS. It requires the GNU multiple precision arithmetic. Make sure that this library is installed before trying to install the NQ.

If you got a complete binary and source distribution for your machine, nothing has to be done if you want to use the NQ for a single architecture. If you want to use the NQ for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the NQ package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a NeXTstation, called `bjerun`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `nq.zoo` (see 56.1). Then you must locate the GAP3 directories containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be be replaced by the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap      gap         1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap      gap       106307 Jan 24 15:16 nq.zoo
gap@tiffy:~ > ls -l
drwxr-xr-x   2 gap      gap         3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap      gap         1024 Nov  8  1991 grp
drwxr-xr-x   2 gap      gap         2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap      gap         2048 Nov 26 09:42 src
drwxr-xr-x   2 gap      gap         1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x nq.zoo
gap@tiffy:~ > ls -l gap3r4p0/pkg/nq
drwxr-xr-x   2 gap    gap       1024 Jan 24 21:00 bin
drwxr-xr-x   2 gap    gap       1024 Jan 19 11:33 examples
drwxr-xr-x   2 gap    gap       1024 Jan 24 21:03 gap
lrwxrwxrwx   1 gap    gap          8 Jan 19 11:33 init.g
drwxr-xr-x   2 gap    gap       1024 Jan 24 21:04 src
-rwxr--r--   1 gap    gap        144 Dec 28 15:08 testNq
```

Switch into the directory `src/` and type `make` to compile the NQ. If the header files for the GNU multiple precision arithmetic are **not** in `/usr/local/include` you must set `GNUINC` to the correct directory. If the library for the GNU multiple precision arithmetic is **not** `/usr/local/lib/libmp.a` you must set `GNULIB`. In your case we first compile the DECstation version. If your operating system **does not** provide a function `getrusage` start make with `COPTS=-DNO_GETRUSAGE`.

```
gap@tiffy:~ > cd gap3r4p0/pkg/nq/src
gap@tiffy:../src > make GNUINC=/usr/gnu/include \
                        GNULIB=/usr/gnu/lib/libmp.a
# you will see a lot of messages
```

Now it is possible to test the standalone.

```
gap@tiffy:../src > cd ..
gap@tiffy:../nq > testNq
```

If `testNq` reports a difference others then machine name, runtime or size, check the GNU multiple precision arithmetic and warnings generated by `make`. If `testNq` succeeded , move the executable to the `bin/` directory.

```
gap@tiffy:../nq > mv src/nq bin/nq-dec-mips-ultrix
```

Now repeat the compilation for the NeXTstation. **Do not** forget to clean up.

```
gap@tiffy:../nq > rlogin bjerun
gap@bjerun:~ > cd gap3r4p0/pkg/nq/src
gap@bjerun:../src > make clean
gap@bjerun:../src > make
# you will see a lot of messages
gap@bjerun:../src > mv nq ../bin/nq-next-m68k-mach
gap@bjerun:../src > exit
gap@tiffy:../src >
```

Switch into the subdirectory `bin/` and create a script which will call the correct binary for each machine. A skeleton shell script is provided in `bin/nq.sh`.

```
gap@tiffy:../src > cd ..
gap@tiffy:../nq > cat > bin/nq
#!/bin/csh
switch ( `hostname` )
  case 'bert':
  case 'tiffy':
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
```

```
    case 'bjerun':
      exec $0-next-m68k-mach $* ;
      breaksw ;
    default:
      echo "nq: sorry, no executable exists for this machine" ;
      breaksw ;
endsw
ctr-D
gap@tiffy:../nq > chmod 755 bin/nq
```

Now it is time to test the package. Assuming that `testNq` worked the following will test the link to GAP3.

```
gap@tiffy:../nq > gap -b
gap> RequirePackage( "nq" );
gap> ReadTest( "gap/nq.tst" );
gap>
```

57.12 SISYPHOS Package

SISYPHOS provides access to implementations of algorithms for dealing with p-groups and their modular group algebras. At the moment only the programs for p-groups are accessible via GAP3. They can be used to compute isomorphisms between p-groups, and automorphism groups of p-groups.

The description of the functions available in the SISYPHOS package is given in chapter 71.

For details about the implementation and the standalone version see the README. This implementation was developed in C by

Martin Wursthorn
Math. Inst. B, 3. Lehrstuhl
Universität Stuttgart

e-mail pluto@machnix.mathematik.uni-stuttgart.de
Tel. +49 (0)711 685 5517
Fax. +49 (0)711 685 5322

57.13 Installing the SISYPHOS Package

SISYPHOS is written in ANSI-C and should run on every UNIX system (and some non-UNIX systems) that provides an ANSI-C Compiler, e.g., the GNU C compiler. SISYPHOS has been ported to IBM RS6000 running AIX 3.2, HP9000 7xx running HP-UX 8.0/9.0, PC 386/486 running Linux, PC 386/486 running DOS or OS/2 with emx and ATARI ST/TT running TOS.

In the example we will assume that you, as user `gap`, are installing the SISYPHOS package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a 486 PC, called `waldorf`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `sisyphos.zoo` (see 56.1). Then you must locate the GAP3 directories containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be be replaced by the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap     1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap   245957 Dec 27 15:16 sisyphos.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x   2 gap     3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap     1024 Nov  8  1991 grp
drwxr-xr-x   2 gap     2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap     2048 Nov 26 09:42 src
drwxr-xr-x   2 gap     1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x sisyphos.zoo
gap@tiffy:~ > ls -l gap3r4p0/pkg/sisyphos
-rw-r--r--  1 gap          9496 Feb 11  1993 README
drwxr-xr-x  3 gap           512 Oct 19 10:24 doc
drwxr-xr-x  2 gap           512 Oct 15 14:30 groups
drwxr-xr-x  2 gap           512 Apr  1  1993 ideal
-rw-r--r--  1 gap         22072 Oct 19 10:23 init.g
drwxr-xr-x  2 gap          1536 Oct 15 14:49 src
```

Switch into the directory `src/`. It contains the makefile for SISYPHOS.

```
gap@tiffy:../src > make
usage: 'make <target>'  where target is one of
'hp700-hpux-gcc2'    for HP 9000/7xx under HP-UX with GNU cc 2
'hp700-hpux-cc'      for HP 9000/7xx under HP-UX with cc
'hp700-hpux-cci'     for HP 9000/7xx under HP-UX with cc -
                       generate version for profile dependent optimization
'hp700-hpux-ccp'     for HP 9000/7xx under HP-UX with cc -
                       relink with profile dependent optimization
'ibm6000-aix-cc'     for IBM RS/6000 under AIX with cc
'ibmpc-linux-gcc2'   for IBM PCs under Linux with GNU cc 2
'ibmpc-emx-gcc2'     for IBM PCs under DOS or OS/2 2.0 with emx
'generic-unix-gcc2'  for other UNIX machines with GNU cc 2
                     this should work on most machines
```

Select the target you need. In our case we first compile the DECstation version. We assume that the command to start GAP3 is `/usr/local/bin/gap` for `tiffy` and `waldorf` and `/rem/tiffy/usr/local/bin/gap` for `bert`.

```
gap@tiffy:../src > make generic-unix-gcc2
# you will see a lot of messages and maybe a few warnings
```

You should test the standalone now. The following command should run without any comment. This will work, however, only for UNIX machines.

```
gap@tiffy:../src > testsis
```

The executables will be collected in the /bin directory, so we move that for the DECstation there.

```
gap@tiffy:../src > mv sis ../bin/sis.ds
```

Now repeat the compilation for the PC. **Do not** forget to clean up.

```
gap@tiffy:../src > rlogin waldorf
gap@waldorf:~ > cd gap3r4p0/pkg/sisyphos/src
gap@waldorf:../src > make clean
gap@waldorf:../src > make generic-unix-gcc2
# you will see a lot of messages and maybe a few warnings
```

Test the executable (under UNIX only), and move it to the right place.

```
gap@waldorf:../src > testsis
gap@waldorf:../src > mv sis ../bin/sis.386bsd
gap@waldorf:../src > exit
gap@tiffy:../src >
```

Switch into the subdirectory bin/ and create a script which will call the correct binary for each machine.

```
gap@tiffy:../src > cd ..
gap@tiffy:../sisyphos > cat > bin/sis
#!/bin/csh
switch ( `hostname` )
  case 'bert':
  case 'tiffy':
    exec ~gap/3.2/pkg/sisyphos/bin/sis.ds $* ;
    breaksw ;
  case 'waldorf':
    exec ~gap/3.2/pkg/sisyphos/bin/sis.386bsd $* ;
    breaksw ;
  default:
    echo "sis: sorry, no executable exists for this machine" ;
    breaksw ;
endsw
ctr-D
gap@tiffy:../sisyphos > chmod 755 bin/sis
```

57.14 Vector Enumeration Package

The Vector Enumeration package provides access to the implementation of the "linear Todd-Coxeter"method for computing matrix representations of finitely presented algebras.

The description of the functions available in the Vector Enumeration package is given in chapter 73.

For details about the implementation and the standalone version see the README. This implementation was developed in C by

Stephen A. Linton
Division of Computer Science

School of Mathematical and Computational Science
University of St. Andrews
North Haugh
St. Andrews
Fife
KY10 2SA
SCOTLAND

e-mail sal@cs.at-andrews.ac.uk
Tel. +44 334 63239
Fax. +44 334 63278

57.15 Installing the Vector Enumeration Package

The Vector Enumerator (VE) is written in C and the package can only be installed under UNIX. It has been tested on DECstation running Ultrix, a 486 running NetBSD, and SUNs running SunOS.

The parts of the package that deal with rationals require the GNU multiple precision arithmetic library GMP. Make sure that this library is installed before trying to install VE.

If you got a complete binary and source distribution for your machine, nothing has to be done if you want to use the VE for a single architecture. If you want to use the VE for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the VE package for use by several users on a network of two DECstations, called `bert` and `tiffy`, and a NeXTstation, called `bjerun`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `ve.zoo` (see 56.1). Then you must locate the GAP3 directories containing `lib/` and `doc/`, this is usually `gap3r4p0` where `0` is to be be replaced by the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap      gap         1024 Nov  8  1991 gap3r4p0
-rw-r--r--   1 gap      gap       106307 Jan 24 15:16 ve.zoo
gap@tiffy:~ > ls -l gap3r4p0
drwxr-xr-x   2 gap      gap         3072 Nov 26 11:53 doc
drwxr-xr-x   2 gap      gap         1024 Nov  8  1991 grp
drwxr-xr-x   2 gap      gap         2048 Nov 26 09:42 lib
drwxr-xr-x   2 gap      gap         2048 Nov 26 09:42 src
drwxr-xr-x   2 gap      gap         1024 Nov 26 09:42 tst
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p0` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo x ve.zoo
 gap@tiffy:~ > ls -l gap3r4p0/pkg/ve
 -rw-r--r--  1 sam        16761 May 10 17:39 Makefile
 -rw-r-----  1 sam         1983 May  6  1993 README
 drwxr-xr-x  2 sam          512 May 10 17:41 bin
 drwxr-xr-x  2 sam          512 May 10 17:34 docs
 drwxr-xr-x  2 sam          512 May 10 17:34 examples
 drwxr-xr-x  3 sam          512 Mar 28 17:55 gap
 -rw-r--r--  1 sam          553 Mar 24 18:18 init.g
 drwxr-xr-x  5 sam         1024 May 10 17:36 src
```

Switch into the directory `ve/` and type `make` to see a list of targets for compilation; then type `make` *target* to compile VE, where *target* is the target that is closest to your machine. If the header files for the GNU multiple precision arithmetic are **not** in `/usr/local/include` you must set `INCDIRGMP` to the correct directory. If the library for the GNU multiple precision arithmetic is **not** `/usr/local/lib/libgmp.a` you must set `LIBDIRGMP`. In this case we first compile the DECstation version.

```
gap@tiffy:~ > cd gap3r4p0/pkg/ve
gap@tiffy:../ve > make INCDIRGMP=/usr/gnu/include \
                   LIBDIRGMP=/usr/gnu/lib/ dec-mips-ultrix-gcc2
# you will see a lot of messages
```

Now repeat the compilation for the NeXTstation. **Do not** forget to clean up.

```
gap@tiffy:../ve > mv bin/me.exe bin/me.dec
gap@tiffy:../ve > mv bin/qme.exe bin/qme.dec
gap@tiffy:../ve > rlogin bjerun
gap@bjerun:~ > cd gap3r4p0/pkg/ve
gap@bjerun:../ve > make clean
# you will see some messages
gap@bjerun:../ve > make next-m68k-mach-gcc2
# you will see a lot of messages
gap@bjerun:../ve > mv bin/me.exe bin/me.next
gap@bjerun:../ve > mv bin/qme.exe bin/qme.next
gap@bjerun:../ve > exit
gap@tiffy:../ve >
```

Switch into the subdirectory `bin/` and create scripts which will call the correct binary for each machine. The shell scripts that are already contained in 'bin/me.sgl' and 'bin/qme.sgl' are suitable only for a single architecture installation.

```
gap@tiffy:../ve > cat > bin/me
#!/bin/csh
switch ( `hostname` )
  case 'bert':
  case 'tiffy':
    exec $0.dec $* ;
    breaksw ;
  case 'bjerun':
    exec $0.next $* ;
    breaksw ;
```

```
    default:
      echo "me/qme/zme: sorry, no executable exists for this machine" ;
      breaksw ;
  endsw
  ctr-D
  gap@tiffy:../ve > chmod 755 bin/me
  gap@tiffy:../ve > ln bin/me bin/qme
```

57.16 The XGap Package

XGAP is a graphical user interface for GAP3, it extends the GAP3 library with functions dealing with graphic sheets and objects. Using these functions it also supplies a graphical interface for investigating the subgroup lattice of a group, giving you easy access to the low index subgroups, prime quotient and Reidemeister-Schreier algorithms and many other GAP3 functions for groups and subgroups. At the moment the only supported window system is X-Windows X11R5, however, programs using the XGAP library functions will run on other platforms as soon as XGAP is available on these. We plan to release a Windows 3.11 version in the near future.

In order to produce a preliminary manual and installation guide for the XGAP package, switch into the directory `gap3r4p4/pkg/xgap/doc` and latex the document `latexme.tex`.

Frank Celler & Susanne Keitemeier

Chapter 58

ANU Pq

The ANU p-quotient program (pq) may be called from GAP3. Using this program, GAP3 provides access to the following: the p-quotient algorithm; the p-group generation algorithm; a standard presentation algorithm; an algorithm to compute the automorphism group of a p-group.

The following section describes the function `Pq`, which gives access to the p-quotient algorithm.

The next section describes the function `PqDescendants`, which gives access to the p-group generation algorithm.

The next sections describe functions for saving results to file (see 58.4 and 58.5).

The next section describes the function `StandardPresentation` which gives access to the standard presentation algorithm and to the algorithm used to compute the automorphism group of a p-group.

The last sections describes the function `IsIsomorphicPGroup` which implements an isomorphism test for p-groups using the standard presentation algorithm.

58.1 Pq

`Pq(` F`, ... )`

Let F be a finitely presented group. Then `Pq` returns the desired p-quotient of F as an ag group.

The following parameters or parameter pairs are supported.

"Prime", p
: Compute the p-quotient for the prime p.

"ClassBound", n
: The p-quotient computed has lower exponent-p class at most n.

"Exponent", n
: The p-quotient computed has exponent n. By default, no exponent law is enforced.

"Metabelian"
: The largest metabelian p-quotient is constructed.

"Verbose"
: The runtime-information generated by the ANU pq is displayed. By default, pq works silently.

"OutputLevel", *n*
: The runtime-information generated by the ANU pq is displayed at output level n, which must be a integer from 0 to 3. This parameter implies "Verbose".

"SetupFile", *name*
: Do not run the ANU pq, just construct the input file and store it in the file *name*. In this case `true` is returned.

Alternatively, you can pass `Pq` a record as a parameter, which contains as entries some (or all) of the above mentioned. Those parameters which do not occur in the record are set to their default values.

See also 58.2.

```
gap> RequirePackage("anupq");
gap> f2 := FreeGroup( 2, "f2" );
Group( f2.1, f2.2 )
gap> Pq( f2, rec( Prime := 2,  ClassBound := 3 ) );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8, G.9, G.10 )
gap> g := f2 / [ f2.1^4, f2.2^4 ];;
gap> Pq( g, rec( Prime := 2, ClassBound := 3 ) );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8 )
gap> Pq( g, "Prime", 2, "ClassBound", 3, "Exponent", 4 );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7 )
gap> g := f2 / [ f2.1^25, Comm(Comm(f2.2,f2.1),f2.1), f2.2^5 ];;
gap> Pq( g, "Prime", 5, "Metabelian", "ClassBound", 5 );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7 )
```

This function requires the package "anupq" (see 57.1).

58.2 PqHomomorphism

`PqHomomorphism(` G`,` *images* `)`

Let G be a p-quotient of F computed using `Pq`. If *images* is a list of images of F`.generators` under an automorphism of F, `PqHomomorphism` will return the corresponding automorphism of G.

```
gap> F := FreeGroup (2, "F");
Group( F.1, F.2 )
gap> G := Pq (F, "Prime", 5, "Class", 2);
Group( G.1, G.2, G.3, G.4, G.5 )
gap> PqHomomorphism (G, [F.2, F.1]);
GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5 ), Group(
G.1, G.2, G.3, G.4, G.5 ), [ G.1, G.2, G.3, G.4, G.5 ],
[ G.2, G.1, G.3^4, G.5, G.4 ] )
```

58.3 PqDescendants

`PqDescendants(` G`, ... )`

Let G be an ag group of prime power order with a consistent power-commutator presentation (see 25.28). `PqDescendants` returns a list of descendants of G.

If G does **not** have p-class 1, then a list of automorphisms of G must be bound to the record component G`.automorphisms` such that G`.automorphisms` together with the inner automorphisms of G generate the automorphism group of G.

One method which may be used to obtain such a generating set for the automorphism group is to call `StandardPresentation`. The record returned has a generating set for the automorphism group of G stored as a component (see 58.6).

The following optional parameters or parameter pairs are supported.

"ClassBound", *n*
: `PqDescendants` generates only descendants with lower exponent-p class at most n. The default value is the exponent-p class of G plus one.

"OrderBound", *n*
: `PqDescendants` generates only descendants of size at most p^n. Note that you cannot set both "OrderBound" and "StepSize".

"StepSize", *n*
: Let n be a positive integer. `PqDescendants` generates only those immediate descendants which are p^n bigger than their parent group.

"StepSize", *l*
: Let l be a list of positive integers such that the sum of the length of l and the exponent-p class of G is equal to the class bound "ClassBound". Then l describes the step size for each additional class.

"AgAutomorphisms"
: The automorphisms stored in G`.automorphisms` are a PAG generating sequence for the automorphism group of G supplied in **reverse** order.

"RankInitialSegmentSubgroups", *n*
: Set the rank of the initial segment subgroup chosen to be n. By default, this has value 0.

"SpaceEfficient"
: The ANU pq performs calculations more slowly but with greater space efficiency. This flag is frequently necessary for groups of large Frattini quotient rank. The space saving occurs because only one permutation is stored at any one time. This option is only available in conjunction with the "AgAutomorphisms" flag.

"AllDescendants"
: By default, only capable descendants are constructed. If this flag is set, compute all descendants.

"Exponent", *n*
: Construct only descendants with exponent n. Default is no exponent law.

"Metabelian"
: Construct only metabelian descendants.

"SubList", *sub*
: Let L be the list of descendants generated. If list *sub* is supplied, `PqDescendants` returns `Sublist(` L`,`*sub* `)`. If an integer n is supplied, `PqDescendants` returns $L[n]$.

"Verbose"

The runtime-information generated by the ANU pq is displayed. By default, pq works silently.

"SetupFile", *name*

Do not run the ANU pq, just construct the input file and store it in the file *name*. In this case `true` is returned.

"TmpDir", *dir*

`PqDescendants` stores intermediate results in temporary files; the location of these files is determined by the value selected by `TmpName`. If your default temporary directory does not have enough free disk space, you can supply an alternative path *dir*. In this case `PqDescendants` stores its intermediate results in a temporary subdirectory of *dir*. Alternatively, you can globally set the variable `ANUPQtmpDir`, for instance in your ".gaprc" file, to point to a suitable location.

Alternatively, you can pass `PqDescendants` a record as a parameter, which contains as entries some (or all) of the above mentioned. Those parameters which do not occur in the record are set to their default values.

Note that you cannot set both "OrderBound" and "StepSize".

In the first example we compute all descendants of the Klein four group which have exponent-2 class at most 5 and order at most 2^6.

```
gap> f2 := FreeGroup( 2, "g" );;
gap> g := AgGroupFpGroup(f2 / [f2.1^2, f2.2^2, Comm(f2.2,f2.1)]);
Group( g.1, g.2 )
gap> g.name := "g";;
gap> l := PqDescendants( g, "OrderBound", 6, "ClassBound", 5,
>                        "AllDescendants" );;
gap> Length(l);
83
gap> Number( l, x -> x.isCapable );
47
gap> List( l, x -> Size(x) );
[ 8, 8, 8, 16, 16, 16, 32, 16, 16, 16, 16, 16, 32, 32, 64, 64, 32,
  32, 32, 32, 32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
  32, 32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 32, 32,
  32, 32, 32, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
  64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64 ]
gap> List( l, x -> Length( PCentralSeries( x, 2 ) ) - 1 );
[ 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
  3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5 ]
```

In the second example we compute all capable descendants of order 27 of the elementary abelian group of order 9. Here, we supply automorphisms which form a PAG generating sequence (in reverse order) for the class 1 group, since this makes the computation more efficient.

```
gap> f2 := FreeGroup( 2, "g" );;
```

```
gap> g := AgGroupFpGroup(f2 / [ f2.1^3, f2.2^3, Comm(f2.1,f2.2) ]);
Group( g.1, g.2 )
gap> g.name := "g";;
gap> g.automorphisms := [];;
gap> GroupHomomorphismByImages(g, g, [g.1, g.2], [g.1^2, g.2^2]);;
gap> Add( g.automorphisms, last );
gap> GroupHomomorphismByImages(g, g, [g.1, g.2], [g.2^2, g.1]);;
gap> Add( g.automorphisms, last );
gap> GroupHomomorphismByImages(g,g,[g.1,g.2],[g.1*g.2^2,g.1^2*g.2^2]);;
gap> Add( g.automorphisms, last );
gap> GroupHomomorphismByImages(g, g, [g.1,g.2], [g.1,g.1^2*g.2]);;
gap> Add( g.automorphisms, last );
gap> GroupHomomorphismByImages(g, g, [g.1, g.2], [g.1^2, g.2]);;
gap> Add( g.automorphisms, last );
gap> l := PqDescendants( g, "OrderBound", 3,
>                           "ClassBound", 2,
>                           "AgAutomorphisms" );;
gap> Length(l);
2
gap> List( l, x -> Size(x) );
[ 27, 27 ]
gap> List( l, x -> Length( PCentralSeries( x, 3 ) ) - 1 );
[ 2, 2 ]
```

In the third example, we compute all capable descendants of the elementary abelian group of order 5^2 which have exponent-5 class at most 3, exponent 5, and are metabelian.

```
gap> f2 := FreeGroup( 2, "g" );;
gap> g := AgGroupFpGroup( f2 / [f2.1^5, f2.2^5, Comm(f2.2,f2.1)] );
Group( g.1, g.2 )
gap> g.name := "g";;
gap> l := PqDescendants(g,"Metabelian","ClassBound",3,"Exponent",5);;
gap> List( l, x -> Length( PCentralSeries( x, 5 ) ) - 1 );
[ 2, 3, 3 ]
gap> List( l, x -> Length( DerivedSeries( x ) ) );
[ 3, 3, 3 ]
gap> List( l, x -> Maximum( List( Elements(x), y -> Order(x,y) ) ) );
[ 5, 5, 5 ]
```

This function requires the package "anupq" (see 57.1).

58.4 PqList

`PqList(` *file* `)`
`PqList(` *file*`,` *sub* `)`
`PqList(` *file*`,` *n* `)`

The function `PqList` reads a file *file* and returns the list L of ag groups defined in this file.

If list *sub* is supplied as a parameter, the function returns `Sublist(` L`,` *sub* `)`. If an integer n is supplied, `PqList` returns $L[n]$.

This function and `SavePqList` (see 58.5) can be used to save and restore a list of descendants (see 58.3).

This function requires the package "anupq" (see 57.1).

58.5 SavePqList

`SavePqList(` *name*`,` *list* `)`

The function `SavePqList` writes a list of descendants *list* to a file *name*.

This function and `PqList` (see 58.4) can be used to save and restore results of `PqDescendants` (see 58.3).

This function requires the package "anupq" (see 57.1).

58.6 StandardPresentation

`StandardPresentation(` F`,` p`, ... )`
`StandardPresentation(` F`,` G`, ... )`

Let F be a finitely presented group. Then `StandardPresentation` returns the standard presentation for the desired p-quotient of F as an ag group.

Let H be the p-quotient whose standard presentation is computed. A generating set for a supplement to the inner automorphism group of H is also returned, stored as the component H`.automorphisms`. Each generator is described by its action on each of the generators of the standard presentation of H.

A finitely-presented group F must be supplied as input. Usually, the user will also supply a prime p and the program will compute the standard presentation for the desired p-quotient of F.

Alternatively, a user may supply an ag group G which is the class 1 p-quotient of F. If this is so, a list of automorphisms of G must be bound to the record component G`.automorphisms` such that G`.automorphisms` together with the inner automorphisms of G generate the automorphism group of G. The presentation for G can be constructed by an initial call to Pq (see 58.1).

Of course, G need not be the class 1 p-quotient of F. However, G`.automorphisms` must contain a description of the automorphism group of G and this is most readily available when G is an elementary abelian group. Where the necessary information is available for a p-quotient of higher class, one can apply the standard presentation algorithm from that class onwards.

The following parameters or parameter pairs are supported.

"ClassBound", n
: The standard presentation is computed for the largest p-quotient of F having lower exponent-p class at most n.

"Exponent", n
: The p-quotient computed has exponent n. By default, no exponent law is enforced.

"Metabelian"
: The p-quotient constructed is metabelian.

"AgAutomorphisms"
: The automorphisms stored in G.`automorphisms` are a PAG generating sequence for the automorphism group of G supplied in **reverse** order.

"Verbose"
: The runtime-information generated by the ANU pq is displayed. By default, pq works silently.

"OutputLevel", n
: The runtime-information generated by the ANU pq is displayed at output level n, which must be a integer from 0 to 3. This parameter implies "Verbose".

"SetupFile", *name*
: Do not run the ANU pq, just construct the input file and store it in the file *name*. In this case `true` is returned.

"TmpDir", *dir*
: `StandardPresentation` stores intermediate results in temporary files; the location of these files is determined by the value selected by `TmpName`. If your default temporary directory does not have enough free disk space, you can supply an alternative path *dir*. In this case `StandardPresentation` stores its intermediate results in a temporary subdirectory of *dir*. Alternatively, you can globally set the variable `ANUPQtmpDir`, for instance in your ".gaprc" file, to point to a suitable location.

Alternatively, you can pass `StandardPresentation` a record as a parameter, which contains as entries some (or all) of the above mentioned. Those parameters which do not occur in the record are set to their default values.

We illustrate the method with the following examples.

```
gap> f2 := FreeGroup( "a", "b" );;
gap> g := f2 / [f2.1^25, Comm(Comm(f2.2,f2.1), f2.1), f2.2^5];
Group( a, b )
gap> StandardPresentation( g, 5, "ClassBound", 10 );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8, G.9, G.10, G.11, G.12,
G.13, G.14, G.15, G.16, G.17, G.18, G.19, G.20, G.21, G.22, G.23,
G.24, G.25, G.26 )
gap> f2 := FreeGroup( "a", "b" );;
gap> g := f2 / [ f2.1^625,
>  Comm(Comm(Comm(Comm(f2.2,f2.1),f2.1),f2.1),f2.1)/Comm(f2.2,f2.1)^5,
>  Comm(Comm(f2.2,f2.1),f2.2), f2.2^625 ];;
gap> StandardPresentation( g, 5, "ClassBound", 15, "Metabelian" );
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8, G.9, G.10, G.11, G.12,
G.13, G.14, G.15, G.16, G.17, G.18, G.19, G.20 )
gap> f4 := FreeGroup( "a", "b", "c", "d" );;
gap> g4 := f4 / [ f4.2^4, f4.2^2 / Comm(Comm (f4.2, f4.1), f4.1),
>                 f4.4^16, f4.1^16 / (f4.3 * f4.4),
>                 f4.2^8 / (f4.4 * f4.3^4) ];
Group( a, b, c, d )
gap> g := Pq( g4, "Prime", 2, "ClassBound", 1 );
Group( G.1, G.2 )
gap> g.automorphisms := [];;
```

```
gap> GroupHomomorphismByImages(g,g,[g.1,g.2],[g.2,g.1*g.2]);;
gap> Add( g.automorphisms, last );
gap> GroupHomomorphismByImages(g,g,[g.1,g.2],[g.2,g.1]);;
gap> Add( g.automorphisms, last );
gap> StandardPresentation(g4,g,"ClassBound",14,"AgAutomorphisms");
Group( G.1, G.2, G.3, G.4, G.5, G.6, G.7, G.8, G.9, G.10, G.11, G.12,
G.13, G.14, G.15, G.16, G.17, G.18, G.19, G.20, G.21, G.22, G.23,
G.24, G.25, G.26, G.27, G.28, G.29, G.30, G.31, G.32, G.33, G.34,
G.35, G.36, G.37, G.38, G.39, G.40, G.41, G.42, G.43, G.44, G.45,
G.46, G.47, G.48, G.49, G.50, G.51, G.52, G.53 )
```

This function requires the package "anupq" (see 57.1).

58.7 IsomorphismPcpStandardPcp

`IsomorphismPcpStandardPcp(` G, S `)`

Let G be a p-group and let S be the standard presentation computed for G by `StandardPresentation`. `IsomorphismPcpStandardPcp` returns the isomorphism from G to S.

We illustrate the function with the following example.

```
gap> F := FreeGroup (6);
Group( f.1, f.2, f.3, f.4, f.5, f.6 )
gap> x := F.1;; y := F.2;; z := F.3;; w := F.4;; a := F.5;; b := F.6;;
gap> R := [x^3 / w, y^3 / w * a^2 * b^2, w^3 / b,
>             Comm (y, x) / z, Comm (z, x), Comm (z, y) / a, z^3 ];;
gap> q := F / R;;
gap> G := Pq (q, "Prime", 3, "ClassBound", 3);
Group( G.1, G.2, G.3, G.4, G.5, G.6 )
gap> S := StandardPresentation (q, 3, "ClassBound", 3);
Group( G.1, G.2, G.3, G.4, G.5, G.6 )
gap> phi := IsomorphismPcpStandardPcp (G, S);
GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
[ G.1, G.2, G.3, G.4, G.5, G.6 ],
[ G.1*G.2^2*G.3*G.4^2*G.5^2, G.1*G.2*G.3*G.5, G.3^2, G.4*G.6^2, G.5,
  G.6 ] )
```

This function requires the package "anupq" (see 57.1).

58.8 AutomorphismsPGroup

`AutomorphismsPGroup(` G `)`
`AutomorphismsPGroup(` G, *output-level*`)`

Let G be a p-group. Then `AutomorphismsPGroup` returns a generating set for the automorphism group of G. Each generator is described by its action on each of the generators of G. The runtime-information generated by the ANU pq is displayed at *output-level*, which must be a integer from 0 to 3.

We illustrate the function using the p-group considered above.

```
gap> Auts := AutomorphismsPGroup (G);
[ GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
    G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ],
    [ G.1, G.2*G.5^2, G.3, G.4, G.5, G.6 ] ),
  GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
    G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ],
    [ G.1, G.2*G.3, G.3, G.4, G.5, G.6 ] ),
  GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
    G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ], [ G.1*G.3^2, G.2, G.3*G.5, G.4,
      G.5, G.6 ] ), GroupHomomorphismByImages( Group( G.1, G.2, G.3,
    G.4, G.5, G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ],
    [ G.1*G.6, G.2*G.6, G.3, G.4, G.5, G.6 ] ),
  GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
    G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ], [ G.1*G.5^2, G.2*G.5, G.3, G.4,
      G.5, G.6 ] ), GroupHomomorphismByImages( Group( G.1, G.2, G.3,
    G.4, G.5, G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ], [ G.1*G.6^2, G.2*G.6, G.3, G.4,
      G.5, G.6 ] ), GroupHomomorphismByImages( Group( G.1, G.2, G.3,
    G.4, G.5, G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ],
    [ G.1*G.4, G.2*G.4*G.6, G.3, G.4*G.6, G.5, G.6 ] ),
  GroupHomomorphismByImages( Group( G.1, G.2, G.3, G.4, G.5,
    G.6 ), Group( G.1, G.2, G.3, G.4, G.5, G.6 ),
    [ G.1, G.2, G.3, G.4, G.5, G.6 ],
    [ G.1^2*G.3^2, G.2^2*G.3, G.3*G.5, G.4^2, G.5^2, G.6^2 ] ) ]
```

This function requires the package "anupq" (see 57.1).

58.9 IsIsomorphicPGroup

`IsIsomorphicPGroup(` *G*, *H* `)`

The functions returns true if G is isomorphic to H. Both groups must be ag groups of prime power order.

```
gap> p1 := Group( (1,2,3,4), (1,3) );
Group( (1,2,3,4), (1,3) )
gap> p2 := SolvableGroup( 8, 5 );
Q8
gap> p3 := SolvableGroup( 8, 4 );
D8
gap> IsIsomorphicPGroup( AgGroup(p1), p2 );
false
gap> IsIsomorphicPGroup( AgGroup(p1), p3 );
```

```
true
```

The function computes and compares the standard presentations for G and H (see 58.6).

This function requires the package "anupq" (see 57.1).

Chapter 59

Automorphism Groups of Special Ag Groups

This chapter describes functions which compute and display information about automorphism groups of finite soluble groups.

The algorithm used for computing the automorphism group requires that the soluble group be given in terms of a special ag presentation. Such presentations are described in the chapter of the GAP3 manual which deals with `Special Ag Groups`. Given a group presented by an arbitrary ag presentation, a special ag presentation can be computed using the function `SpecialAgGroup`.

The automorphism group is returned as a standard GAP3 group record. Automorphisms are represented by their action on the sag group generating set of the input group. The order of the automorphism group is also computed.

The performance of the automorphism group algorithm is highly dependent on the structure of the input group. Given two groups with the same sequence of LG-series factor groups it will usually take much less time to compute the automorphism group of the one with the larger automorphism group. For example, it takes less than 1 second (Sparc 10/52) to compute the automorphism group of the exponent 7 extraspecial group of order 7^3. It takes more than 40 seconds to compute the automorphism group of the exponent 49 extraspecial group of order 7^3. The orders of the automorphism groups are 98784 and 2058 respectively. It takes only 20 minutes (Sparc 10/52) to compute the automorphism group of the 2-generator Burnside group of exponent 6, a group of order $2^{28} \cdot 3^{25}$ whose automorphism group has order $2^{40} \cdot 3^{53} \cdot 5 \cdot 7$; note, however, that it can take substantially longer than this to compute the automorphism groups of some of the groups of order 64 (for nilpotent groups one should use the function `AutomorphismsPGroup` from the ANU PQ package instead).

The following section describes the function that computes the automorphism group of a special ag group (see 59.1). It is followed by a description of automorphism group elements and their operations (see 59.2 and 59.3). Functions for obtaining some structural information about the automorphism group are described next (see 59.4, 59.5 and 59.6). Finally, a function that converts the automorphism group into a form which may be more suitable for some applications is described (see 59.7).

59.1 AutGroupSagGroup

`AutGroupSagGroup(`G`)`
`AutGroupSagGroup(`G`, `l`)`

Given a special ag group G, the function `AutGroupSagGroup` computes the automorphism group of G. It returns a group generated by automorphism group elements (see 59.2). The order of the resulting automorphism group can be obtained by applying the function `Size` to it.

If the optional argument l is supplied, the automorphism group of G/G_l is computed, where G_l is the l-th term of the LG-series of G (see `More about Special Ag Groups`).

```
gap> C6 := CyclicGroup(AgWords, 6);;
gap> S3 := SymmetricGroup(AgWords, 3);;
gap> H := WreathProduct(C6,S3);;
gap> G := SpecialAgGroup(H / Centre(H));;
gap> G := RenamedGensSagGroup(G, "g"); # rename gens of G to [g1,g2,..,g12]
Group( g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 )
gap> G.name := "G";;
gap> A := AutGroupSagGroup(G);
Group( Aut(G, [ g1*g2, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12
 ]), Aut(G, [ g1, g2, g3^2, g4^2*g6^2*g7, g5^2*g6*g7^2, g6*g8^2,
  g7*g8^2, g8^2, g10*g11, g10, g9*g10, g9*g11*g12 ]), Aut(G,
[ g1, g2, g3, g4, g5^2*g6*g7^2, g6*g7, g7^2, g8^2, g9, g10, g11, g12
 ]), Aut(G, [ g1, g2, g3, g4*g6*g7^2, g5*g6^2*g7, g6, g7, g8, g9, g10,
  g11, g12 ]), Aut(G, [ g1, g2, g3, g4, g5*g6*g7^2, g6, g7, g8, g9,
  g10, g11, g12 ]), Aut(G, [ g1, g2, g3, g4^2, g5*g6^2*g7, g6^2*g8,
  g7^2*g8, g8, g10*g11, g10, g9*g10, g9*g11*g12 ]), Aut(G,
[ g1, g2, g3, g4*g6^2*g7, g5*g6*g7^2, g6, g7, g8, g9, g10, g11, g12
 ]), InnerAut(G, g1), InnerAut(G, g3), InnerAut(G, g4), InnerAut(G,
g5), InnerAut(G, g6), Aut(G, [ g1, g2, g3*g7*g8, g4, g5, g6*g8, g7,
  g8, g9, g10, g11, g12 ]), InnerAut(G, g7*g8), Aut(G,
[ g1, g2, g3, g4, g5*g8, g6, g7, g8, g9, g10, g11, g12 ]), InnerAut(G,
g8^2), Aut(G, [ g1, g2, g3, g4, g5, g6, g7, g8, g9, g9*g11, g9*g10,
  g10*g11*g12 ]), Aut(G, [ g1, g2, g3, g4, g5, g6, g7, g8, g10*g12,
  g10, g9*g11*g12, g9*g10 ]), InnerAut(G, g10), InnerAut(G,
g11), InnerAut(G, g12), InnerAut(G, g9) )
gap> Size(A);
30233088
gap> PrimePowersInt(last);
[ 2, 9, 3, 10 ]
```

The size of the outer automorphism group is easily computed as follows.

```
gap> innersize := Size(G) / Size(Centre(G));
23328
gap> outersize := Size(A) / innersize;
1296
```

59.2 Automorphism Group Elements

An element a of an automorphism group is a group element record with the following additional components:

`isAut`
: Is bound to `true` if a is an automorphism record.

`group`
: Is the special ag group G on which the automorphism a acts.

`images`
: Is the list of images of the generating set of G under a. That is, `a.images[i]` is the image of `G.generators[i]` under the automorphism.

The following components may also be defined for an automorphism group element:

`inner`
: If this component is bound, then it is either an element g of G indicating that a is the inner automorphism of G induced by g, or it is `false` indicating that a is not an inner automorphism.

`weight`
: This component is set for the elements of the generating set of the full automorphism group of a sag group. It stores the weight of the generator (see 59.4).

Along with most of the functions that can be applied to any group elements (e.g. `Order` and `IsTrivial`), the following functions are specific to automorphism group elements:

`IsAut(`*a*`)`

The function `IsAut` returns `true` if a is an automorphism record, and `false` otherwise.

`IsInnerAut(`*a*`)`

Returns `true` if a is an inner automorphism, and `false` otherwise. If `a.inner` is already bound, then the information stored there is used. If `a.inner` is not bound, `IsInnerAut` determines whether a is an inner automorphism, and sets `a.inner` appropriately before returning the answer.

59.3 Operations for Automorphism Group Elements

a `=` *b*

For automorphism group elements a and b, the operator `=` evaluates to `true` if the automorphism records correspond to the same automorphism, and `false` otherwise. Note that this may return `true` even when the two records themselves are different (one of them may have more information stored in it).

a `*` *b*

For automorphism group elements a and b, the operator `*` evaluates to the product ab of the automorphisms.

a / *b*

For automorphism group elements a and b, the operator / evaluates to the quotient ab^{-1} of the automorphisms.

a ^ *i*

For an automorphism group element a and an integer i, the operator ^ evaluates to the i-th power a^i of a.

a ^ *b*

For automorphism group elements a and b, the operator ^ evaluates to the conjugate $b^{-1}ab$ of a by b.

`Comm(`*a*`, `*b*`)`

The function `Comm` returns the commutator $a^{-1}b^{-1}ab$ of the two automorphism group elements a and b.

g ^ *a*

For a sag group element g and an automorphism group element a, the operator ^ evaluates to the image g^a of the ag word g under the automorphism a. The sag group element g must be an element of `a.group`.

S ^ *a*

For a subgroup S of a sag group and an automorphism group element a, the operator ^ evaluates to the image S^a of the subgroup S under the automorphism a. The subgroup S must be a subgroup of `a.group`.

list * *a*
a * *list*

For a list *list* and an automorphism group element a, the operator * evaluates to the list whose i-th entry is *list*`[`*i*`] * `*a* or *a*` * `*list*`[`*i*`]` respectively.

list ^ *a*

For a list *list* and an automorphism group element a, the operator ^ evaluates to the list whose i-th entry is *list*`[`*i*`] ^ `*a*.

Note that the action of automorphism group elements on the elements of the sag group via the operator ^ corresponds to the default action `OnPoints` (see `Other Operations`) so that the functions `Orbit` and `Stabilizer` can be used in the natural way. For example:

```
gap> Orbit(A, G.7);
[ g7, g7*g8^2, g7^2, g7^2*g8, g7*g8, g7^2*g8^2 ]
gap> Length(last);
6
```

```
gap> S := Subgroup(G, [G.11, G.12]);
Subgroup( G, [ g11, g12 ] )
gap> Size(S);
4
gap> Orbit(A, S);
[ Subgroup( G, [ g11, g12 ] ), Subgroup( G, [ g9*g10, g9*g11*g12 ] ) ]
gap> Intersection(last);
Subgroup( G, [  ] )
```

59.4 AutGroupStructure

`AutGroupStructure(`A`)`

The generating set of the automorphism group returned by `AutGroupSagGroup` is closely related to a particular subnormal series of the automorphism group. This function displays a description of the factors of this series.

Let A be the automorphism group of G. Let $G = G_1 > G_2 > \ldots > G_m > G_{m+1} = 1$ be the LG-series of G (see `More about Special Ag Groups`). For $0 \leq i \leq m$ let A_{2i+1} be the subgroup of A containing all those automorphisms which induce the identity on G/G_{i+1}. Clearly $A_1 = A$ and $A_{2m+1} = 1$. Furthermore, let A_{2i+2} be the subgroup of A_{2i+1} containing those automorphisms which also act trivially on the quotient G_i/G_{i+1}. Note that A_2/A_3 is always trivial. Thus the subnormal series

$$A = A_1 \geq A_2 \geq \ldots \geq A_{2m+1} = 1$$

of A is obtained. The subgroup A_i is the **weight** i subgroup of A. The **weight** of a generator α of A is defined to be the least i such that $\alpha \in A_i$.

The function `AutGroupStructure` takes as input an automorphism group A computed using `AutGroupSagGroup` and prints out a description of the non-trivial factors of the subnormal series of the automorphism group A.

The factor of **weight** i is A_i/A_{i+1}. A factor of even weight is an elementary abelian group, and it is described by giving its order. A factor of odd weight is described by giving a generating set for a faithful representation of it as a matrix group acting on a layer of the LG-series of G (the weight $2i-1$ factor acts on the LG-series layer G_i/G_{i+1}).

```
gap> AutGroupStructure(A);;

 Order of full automorphism group is 30233088 = 2^9 * 3^10

 Factor of size 2 (matrix group, weight 1)
  Field: GF(2)
    [1 1]
    [0 1]

 Factor of size 2 (matrix group, weight 3)
  Field: GF(3)
    [2]

 Factor of size 36 = 2^2 * 3^2 (matrix group, weight 5)
```

```
 Field: GF(3)
   [1 0 0]    [1 0 1]    [1 0 0]    [2 0 0]    [1 0 2]
   [0 2 1]    [0 1 2]    [0 1 1]    [0 1 2]    [0 1 1]
   [0 0 1]    [0 0 1]    [0 0 1]    [0 0 2]    [0 0 1]

   [2 0 0]    [1 0 1]
   [0 2 0]    [0 1 0]
   [0 0 2]    [0 0 1]

Factor of size 27 = 3^3 (elementary abelian, weight 6)

Factor of size 3 (elementary abelian, weight 8)

Factor of size 27 = 3^3 (elementary abelian, weight 10)

Factor of size 6 = 2 * 3 (matrix group, weight 11)
 Field: GF(2)
   [1 0 0 0]    [0 1 0 1]
   [1 0 1 0]    [0 1 0 0]
   [1 1 0 0]    [1 0 1 1]
   [0 1 1 1]    [1 1 0 0]

Factor of size 16 = 2^4 (elementary abelian, weight 12)
```

As mentioned earlier, each generator of the automorphism group has its weight stored in the record component `weight`.

```
gap> List(Generators(A), a -> a.weight);
[ 1, 3, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 8, 10, 10, 10, 11, 11, 12, 12,
  12, 12 ]
```

Note that the subgroup A_i of A is generated by the elements of the generating set of A whose weights are at least i. Hence, in analogy to strong generating sets of permutation groups, the generating set of A is a **strong generating set** relative to the chain of subgroups A_i.

The generating set of a matrix group displayed by `AutGroupStructure` corresponds directly to the list of elements of the corresponding weight in `A.generators`. In the example above, the first matrix listed at weight 5 corresponds to `A.generators[3]`, and the last matrix listed at weight 5 corresponds to `A.generators[9]`.

It is also worth noting that the generating set for an automorphism group returned by `AutGroupSagGroup` can be heavily redundant. In the example given above, the weight 5 matrix group can be generated by just three of the seven elements listed (for example elements 1, 5 and 6). The other four elements can be discarded from the generating set for the matrix group, and the corresponding elements of the generating set for A can also be discarded.

59.5 AutGroupFactors

`AutGroupFactors(`A`)`

The function `AutGroupFactors` takes as input an automorphism group A computed by `AutGroupSagGroup` and returns a list containing descriptions of the non-trivial factors A_i/A_{i+1} (see 59.4). Each element of this list is either a list $[p, e]$ which indicates that the factor is elementary abelian of order p^e, or a matrix group which is isomorphic to the corresponding factor.

```
gap> fact:=AutGroupFactors(A);;
gap> F := fact[3];;
gap> D := DerivedSubgroup(F);;
gap> Nice(Generators(D));
  Field: GF(3)
    [1 0 0]
    [0 1 2]
    [0 0 1]
gap> S := SylowSubgroup(F,2);;
gap> Nice(Generators(S));
  Field: GF(3)
    [2 0 0]     [1 0 0]
    [0 1 1]     [0 2 2]
    [0 0 2]     [0 0 1]
```

Of course, the factors of the returned series can be examine further. For example

```
gap> F := fact[3];;
gap> D := DerivedSubgroup(F);;
gap> Nice(Generators(D));
  Field: GF(3)
    [1 0 0]
    [0 1 2]
    [0 0 1]
gap> S := SylowSubgroup(F,2);;
gap> Nice(Generators(S));
  Field: GF(3)
    [2 0 0]     [1 0 0]
    [0 1 1]     [0 2 2]
    [0 0 2]     [0 0 1]
```

59.6 AutGroupSeries

`AutGroupSeries(`A`)`

The function `AutGroupSeries` takes as input an automorphism group A computed by `AutGroupSagGroup` and returns a list containing those subgroups A_i of A which give non-trivial quotients A_i/A_{i+1} (see 59.4).

```
gap> series:=AutGroupSeries(A);;
gap> series[7].weight;
11
```

```
gap> series[8].weight;
12
```

Each of the subgroups in the list has its weight stored in record component `weight`.

```
gap> series[7].weight;
11
gap> series[8].weight;
12
```

59.7 AutGroupConverted

`AutGroupConverted (`*A*`)`

Convert the automorphism group returned by `AutGroupSagGroup` into a group generated by `GroupHomomorphismByImages` records, and return the resulting group. Note that this function should not be used unless absolutely necessary, since operations for elements of the resulting group are substantially slower than operations with automorphism records.

```
gap> H := AutGroupConverted(A);
Group( GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g2, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3^2, g4^2*g6^2*g7, g5^2*g6*g7^2, g6*g8^2, g7*g8^2, g8^2,
  g10*g11, g10, g9*g10, g9*g11*g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4, g5^2*g6*g7^2, g6*g7, g7^2, g8^2, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4*g6*g7^2, g5*g6^2*g7, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4, g5*g6*g7^2, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4^2, g5*g6^2*g7, g6^2*g8, g7^2*g8, g8, g10*g11, g10,
  g9*g10, g9*g11*g12 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4*g6^2*g7, g5*g6*g7^2, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4^2, g5^2, g6^2*g7^2*g8^2, g7*g8^2, g8^2, g10*g11, g10,
  g9*g10, g9*g11*g12 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4*g6*g7^2, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
```

```
[ g1*g4^2, g2, g3*g6^2*g7, g4, g5*g7^2*g8, g6*g8^2, g7*g8^2, g8,
  g10*g11, g9*g10*g12, g11*g12, g11
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g5^2, g2, g3, g4*g7*g8^2, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g6^2*g7*g8, g2, g3, g4*g8, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3*g7*g8, g4, g5, g6*g8, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4*g8, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4, g5*g8, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g8, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g9*g11, g9*g10, g10*g11*g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4, g5, g6, g7, g8, g10*g12, g10, g9*g11*g12, g9*g10
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1, g2, g3, g4*g9*g12, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g9*g10*g11, g2, g3, g4*g12, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g9*g11, g2, g3, g4*g11*g12, g5, g6, g7, g8, g9, g10, g11, g12
 ] ), GroupHomomorphismByImages( G, G,
[ g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12 ],
[ g1*g9*g10*g11, g2, g3, g4*g9*g10*g11, g5, g6, g7, g8, g9, g10, g11,
  g12 ] ) )
```

Chapter 60

Cohomology

This chapter describes functions which may be used to perform certain cohomological calculations on a finite group G.

These include:

(i) The p-part Mul_p of the Schur multiplier *Mul* of G, and a presentation of a covering extension of Mul_p by G, for a specified prime p;
(ii) The dimensions of the first and second cohomology groups of G acting on a finite dimensional KG module M, where K is a field of prime order; and
(iii) Presentations of split and nonsplit extensions of M by G.

All of these functions require G to be defined as a finite permutation group. The functions which compute presentations require, in addition, a presentation of G. Finally, the functions which operate on a module M require the module to be defined by a list of matrices over K. This situation is handled by first defining a *GAP* record, which contains the required information. This is done using the function `CHR`, which must be called before any of the other functions. The remaining functions operate on this record.

If no presentation of the permutation group G is known, and G has order at most 32767, then a presentation can be computed using the function `CalcPres`. On the other hand, if you start with a finitely presented group, then you can create a permutation representation with the function `PermRep` (although there is no guarantee that the representation will be faithful ingeneral).

The functions all compute and make use of a descending sequence of subgroups of G, starting at G and ending with a Sylow p-subgroup of G, and it is usually most efficient to have the indices of the subgroups in this chain as small as possible. If you get a warning message, and one of the function fails because the indices in the chain computed are too large, then you can try to remedy matters by supplying your own chain. See Section 60.10 for more details, and an example.

If you set the external variable `InfoCohomolofy` to the value `Print`, then a small amount of information will be printed, indicating what is happening. If *chr* is the cohomology record you are working with, and you set the field *chr*`.verbose` to the value `true`, then you will see all the output of the external programs.

60.1 CHR

`CHR(`G`, `p`, [`F`], [`*mats*`] )`

`CHR` constructs a cohomology-record, which is used as a parameter for all of the other functions in this chapter. G must be a finite permutation group, and p a prime number. If present, F must either be zero or a finitely presented group with the same number of generators as G, of which the relators are satisfied by the generators of G. In fact, to obtain meaningful results, F should almost certainly be isomorphic to G. If present, *mats* should be a list of invertible matrices over the finite field $K = GF(p)$. The list should have the same length as the number of generators of G, and the matrices should correspond to these generators, and define a $GF(p)G$-module, which we will denote by M.

60.2 SchurMultiplier

`SchurMultiplier(`*chr*`)`

chr must be a cohomology-record that was created by a call of `CHR(`G`,`p`,[`F`],[`*mats*`])`. `SchurMultiplier` calculates the p-part Mul_p of the Schur multiplier *Mul* of G. The result is returned as a list of integers, which are the abelian invariants of Mul_p. If the list is empty, then Mul_p is trivial.

60.3 CoveringGroup

`CoveringGroup(`*chr*`)`

chr must be a cohomology-record, created by a call of `CHR(`G`,`p`,`F`,[`*mats*`])`, where F is a finitely presented group. `CoveringGroup` calculates a presentation of a covering extension of Mul_p by G, where Mul_p is the p-part of the Schur multiplier *Mul* of G. The set of generators of the finitely presented group that is returned is a union of two sets, which are in one-one correspondence with the generators of F and of Mul_p, respectively.

The relators fall into three classes:

a) Those that specify the orders of the generators of Mul_p;
b) Those that say that the generators of Mul_p are central; and
c) Those that give the values of the relators of F as elements of Mul_p.

60.4 FirstCohomologyDimension

`FirstCohomologyDimension(`*chr*`)`

chr must be a cohomology-record, created by a call of `CHR(`G`,`p`,`F`,`*mats*`)`. (If there is no finitely presented group F involved, then the third parameter of `CHR` should be given as 0.) `FirstCohomologyDimension` calculates and returns the dimension over $K = GF(p)$ of the first cohomology group $H^1(G, M)$ of the group G in its action on the module M defined by the matrices *mats*.

60.5 SecondCohomologyDimension

`SecondCohomologyDimension(`*chr*`)`

chr must be a cohomology-record, created by a call of `CHR(`*G*`,`*p*`,`*F*`,`*mats*`)`. (If there is no finitely presented group F involved, then the third parameter of `CHR` should be given as 0.) `SecondCohomologyDimension` calculates and returns the dimension over $K = GF(p)$ of the second cohomology group $H^2(G, M)$ of the group G in its action on the module M defined by the matrices *mats*.

60.6 SplitExtension

`SplitExtension(`*chr*`)`

chr must be a cohomology-record, created by a call of `CHR(`*G*`,`*p*`,`*F*`,`*mats*`)`, where F is a finitely presented group. `SplitExtension` returns a presentation of the split extension of the module M defined by the matrices *mats* by the group G. This is a straightforward calculation, and involves no call of the external cohomology programs. It is provided here for convenience.

60.7 NonsplitExtension

`NonsplitExtension(`*chr*`, [`*vec*`] )`

chr must be a cohomology-record, created by a call of `CHR(`*G*`,`*p*`,`*F*`,`*mats*`)`, where F is a finitely presented group. If present, *vec* must be a list of integers of length equal to the dimension over $K = GF(p)$ of the second cohomology group $H^2(G, M)$ of the group G in its action on the module M defined by the matrices *mats*. `NonsplitExtension` calculates and returns a presentation of a nonsplit extension of M by G. Since there may be many such extensions, and the equivalence classes of these extensions are in one-one correspondence with the nonzero elements of $H^2(G, M)$, the optional second parameter can be used to specify an element of $H^2(G, M)$ as a vector. The default value of this vector is `[1,0,...,0]`. The set of generators of the finitely presented group that is returned is a union of two sets, which are in one-one correspondence with the generators of F and of M (as an abelian group), respectively.

The relators fall into three classes:

a) Those that say that M is an abelian group of exponent p;
b) Those that define the action of the generators of F on those of M; and
c) Those that give the values of the relators of F as elements of M.

(*Note*:It is not particularly efficient to call `SecondCohomologyDimension` first to calculate the dimension of $H^2(G, M)$, which must of course be known if the second parameter is to be given; it is preferable to call `NonsplitExtension` immediately without the second parameter (which will return one nonsplit extension), and then to call `SecondCohomologyDimension`, which will at that stage return the required dimension immediately - all subsequent calls of `NonsplitExtension` on *chr* will also yield immediate results.)

60.8 CalcPres

`CalcPres(`*chr*`)`

`CalcPres` computes a presentation of the permutation group *chr*`.permgp` on the same generators as *chr*`.permgp`, and stores it as *chr*`.fpgp`. It currently only works for groups of order up to 32767, although that could easily be increased if required.

60.9 PermRep

`PermRep(`*G*`, `*K*`)`

`PermRep` calculates the permutation representation of the finitely presented group F on the right cosets of the subgroup K, and returns it as a permutation group of which the generators correspond to those of F. It simply calls the GAP3 Todd-Coxeter function. Of course, there is no guarantee in general that this representation will be faithful.

60.10 Further Information

Suppose, as usual, that the cohomology record *chr* was constructed with the call `CHR(`*G*`,`*p*`,[`*F*`],[`*mats*`])`. All of the functions make use of a strictly decreasing chain of subgroups of the permutation group G starting with G itself and ending with a Sylow p-subgroup P of G. In general, the programs run most efficiently if the indices between successive terms in this sequence are as small as possible. By default, GAP3 will attempt to find a suitable chain, when you call the first cohomology function on *chr*. However, you may be able to construct a better chain yourself. If so, then you can do this by assigning the record field *chr*`.chain` to the list L of subgroups that you wish to use. You should do that before calling any of the cohomology functions. Remeber that the first term in the list must be G itself, the sequence of subgroups must be strictly decreasing, and the last term must be equal to the Sylow subgroup stored as *chr*`.sylow`. (You can change *chr*`.sylow` to a different Sylow p-subgroup if you like.) Here is a slightly contrived example of this process.

```
gap> RequirePackage( "cohomolo" );
gap> G:=AlternatingGroup(16);;
gap> chr:=CHR(G,2);;
gap> InfoCohomology:=Print;;
gap> SchurMultiplier(chr);
#Indices in the subgroup chain are:  2027025 315
#WARNING: An index in the subgroup chain found is larger than 50000.
#This calculation may fail. See manual for possible remedies.
#I  Cohomology package: Calling external program.
#I  External program complete.
Error, 'Cohomology' failed for some reason.
 in
Cohomology( chr, true, false, false, TmpName(  ) ) called from
SchurMultiplier( chr ) called from
main loop
brk> quit;
```

The first index in the chain found by GAP was hopelessly large. Let's try and do better.

```
gap> P:=chr.sylow;;
gap> H1:=Subgroup(G, [(1,2)(9,10), (2,3,4,5,6,7,8),
>         (1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)(8,16)]);;
gap> Index(G,H1);
6435
gap> H2:=Subgroup(H1, [(1,2)(5,6), (1,2)(9,10), (2,3,4),
>         (1,5)(2,6)(3,7)(4,8),
```

```
>         (1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)(8,16)]);;
gap> Index(H1,H2);
1225
gap> IsSubgroup(H2,P);
true
```

If that had been false, we could have replaced chr.sylow by a Sylow 2-subgroup of H2. As it is true, we just continue.

```
gap> Index(H2,P);
81
gap> chr.chain := [G,H1,H2,P];;
gap> SchurMultiplier(chr);
#I  Cohomology package: Calling external program.
#I  External program complete.
#I  Removing temporary files.
[ 2 ]
```

Chapter 61

CrystGap–The Crystallographic Groups Package

The CrystGap package provides functions for the computation with affine crystallographic groups, in particular space groups. Also provided are some functions dealing with related linear matrix groups, such as point groups. For the definition of the standard crystallographic notions we refer to the International Tables [TH95], in particular the chapter by Wondratschek [Won95], and to the introductory chapter in [BBN+78]. Some material can also be found in the chapters 38.13 and 38.12. The principal algorithms used in this package are described in [EGN97b], a preprint of which in included in the `doc` directory of this package.

CrystGap is implemented in the GAP3 language, and runs on any system supporting GAP3 3.4.4. The function `WyckoffLattice`, however, requires the share package XGap, which in turn runs only under Unix. The functions described in this chapter can be used only after loading CrystGap with the command

```
gap> RequirePackage( "cryst" );
```

CrystGap has been developed by

Bettina Eick
Lehrstuhl D für Mathematik, RWTH Aachen, D-52056 Aachen, Germany
e-mail: `Bettina.Eick@math.RWTH-Aachen.de`

Franz Gähler
Centre de Physique Théorique, Ecole Polytechnique, F-91128 Palaiseau, France
e-mail: `gaehler@pth.polytechnique.fr`

Werner Nickel
School of Mathematical and Computational Sciences, University of St Andrews, St Andrews, Fife KY16 9SS, Scotland
e-mail: `werner@dcs.st-and.ac.uk`

Please send bug reports, suggestions and other comments to any of these e-mail addresses.

The first and third authors acknowledge financial support from the Graduiertenkolleg *Analyse und Konstruktion in der Mathematik*. The second author was supported by the Swiss

Bundesamt für Bildung und Wissenschaft in the framework of the HCM programme of the European Community. This collaboration was in part made possible by financial support from the HCM project *Computational Group Theory*.

61.1 Crystallographic Groups

An affine crystallographic group G is a subgroup of the group of all Euclidean motions of d-dimensional space, with the property that its subgroup T of all pure translations is a freely abelian, normal subgroup of G, which has rank at most equal to d, and which has finite index in G.

In this package, the term **CrystGroup** always refers to such an **affine** crystallographic group. Linear matrix groups, whether crystallographic or not, will carry different designations (see below). CrystGroups are represented as special matrix groups, whose elements are affine matrices of the form

```
[ M 0 ]
[ t 1 ]
```

acting on row vectors $(x, 1)$ from the right. Note that this is different from the crystallographic convention, where matrices usually act from the left on column vectors (see also 38.13). We have adopted this convention to maintain compatibility with the rest of GAP3.

The "linear"parts M of the elements of a CrystGroup G generate the **point group** P of G, which is isomorphic to the quotient G/T. There is a natural homomorphism from G to P, whose kernel is T. The translation vectors of the elements of T generate a free Z-module L, called the **translation lattice** of G. CrystGroups can be defined with respect to any basis of Euclidean space, but internally most computations will be done in a basis which contains a basis of L (see 61.3).

CrystGroups carry a special operations record `CrystGroupOps`, and are identified with a tag `isCrystGroup`. CrystGroups must be constructed with a call to `CrystGroup` (see 61.4) which sets the tag `isCrystGroup` to `true`, and sets the operations record to `CrystGroupOps`.

Warning: The groups in GAP3's crystallographic groups library (see 38.13), whether they are extracted with `SpaceGroup` or `TransposedSpaceGroup`, are **not** CrystGroups in the sense of this package, because CrystGroups have different record entries and a different operations record. However, a group extracted with `TransposedSpaceGroup` from that library can be converted to a CrystGroup by a call to `CrystGroup` (see 61.4).

61.2 Space Groups

A CrystGroup which has a translation subgroup of full rank is called a **space group**. Certain functions are available only for space groups, and not for general CrystGroups, notably all functions dealing with Wyckoff positions (see 61.17).

Space groups which are equivalent under conjugation in the affine group (shortly: affine equivalent space groups) are said to belong to the same **space group type**. As is well known, in three dimensions there are 219 such space group types (if only conjugation by transformations with positive determinant is allowed, there are 230).

Representatives of all space group types in dimensions 2, 3 and 4 can be obtained from the crystallographic groups library contained in GAP3 (see 38.13). They must be extracted with the function `CrystGroup`, and not with the usual extraction functions `SpaceGroup` and `TransposedSpaceGroup` of that library, as these latter functions return groups which do not have an operations record that would allow to compute with them. `CrystGroup` accepts exactly the same arguments as `SpaceGroup` and `TransposedSpaceGroup`. It returns the same group as `TransposedSpaceGroup`, but equipped with a working operations record.

Space group types (and thus space groups) are classified into $\mathbb{Z}$-classes and $\mathbb{Q}$-classes. Two space groups belong to the same $\mathbb{Z}$-class if their point groups, expressed in a basis of their respective translation lattices, are conjugate as subgroups of $GL(d,\mathbb{Z})$. If the point groups are conjugate as subgoups of $GL(d,\mathbb{Q})$, the two space groups are said to be in the same $\mathbb{Q}$-class. This provides also a classification of point groups (expressed in a lattice basis, i.e., integral point groups) into $\mathbb{Z}$-classes and $\mathbb{Q}$-classes.

For a given finite integral matrix group P, representing a point group expressed in a lattice basis, a set of representative space groups for each space group type in the $\mathbb{Z}$-class of P can be obtained with `SpaceGroupsPointGroup` (see 61.16). If, moreover, the normalizer of P in $GL(d,\mathbb{Z})$ is known (see 61.23), exactly one representative is obtained for each space group type. Representatives of all $\mathbb{Z}$-classes of maximal irreducible finite point groups are contained in a GAP3 library (see 38.12) in all dimensions up to 11, and in prime dimensions up to 23. For some other dimensions, at least $\mathbb{Q}$-class representatives are available.

Important information about a space group is contained in its **affine normalizer** (see 61.27), which is the normlizer of the space group in the affine group. In a way, the affine normalizer can be regarded as the symmetry of the space group.

Warning: Groups which are called space groups in this manual should not be confused with groups extracted with `SpaceGroup` from the crystallographic groups library (see 38.13). The latter are not CrystGroups in the sense of this package.

61.3 More about Crystallographic Groups

In this section we describe how a CrystGroup G is represented internally. The casual user can skip this section in a first reading. Although the generators of a CrystGroup can be specified with respect to any basis, most computations are done internally in a special, standard basis, which is stored in `G.internalBasis`. The results are translated into the user-specified basis only afterwards. `G.internalBasis` consists of a (standard) basis of the translation lattice of G, complemented, if necessary, with suitable standard basis vectors. The standard basis of the translation lattice is stored in `G.translations`.

As soon as `G.internalBasis` has been determined, both the CrystGroup G and its point group P obtain a component `internalGenerators`. For the point group P, the component `P.internalGenerators` contains a set of generators of P, expressed with respect to the `internalBasis` of G, whereas for the CrystGroup G the component `G.internalGenerators` contains a set of homomorphic preimages of `P.internalGenerators` in G, also expressed in the `internalBasis` of G. Thus `G.internalGenerators` does not contain any translation generators. These are easy to add, however: With respect to the internal basis, the translations are generated by the first k standard basis vectors, where k is the rank of the translation lattice.

Note that the `internalGenerators` of both a point group P and a CrystGroup G may be changed by some functions, notably by `FpGroup`. Thus they need not have any obvious connection to `P.generators` and `G.generators`, respectively. Internal record entries of a CrystGroup should **never be changed** by the user.

61.4 CrystGroup

`CrystGroup(` *matgroup* `)`

`CrystGroup(` *generating matrices* `)`

`CrystGroup(` *list of generators*`,` *identity* `)`

`CrystGroup(` *integers* `)`

`CrystGroup(` *string* `)`

`CrystGroup` accepts as arguments either a group of affine matrices, or a list of generating affine matrices, or an argument identifying a space group from the crystallographic groups library, i.e., a list of two or five integers, or a string containing a Hermann-Mauguin symbol, and converts it into a CrystGroup in the sense of this package. `CrystGroup` tests whether the generators are indeed affine matrices.

61.5 IsCrystGroup

`IsCrystGroup(` G `)`

tests whether `G.isCrystGroup` is present and `true`. `G.isCrystGroup` is set by `CrystGroup`.

61.6 PointGroup

`PointGroup(` G `)`

extracts the point group P of a space group G, binds it to `G.pointGroup`, and returns it. It also determines the homomorphism from G to P, and binds it to `G.pointHomom`. A point group P has always a component `P.isPointGroup` set to `true`, and a component `P.crystGroup` containing the CrystGroup from which it was constructed.

61.7 TranslationsCrystGroup

`TranslationsCrystGroup(` G `)`

determines a basis of the translation lattice of G, binds it to `G.translations`, and returns it. Note that this translation lattice is always invariant under the point group P of G. If `G.translations` is not yet present, a finite presentation of P needs to be determined. A basis of the translation lattice can also be added by the user, with `AddTranslationsCrystGroup` (see 61.8).

Warning: The component `G.translations` must **never** be set by hand. The functions `TranslationsCrystGroup` and `AddTranslationsCrystGroups` have important (and wanted) side effects.

61.8 AddTranslationsCrystGroup

`AddTranslationsCrystGroup(` G`,` *basis* `)`

Since `TranslationsCrystGroup` (see 61.7) needs a presentation of the point group, the computation of `G.translations` can be rather time consuming. If a basis of the translation lattice is known, `AddTranslationsCrystGroup` can be used to add this knowledge to a CrystGroup. If `G.translations` is already known, its value is kept without further notice. It is the responsibility of the user that the basis handed over to `AddTranslationsCrystGroup` is a correct basis of the translation lattice. In case of doubt, the function `CheckTranslations` (see 61.9) can be used to check whether the basis added was indeed correct.

Warning: The component `G.translations` must **never** be set by hand. The functions `TranslationsCrystGroup` and `AddTranslationsCrystGroups` have important (and wanted) side effects.

61.9 CheckTranslations

`CheckTranslations(` G `)`

checks whether `G.translations` is indeed correct. If `G.translations` is incorrect, a warning message is printed, otherwise GAP3 remains silent. In the case of an incorrect translation basis a new CrystGroup must be created, and the computations must be started afresh, because the wrong translation basis may have produced wrong information components. `CheckTranslations` is useful if a basis has been added with `AddTranslationsCrystGroup`, and doubts arise later whether the basis added was correct.

61.10 ConjugatedCrystGroup

`ConjugatedCrystGroup(` G, c `)`

returns a new CrystGroup which is a conjugate of G. The conjugator c can either be a d-dimensional linear matrix (which then is complemented with the zero translation), or a $(d+1)$-dimensional affine matrix. The generators are conjugated as $g^c = cgc^{-1}$. Some components which are bound in G are copied and translated to the new basis, in particular `G.generators`, `G.translations`, `G.internalBasis`, and `G.wyckoffPositons`. If `G.internalBasis` is bound,

`ConjugatedCrystGroup( G, G.internalBasis )`

returns a CrystGroup whose translation lattice (of rank k) is generated by the first k rows of the identity matrix. `ConjugatedCrystGroup` allows as input only a parent CrystGroup.

61.11 FpGroup for point groups

`FpGroup(` P `)`

computes a finite presentation of the point group P, and binds it to `P.fpGroup`. If P (and thus its CrystGroup `G := P.crystGroup`) is solvable, a power-commutator presentation is returned.

Warning: If P is solvable, the abstract generators are not necessarily isomorphic images of `P.generators` (see 61.3).

61.12 FpGroup for CrystGroups

`FpGroup(` G `)`

computes a finite presentation of the CrystGroup G, and binds it to `G.fpGroup`. If the point group (and thus G) is solvable, a power-commutator presentation is returned. The presentation is always an extension of the presentation of the point group (which is computed if necessary).

Warning: The abstract generators of the presentation are not necessarily isomorphic images of `G.generators` (see 61.3).

61.13 MaximalSubgroupsRepresentatives

`MaximalSubgroupsRepresentatives( S, "translationEqual", [, ind ] )`

`MaximalSubgroupsRepresentatives( S, "classEqual", ind )`

`MaximalSubgroupsRepresentatives( S, ind )`

returns a list of conjugacy class representatives of maximal subgroups of the CrystGroup S. If *ind* is present, which must be a prime or a list of primes, only those subgroups are returned whose index is a power of a prime contained in or equal to *ind*. If the flag "translationEqual"is present, only those subgroups are returned which are translation-equal (translationengleich) with S. If the *flag* "classEqual"is present, only those subgroups are return which are class-equal (klassengleich) with S. *ind* is optional only if the flag "latticeEqual"is present. In all other cases, *ind* is required.

61.14 IsSpaceGroup

`IsSpaceGroup(` S `)`

determines whether the CrystGroup S is a space group (see 61.1).

61.15 IsSymmorphicSpaceGroup

`IsSymmorphicSpaceGroup(` S `)`

determines whether the space group S is symmorphic. A space group is called **symmorphic** if it is equivalent to a semidirect product of its point group with its translation subgroup.

61.16 SpaceGroupsPointGroup

`SpaceGroupsPointGroup(` P `)`

`SpaceGroupsPointGroup(` P`,` *normalizer elements* `)`

where P is any finite subgroup of $GL(d, Z)$, returns a list of all space groups with point group P, up to conjugacy in the full translation group of Euclidean space. All these space groups are returned as CrystGroups in standard representation. If a second argument is present, which must be a list of elements of the normalizer of P in $GL(d, Z)$, only space groups inequivalent under conjugation with these elements are returned. If these normalizer elements, together with P, generate the full normalizer of P in $GL(d, Z)$, then exactly one representative of each space group type is obtained.

61.17 Wyckoff Positions

A Wyckoff position of a space group G is an equivalence class of points in Euclidean space, having stabilizers which are conjugate subgroups of G. Apart from a subset of lower dimension, which contains points with even bigger stabilizers, a Wyckoff position consists of a G-orbit of some affine subspace A. A Wyckoff position W therefore can be specified by a representative affine subspace A and its stabilizer subgroup. In CrystGap, a Wyckoff position W is represented as a record with the following components:

`W.basis`
: Basis of the linear space L parallel to A. This basis is also a basis of the intersection of L with the translation lattice of S.
Can be extracted with `WyckoffBasis( W )`.

`W.translation`
: `W.translation` is such that `A = L + W.translation`.
Can be extracted with `WyckoffTranslation( W )`.

`W.stabilizer`
: The stabilizer subgroup of any generic point in A.
Can be extracted with `WyckoffStabilizer( W )`.

`W.class`
: Wyckoff positions carry the same class label if and only if their stabilizers have point groups which are conjugate subgroups of the point group of S.
Can be extracted with `WyckoffPosClass( W )`.

`W.spaceGroup`
: The space group of which it is a Wyckoff position.
Can be extracted with `WyckoffSpaceGroup( W )`.

`W.isWyckoffPosition`
: A flag identifying the record as a Wyckoff position. It is set to true.
Can be tested with `IsWyckoffPosition( W )`.

`W.operations`
: The operations record of a Wyckoff position. It currently contains only a Print function.

61.18 WyckoffPositions

`WyckoffPositions(` *G* `)`

returns the list of all Wyckoff positions of the space group G.

61.19 WyckoffPositionsByStabilizer

`WyckoffPositionsByStabilizer(` *G*`,` *U* `)`,

where G is a space group and U a subgroup of the point group or a list of such subgroups, determines only the Wyckoff positions (see 61.18) having a representative affine subspace whose stabilizer has a point group equal to the subgroup U or contained in the list U, respectively.

61.20 WyckoffPositionsQClass

`WyckoffPositionsQClass(` *G*`,` *S* `)`

For space groups with larger point groups, most of the time in the computation of Wyckoff positions (see 61.18) is spent computing the subgroup lattice of the point group. If Wyckoff positions are needed for several space groups which are in the same Q class, and therefore have isomorphic point groups, one can avoid recomputing the same subgroup lattice for each of them as follows. For the computation of the Wyckoff positions of the first space group S one uses a call to `WyckoffPositions`. For the remaining space groups, S is then passed as a second argument to `WyckoffPositionsQClass(` G`,` S `)`, which uses some of the results already obtained for S.

61.21 WyckoffOrbit

`WyckoffOrbit( W )`

takes a Wyckoff position W (see 61.17) and returns a list of Wyckoff positions which are different representations of W, such that the representative affine subspaces of these representations form an orbit under the space group G of W, modulo lattice translations.

61.22 WyckoffLattice

`WyckoffLattice(` G `)`

If a point x in a Wyckoff position W_1 has a stabilizer which is a subgroup of the stabilizer of some point y in a Wyckoff position W_2, then the closure of W_1 will contain W_2. These incidence relations are best represented in a graph. `WyckoffLattice(` G `)` determines and displays this graph using XGAP (note that XGAP runs only under Unix plus the X Window System). Each Wyckoff position is represented by a vertex. If W_1 contains W_2, its vertex is placed below that of W_2 (i.e., Wyckoff positions with bigger stabilizers are placed higher up), and the two are connected, either directly (if there is no other Wyckoff position in between) or indirectly. With the left mouse button and with the XGAP `CleanUp` menu it is possible to change the layout of the graph (see the XGAP manual). When clicking with the right mouse button on a vertex, a pop up menu appears, which allows to obtain the following information about the representative affine subspace of the Wyckoff position:

`StabDim`:
: Dimension of the affine subspace of stable points.

`StabSize`:
: Size of the stabilizer subgroup.

`ClassSize`:
: Number of Wyckoff positions having a stabilizer whose point group is in the same subgroup conjugacy class.

`IsAbelian`, `IsCyclic`, `IsNilpotent`, `IsPerfect`, `IsSimple`, `IsSolvable`:
: Information about the stabilizer subgroup.

`Isomorphism`:
: Isomorphism type of the stabilizer subgroup. Works only for small sizes.

`ConjClassInfo`:
: Prints (in the GAP3 window) information about each of the conjugacy classes of the stabilizer, namely the order, the trace and the determinant of its elements, and the size of the conjugacy class. Note that trace refers here only to the trace of the point group part, without the trailing 1 of the affine matrix.

`Translation`:
The representative point of the affine subspace.

`Basis`:
The basis of the linear space parallel to the affine subspace.

61.23 NormalizerGL

`NormalizerGL( G )`,

where G is a finite subgroup of *GL(d,Z)*, returns the normalizer of G in *GL(d,Z)*. At present, this function is available only for groups which are the point group of a CrystGroup extracted from the space group library.

61.24 CentralizerGL

`CentralizerGL( G )`,

where G is a finite subgroup of *GL(d,Z)*, returns the centralizer of G in *GL(d,Z)*. At present, this function is available only for groups which are the point group of a CrystGroup extracted from the space group library.

61.25 PointGroupsBravaisClass

`PointGroupsBravaisClass( B )`

`PointGroupsBravaisClass( B [, norm ] )`

where B is a finite integral matrix group, returns a list of representatives of those conjugacy classes of subgroups of B which are in the same Bravais class as B. These representatives are returned as parent groups, not subgroups. If B is a Bravais group, the list contains a representative of each point group in the Bravais class of B. If a second argument is present, which must be a list of elements of the normalizer of B in $GL(d,\mathbb{Z})$, only subgroups inequivalent under conjugation with these elements are returned.

61.26 TranslationNormalizer

`TranslationNormalizer( S )`

returns the normalizer of the space group S in the full translation group. At present, this function is implemented only for space groups, not for general CrystGroups. The translation normalizer *TN* of S may contain a continuous subgroup C. A basis of the space of such continuous translations is bound in `TN.continuousTranslations`. Since this subgroup is not finitely generated, it is **not** contained in the group generated by `TN.generators`. Properly speaking, the translation normalizer is the span of *TN* and C together.

61.27 AffineNormalizer

`AffineNormalizer( S )`

returns the affine normalizer of the space group S. The affine normalizer contains the translation normalizer as a subgroup. Similarly as with `TranslationNormalizer`, the subgroup

C of continuous translations, which is not finitely generated, is not part of the group that is returned. However, a basis of the space of continuous translations is bound in the component `continuousTranslations`.

At present, this function is available only for for space groups, not for general `CrystGroups`. Moreover, the `NormalizerGL` (see 61.23) of the point group of S must be know, which currently is the case only for CrystGroups extracted from the space group library.

61.28 AffineInequivalentSubgroups

`AffineInequivalentSubgroups( sub )`

takes as input a list of subgroups with common parent space group S, and returns a sublist of those which are affine inequivalent. For this, the affine normalizer of S is required, which currently is available only if S is a space group extracted from the space groups library.

61.29 Other functions for CrystGroups

In the operations record of a `CrystGroup` many of the usual GAP3 functions are replaced with a `CrystGroup` specific implementation. For other functions the default implementation can be used. Since `CrystGroups` are matrix groups, all functions which work for a finite matrix group should work also for a finite `CrystGroup` (i.e., one which contains no pure translations). Of course, functions which require a **finite** group as input will work only for finite `CrystGroups`. Following is a (probably not exhaustive) list of functions that are known to work for also for **infinite** `CrystGroups`.

```
in
Parent, IsParent, Group, IsGroup
Subgroup, IsSubgroup, AsSubgroup, Index
Centralizer, Centre, Normalizer
Closure, NormalClosure
Intersection, NormalIntersection
ConjugacyClassSubgroups, ConjugateSubgroups
DerivedSubgroup, CommutatorSubgroup, Core
DerivedSeries, SubnormalSeries
FactorGroup, CommutatorFactorGroup
ConjugateSubgroup, TrivialSubgroup
IsAbelian, IsCentral, IsTrivial
IsNormal, IsSubnormal, IsPerfect, IsSolvable
```

The following functions work for `CrystGroups` **provided** the subgroup H has **finite index** in G. The elements of the resulting domain are given in ascending order (with respect to an ad hoc, but fixed ordering).

```
Cosets( G, H )
RightCosets( G, H )
LeftCosets( G, H )
```

The following functions dealing with group operations work for `CrystGroups` provided the orbits of the action are **finite**. Since `CrystGroups` are not finite in general, this is a non-trivial requirement, and so some care is needed.

```
Orbit( G, d, opr )
Orbits( G, D, opr )
OrbitLengths( G, D, opr )
Stabilizer( G, d, opr )
RepresentativeOperation( G, d, e, opr )
RepresentativesOperation( G, d, opr )
```

The following functions have a `CrystGroup` specific implementation, but work for **finite** `CrystGroups` only:

```
Elements( G )
ConjugacyClasses( G )
PermGroup( G )
SylowSubgroup( G, p )
```

61.30 Color Groups

Elements of a color group C are colored in the following way. The elements having the same color as `C.identity` form a subgroup H, which has finite index n in C. H is called the `ColorSubgroup` of C. Elements of C have the same color if and only if they are in the same right coset of H in C. A fixed list of right cosets of H in C, called `ColorCosets`, therefore determines a labelling of the colors, which runs from 1 to n. Elements of H by definition have color 1, i.e., the coset with representative `C.identity` is always the first element of `ColorCosets`. Right multiplication by a fixed element g of C induces a permutation *p(g)* of the colors of the parent of C. This defines a natural homomorphism of C into the permutation group of degree n. The image of this homomorphism is called the `ColorPermGroup` of C, and the homomorphism to it is called the `ColorHomomorphism` of C.

61.31 ColorGroup

A color group is constructed with

`ColorGroup( G, H ),`

which returns a colored copy of G, with color subgroup H. G must be a parent group, and H must be a finite index subgroup of G. Color subgroups must be constructed as subgroups of color parent groups, and not by coloring uncolored subgroups. Subgroups of color groups will inherit the coloring of their parent, including the labelling of the colors.

Color groups are identified with a tag `isColorGroup`. They always have a component `colorSubgroup`. Color parent groups moreover always have a component `colorCosets`, which fixes a labelling of the colors.

Groups which may be colored include, in particular, `CrystGroups`, but coloring of any finite group, such as a finite matrix group or permutation group, should work as well.

61.32 IsColorGroup

`IsColorGroup( G )`

checks whether `G.isColorGroup` is bound and true.

61.33 ColorSubgroup

`ColorSubgroup( G )`

returns the color subgroup of G.

61.34 ColorCosets

`ColorCosets( G )`

returns the color cosets of G.

61.35 ColorOfElement

`ColorOfElement( G, elem )`

returns the color of an element.

61.36 ColorPermGroup

`ColorPermGroup( G )`

returns the ColorPermGroup of G, which is the permutation group induced by G acting on the colors of the parent of G.

61.37 ColorHomomorphism

`ColorHomomorphism( G )`

returns the homomomorphism from G to its ColorPermGroup.

61.38 Subgroup for color groups

If C is a color group,

`Subgroup( C, [elems] )`

returns a colored subgroup of C, whereas

`C.operations.UncoloredSubgroup( C, [elems] )`

returns an ordinary, uncolored subgroup.

61.39 PointGroup for color CrystGroups

If C is a color `CrystGroup` whose color subgroup is lattice-equal (or translationengleich) with C, the point group of C can consistently be colored. In that case,

`PointGroup( C )`

returns a colored point group. Otherwise, the point group will be uncolored. An uncolored point group can always be obtained with

`C.operations.UncoloredPointGroup( C )`

61.40 Inequivalent colorings of space groups

Two colorings of a space group S are equivalent if the two `ColorSubgroups` are conjugate in the affine normalizer of S.

`AffineInequivalentSubgroups( L )`

where L is a list of sub space groups with a common parent space group S, returns a list of affine inequivalent subgroups from L. At present, this routine is supported only for CrystGroups constructed from the space group library.

A list of prime index p subgroups of S (actually, a list of conjugacy class representatives of such subgroups) can be obtained with

`Filtered( MaximalSubgroupsRepresentatives( S, p ), U -> U.index = p )`

These two routines together therefore allow to determine all inequivalent colorings of S with p colors.

Chapter 62

The Double Coset Enumerator

62.1 Double Coset Enumeration

Double Coset Enumeration (DCE) can be seen either as a space- (and time-) saving variant of ordinary Coset Enumeration (the Todd-Coxeter procedure), as a way of constructing finite quotients of HNN-extensions of known groups or as a way of constructing groups given by symmetric presentations in a sense defined by Robert Curtis. A double coset enumeration works with a finitely-presented group G, a finitely generated subgroup H (given by generators) and a finite subgroup K, given explicitly, usually as a permutation group. The action of G on the cosets of H divides into orbits under K, and is constructed as such, using only a relatively small amount of information for each orbit.

The next two sections 62.2 and 62.3 describe the authorship of the package, and the simple procedure for installing it.

In 62.4 the calculation performed by the double coset enumerator, and the meaning of the input is described more precisely. The following sections: 62.5, 62.6 and 62.7 describe how the input is organized as `GAP3` data, and a number of examples are given in 62.8.

The data structure returned by DCE is described in 62.9 and the control of the comments printed during calculation in 62.10. Succeeding sections: 62.11, 62.12, 62.13, 62.14, 62.15 and 62.16 describe the basic functions used to run DCE, extract information from the result, and save and restore double coset tables. The use of these functions is shown in 62.17.

The user can exert considerable control over the behaviour of DCE, as described in 62.18 and 62.19.

Since double coset enumeration can construct permutation representations of very high degree, it may not be feasible to extract permutations from the result. Nevertheless, some analysis of the permutation representation may be possible. This is described in 62.20 and the functions used are documented in: 62.21, 62.22 and 62.23 and demonstrated in 62.24.

Finally, the link with Robert Curtis' notion of a symmetric presentation is described in 62.25 with detailed documentation in 62.26 and 62.27.

More detailed documentation of the data structures used in double coset enumeration, and the internal functions available to access them is found in the document "GAP Double Coset Enumerator – Internals", found in the `doc` directory of the `dce` package.

62.2 Authorship and Contact Information

The dce package was written by Steve Linton of the Division of Computer Science, University of St. Andrews, North Haugh, St. Andrews, Fife, KY16 9SS, UK
e-mail: sal@dcs.st-and.ac.uk, and any problems or questions should be directed to him.

The work was done mainly during a visit to Lehrstuhl D f'ur Mathematik, RWTH-Aachen, Aachen, Germany, and the author gratefuly acknowledges the hospitality of Lehrstuhl D and the financial support of the Deutsche Forschungsgemeinschaft.

62.3 Installing the DCE Package

The DCE package is completely written in the GAP3 language, it does not require any additional programs and/or compilations. It will run on any computer that runs GAP3. In the following we will describe the installation under UNIX. The installation on the Atari ST, TT or IBM PC is similar.

In the example we give we will assume that GAP3 is installed in the home directory of a pseudo user gap and that you, as user gap, want to install the DCE package. Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file dce.zoo (see 56.1). Then you must locate the GAP3 directories containing lib/ and doc/, this is usually gap3r4p2 where 2 is to be replaced by the current the patch level.

```
user@host:~ > ls -l
drwxr-xr-x  11 gap      gap         1024 Jul  8 14:05 gap3r4p2
-rw-r--r--   1 gap      gap        76768 Sep 11 12:33 dce.zoo
user@host:~ > ls -l gap3r4p2
drwxr-xr-x   2 gap      gap         3072 Aug 26 11:53 doc
drwxr-xr-x   2 gap      gap         1024 Jul  8 14:05 grp
drwxr-xr-x   2 gap      gap         2048 Aug 26 09:42 lib
drwxr-xr-x   2 gap      gap         2048 Aug 26 09:42 src
drwxr-xr-x   2 gap      gap         1024 Aug 26 09:42 tst
```

Unpack the package using unzoo (see 56.3). Note that you must be in the directory containing gap3r4p2 to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
user@host:~ > unzoo x dce.zoo
user@host:~ > ls -l gap3r4p2/pkg/dce
-rw-r--r--   1 gap      gap         1536 Nov 22 04:16 README
-rw-r--r--   1 gap      gap       116553 Nov 22 04:02 init.g
-rw-r--r--   1 gap      gap        48652 Nov 22 04:18 dce.tex
-rw-r--r--   1 gap      gap       549708 Nov 22 04:18 dce.dvi
-rw-r--r--   1 gap      gap        14112 Nov 22 04:18 dce-inte.tex
-rw-r--r--   1 gap      gap       116553 Nov 22 03:41 dce.g
```

Copy the file dce.tex into the doc/ directory, and edit manual.tex (also in the doc/ directory) and add a line \Include{dce} after the line \Include{cohomolo} near the end of the file. Finally run latex again (see 56.3).

```
user@host:~ > cd gap3r4p2/pkg/dce
user@host:../dce > cp dce.tex ../../doc
user@host:../dce > cd ../../doc
user@host:../doc > vi manual.tex # and add the necessary line
user@host:../doc > latex manual
# a few messages about undefined references
user@host:../doc > latex manual
# a few messages about undefined references
user@host:../doc > makeindex manual
# 'makeindex' prints some diagnostic output
user@host:../doc > latex manual
# there should be no warnings this time
```

Now it is time to test the installation. Let us assume that the executable of GAP3 lives in `src/` and is called `gap`.

```
user@host:~/gap3r4p2 > src/gap -b
gap> RequirePackage( "dce" );
gap> k := SymmetricGroup(3);
Group( (1,3), (2,3) )
gap> c := AbstractGenerator("c");;
gap> d := AbstractGenerator("d");;
gap> S5Pres := rec(
>     groupK := k,
>     gainGroups := [rec(), rec(dom := 3)],
>     gens := [rec(name := c, invol := true, wgg := 2),
>              rec(name := d, invol := true, wgg := 1)],
>     relators := [DCEWord(k,c*d)^3,DCEWord(k,[(2,3),c])^3],
>     subgens := [DCEWord(k,(1,2,3)), DCEWord(k,(1,2)), DCEWord(k,c)]);
rec(
  groupK := Group( (1,3), (2,3) ),
  gainGroups := [ rec(
           ), rec(
          dom := 3 ) ],
  gens := [ rec(
          name := c,
          invol := true,
          wgg := 2 ), rec(
          name := d,
          invol := true,
          wgg := 1 ) ],
  relators :=
   [ DCEWord(Group( (1,3), (2,3) ),[c, d])^3, DCEWord(Group( (1,3),
        (2,3) ),[(2,3), c])^3 ],
  subgens :=
   [ DCEWord(Group( (1,3), (2,3) ),[(1,2,3)]), DCEWord(Group( (1,3),
        (2,3) ),[(1,2)]), DCEWord(Group( (1,3), (2,3) ),[c]) ] )
gap> u := DCE(S5Pres);
#I Set up generators and inverses
```

```
#I Set up column structure: 4 columns
#I Pre-processed relators
#I Done subgroup generators
#I Also done relators in subgroup
#I Pushing at weight 3
#I      1 double 1 single 1 blanks
#I 1 DCEWord(K,[c, d])^3
#I   1 cases
#I 1 DCEWord(K,[(2,3), c])^3
#I   1 cases
#I Pushing at weight 5
#I      3 double 5 single 1 blanks
#I 2 DCEWord(K,[c, d])^3
#I   1 cases
#I 2 DCEWord(K,[(2,3), c])^3
#I   1 cases
#I 3 DCEWord(K,[c, d])^3
#I   2 cases
#I 3 DCEWord(K,[(2,3), c])^3
#I   3 cases
#I Pushing at weight 101
#I      3 double 5 single 0 blanks
#I 1 DCEWord(K,[c, c])
#I   1 cases
#I 1 DCEWord(K,[d, d])
#I   1 cases
#I Pushing at weight 103
#I      3 double 5 single 0 blanks
#I 2 DCEWord(K,[c, c])
#I   1 cases
#I 2 DCEWord(K,[d, d])
#I   1 cases
#I 3 DCEWord(K,[c, c])
#I   2 cases
#I 3 DCEWord(K,[d, d])
#I   1 cases
<< Double coset table "No name" closed 3 double 5 single >>
```

If `RequirePackage` signals an error check the permissions of the subdirectories `pkg/` and `dce/`.

62.4 Mathematical Introduction

Coset Enumeration can be considered as a means of constructing a permutation representation of a finitely-presented group. Let G be such a group, and let $\Omega = H\backslash G$ be the set of right cosets of a subgroup H, on which G acts. Let K be a subgroup of G. The action of K will divide Ω into orbits corresponding to the double cosets $H\backslash G/K$. Now, suppose that $x \in G$ and let $L = K \cap K^{x^{-1}}$. Let $D \in H\backslash G/K$ be a double coset and let d be a fixed single coset contained in it (so that $D = dK$). Let $l \in L$. Then

$$(dl)x = (dx)l^x \in (dx)K$$

so that the action of x on Ω can be computed from its action on a set of orbit representatives of L and its action on L, which takes place within K. If L is large this can provide a considerable saving of space. This space saving is the motivation for double coset enumeration. The group L is called the **gain group** of x, since the space saving is approximately a factor of $|L|$.

The input to the double coset enumeration algorithm includes a specification of a group K, and of a set of generators X. For each $x \in X$, a pair of subgroups $L_x, L^{(x)} \leq K$ is given, together with an isomorphism $\theta_x : L_x \to L^{(x)}$. This information defines a group F, obtained from the free product of K with the free group F_X by requiring that each x act by conjugation on L_x according to the map θ_x. Technically F is a multiple HNN-extension of K.

The final parts of the input (mathematically speaking, in practice additional input is used to guide the program towards efficiency) are a set of relators R and a set of subgroup generators W, consisting of elements of the free product of K and F_X, that is words composed of the letters x and elements of K.

The algorithm then constructs a compact representation of the action of a group $G = F/N$, where $N = \langle R \rangle^F$, on the set Ω of cosets of $H = \langle W \rangle N/N$. This can also be viewed as a permutation action of F, with kernel N and point stabiliser $\langle W \rangle N$. We take this view to avoid writing KN/N all the time.

This representation is organized in terms of the orbits (double cosets) of K on Ω. For each orbit D, an arbitrary representative $d \in D$ is chosen, and the group $M_d = \mathrm{Stab}_K(d)$ is recorded (as a subgroup of K). For historical reasons this group is known as the "muddle group of the double coset. This allows us to refer to elements of Ω by expressions of the form dk, with

$$d_1k_1 = d_2k_2 \iff d_1 = d_2 \text{ and } k_1k_2^{-1} \in M_{d_1}.$$

We call such an expression a **name** for the element of Ω.

In addition for each $x \in X$, and for each orbit of L_x contained in D, with representative dk, a name for the point dkx is recorded. By the arguments of the initial paragraph, the action of x on any dk can then be computed, and the action of K is by right multiplication, so the full action of F (or equivalently G) is available.

62.5 Gain Group Representation

In the representation described in section 62.4, computing the action of a generator x on a double coset named dk depends on finding the L_x-orbit representative of dk. The L_x orbits lying in $D = d^K$ correspond to the double cosets $M_d \backslash K/L_x$ and so to the orbits of M_d on the left cosets L_x.

The effect of this is that the program spends most of its time computing with the action of K on the left cosets of the various groups L_x. If this action can be represented in some more direct way, such as an action on points, tuples or sets, then there is a huge performance gain. The input format of the program is set up to reflect this. Each gain group L_x is specified by giving an action of K on some domain which is permutationally equivalent to the action of K on left cosets of L_x.

It sometimes happens that two generators x and y have identical, or conjugate, gain groups. The program does a considerable amount of pre-computation with each gain group, and builds some potentially large data structures, so it is sensible to combine these for identical or conjugate gain groups. To allow this, the gain groups are specified as one part of the input, and then another part specifies, for each generator, a reference to the gain group and possibly a conjugating element.

62.6 DCE Words

As indicated in section 62.4, the relators and subgroup generators are specified as elements of the free product, $K * F_X$ which is to say products of elements of K and generators from X (and their inverses). These are represented in GAP3 as DCE Words, created using the `DCEWord` function. This is called as `DCEWord(K, l)` where `l` is an element of `K`, a word in abstract generators or a list of these. DCE Words are in `GroupElements` and can be multiplied (when the groups K match), inverted, raised to powers and so forth.

Note that the abstract generators are used here simply as place-holders. Although, in general, creating abstract generators with `AbstractGenerator` rather than `FreeGroup` is a bad idea, it will not cause problems here. A new version of this package will be produced for GAP3 4 which will avoid this problem.

62.7 DCE Presentations

The input to the GAP3 Double Coset Enumerator is presented as a record. This has the following compulsory components.

`groupK` The group K, given as a GAP3 group. In general, it is best to represent K as a permutation group of low degree.

`gainGroups` This specifies the types (K-conjugacy classes) of gain groups L associated with the generators. It takes the form of a list of records, each with the following components:

- `dom` – A representative of a set on which K acts in the same way that it acts on the left cosets of L. If this is not given then $L = K$ and other fields are set accordingly.
- `op` – The operation of K on this set. This should be a GAP3 operation such as `OnPoints`. If `op` is not given, and `dom` is an integer then `op` defaults to `OnPoints`. If `op` is not given and `dom` is a set, then the `op` defaults to `OnSets`.

`gens` This field specifies the generators (the set X). It is a list of records, each with the fields:

- `name` – The abstract generator that will be used to denote this generator in the relations and subgroup generators.
- `invol` – A Boolean value indicating whether this generator should be considered as its own inverse. Default `false`.
- `inverse` – The 'name' of the inverse of this generator. This field is ignored if `invol` is present. If both `inverse` and `invol` are absent then a new generator will be created to be an inverse.

`wgg` – The index (in `gainGroups`) of the gain group of this generator (up to conjugacy).

`ggconj` – The gain group conjugator. The actual gain group of this generator will be that defined by entry `wgg` of the `gainGroups` list, conjugated by the element `ggconj` (of K). If this field is absent then it is taken to be the identity of K.

`action` – This specifies the isomorphism θ_x induced by x between L_x and $L_{x^{-1}}$. It can be `false`, indicating no action, an element of K, indicating action by conjugation, or it can be an explicit isomorphism. The default is `false`. If an explicit homomorphism is given and the the field `invol` is not present, then the field `inverse` must be present; that is, a generator inverse to x cannot be synthesized in this case.

`relators` The relations of the presentation, as a list of DCE Words. Certain additional fields may be added to the words (which are represented as records) to optimize the calculation. These are described below.

`subgens` The generators of H, as a list of DCE Words.

62.8 Examples of Double Coset Enumeration

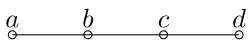
To save space and avoid clutter the examples are shown without the `gap>` and `>` prompts, as they might appear in an input file. For examples of DCE in operation see 62.17 and 62.19.

The Symmetric group of degree 5

It is well known that

$$G = S_5 = \langle\, a, b, c, d \quad | \quad a^2 = b^2 = c^2 = d^2 = (ab)^3 = (bc)^3 = (cd)^3 = (ac)^2 = (ad)^2 = (bd)^2 = 1 \,\rangle\,.$$

For brevity we denote this presentation by a Coxeter diagram:

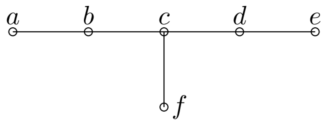


We let $K = \langle\, a, b \,\rangle \cong S_3$ and identify a with $(1,2)$ and b with $(2,3)$. Then G is generated by K and $X = \{c, d\}$. We can see from the presentation that $L_c = \langle\, a \,\rangle = \mathrm{Stab}_K(3)$, while $L_d = K$. We set $H = \langle\, a, b, c \,\rangle \cong S_4$ and obtain the following presentation:

```
gap> k := SymmetricGroup(3);
Group( (1,3), (2,3) )
gap> c := AbstractGenerator("c");;
gap> d := AbstractGenerator("d");;
gap> S5Pres := rec(
>    groupK := k,
>    gainGroups := [rec(),           # default to L=K
>                   rec(dom := 3)],  # default to action on points
>    gens := [rec(name := c, invol := true, wgg := 2),
>             rec(name := d, invol := true, wgg := 1)],
>    relators := [DCEWord(k,c*d)^3,DCEWord(k,[(2,3),c])^3],
>    subgens := [DCEWord(k,(1,2,3)), DCEWord(k,(1,2)), DCEWord(k,c)]);;
```

The Weyl Group of Type E_6

We consider another group given by a Coxeter presentation



This time we take $K = H = \langle a, b, c, d, e \rangle \cong S_6$ and obtain the presentation:

```
gap> k := SymmetricGroup(6);
Group( (1,6), (2,6), (3,6), (4,6), (5,6) )
gap> f := AbstractGenerator("f");;
gap> WE6Pres := rec (
>    groupK := k,
>    gainGroups := [rec(dom := [1,2,3])],  # action defaults to OnSets
>    gens := [rec(name := f,wgg := 1, invol := true)],
>    relators := [DCEWord(k,[(3,4),f])^3],
>    subgens := [DCEWord(k,(1,2,3,4,5,6)),DCEWord(k,(1,2))] );;
```

S_5 revisited

To illustrate other features of the program, we consider the presentation of S_5 again, but this time we choose $K = \langle b, c \rangle \cong S_3$ and identify b with $(1,2)$ and c with $(2,3)$. Now $L_a = \langle c \rangle = \mathrm{Stab}_K(1)$, while $L_d = \langle b \rangle = \mathrm{Stab}_K(3)$. These are two conjugate subgroups of K, so we can use the `ggconj` feature to combine the data structures for them.

We can present this as:

```
gap> k := SymmetricGroup(3);
Group( (1,3), (2,3) )
gap> a := AbstractGenerator("a");;
gap> d := AbstractGenerator("d");;
gap> b := (1,2);;
gap> c := (2,3);;
gap> S5PresA := rec (
>    groupK := k,
>    gainGroups := [rec(dom := 1)],
>    gens := [rec(name := a, invol := true, wgg := 1),
>             rec(name := d, invol := true, wgg := 1, ggconj := (1,3))],
>    relators := [DCEWord(k,[c,d])^3,DCEWord(k,[a,b])^3,
>                 DCEWord(k,[a,d])^2],
>    subgens := [DCEWord(k,(1,2,3)),DCEWord(k,(1,2)), DCEWord(k,a)] );;
```

The Harada-Norton Group

The almost-simple group $HN:2$ can be constructed as follows. Take the symmetric group S_{12} acting naturally on $\{1, \ldots, 12\}$ and let L be the stabiliser of $\{1, 2, 3, 4, 5, 6\}$. Then $L \cong S_6 \times S_6$. Extend S_{12} by adjoining an element a which normalizes L and acts on each factor S_6 by its outer automorphism. Impose the additional relations $a^2 = 1$ and $(a(6,7))^5 = 1$.

With $H = K$, this construction translates directly into DCE input:

```
gap> a := AbstractGenerator("a");;
gap> K := Group((1,2,3,4,5,6,7,8,9,10,11,12),(1,2));;
gap> L := Stabilizer(K,[1,2,3,4,5,6],OnSets);;
gap> f := GroupHomomorphismByImages( L,L,
>    [(1,5,4,3,2),(5,6),(12,8,9,10,11),(7,8)],
>    [(1,5,4,3,2),(1,4)(2,3)(5,6),(12,8,9,10,11),(12,9)(10,11)(7,8)] );;
gap> HNPres := rec(
>    groupK := K,
>    gainGroups := [ rec(dom := [1,2,3,4,5,6], op := OnSets)],
>    gens := [ rec( name := a, invol := true, wgg := 1, action := f)],
>    relators := [DCEWord(K,[a,(6,7)])^5],
>    strategy := rec(whichStrategy := "HLT", EC := [1140000]),
>    subgens := [(1,2,3,4,5,6,7,8,9,10,11,12),(1,2)]);;
```

A Non-permutation Example

The programs were written with the case of K a permutation group uppermost in the author's mind, however other representations are possible.

In this example, we represent the symmetric group S_4 as an AG-group in the Coxeter presentation of S_6. This example also demonstrates the explicit use of the action of K on left cosets of L_x, when no suitable action on points, sets or similar is available.

```
gap> k := AgGroup(SymmetricGroup(4));
Group( g1, g2, g3, g4 )
gap> a := PreImage(k.bijection,(1,2));
g1
gap> b := PreImage(k.bijection,(2,3));
g1*g2
gap> c := PreImage(k.bijection,(3,4));
g1*g4
gap> d := AbstractGenerator("d");;
gap> e := AbstractGenerator("e");;
gap> l := Subgroup(k,[a,b]);
Subgroup( Group( g1, g2, g3, g4 ), [ g1, g1*g2 ] )
gap> OurOp := function(cos,g)
>     return g^-1*cos;                   # note the inversion
> end;;
gap> Pres := rec (
>    groupK := k,
>    gainGroups := [rec(dom := k.identity*l, op := OurOp),rec()],
>    gens := [rec(name := d, invol := true, wgg := 1),
>             rec(name := e, invol := true, wgg := 2)],
>    relators := [DCEWord(k,d*e)^3,DCEWord(k,[c,d])^3],
>    subgens := [DCEWord(k,a),DCEWord(k,b), DCEWord(k,c),DCEWord(k,d)]);;
```

62.9 The DCE Universe

The various user functions described below operate on a record called a **DCE Universe**.

This is created by the function `DCESetup` (or by `DCE`, which calls `DCESetup`) and is then passed as first argument to all other DCE functions. The following fields are likely to be of most interest:

`K` The group K. For brevity this group is given the `name` "K".

`pres` The presentation from which this universe was created.

`isDCEUniverse` Always `true`.

`status` A string describing the status of the enumeration. Values include:

"in end game" – The program believes that the enumeration is almost complete and has shifted to a Felsch-like strategy to try and finish it.

"early-closed" – The table is closed, that is has no blank entries, but the program has not actually proved that the permutation representation described satisfies all the relations. The program will stop under these circumstances if the degree falls within a range set by the user (see 62.18

"running" – Enumeration is in progress.

"closed" – The enumeration has been completed.

"Setting up" – The data structures are still being initialized.

"Set up" – The data structures are initialized but computation has not yet started.

`degree` The number of single cosets represented by the current double coset table.

`dcct` The number of double cosets in the current table.

62.10 Informational Messages from DCE

`InfoDCE1`

`InfoDCE2`

`InfoDCE3`

`InfoDCE4`

`DCEInfoPrint`

The level of information printed by the programs can be controlled by setting the variables `InfoDCE1`, `InfoDCE2`, `InfoDCE3` and `InfoDCE4`. These can be (sensibly) set to either `DCEInfoPrint` or to `Ignore`. By default `InfoDCE1` is set to `DCEInfoPrint` and the rest to `Ignore`. Setting further variables to `DCEInfoPrint` produces more detailed comments. The higher numbered variables are intended mainly for debugging.

62.11 DCE

`DCE(`*pres*`)`

The basic command to run the double coset enumerator is `DCE`. This takes one argument, the presentation record in the format described above, and returns a DCE Universe of status "closed" or "early-closed". The exact details of operation are controlled by various fields in the input structure, as described in 62.18.

62.12 DCESetup

`DCESetup(`*pres*`)`

This function is called by `DCE` to initialize all the data structures needed. It returns a DCE Universe of status "Set up".

62.13 DCEPerm

`DCEPerm(`*universe*`,`*word*`)`

This function computes the permutation action of the DCEWord *word* on the single cosets described by *universe*. The status of *universe* should be "closed" or "early-closed". The first time this function (or `DCEPerms`) is called some large data structures are computed and stored in *universe*.

62.14 DCEPerms

`DCEPerms(`*universe*`)`

This function returns a list of permutations which generate the permutation group described by *universe*, which should have status "closed" or "early-closed". The permutations correspond to the generators X of the presentation (except any which are inverses of preceding generators) and then to the generators of K.

62.15 DCEWrite

`DCEWrite(`*universe*`,`*filename*`)`

This function writes selected information from the DCE Universe *universe* onto the file *filename* in a format suitable for recovery with `DCERead`.

62.16 DCERead

`DCERead(`*universe*`,`*filename*`)`

This function recovers the information written to file *filename* by `DCEWrite`. *universe* must be a DCE Universe of status "Set up", created from exactly the same presentation as was used to create the universe originally written to the file.

62.17 Example of DCE Functions

We take the first example presentation above, run it and demonstrate the above functions on the result.

```
gap> k := S5Pres.groupK;;
gap> c := S5Pres.gens[1].name;;
gap> d := S5Pres.gens[2].name;;
gap> u := DCE(S5Pres);
#I Set up generators and inverses
#I Set up column structure: 4 columns
```

```
#I Pre-processed relators
#I Done subgroup generators
#I Also done relators in subgroup
#I Pushing at weight 3
#I      1 double 1 single 1 blanks
#I 1 DCEWord(K,[c, d])^3
#I   1 cases
#I 1 DCEWord(K,[(2,3), c])^3
#I   1 cases
#I Pushing at weight 5
#I      3 double 5 single 1 blanks
#I 2 DCEWord(K,[c, d])^3
#I   1 cases
#I 2 DCEWord(K,[(2,3), c])^3
#I   1 cases
#I 3 DCEWord(K,[c, d])^3
#I   2 cases
#I 3 DCEWord(K,[(2,3), c])^3
#I   3 cases
#I Pushing at weight 101
#I      3 double 5 single 0 blanks
#I 1 DCEWord(K,[c, c])
#I   1 cases
#I 1 DCEWord(K,[d, d])
#I   1 cases
#I Pushing at weight 103
#I      3 double 5 single 0 blanks
#I 2 DCEWord(K,[c, c])
#I   1 cases
#I 2 DCEWord(K,[d, d])
#I   1 cases
#I 3 DCEWord(K,[c, c])
#I   2 cases
#I 3 DCEWord(K,[d, d])
#I   1 cases
<< Double coset table "No name" closed 3 double 5 single >>
gap> u.degree;
5
gap> u.status;
"closed"
gap> u.dcct;
3
gap> a1 := DCEWord(k,(1,2));
DCEWord(K,[(1,2)])
gap> b1 := DCEWord(k,(2,3));
DCEWord(K,[(2,3)])
gap> c1 := DCEWord(k,c);
DCEWord(K,[c])
```

```
gap> d1 := DCEWord(k,d);
DCEWord(K,[d])
gap> DCEPerm(u,a1);
#I Starting To Add Cosets
#I Done cosets, starting image
(4,5)
gap> DCEPerm(u,a1*c1*b1);
#I Done cosets, starting image
(2,4,5,3)
gap> DCEPerms(u);
#I Done cosets, starting image
#I Done cosets, starting image
#I Done cosets, starting image
#I Done cosets, starting image
[ (2,3), (1,2), (3,5), (3,4) ]
gap> DCEWrite(u,"s5.dct");
gap> u1 := DCESetup(S5Pres);
#I Set up generators and inverses
#I Set up column structure: 4 columns
#I Pre-processed relators
<< Double coset table "No name" Set up >>
gap> DCERead(u1,"s5.dct");
#I Read the file
gap> u1;
<< Double coset table "No name" closed 3 double 5 single >>
gap> DCEPerms(u1);
#I Starting To Add Cosets
#I Done cosets, starting image
#I Done cosets, starting image
#I Done cosets, starting image
#I Done cosets, starting image
[ (2,3), (1,2), (3,5), (3,4) ]
```

62.18 Strategies for Double Coset Enumeration

As with the Todd-Coxeter algorithm, the order of defining new (double) cosets and applying relations can make a huge difference to the performance of the algorithm. There is considerable scope for user control of the strategy followed by the DCE program. This is exercised by setting the `strategy` field in the presentation record (and less importantly by adding various fields to the relators). This field should be set to a record, for which various fields are meaningful. The most important is `whichStrategy`, which should take one of three values:

"HLT" A weighted Haselgrove-Leech-Trotter strategy. This is the default.

"Felsch" A pure Felsch strategy.

"Havas" A family of hybrid strategies, controlled by three parameters: `FF` which regulates the use of the preferred definition list to ensure that all definitions get made eventually (high values use the list more); HavN which is the number of double cosets that will be filled by definition before the relators are pushed from HavK double cosets.

When it completes successfully HLT is generally much the fastest strategy.

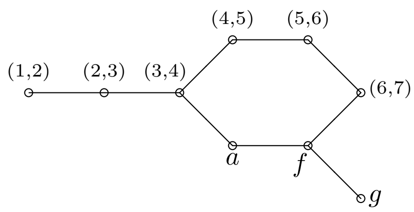
Apart from the fields FF, HavN and HavK, the other meaningful field in the strategy record is EC, which is the set (usually a range) of degrees at which early-closing is allowed. Even if you know the exact degree of the final representation it is worth-while allowing some "slack" so that the "end-game" strategy can come into play.

The "HLT" strategy can be fine-tuned by setting "weights" on the relators. Weights are integers, and a relator with higher weight will be used less than one with lower weight. This is done by adding a field weight to the relator record. The default weight is the base two logarithm of the length of the relator (after consecutive elements of K in the relator have been combined).

Finally, setting the insg field of a relator causes it to be used as a subgroup generator as well.

62.19 Example of Double Coset Enumeration Strategies

We look at a presentation for the sporadic group Fi_{22}, given by the Coxeter diagram:



with the additional relation $(f(4,5)(6,7)(3,4)(5,6)a)^4 = 1$ (the "hexagon" relation).

As indicated by the labels on the diagram we take $K = S_7$. The subgroup generated by all the nodes except the left-most has index 3510. An enumeration over that subgroup is coded as:

```
gap> k := SymmetricGroup(7);
Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) )
gap>
gap> aname := AbstractGenerator("a");; a := DCEWord(k,aname);
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[a])
gap> fname := AbstractGenerator("f");; f := DCEWord(k,fname);
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[f])
gap> gname := AbstractGenerator("g");; g := DCEWord(k,gname);
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[g])
gap>
gap> hexagon := (f*DCEWord(k,(4,5)*(6,7)*(3,4)*(5,6))*a)^4;
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7),
(6,7) ),[f, (3,4,6,7,5), a])^4
gap> hexagon.name := "hex";
"hex"
gap>
```

```
gap> rel1 := (a*DCEWord(k,(3,4)))^3;
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[a, (3,4)])^
3
gap> rel2 := (f*DCEWord(k,(6,7)))^3;
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[f, (6,7)])^
3
gap> rel3 := (a*g)^2;
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[a, g])^2
gap> rel4 := (f*g)^3;
DCEWord(Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ),[f, g])^3
gap>
gap> F22Pres := rec(
>    groupK := k,
>    gainGroups := [ rec(), rec(dom := 7, op := OnPoints),
>                    rec(dom := [1,2,3], op := OnSets)],
>    gens := [ rec( name := aname, invol := true, wgg := 3),
>              rec (name := fname, invol := true, wgg := 2),
>              rec (name := gname, invol := true, wgg := 1)],
>    relators := [rel1,rel2,rel4,rel3,hexagon],
>    subgens := [(2,3,4,5,6,7),(3,2),f,a,g] );;
```

HLT Strategy

As given, this presentation will use the default HLT strategy. On a SparcStation 10-41 this enumeration takes 60.8 CPU seconds and defines a total of 95 double cosets (for a final total of 24).

Since we know the correct index in this example, we can use early-closing, by setting

```
gap> F22Pres.strategy := rec(EC := [3510]);
rec(
  EC := [ 3510 ] )
gap> DCE(F22Pres);
#I Set up generators and inverses
#I Set up column structure: 43 columns
#I Pre-processed relators
#I Done subgroup generators
#I Also done relators in subgroup
#I Pushing at weight 3
#I      1 double 7 single 2 blanks
#I 1 DCEWord(K,[a, (3,4)])^3
#I   4 cases
...
```

The calculation proceeds identically until, after 40 seconds, it reaches a table with 3510 single cosets and only four blank entries. The program then changes strategies and attempts to fill the blanks as seen in the following piece of output:

```
...
#I 13 hex
#I   70 cases
```

```
#I 22 DCEWord(K,[a, (3,4)])^3
#I   39 cases
#I 22 DCEWord(K,[f, (6,7)])^3
#I   9 cases
#I 22 DCEWord(K,[f, g])^3
#I   3 cases
#I 22 DCEWord(K,[a, g])^2
#I   8 cases
#I 15 hex
#I   90 cases
#I Entering Pre-early closing 24 3510 4
#I 48 DCEWord(K,[a, (3,4)])^3
#I   29 cases
#I 48 DCEWord(K,[f, (6,7)])^3
#I   5 cases
<< Double coset table "No name" early-closed 24 double 3510 single >>
```

This succeeds after a further 2 seconds, producing a closed table. This method actually defines more double cosets (97), but is much faster.

We can cause the change of strategies to occur a little earlier by widening the range of acceptable indices. With:

```
gap> F22Pres.strategy := rec(EC := [3500..3600]);
rec(
  EC := [ 3500 .. 3600 ] )
gap> u := DCE(F22Pres);
#I Set up generators and inverses
#I Set up column structure: 43 columns
#I Pre-processed relators
#I Done subgroup generators
#I Also done relators in subgroup
#I Pushing at weight 3
#I      1 double 7 single 2 blanks
#I 1 DCEWord(K,[a, (3,4)])^3
#I   4 cases
...
```

With this option we see:

```
...
#I 13 hex
#I   70 cases
#I Entering Pre-early closing 24 3516 18
#I 22 DCEWord(K,[a, (3,4)])^3
#I   39 cases
#I 22 DCEWord(K,[f, (6,7)])^3
#I   9 cases
#I 22 DCEWord(K,[f, g])^3
#I   3 cases
#I 22 DCEWord(K,[a, g])^2
```

```
#I   8 cases
#I 22 hex
#I   130 cases
#I 22 DCEWord(K,[a, a])
#I   8 cases
#I 22 DCEWord(K,[f, f])
#I   3 cases
#I 22 DCEWord(K,[g, g])
#I   1 cases
#I 36 DCEWord(K,[a, (3,4)])^3
#I   39 cases
#I 36 DCEWord(K,[f, (6,7)])^3
#I   9 cases
#I 36 DCEWord(K,[f, g])^3
#I   3 cases
#I 36 DCEWord(K,[a, g])^2
#I   8 cases
#I 36 hex
#I   130 cases
<< Double coset table "No name" early-closed 24 double 3510 single >>
```

and a run time of about 37 seconds.

Apart from the early-closing criteria, we can tune the behaviour of the HLT algorithm by varying the relator weights. We can see the default weights by doing:

```
gap> List(u.relators,r->[r,r.weight]);
[ [ DCEWord(K,[a, (3,4)])^3, 2 ], [ DCEWord(K,[f, (6,7)])^3, 2 ],
  [ DCEWord(K,[f, g])^3, 2 ], [ DCEWord(K,[a, g])^2, 2 ], [ hex, 3 ],
  [ DCEWord(K,[a, a]), 100 ], [ DCEWord(K,[f, f]), 100 ],
  [ DCEWord(K,[g, g]), 100 ] ]
```

The relators with weight 100 are simply added automatically to ensure that the algorithm cannot terminate without closing the table.

We could emulate the unweighted HLT algorithm by setting `hexagon.weight:= 2;`

This produces significantly worse performance, as the long hexagon relation is pushed more often than necessary. On the other hand increasing its weight to 4 also produces worse performance than the default, because unnecessarily much of the infinite hyperbolic reflection group (defined by the other relations) is constructed.

Felsch Strategies

We can try this presentation with the Felsch strategy by simply setting:

```
F22Pres.strategy := rec(whichStrategy := "Felsch",EC := [3500..3600]);
```

Using this strategy the enumeration takes longer (92 seconds), but defines only 35 double cosets in total. The Felsch algorithm can often be improved by adding the longer relators as redundant subgroup generators. We can try this by setting `hexagon.insg := true;` but the improvement is very slight (to 91 seconds and 35 double cosets).

Hybrid strategy

We can access the hybrid methods be setting `F22Pres.strategy.whichStrategy := "Havas";` We first look at the preferred definition list alone, by setting

```
gap> strat := F22Pres.strategy;
rec(
  EC := [ 3500 .. 3600 ] )
gap> strat.FF := 5;
5
gap> strat.HavN := 100;
100
gap> strat.HavK := 0;
0
```

This turns out to be significantly worse than the simple Felsch algorithm, defining 56 double cosets and taking 145 seconds. Smaller values for `FF` produce performance closer to the simple Felsch.

By setting

```
gap> strat.FF := 1;
1
gap> strat.HavN := 5;
5
gap> strat.HavK := 2;
2
```

We can try a hybrid strategies (without the PDL). This runs in about 100 seconds, making 41 definitions.

62.20 Functions for Analyzing Double Coset Tables

The functions `DCEPerm` and `DCEPerms` have already been described, while elementary information (such as the numbers of single and double cosets) can be read directly from the DCE Universe produced by an enumeration. When the number of single cosets is large, however, as in the example of $HN:2$ above, `DCEPerm` requires an improbably large amount of space, so permutations cannot sensibly be obtained. However some analysis of the permutation representation is possible directly from the double coset table.

Specifically, functions exist to study the orbits of H, and compute their sizes and the collapsed adjacency matrices of the orbital graphs. The performance of these functions depends crucially on the size of the group $M = H \cap K$, which will always be the muddle group of the first double coset HK. When $M = K$, so that $K \leq H$, then each orbit of H is just a union of double cosets and the algorithms are fast, whereas when $M = 1$ there no benefit over extracting permutations.

62.21 DCEColAdj

`DCEColAdj(`*universe*`)`

This function computes the complete set of collapsed adjacency matrices (incidence matrices) for all the orbital graphs in the permutation action implied by *universe*, which must be a DCE Universe of status "closed" or "early-closed". For very large degrees, and/or if some of the subgroup generators are long words, this function can take infeasibly long, so some other functions are provided for partial calculations.

62.22 DCEHOrbits

`DCEHOrbits(`*universe*`)`

This function determines the orbits of H, as unions of orbits of $M = H \cap K$. Various additions are made to the data structures in *universe*, which are described in detail elsewhere. The most comprehensible field is `u.orbsizes` which gives the number of points (single cosets) in the orbits.

62.23 DCEColAdjSingle

`DCEColAdjSingle(`*universe*`,`*orbnum*`)`

This function determines the single collapsed adjacency matrix corresponding to orbital graph number *orbnum* (in the ordering of `<universe>.orbsizes`). This takes time roughly proportional to `<universe>.orbsizes[<orbnum>]`, so that extracting the adjacency matrices corresponding to small orbits in large representations is possible.

62.24 Example of DCEColAdj

We return to the hexagon presentation for Fi_{22}, and join it just as the double coset enumeration is finishing:

```
gap> InfoDCE1 := Ignore;
function (...) internal; end
gap> u := DCE(F22Pres);
<< Double coset table "No name" early-closed 24 double 3510 single >>
gap> InfoDCE1 := DCEInfoPrint;;
gap> DCEHOrbits(u);
#I Completed preliminaries, index of M is  7
#I Annotated table
#I Completed orbit  1  size  1
#I Completed orbit  2  size  2816
#I Completed orbit  3  size  693
gap> u.orbsizes;
[ 1, 2816, 693 ]
gap> DCEColAdj(u);
#I Added contribution from  1 part 1
#I Added contribution from  1 part 2
#I Added contribution from  2 part 1
#I Added contribution from  2 part 5
#I Added contribution from  3 part 1
#I Added contribution from  4 part 1
               ...
#I Added contribution from  70 part 1
#I Added contribution from  70 part 3
#I Added contribution from  70 part 4
#I Added contribution from  70 part 5
#I Added contribution from  70 part 7
```

```
#I Added contribution from  77 part 1
#I Added contribution from  77 part 5
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 0, 2816, 0 ], [ 1, 2248, 567 ], [ 0, 2304, 512 ] ],
  [ [ 0, 0, 693 ], [ 0, 567, 126 ], [ 1, 512, 180 ] ] ]
gap> DCEColAdjSingle(u,3);
#I Added contribution from  2 part 5
#I Added contribution from  6 part 5
#I Added contribution from  7 part 3
#I Added contribution from  11 part 2
#I Added contribution from  13 part 5
#I Added contribution from  15 part 4
#I Added contribution from  19 part 1
#I Added contribution from  20 part 5
#I Added contribution from  22 part 4
#I Added contribution from  23 part 4
#I Added contribution from  26 part 1
#I Added contribution from  36 part 3
#I Added contribution from  42 part 7
#I Added contribution from  44 part 7
#I Added contribution from  61 part 1
#I Added contribution from  65 part 7
#I Added contribution from  70 part 5
[ [ 0, 0, 693 ], [ 0, 567, 126 ], [ 1, 512, 180 ] ]
```

62.25 Double Coset Enumeration and Symmetric Presentations

R.T. Curtis has defined the notion of a symmetric presentation: given a group K, permuting a set S, we consider a generating set X in bijection with S, with conjugation by K permuting X as K permutes S. A symmetric presentation is such a set up, together with relations given in terms of the elements of K and T.

It is not hard to see that, at least when K is transitive on S, this is equivalent to the set up for double coset enumeration, with one generator t, and gain group equal to the point stabiliser in K of some $s_0 \in S$. The relations can be written in terms of K, t and conjugates of t by K, and so in terms of K and t.

62.26 SetupSymmetricPresentation

`SetupSymmetricPresentation(`K`,`*name* `[,` *base* `[,` *op*`]])`

The function `SetupSymmetricPresentation` implements the equivalence between presentations for DCE and Symmetric Presentations in the sense of Curtis. The argument K is the group acting, and *name* is an `AbstractGenerator` that will be used as t. The optional arguments *base* and *op* can be used to specify s_0 and the action of K on S. *base* defaults to 1 and *op* to `OnPoints`.

The function returns a record with two components:

`skeleton` a partial DCE Presentation. The fields `K`, `gainGroups` and `gens` are bound. Fields `relators` and `subgens` must still be added, and `name` and `strategy` may be added, before enumeration.

`makeGen` is a function which converts elements of `Orbit(K,base,op)` into DCE-Words for the corresponding symmetric Generators.

62.27 Examples of DCE and Symmetric Presentations

M_{12}

The following input gives a symmetric presentation of the Mathieu group M_{12}:

```
gap> t := AbstractGenerator("t");;
gap> K := Group((1,2,3,4,5),(1,2,3));
Group( (1,2,3,4,5), (1,2,3) )
gap> SGrec := SetupSymmetricPresentation(K,t);
rec(
  skeleton := rec(
      groupK := Group( (1,2,3,4,5), (1,2,3) ),
      gainGroups := [ rec(
              dom := 1,
              op := function (...) internal; end ) ],
      gens := [ rec(
              name := t,
              wgg := 1 ) ] ),
  makeGen := function ( pt ) ... end )
gap> t := SGrec.makeGen;
function ( pt ) ... end
gap> Pres := SGrec.skeleton;
rec(
  groupK := Group( (1,2,3,4,5), (1,2,3) ),
  gainGroups := [ rec(
          dom := 1,
          op := function (...) internal; end ) ],
  gens := [ rec(
          name := t,
          wgg := 1 ) ] )
gap> Pres.name := "M12 Symmetric";
"M12 Symmetric"
gap> Pres.strategy := rec(EC := [1000..3000]);
rec(
  EC := [ 1000 .. 3000 ] )
gap> Pres.relators := [t(1)^3,(t(1)/t(2))^2*DCEWord(K,(3,4,5))];
[ DCEWord(Group( (1,2,3,4,5), (1,2,3) ),[t])^3,
  DCEWord(Group( (1,2,3,4,5),
    (1,2,3) ),[t, (1,3,4,5,2), t^-1, (1,2,5,4,3), t, (1,3,4,5,2), t^-1\
, (1,2,5,4,3), (3,4,5)]) ]
gap> Pres.subgens := [DCEWord(K,(1,2,3,4,5)),DCEWord(K,(1,2,3)),
>                   (DCEWord(K,(1,2,3,4,5))*t(1))^8];
```

```
[ DCEWord(Group( (1,2,3,4,5), (1,2,3) ),[(1,2,3,4,5)]),
  DCEWord(Group( (1,2,3,4,5), (1,2,3) ),[(1,2,3)]),
  DCEWord(Group( (1,2,3,4,5), (1,2,3) ),[(1,2,3,4,5), t])^8 ]
gap> Pres.relators[1].weight := 2;;   # default weight is too low
```

DCE enumerates this presentation in a few seconds.

```
gap> InfoDCE1 := Ignore;
function (...) internal; end
gap> u := DCE(Pres);
<< Double coset table "M12 Symmetric" early-closed 47 double
1584 single >>
gap> time;
5400
```

$He:2$

The following is a presentation of $He:2$ generated by 180 symmetric generators of order 7 permuted by $3S_7 \times 2$. This is really 30 generators permuted monomially, but we don't have monomial groups in GAP.

The following can be placed in an input file `he2.g`.

```
#
# The group K we want is 3S7 x 2. We make this from a handy
# representation of 3S7
#
DoubleP := function(p,n)
    local l;
    l := OnTuples([1..n],p);
    Append(l,l+n);
    return PermList(l);
end;

Swap := function(n)
    return PermList(Concatenation([n+1..2*n],[1..n]));
end;

K := Group(
    DoubleP((1, 2)( 3, 5)( 4, 7)( 6,10)( 8,12)( 9,14)(11,17)(13,20)
        (15,23)(16,25)(18,28)(19,30)(21,33)(22,35)(24,37)(26,40)(27,41)
        (29,44)(31,47)(32,49)(34,51)(36,54)(38,57)(39,46)(42,61)(43,63)
        (45,66)(48,53)(50,70)(52,60)(55,73)(56,65)(58,76)(59,78)(62,75)
        (64,80)(67,84)(68,74)(69,77)(71,85)(72,86)(79,89)(81,88)(82,87)
        (83,90),90),
    DoubleP(( 1, 3, 6)(44,65,49)
        (2,4,8,13,21,34,52,10,16,26,28,43,64,82,5,9,15,24,38,58,77)
        (7,11,18,29,45,67,63,14,22,20,32,50,61,33,25,39,37,56,75,86,57)
        (12,19,31,48,69,51,71,23,36,55,74,87,76,88,40,59,79,41,60,80,90)
        (17,27,42,62,81,47,30,46,68,84,70,85,89,78,35,53,72,66,83,73,54)
        ,90),Swap(90) );
```

```
#
# Now lets get the generators we want
#
x := DCEWord(K,K.1);
y := DCEWord(K,K.2);
a := DCEWord(K,K.3);
#
# And the name for our generator outside K
#
t := AbstractGenerator("t");
#
# Now we can specify our setup
#
SGrec := SetupSymmetricPresentation(K,t);
SG := SGrec.makeGen;
Pres := SGrec.skeleton;
#
# We still have to put some fields in the presentation
#
Pres.name := "He:2 Symmetric";
Pres.relators := [
    SG(1)^7,(SG(1)* SG(2))^2,
    SG(1)^2 / SG(3),
    y^-7 / (SG(1)^-1*SG(2)^-2*SG(1)^2*SG(2)),
    y^9 /  Comm(SG(1),SG(65)),
    SG(1)*SG(91),
    DCEWord(K,DoubleP((1,2)(3,5)(4,76)(6,10)(7,58)(8,12)(9,80)(11,70)
            (13,20)(14,64)(15,23)(16,51)(17,50)(18,28)(19,42)(21,87)
            (22,62)(24,37)(25,34)(26,40)(27,32)(29,68)(30,61)(31,85)
            (33,82)(35,75)(36,72)(38,60)(39,66)(41,49)(43,69)(44,74)
            (45,46)(47,71)(48,65)(52,57)(53,56)(54,86)(55,81)(59,84)
            (63,77)(67,78)(73,88)(79,83)(89,90),90)) /
    (SG(1)*SG(2)^2*SG(1)^2*SG(2)) ];
Pres.subgens := [t,x,x^(y^3)*x^(y^-1*x*y^-2),
                    Comm(x,y^-1*x*y^-1),Comm(x,y*x*y^2),a];
Pres.strategy := rec(EC := [8000..12000]);
```

We can run this example quietly:

```
gap> Read("he2.g");
gap> InfoDCE1 := Ignore;
function (...) internal; end
gap> u := DCE(Pres);
<< Double coset table "He:2 Symmetric" early-closed 9 double
8330 single >>
gap> time;
126716
```

Chapter 63

GLISSANDO

GLISSANDO (version 1.0) is a share library package that implements a GAP3 library of small semigroups and near-rings. The library files can be systematically searched for near-rings and semigroups with certain properties.

The GLISSANDO package (version 1.0) was written by

Christof Nöbauer
Institut für Mathematik
Johannes Kepler Universität Linz
4040 Linz, Austria

e-mail noebsi@bruckner.stoch.uni-linz.ac.at

and supported by the

Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*, Project P11486-TEC.

63.1 Installing the Glissando Package

The GLISSANDO package is completely written in the GAP3 language, it does not require any additional programs and/or compilations. It will run on any computer that runs GAP3. To access GLISSANDO, use `RequirePackage( "gliss" );` (see 57.1).

63.2 Transformations

A **transformation** is a mapping with equal source and range, say X. For example, X may be a set or a group. A transformation on X then acts on X by transforming each element of X into (precisely one) element of X.

Note that a transformation is just a special case of a mapping. So all GAP3 functions that work for mappings will also work for transformations.

For the following, it is important to keep in mind that in GAP3 sets are represented by **sorted** lists without holes and duplicates. Throughout this section, let X be a set or a group with n elements. A transformation on X is uniquely determined by a list of length n without holes and with entries which are integers between 1 and n.

For example, for the set `X := [1,2,3]`, the list `[1,1,2]` determines the transformation on X which transforms 1 into 1, 2 into 1, and 3 into 2.

Analogously, for the cyclic group of order 3: `C3`, with (the uniquely ordered) set of elements `[(),(1,2,3),(1,3,2)]`, the list `[2,3,3]` determines the transformation on `C3` which transforms `()` into `(1,2,3)`, `(1,2,3)` into `(1,3,2)`, and `(1,3,2)` into `(1,3,2)`.

Such a list which on a given set or group uniquely determines a transformation will be called **transformation list** (short **tfl**).

Transformations are created by the constructor functions `Transformation` or `AsTransformation` and they are represented by records that contain all the information about the transformations.

63.3 Transformation

`Transformation(` *obj*`,` *tfl* `)`

The constructor function `Transformation` returns the transformation determined by the transformation list *tfl* on *obj* where *obj* must be a group or a set.

```
gap> t1:=Transformation([1..3],[1,1,2]);
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
gap> g:=Group((1,2),(3,4));
Group( (1,2), (3,4) )
gap> gt := Transformation(g,[1,1,2,5]);
Error, Usage: Transformation( <obj>, <tfl> ) where <obj> must be a set
or a group and <tfl> must be a valid transformation list for <obj> in
Transformation( g, [ 1, 1, 2, 5 ] ) called from
main loop
brk>
gap> gt := Transformation( g, [4,2,2,1] );
Transformation( Group( (1,2), (3,4) ), [ 4, 2, 2, 1 ] )
```

63.4 AsTransformation

`AsTransformation(` *map* `)`

The constructor function `AsTransformation` returns the mapping *map* as transformation. Of course, this function can only be applied to mappings with equal source and range, otherwise an error will be signaled.

```
gap> s3:=Group((1,2),(1,2,3));
Group( (1,2), (1,2,3) )
gap> i:=InnerAutomorphism(s3,(2,3));
InnerAutomorphism( Group( (1,2), (1,2,3) ), (2,3) )
gap> AsTransformation(i);
Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 6, 5, 4, 3 ] )
```

63.5 IsTransformation

`IsTransformation(` *obj* `)`

`IsTransformation` returns `true` if the object *obj* is a transformation and `false` otherwise.

```
gap> IsTransformation( [1,1,2] );
false                              # a list is not a transformation
gap> IsTransformation( (1,2,3) );
false                              # a permutation is not a transformation
gap> IsTransformation( t1 );
true
```

63.6 IsSetTransformation

`IsSetTransformation(` *obj* `)`

`IsSetTransformation` returns `true` if the object *obj* is a set transformation and `false` otherwise.

```
gap> IsSetTransformation( t1 );
true
gap> g:= Group((1,2),(3,4));
Group( (1,2), (3,4) )
gap> gt:=Transformation(g,[4,2,2,1]);
[ 4, 2, 2, 1 ]
gap> IsSetTransformation( gt );
false
```

63.7 IsGroupTransformation

`IsGroupTransformation(` *obj* `)`

`IsGroupTransformation` returns `true` if the object *obj* is a group transformation and `false` otherwise.

```
gap> IsGroupTransformation( t1 );
false
gap> IsGroupTransformation( gt );
true
```

Note that transformations are defined to be either a set transformation or a group transformation.

63.8 IdentityTransformation

`IdentityTransformation(` *obj* `)`

`IdentityTransformation` is the counterpart to the GAP3 standard library function `IdentityMapping`. It returns the identity transformation on *obj* where *obj* must be a group or a set.

```
gap> IdentityTransformation( [1..3] );
Transformation( [ 1, 2, 3 ], [ 1, 2, 3 ] )
gap> IdentityTransformation( s3 );
Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 3, 4, 5, 6 ] )
```

63.9 Kernel for transformations

`Kernel( t )`

For a transformation t on X, the **kernel** of t is defined as an equivalence relation $Kernel(t)$ as: $\forall x, y \in X:\ (x, y) \in Kernel(t)$ iff $t(x) = t(y)$.

`Kernel` returns the kernel of the transformation t as a list `l` of lists where each sublist of `l` represents an equivalence class of the equivalence relation *Kernel(t)*.

```
gap> t:=Transformation( [1..5], [2,3,2,4,4] );
Transformation( [ 1, 2, 3, 4, 5 ], [ 2, 3, 2, 4, 4 ] )
gap> Kernel( t );
[ [ 1, 3 ], [ 2 ], [ 4, 5 ] ]
```

63.10 Rank for transformations

`Rank( t )`

For a transformation t on X, the **rank** of t is defined as the size of the image of t, i.e. $|\ \{t(x)\ \mid\ x \in X\}\ |$, or, in GAP3 language: `Length( Image( t ) )`.

`Rank` returns the rank of the transformation t.

```
gap> t1;
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
gap> Rank( t1 );
2
gap>
gap> gt;
Transformation( Group( (1,2), (3,4) ), [ 4, 2, 2, 1 ] )
gap> Rank(gt);
3
```

63.11 Operations for transformations

t1 `*` *t2*

The product operator * returns the transformation which is obtained from the transformations *t1* and *t2*, by composition of *t1* and *t2* (i.e. performing *t2* **after** *t1*). This function works for both set transformations as well as group transformations.

```
gap> t1:=Transformation( [1..3], [1,1,2] );
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
gap> t2:=Transformation( [1..3], [2,3,3] );
Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] )
gap> t1*t2;
Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] )
gap> t2*t1;
Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] )
```

t1 `+` *t2*

The add operator `+` returns the group transformation which is obtained from the group transformations *t1* and *t2* by pointwise addition of *t1* and *t2*. (Note that in this context addition means performing the GAP3 operation `p * q` for the corresponding permutations `p` and `q`).

t1 `-` *t2*

The subtract operator `-` returns the group transformation which is obtained from the group transformations *t1* and *t2* by pointwise subtraction of *t1* and *t2*. (Note that in this context subtraction means performing the GAP3 operation `p * q^-1` for the corresponding permutations `p` and `q`).

Of course, those two functions `+` and `-` work only for group transformations.

```
gap> g:=Group( (1,2,3) );
Group( (1,2,3) )
gap> gt1:=Transformation( g, [2,3,3] );
Transformation( Group( (1,2,3) ), [ 2, 3, 3 ] )
gap> gt2:=Transformation( g, [1,3,2] );
Transformation( Group( (1,2,3) ), [ 1, 3, 2 ] )
gap> gt1+gt2;
Transformation( Group( (1,2,3) ), [ 2, 2, 1 ] )
gap> gt1-gt2;
Transformation( Group( (1,2,3) ), [ 2, 1, 2 ] )
```

63.12 DisplayTransformation

`DisplayTransformation(` *t* `)`

`DisplayTransformation` nicely displays a transformation *t*.

```
gap> t:=Transformation( [1..5], [3,3,2,1,4] );
Transformation( [ 1, 2, 3, 4, 5 ], [ 3, 3, 2, 1, 4 ] )
gap> DisplayTransformation( t );
Transformation on [ 1, 2, 3, 4, 5 ]:
  1 -> 3
  2 -> 3
  3 -> 2
  4 -> 1
  5 -> 4
gap>
```

63.13 Transformation records

As almost all objects in GAP3, transformations, too, are representend by records. Such a transformation record has the following components:

`isGeneralMapping`
: this is always `true`, since in particular, any transformation is a general mapping.

`domain`
: the entry of this record field is `Mappings`.

`isMapping`
: this is always `true` since a transformation is in particular a single valued mapping.

`isTransformation`
: always `true` for a transformation.

`isSetTransformation`
: this exists and is set to `true` for set transformations exclusively.

`isGroupTransformation`, `isGroupElement`
: these two exist and are set to `true` for group transformations exclusively.

`elements`
: this record field holds a list of the elements of the source.

`source`, `range`
: both entries contain the same set in case of a set transformation, resp. the same group in case of a group transformation.

`tfl`
: this contains the transformation list which uniquely determines the transformation.

`operations`
: the operations record of the transformation. E.g. * or `=`, etc. can be found here.

`image`, `rank`, `ker`
: these are bound and contain image, rank, kernel in case they have already been computed for the transformation.

63.14 Transformation Semigroups

Having established transformations and being able to perform the associative operation **composition** (which in GAP3 is denoted as `*` with them, the next step is to consider **transformation semigroups**.

All functions described in this section are intended for **finite** transformation semigroups, in particular transformation semigroups on a finite set or group X. A transformation semigroup is created by the constructor function `TransformationSemigroup` and it is represented by a record that contains all the information about the transformation semigroup.

63.15 TransformationSemigroup

`TransformationSemigroup(` t_1`, ...,` t_n `)`
`TransformationSemigroup( [` t_1`, ...,` t_n `] )`

When called in this form, the constructor function `TransformationSemigroup` returns the transformation semigroup generated by the transformations t_1, ..., t_n. There is another way to call this function:

`TransformationSemigroup(` n `)`

If the argument is a positive integer n, `TransformationSemigroup` returns the semigroup of all transformations on the set $\{1, 2, \ldots, n\}$.

```
gap> t1 := Transformation( [1..3], [1,1,2] );
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
gap> t2 := Transformation( [1..3], [2,3,3] );
Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] )
gap> s:=TransformationSemigroup( t1, t2 );
```

```
TransformationSemigroup( Transformation( [ 1, 2, 3 ],
[ 1, 1, 2 ], Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) )
gap> s27 := TransformationSemigroup( 3 );
TransformationSemigroup( Transformation( [ 1, 2, 3 ],
[ 1, 1, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 1, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 1, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 2, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 2, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 2, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 3, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 3, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 1, 3, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 1, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 1, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 1, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 2, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 2, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 2, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 3, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 3, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 2, 3, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 1, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 1, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 1, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 2, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 2, 2 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 2, 3 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 3, 1 ] ), Transformation( [ 1, 2, 3 ],
[ 3, 3, 2 ] ), Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) )
```

63.16 IsSemigroup

`IsSemigroup( `*obj*` )`

`IsSemigroup` returns `true` if the object *obj* is a semigroup and `false` otherwise. This function simply checks whether the record component *obj*`.isSemigroup` is bound and is `true`.

```
gap> IsSemigroup( t1 );
false                          # a transformation is not a semigroup
gap> IsSemigroup( Group( (1,2,3) ) );
false                          # a group is not a semigroup
gap> IsSemigroup( s27 );
true
```

63.17 IsTransformationSemigroup

`IsTransformationSemigroup( `*obj*` )`

`IsTransformationSemigroup` returns `true` if the object *obj* is a transformation semigroup and `false` otherwise.

```
gap> IsTransformationSemigroup( s27 );
true
```

63.18 Elements for semigroups

`Elements(` *sg* `)`

`Elements` computes the elements of the semigroup *sg*. Note: the GAP3 standard library dispatcher function `Elements` calls the function *sg*`.operations.Elements` which performs a simple closure algorithm.

```
gap> t1 := Transformation( [1..3], [1,1,2] );
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
gap> t2 := Transformation( [1..3], [2,3,3] );
Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] )
gap> s := TransformationSemigroup( t1, t2 );
TransformationSemigroup( Transformation( [ 1, 2, 3 ],
[ 1, 1, 2 ] ), Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) )
gap> Elements( s );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ]
```

63.19 Size for semigroups

`Size(` *sg* `)`

`Size` returns the number of elements in *sg*.

```
gap> Size( s );
7
```

63.20 DisplayCayleyTable for semigroups

`DisplayCayleyTable(` *sg* `)`

`DisplayCayleyTable` prints the Cayley table of the semigroup *sg*. Note: The dispatcher function `DisplayCayleyTable` calls the function *sg*`.operations.DisplayTable` which performs the actual printing. `DisplayCayleyTable` has no return value.

```
gap> DisplayCayleyTable( s );
Let:
s0 := Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] )
s1 := Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
s2 := Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] )
```

```
s3 := Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] )
s4 := Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] )
s5 := Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] )
s6 := Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] )


  *  | s0 s1 s2 s3 s4 s5 s6
 --------------------------
  s0 | s0 s0 s0 s3 s3 s3 s6
  s1 | s0 s0 s1 s3 s3 s4 s6
  s2 | s0 s0 s2 s3 s3 s5 s6
  s3 | s0 s0 s3 s3 s3 s6 s6
  s4 | s0 s1 s3 s3 s4 s6 s6
  s5 | s0 s2 s3 s3 s5 s6 s6
  s6 | s0 s3 s3 s3 s6 s6 s6
```

63.21 IdempotentElements for semigroups

`IdempotentElements(` *sg* `)`

An element i of a semigroup $(S, \cdot)$ is called an **idempotent** (element) iff $i \cdot i = i$.

The function `IdempotentElements` returns a list of those elements of the semigroup *sg* that are idempotent. (Note that for a finite semigroup this can never be the empty list).

```
gap> IdempotentElements( s );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ]
```

63.22 IsCommutative for semigroups

`IsCommutative(` *sg* `)`

A semigroup $(S, \cdot)$ is called **commutative** if $\forall a, b \in S : \; a \cdot b = b \cdot a$.

The function `IsCommutative` returns the according value `true` or `false` for a semigroup *sg*.

```
gap> IsCommutative( s );
false
```

63.23 Identity for semigroups

`Identity(` *sg* `)`

An element i of a semigroup $(S, \cdot)$ is called an **identity** iff $\forall s \in S : \; s \cdot i = i \cdot s = s$. Since for two identities, i, j: $i = i \cdot j = j$, an identity is unique if it exists.

The function `Identity` returns a list containing as single entry the identity of the semigroup *sg* if it exists or the empty list `[ ]` otherwise.

```
gap> Identity( s );
```

```
[  ]
gap> tr1 := Transformation( [1..3], [1,1,1] );
Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] )
gap> tr2 := Transformation( [1..3], [1,2,2] );
Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] )
gap> sg := TransformationSemigroup( tr1, tr2 );
TransformationSemigroup( Transformation( [ 1, 2, 3 ],
[ 1, 1, 1 ] ), Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ) )
gap> Elements( sg );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ) ]
gap> Identity( sg );
[ Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ) ]
```

The last example shows that the identity element of a transformation semigroup on a set X needs not necessarily be the identity transformation on X.

63.24 SmallestIdeal

`SmallestIdeal(` *sg* `)`

A subset I of a semigroup $(S, \cdot)$ is defined as an **ideal** of S if $\forall i \in I, s \in S: \ i{\cdot}s \in I \ \& \ s{\cdot}i \in I$. An ideal I is called **minimal**, if for any ideal J, $J \subseteq I$ implies $J = I$. If a minimal ideal exists, then it is unique and therefore the **smallest** ideal of S.

The function `SmallestIdeal` returns the smallest ideal of the transformation semigroup *sg*. Note that for a finite semigroup the smallest ideal always exists. (Which is not necessarily true for an arbitrary semigroup).

```
gap> SmallestIdeal( s );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ]
```

63.25 IsSimple for semigroups

`IsSimple(` *sg* `)`

A semigroup S is called **simple** if it has no honest ideals, i.e. in case that S is finite the smallest ideal of S equals S itself.

The GAP3 standard library dispatcher function `IsSimple` calls the function `sg.operations.IsSimple` which checks if the semigroup *sg* equals its smallest ideal and if so, returns `true` and otherwise `false`.

```
gap> IsSimple( s );
false
gap> c3 := TransformationSemigroup( Transformation( [1..3],
>                                   [2,3,1] ) );
TransformationSemigroup( Transformation( [ 1, 2, 3 ], [ 2, 3, 1 ] ) )
gap> IsSimple( c3 );
true
```

63.26 Green

`Green(` *sg*`,` *string* `)`

Let $(S, \cdot)$ be a semigroup and $a \in S$. The set $a \cdot S^1 := a \cdot S \cup \{a\}$ is called the principal right ideal generated by a. Analogously, $S^1 \cdot a := S \cdot a \cup \{a\}$ is called the principal left ideal generated by a and $S^1 \cdot a \cdot S^1 := S \cdot a \cdot S \cup S \cdot a \cup a \cdot S \cup \{a\}$ is called the principal ideal generated by a.

Now, Green's equivalence relation $\mathcal{L}$ on S is defined as: $(a, b) \in \mathcal{L} :\Leftrightarrow\ S^1 \cdot a = S^1 \cdot b$ i.e. a and b generate the same principal left ideal. Similarly: $(a, b) \in \mathcal{R} :\Leftrightarrow\ a \cdot S^1 = b \cdot S^1$ i.e. a and b generate the same principal right ideal and $(a, b) \in \mathcal{J} :\Leftrightarrow\ S^1 \cdot a \cdot S^1 = S^1 \cdot b \cdot S^1$ i.e. a and b generate the same principal ideal. $\mathcal{H}$ is defined as the intersection of $\mathcal{L}$ and $\mathcal{R}$ and $\mathcal{D}$ is defined as the join of $\mathcal{L}$ and $\mathcal{R}$.

In a finite semigroup, $\mathcal{D} = \mathcal{J}$.

The arguments of the function `Green` are a finite transformation semigroup *sg* and a one character string *string* where *string* must be one of the following: `"L"`, `"R"`, `"D"`, `"J"`, `"H"`. The return value of `Green` is a list of lists of elements of *sg* representing the equivalence classes of the according Green's relation.

```
gap> s;
TransformationSemigroup( Transformation( [ 1, 2, 3 ],
[ 1, 1, 2 ] ), Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) )
gap> Elements( s );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ]
gap> Green( s, "L" );
[ [ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) ] ]
gap> Green( s, "R" );
[ [ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ] ]
gap> Green( s, "H" );
[ [ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ) ],
```

```
  [ Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ] ]
gap> Green( s, "D" );
[ [ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ],
  [ Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
      Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ) ] ]
```

63.27 Rank for semigroups

`Rank(` *sg* `)`

The **rank** of a transformation semigroup S is defined as the minimal rank of the elements of S, i.e. $\min\{rank(s) \mid s \in S\}$.

The function `Rank` returns the rank of the semigroup *sg*.

```
gap> Rank( s );
1
gap> c3;
TransformationSemigroup( Transformation( [ 1, 2, 3 ], [ 2, 3, 1 ] ) )
gap> Rank( c3 );
3
```

63.28 LibrarySemigroup

`LibrarySemigroup(` *size*`,` *num* `)`

The semigroup library contains all semigroups of sizes 1 up to 5, classified into classes of isomorphic semigroups. `LibrarySemigroup` retrieves a representative of an isomorphism class from the semigroup library and returns it as a transformation semigroup. The parameters of `LibrarySemigroup` are two positive integers: *size* must be in $\{1, 2, 3, 4, 5\}$ and indicates the size of the semigroup to be retrieved, *num* indicates the number of an isomorphism class.

```
gap> ls := LibrarySemigroup( 4, 123 );
TransformationSemigroup( Transformation( [ 1, 2, 3, 4 ],
[ 1, 1, 3, 3 ] ), Transformation( [ 1, 2, 3, 4 ],
[ 1, 2, 3, 4 ] ), Transformation( [ 1, 2, 3, 4 ],
[ 1, 3, 3, 1 ] ), Transformation( [ 1, 2, 3, 4 ], [ 1, 4, 3, 2 ] ) )
gap> DisplayCayleyTable( ls );
Let:
s0 := Transformation( [ 1, 2, 3, 4 ], [ 1, 1, 3, 3 ] )
s1 := Transformation( [ 1, 2, 3, 4 ], [ 1, 2, 3, 4 ] )
s2 := Transformation( [ 1, 2, 3, 4 ], [ 1, 3, 3, 1 ] )
```

```
s3 := Transformation( [ 1, 2, 3, 4 ], [ 1, 4, 3, 2 ] )


  *  |  s0 s1 s2 s3
 ------------------
  s0 | s0 s0 s0 s0
  s1 | s0 s1 s2 s3
  s2 | s2 s2 s2 s2
  s3 | s2 s3 s0 s1
```

In dependence of *size*, *num* must be one of the following:

size	*num*
1	$1 \leq num \leq 1$
2	$1 \leq num \leq 5$
3	$1 \leq num \leq 24$
4	$1 \leq num \leq 188$
5	$1 \leq num \leq 1915$

63.29 Transformation semigroup records

Transformation Semigroups are implemented as records. Such a transformation semigroup record has the following components:

`isDomain`, `isSemigroup`
: these two are always `true` for a transformation semigroup.

`isTransformationSemigroup`
: this is bound and `true` only for transformation semigroups.

`generators`
: this holds the set of generators of a transformation semigroup.

`multiplication`
: this record field contains a function that represents the binary operation of the semigroup that can be performed on the elements of the semigroup. For transformation semigroups this equals of course, composition. Example:

```
gap> elms := Elements( s );
[ Transformation( [ 1, 2, 3 ], [ 1, 1, 1 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 1, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 2, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 2, 3, 3 ] ),
  Transformation( [ 1, 2, 3 ], [ 3, 3, 3 ] ) ]
gap> s.multiplication( elms[5], elms[2] );
Transformation( [ 1, 2, 3 ], [ 1, 1, 2 ] )
```

`operations`
: this is the operations record of the semigroup.

`size`, `elements`, `rank`, `smallestIdeal`, `IsFinite`, `identity`
these entries become bound if the according functions have been performed on the semigroup.

`GreenL`, `GreenR`, `GreenD`, `GreenJ`, `GreenH`
these are entries according to calls of the function `Green` with the corresponding parameters.

63.30 Near-rings

In section 77 we introduced transformations on sets and groups. We used set transformations together with composition `*` to construct transformation semigroups in section 63.14. In section 77 we also introduced the operation of pointwise addition `+` for group transformations. Now we are able to use these group transformations together with pointwise addition `+` and composition `*` to construct (right) near-rings.

A **(right) near-ring** is a nonempty set N together with two binary operations on N, $+$ and $\cdot$ s.t. $(N,+)$ is a group, $(N,\cdot)$ is a semigroup, and $\cdot$ is right distributive over $+$, i.e. $\forall n_1, n_2, n_3 \in N: \ (n_1+n_2)\cdot n_3 = n_1 \cdot n_3 + n_2 \cdot n_3$.

Here we have to make a **remark**: we let transformations act from the right; yet in order to get a right transformation near-ring transformations must act from the left, hence we define a near-ring multiplication $\cdot$ of two transformations, t_1, t_2 as $t_1 \cdot t_2 := t_2 * t_1$.

There are three possibilities to get a near-ring in GAP3: the constructor function `Nearring` can be used in two different ways or a near-ring can be extracted from the near-rings library by using the function `LibraryNearring`. All functions described here were programmed for permutation groups and they also work fine with them; other types of groups (such as AG groups) are not supported.

Near-rings are represented by records that contain the necessary information to identify them and to do computations with them.

63.31 IsNrMultiplication

`IsNrMultiplication(` *G*`,` *mul* `)`

The arguments of the function `IsNrMultiplication` are a permutation group G and a GAP3 function *mul* which has two arguments `x` and `y` which must both be elements of the group G and returns an element `z` of G s.t. *mul* defines a binary operation on G.

`IsNrMultiplication` returns `true` (`false`) if *mul* is (is not) a near-ring multiplication on G i.e. it checks whether it is well-defined, associative and right distributive over the group operation of G.

```
gap> g := Group( (1,2), (1,2,3) );
Group( (1,2), (1,2,3) )
gap> mul_r := function(x,y) return x; end;
function ( x, y ) ... end
gap> IsNrMultiplication( g, mul_r );
true
gap> mul_l := function(x,y) return y; end;
function ( x, y ) ... end
```

```
gap> IsNrMultiplication( g, mul_l );
specified multiplication is not right distributive.
false
```

63.32 Nearring

`Nearring(` G, *mul* `)`

In this form the constructor function `Nearring` returns the near-ring defined by the permutation group G and the near-ring multiplication *mul*. (For a detailed explanation of *mul* see 63.31). `Nearring` calls `IsNrMultiplication` in order to make sure that *mul* is really a near-ring multiplication.

```
gap> g := Group( (1,2,3) );
Group( (1,2,3) )
gap> mul_r := function(x,y) return x; end;
function ( x, y ) ... end
gap> n := Nearring( g, mul_r );
Nearring( Group( (1,2,3) ), function ( x, y )
    return x;
end )
gap> DisplayCayleyTable( n );
Let:
n0 := ()
n1 := (1,2,3)
n2 := (1,3,2)

  +  | n0 n1 n2
 ---------------
  n0 | n0 n1 n2
  n1 | n1 n2 n0
  n2 | n2 n0 n1

  *  | n0 n1 n2
 ---------------
  n0 | n0 n0 n0
  n1 | n1 n1 n1
  n2 | n2 n2 n2
```

`Nearring(` t_1, ..., t_n `)`
`Nearring( [`t_1, ..., t_n`] )`

In this form the constructor function `Nearring` returns the near-ring generated by the group transformations $t_1, \ldots, t_n$. All of them must be transformations on the same permutation group.

Note that `Nearring` allows also a list of group transformations as argument, which makes it possible to call `Nearring` e.g. with a list of endomorphisms generated by the function `Endomorphisms` (see 63.71), which for a group G allows to compute $E(G)$; `Nearring` called with the list of all inner automorphisms of a group G would return $I(G)$.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
gap> n := Nearring( t );
Nearring( Transformation( Group( (1,2) ), [ 2, 1 ] ) )
gap> DisplayCayleyTable( n );
Let:
n0 := Transformation( Group( (1,2) ), [ 1, 1 ] )
n1 := Transformation( Group( (1,2) ), [ 1, 2 ] )
n2 := Transformation( Group( (1,2) ), [ 2, 1 ] )
n3 := Transformation( Group( (1,2) ), [ 2, 2 ] )

  +  | n0 n1 n2 n3
 ------------------
  n0 | n0 n1 n2 n3
  n1 | n1 n0 n3 n2
  n2 | n2 n3 n0 n1
  n3 | n3 n2 n1 n0

  *  | n0 n1 n2 n3
 ------------------
  n0 | n0 n0 n0 n0
  n1 | n0 n1 n2 n3
  n2 | n3 n2 n1 n0
  n3 | n3 n3 n3 n3

gap> g := Group( (1,2), (1,2,3) );
Group( (1,2), (1,2,3) )
gap> e := Endomorphisms( g );
[ Transformation( Group( (1,2), (1,2,3) ), [ 1, 1, 1, 1, 1, 1 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 2, 1, 1, 2 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 6, 5, 4, 3 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 2, 5, 4, 6 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 3, 1, 1, 3 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 6, 4, 5, 2 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 2, 4, 5, 3 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 3, 5, 4, 2 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 6, 1, 1, 6 ] ),
  Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 3, 4, 5, 6 ] ) ]
gap> nr := Nearring( e );      # the endomorphisms near-ring on S3
Nearring( Transformation( Group( (1,2), (1,2,3) ), [ 1, 1, 1, 1, 1, 1
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 2, 1, 1, 2
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 3, 4, 5, 6
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 2, 6, 5, 4, 3
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 2, 5, 4, 6
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 3, 1, 1, 3
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 3, 6, 4, 5, 2
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 2, 4, 5, 3
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 3, 5, 4, 2
```

```
 ] ), Transformation( Group( (1,2), (1,2,3) ), [ 1, 6, 6, 1, 1, 6 ] ) )
gap> Size( nr );
54
```

63.33 IsNearring

`IsNearring( obj )`

`IsNearring` returns `true` if the object *obj* is a near-ring and `false` otherwise. This function simply checks if the record component *obj*`.isNear-ring` is bound to the value `true`.

```
gap> n := LibraryNearring( "C3", 4 );
LibraryNearring( "C3", 4 )
gap> IsNearring( n );
true
gap> IsNearring( nr );
true
gap> IsNearring( Integers );
false            # Integers is a ring record, not a near-ring record
```

63.34 IsTransformationNearring

`IsTransformationNearring( obj )`

`IsTransformationNearring` returns `true` if the object *obj* is a transformation near-ring and `false` otherwise. `IsTransformationNearring` simply checks if the record component *obj*`.isTransformationNearring` is bound to `true`.

```
gap> IsTransformationNearring( nr );
true
gap> IsTransformationNearring( n );
false
```

63.35 LibraryNearring

`LibraryNearring( grp_name, num )`

`LibraryNearring` retrieves a near-ring from the near-rings library files. *grp_name* must be one of the following strings indicating the name of the according group: `"C2"`, `"C3"`, `"C4"`, `"V4"`, `"C5"`, `"C6"`, `"S3"`, `"C7"`, `"C8"`, `"C2xC4"`, `"C2xC2xC2"`, `"D8"`, `"Q8"`, `"C9"`, `"C3xC3"`, `"C10"`, `"D10"`, `"C11"`, `"C12"`, `"C2xC6"`, `"D12"`, `"A4"`, `"T"`, `"C13"`, `"C14"`, `"D14"`, `"C15"`, *num* must be an integer which indicates the number of the class of near-rings on the specified group.

```
gap> n := LibraryNearring( "V4", 13 );
LibraryNearring( "V4", 13 )
```

In dependence of *grp_name*, *num* must be one of the following:

grp_name	*num*	*grp_name*	*num*
"C2"	$1 \leq num \leq 3$	"C3xC3"	$1 \leq num \leq 264$
"C3"	$1 \leq num \leq 5$	"C10"	$1 \leq num \leq 329$
"C4"	$1 \leq num \leq 12$	"D10"	$1 \leq num \leq 206$
"V4"	$1 \leq num \leq 23$	"C11"	$1 \leq num \leq 139$
"C5"	$1 \leq num \leq 10$	"C12"	$1 \leq num \leq 1749$
"C6"	$1 \leq num \leq 60$	"C2xC6"	$1 \leq num \leq 3501$
"S3"	$1 \leq num \leq 39$	"D12"	$1 \leq num \leq 48137$
"C7"	$1 \leq num \leq 24$	"A4"	$1 \leq num \leq 483$
"C8"	$1 \leq num \leq 135$	"T"	$1 \leq num \leq 824$
"C2xC4"	$1 \leq num \leq 1159$	"C13"	$1 \leq num \leq 454$
"C2xC2xC2"	$1 \leq num \leq 834$	"C14"	$1 \leq num \leq 2716$
"D8"	$1 \leq num \leq 1447$	"D14"	$1 \leq num \leq 1821$
"Q8"	$1 \leq num \leq 281$	"C15"	$1 \leq num \leq 3817$
"C9"	$1 \leq num \leq 222$		

63.36 DisplayCayleyTable for near-rings

`DisplayCayleyTable(` *nr* `)`

`DisplayCayleyTable` prints the additive and multiplicative Cayley tables of the near-ring *nr*. This function works the same way as for semigroups; the dispatcher function `Display-CayleyTable` calls *nr*`.operations.DisplayTable` which performs the actual printing.

```
gap> DisplayCayleyTable( LibraryNearring( "V4", 22 ) );
Let:
n0 := ()
n1 := (3,4)
n2 := (1,2)
n3 := (1,2)(3,4)

  +  | n0 n1 n2 n3
 ------------------
  n0 | n0 n1 n2 n3
  n1 | n1 n0 n3 n2
  n2 | n2 n3 n0 n1
  n3 | n3 n2 n1 n0

  *  | n0 n1 n2 n3
 ------------------
  n0 | n0 n0 n0 n0
  n1 | n0 n1 n2 n3
  n2 | n2 n2 n2 n2
  n3 | n2 n3 n0 n1
```

63.37 Elements for near-rings

`Elements(` *nr* `)`

The function `Elements` computes the elements of the near-ring *nr*. As for semigroups the GAP3 standard library dispatcher function `Elements` calls *nr*`.operations.Elements` which simply returns the elements of *nr*`.group` if *nr* is not a transformation near-ring or – if *nr* is a transformation near-ring – performs a simple closure algorithm and returns a set of transformations which are the elements of *nr*.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
gap> Elements( Nearring( t ) );
[ Transformation( Group( (1,2) ), [ 1, 1 ] ),
  Transformation( Group( (1,2) ), [ 1, 2 ] ),
  Transformation( Group( (1,2) ), [ 2, 1 ] ),
  Transformation( Group( (1,2) ), [ 2, 2 ] ) ]
gap> Elements( LibraryNearring( "C3", 4 ) );
[ (), (1,2,3), (1,3,2) ]
```

63.38 Size for near-rings

`Size(` *nr* `)`

`Size` returns the number of elements in the near-ring *nr*.

```
gap> Size( LibraryNearring( "C3", 4 ) );
3
```

63.39 Endomorphisms for near-rings

`Endomorphisms(` *nr* `)`

`Endomorphisms` computes all the endomorphisms of the near-ring *nr*. The endomorphisms are returned as a list of transformations. In fact, the returned list contains those endomorphisms of *nr*`.group` which are also endomorphisms of the near-ring *nr*.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
gap> nr := Nearring( t );
Nearring( Transformation( Group( (1,2) ), [ 2, 1 ] ) )
gap> Endomorphisms( nr );
[ Transformation( Group( (1,2)(3,4), (1,3)(2,4) ), [ 1, 1, 1, 1 ] ),
  Transformation( Group( (1,2)(3,4), (1,3)(2,4) ), [ 1, 2, 2, 1 ] ),
  Transformation( Group( (1,2)(3,4), (1,3)(2,4) ), [ 1, 2, 3, 4 ] ) ]
```

63.40 Automorphisms for near-rings

`Automorphisms(` *nr* `)`

`Automorphisms` computes all the automorphisms of the near-ring *nr*. The automorphisms are returned as a list of transformations. In fact, the returned list contains those automorphisms of *nr*`.group` which are also automorphisms of the near-ring *nr*.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
```

```
gap> nr := Nearring( t );
Nearring( Transformation( Group( (1,2) ), [ 2, 1 ] ) )
gap> Automorphisms( nr );
[ Transformation( Group( (1,2)(3,4), (1,3)(2,4) ), [ 1, 2, 3, 4 ] ) ]
```

63.41 FindGroup

`FindGroup(` *nr* `)`

For a transformation near-ring *nr*, this function computes the additive group of *nr* as a GAP3 permutation group and stores it in the record component *nr*`.group`.

```
gap> t := Transformation( Group( (1,2) ), [2,1] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
gap> n := Nearring( t );
Nearring( Transformation( Group( (1,2) ), [ 2, 1 ] ) )
gap> g := FindGroup( n );
Group( (1,2)(3,4), (1,3)(2,4) )
gap> Elements( g );
[ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ]
gap> Elements( n );
[ Transformation( Group( (1,2) ), [ 1, 1 ] ),
  Transformation( Group( (1,2) ), [ 1, 2 ] ),
  Transformation( Group( (1,2) ), [ 2, 1 ] ),
  Transformation( Group( (1,2) ), [ 2, 2 ] ) ]
```

63.42 NearringIdeals

`NearringIdeals(` *nr* `)`
`NearringIdeals(` *nr*, "l" `)`
`NearringIdeals(` *nr*, "r" `)`

`NearringIdeals` computes all (left) (right) ideals of the near-ring *nr*. The return value is a list of subgroups of the additive group of *nr* representing the according ideals. In case that *nr* is a transformation near-ring, `FindGroup` is used to determine the additive group of *nr* as a permutation group. If the optional parameters `"l"` or `"r"` are passed, all left resp. right ideals are computed.

```
gap> n := LibraryNearring( "V4", 11 );
LibraryNearring( "V4", 11 )
gap> NearringIdeals( n );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ), V4 ]
gap> NearringIdeals( n, "r" );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ), V4 ]
gap> NearringIdeals( n, "l" );
[ Subgroup( V4, [  ] ), Subgroup( V4, [ (3,4) ] ),
  Subgroup( V4, [ (1,2) ] ), Subgroup( V4, [ (1,2)(3,4) ] ), V4 ]
```

63.43 InvariantSubnearrings

`InvariantSubnearrings(` *nr* `)`

A subnear-ring $(M, +, \cdot)$ of a near-ring $(N, +, \cdot)$ is called an **invariant subnear-ring** if both, $M \cdot N \subseteq M$ and $N \cdot M \subseteq M$.

The function `InvariantSubnearrings` computes all invariant subnear-rings of the near-ring *nr*. The function returns a list of near-rings representing the according invariant subnear-rings. In case that *nr* is a transformation near-ring, `FindGroup` is used to determine the additive group of *nr* as a permutation group.

```
gap> InvariantSubnearrings( LibraryNearring( "V4", 22 ) );
[ Nearring( Subgroup( V4, [ (1,2) ] ), function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ), Nearring( V4, function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ) ]
```

63.44 Subnearrings

`Subnearrings(` *nr* `)`

The function `Subnearrings` computes all subnear-rings of the near-ring *nr*. The function returns a list of near-rings representing the according subnear-rings. In case that *nr* is a transformation near-ring, `FindGroup` is used to determine the additive group of *nr* as a permutation group.

```
gap> Subnearrings( LibraryNearring( "V4", 22 ) );
[ Nearring( Subgroup( V4, [  ] ), function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ), Nearring( Subgroup( V4, [ (3,4) ] ), function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ), Nearring( Subgroup( V4, [ (1,2) ] ), function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ), Nearring( V4, function ( x, y )
        return elms[tfle.(f[Position( elms, y )])[Position( elms, x )]
           ];
    end ) ]
```

63.45 Identity for near-rings

`Identity(` *nr* `)`

The function `Identity` returns a list containing the identity of the multiplicative semigroup of the near-ring *nr* if it exists and the empty list `[  ]` otherwise.

```
gap> Identity( LibraryNearring( "V4", 22 ) );
[ (3,4) ]
```

63.46 Distributors

`Distributors(` *nr* `)`

An element x of a near-ring $(N, +, \cdot)$ is called a **distributor** if $x = n_1 \cdot (n_2 + n_3) - (n_1 \cdot n_2 + n_1 \cdot n_3)$ for some elements n_1, n_2, n_3 of N.

The function `Distributors` returns a list containing the distributors of the near-ring *nr*.

```
gap> Distributors( LibraryNearring( "S3", 19 ) );
[ (), (1,2,3), (1,3,2) ]
```

63.47 DistributiveElements

`DistributiveElements(` *nr* `)`

An element d of a right near-ring $(N, +, \cdot)$ is called a **distributive element** if it is also left distributive over all elements, i.e. $\forall n_1, n_2 \in N : \ d \cdot (n_1 + n_2) = d \cdot n_1 + d \cdot n_2$.

The function `DistributiveElements` returns a list containing the distributive elements of the near-ring *nr*.

```
gap> DistributiveElements( LibraryNearring( "S3", 25 ) );
[ (), (1,2,3), (1,3,2) ]
```

63.48 IsDistributiveNearring

`IsDistributiveNearring(` *nr* `)`

A right near-ring N is called **distributive near-ring** if its multiplication is also left distributive.

The function `IsDistributiveNearring` simply checks if all elements are distributive and returns the according boolean value `true` or `false`.

```
gap> IsDistributiveNearring( LibraryNearring( "S3", 25 ) );
false
```

63.49 ZeroSymmetricElements

`ZeroSymmetricElements(` *nr* `)`

Let $(N, +, \cdot)$ be a right near-ring and denote by 0 the neutral element of $(N, +)$. An element n of N is called a **zero-symmetric element** if $n \cdot 0 = 0$.

Remark: note that in a **right** near-ring $0 \cdot n = 0$ is true for all elements n.

The function `ZeroSymmetricElements` returns a list containing the zero-symmetric elements of the near-ring *nr*.

```
gap> ZeroSymmetricElements( LibraryNearring( "S3", 25 ) );
[ (), (1,2,3), (1,3,2) ]
```

63.50 IsAbstractAffineNearring

`IsAbstractAffineNearring(` *nr* `)`

A right near-ring N is called **abstract affine** if its additive group is abelian and its zero-symmetric elements are exactly its distributive elements.

The function `IsAbstractAffineNear-ring` returns the according boolean value `true` or `false`.

```
gap> IsAbstractAffineNearring( LibraryNearring( "S3", 25 ) );
false
```

63.51 IdempotentElements for near-rings

`IdempotentElements(` *nr* `)`

The function `IdempotentElements` returns a list containing the idempotent elements of the multiplicative semigroup of the near-ring *nr*.

```
gap> IdempotentElements( LibraryNearring( "S3", 25 ) );
[ (), (2,3) ]
```

63.52 IsBooleanNearring

`IsBooleanNearring(` *nr* `)`

A right near-ring N is called **boolean** if all its elements are idempotent with respect to multiplication.

The function `IsBooleanNearring` simply checks if all elements are idempotent and returns the according boolean value `true` or `false`.

```
gap> IsBooleanNearring( LibraryNearring( "S3", 25 ) );
false
```

63.53 NilpotentElements

`NilpotentElements(` *nr* `)`

Let $(N, +, \cdot)$ be a near-ring with zero 0. An element n of N is called **nilpotent** if there is a positive integer k such that $n^k = 0$.

The function `NilpotentElements` returns a list of sublists of length 2 where the first entry is a nilpotent element n and the second entry is the smallest k such that $n^k = 0$.

```
gap> NilpotentElements( LibraryNearring( "V4", 4 ) );
[ [ (), 1 ], [ (3,4), 2 ], [ (1,2), 3 ], [ (1,2)(3,4), 3 ] ]
```

63.54 IsNilNearring

`IsNilNearring(` *nr* `)`

A near-ring N is called **nil** if all its elements are nilpotent.

The function `IsNilNearring` checks if all elements are nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilNearring( LibraryNearring( "V4", 4 ) );
true
```

63.55 IsNilpotentNearring

`IsNilpotentNearring(` *nr* `)`

A near-ring N is called **nilpotent** if there is a positive integer k, s.t. $N^k = \{0\}$.

The function `IsNilpotentNearring` tests if the near-ring *nr* is nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilpotentNearring( LibraryNearring( "V4", 4 ) );
true
```

63.56 IsNilpotentFreeNearring

`IsNilpotentFreeNearring(` *nr* `)`

A near-ring N is called **nilpotent free** if its only nilpotent element is 0.

The function `IsNilpotentFreeNearring` checks if 0 is the only nilpotent and returns the according boolean value `true` or `false`.

```
gap> IsNilpotentFreeNearring( LibraryNearring( "V4", 22 ) );
true
```

63.57 IsCommutative for near-rings

`IsCommutative(` *nr* `)`

A near-ring $(N, +, \cdot)$ is called **commutative** if its multiplicative semigroup is commutative.

The function `IsCommutative` returns the according value `true` or `false`.

```
gap> IsCommutative( LibraryNearring( "C15", 1235 ) );
false
gap> IsCommutative( LibraryNearring( "V4", 13 ) );
true
```

63.58 IsDgNearring

`IsDgNearring(` *nr* `)`

A near-ring $(N, +, \cdot)$ is called **distributively generated (d.g.)** if $(N, +)$ is generated additively by the distributive elements of the near-ring.

The function `IsDgNearring` returns the according value `true` or `false` for a near-ring *nr*.

```
gap> IsDgNearring( LibraryNearring( "S3", 25 ) );
false
gap> IsDgNearring( LibraryNearring( "S3", 1 ) );
true
```

63.59 IsIntegralNearring

`IsIntegralNearring(` *nr* `)`

A near-ring $(N, +, \cdot)$ with zero element 0 is called **integral** if it has no zero divisors, i.e. the condition $\forall n_1, n_2 : \; n_1 \cdot n_2 = 0 \Rightarrow n_1 = 0 \vee n_2 = 0$ holds.

The function `IsIntegralNearring` returns the according value `true` or `false` for a near-ring *nr*.

```
gap> IsIntegralNearring( LibraryNearring( "S3", 24 ) );
true
gap> IsIntegralNearring( LibraryNearring( "S3", 25 ) );
false
```

63.60 IsPrimeNearring

`IsPrimeNearring(` *nr* `)`

A near-ring $(N, +, \cdot)$ with zero element 0 is called **prime** if the ideal $\{0\}$ is a prime ideal.

The function `IsPrimeNearring` checks if *nr* is a prime near-ring by using the condition *for all non-zero ideals* $I, J: \; I \cdot J \neq \{0\}$ and returns the according value `true` or `false`.

```
gap> IsPrimeNearring( LibraryNearring( "V4", 11 ) );
false
```

63.61 QuasiregularElements

`QuasiregularElements(` *nr* `)`

Let $(N, +, \cdot)$ be a right near-ring. For an element $z \in N$, denote the left ideal generated by the set $\{n - n \cdot z \mid n \in N\}$ by L_z. An element z of N is called **quasiregular** if $z \in L_z$.

The function `QuasiregularElements` returns a list of all quasiregular elements of a near-ring *nr*.

```
gap> QuasiregularElements( LibraryNearring( "V4", 4 ) );
[ (), (3,4), (1,2), (1,2)(3,4) ]
```

63.62 IsQuasiregularNearring

`IsQuasiregularNearring(` *nr* `)`

A near-ring N is called **quasiregular** if all its elements are quasiregular.

The function `IsQuasiregularNearring` simply checks if all elements of the near-ring *nr* are quasiregular and returns the according boolean value `true` or `false`.

```
gap> IsQuasiregularNearring( LibraryNearring( "V4", 4 ) );
true
```

63.63 RegularElements

`RegularElements(` *nr* `)`

Let $(N, +, \cdot)$ be a near-ring. An element n of N is called **regular** if there is an element x such that $n \cdot x \cdot n = n$.

The function `RegularElements` returns a list of all regular elements of a near-ring *nr*.

```
gap> RegularElements( LibraryNearring( "V4", 4 ) );
[ () ]
```

63.64 IsRegularNearring

`IsRegularNearring(` *nr* `)`

A near-ring N is called **regular** if all its elements are regular.

The function `IsRegularNearring` simply checks if all elements of the near-ring *nr* are regular and returns the according boolean value `true` or `false`.

```
gap> IsRegularNearring( LibraryNearring( "V4", 4 ) );
false
```

63.65 IsPlanarNearring

`IsPlanarNearring(` *nr* `)`

Let $(N, +, \cdot)$ be a right near-ring. For $a, b \in N$ define the equivalence relation $\equiv$ by $a \equiv b :\Leftrightarrow \forall n \in N :\ n \cdot a = n \cdot b$. If $a \equiv b$ then a and b are called **equivalent multipliers**. A near-ring N is called **planar** if $\mid N/_{\equiv} \mid \geq 3$ and if every equation of the form $x \cdot a = x \cdot b + c$ has a unique solution for two non equivalent multipliers a and b.

The function `IsPlanarNearring` returns the according value `true` or `false` for a near-ring *nr*.

Remark: this function works only for library near-rings, i.e. near-rings which are constructed by using the function `LibraryNearring`.

```
gap> IsPlanarNearring( LibraryNearring( "V4", 22 ) );
false
```

63.66 IsNearfield

`IsNearfield(` *nr* `)`

Let $(N, +, \cdot)$ be a near-ring with zero 0 and denote by N^* the set $N - \{0\}$. N is a **nearfield** if $(N^*, \cdot)$ is a group.

The function `IsNearfield` tests if *nr* has an identity and if every non-zero element has a multiplicative inverse and returns the according value `true` or `false`.

```
gap> IsNearfield( LibraryNearring( "V4", 16 ) );
true
```

63.67 LibraryNearringInfo

`LibraryNearringInfo(` *group_name*`,` *list*`,` *string* `)`

This function provides information about the specified library near-rings in a way similar to how near-rings are presented in the appendix of [Pil83]. The parameter *group_name* specifies the name of a group; valid names are exactly those which are also valid for the function `LibraryNearrings` (cf. section 63.35).

The parameter *list* must be a non-empty list of integers defining the classes of near-rings to be printed. Naturally, these integers must all fit into the ranges described in section 63.35 for the according groups.

The third parameter *string* is optional. *string* must be a string consisting of one or more (or all) of the following characters: `l`, `m`, `i`, `v`, `s`, `e`, `a`. Per default, (i.e. if this parameter is not specified), the output is minimal. According to each specified character, the following is added:

`c`
: print the meaning of the letters used in the output.

`m`
: print the multiplication tables.

`i`
: list the ideals.

`l`
: list the left ideals.

`r`
: list the right ideals.

`v`
: list the invariant subnear-rings.

`s`
: list the subnear-rings.

`e`
: list the near-ring endomorphisms.

`a`
: list the near-ring automorphisms.

Examples:

`LibraryNearringInfo( "C3", [ 3 ], "lmivsea")` will print all available information on the third class of near-rings on the group C_3.

`LibraryNearringInfo( "C4", [ 1..12 ] )` will provide a minimal output for all classes of near-rings on C_4.

`LibraryNearringInfo( "S3", [ 5, 18, 21 ], "mi")` will print the minimal information plus the multiplication tables plus the ideals for the classes 5, 18, and 21 of near-rings on the group S_3.

63.68 Nearring records

The record of a nearring has the following components:

`isDomain`, `isNearring`
: these two are always `true` for a near-ring.

`isTransformationNearring`
: this is bound and `true` only for transformation near-rings (i.e. those near-rings that are generated by group transformations by using the constructor function `Nearring` in the second form).

`generators`
: this is bound only for a transformation near-ring and holds the set of generators of the transformation near-ring.

`group`
: this component holds the additive group of the near-ring as permutation group.

`addition`, subtraction, `multiplication`
: these record fields contain the functions that represent the binary operations that can be performed with the elements of the near-ring on the additive group of the near-ring (addition, subtraction) resp. on the multiplicative semigroup of the near-ring (multiplication)

```
gap> nr := Nearring( Transformation( Group( (1,2) ), [2,1] ) );
Nearring( Transformation( Group( (1,2) ), [ 2, 1 ] ) )
gap> e := Elements( nr );
[ Transformation( Group( (1,2) ), [ 1, 1 ] ),
  Transformation( Group( (1,2) ), [ 1, 2 ] ),
  Transformation( Group( (1,2) ), [ 2, 1 ] ),
  Transformation( Group( (1,2) ), [ 2, 2 ] ) ]
gap> nr. addition( e[2], e[3] );
Transformation( Group( (1,2) ), [ 2, 2 ] )
gap> nr.multiplication( e[2], e[4] );
Transformation( Group( (1,2) ), [ 2, 2 ] )
gap> nr.multiplication( e[2], e[3] );
Transformation( Group( (1,2) ), [ 2, 1 ] )
```

`operations`
: this is the operations record of the near-ring.

`size`, `elements`, `endomorphisms`, `automorphisms`
: these entries become bound if the according functions have been performed on the near-ring.

63.69 Supportive Functions for Groups

In order to support near-ring calculations, a few functions for groups had to be added to the standard GAP3 group library functions.

63.70 DisplayCayleyTable for groups

`DisplayCayleyTable(` *group* `)`

`DisplayCayleyTable` prints the Cayley table of the group *group*. This function works the same way as for semigroups and near-rings: the dispatcher function `DisplayCayleyTable` calls *group*`.operations.DisplayTable` which performs the printing.

```
gap> g := Group( (1,2), (3,4) );          # this is Klein's four group
Group( (1,2), (3,4) )
gap> DisplayCayleyTable( g );
Let:
g0 := ()
g1 := (3,4)
g2 := (1,2)
g3 := (1,2)(3,4)
```

```
 +  | g0 g1 g2 g3
-----------------
g0  | g0 g1 g2 g3
g1  | g1 g0 g3 g2
g2  | g2 g3 g0 g1
g3  | g3 g2 g1 g0
```

63.71 Endomorphisms for groups

`Endomorphisms( ` *group* ` )`

`Endomorphisms` computes all the endomorphisms of the group *group*. This function is most essential for computing the near-rings on a group. The endomorphisms are returned as a list of transformations s.t. the identity endomorphism is always the last entry in this list. For each transformation in this list the record component `isGroupEndomorphism` is set to `true` and if such a transformation is in fact an automorphism then in addition the record component `isGroupAutomorphism` is set to `true`.

```
gap> g := Group( (1,2,3) );
Group( (1,2,3) )
gap> Endomorphisms( g );
[ Transformation( Group( (1,2,3) ), [ 1, 1, 1 ] ),
  Transformation( Group( (1,2,3) ), [ 1, 3, 2 ] ),
  Transformation( Group( (1,2,3) ), [ 1, 2, 3 ] ) ]
```

63.72 Automorphisms for groups

`Automorphisms( ` *group* ` )`

`Automorphisms` computes all the automorphisms of the group *group*. The automorphisms are returned as a list of transformations s.t. the identity automorphism is always the last entry in this list. For each transformation in this list the record components `isGroupEndomorphism` and `isGroupAutomorphism` are both set to `true`.

```
gap> d8 := Group( (1,2,3,4), (2,4) ); # dihedral group of order 8
Group( (1,2,3,4), (2,4) )
gap> a := Automorphisms( d8 );
[ Transformation( Group( (1,2,3,4), (2,4) ), [ 1, 2, 8, 7, 5, 6, 4, 3
     ] ), Transformation( Group( (1,2,3,4), (2,4) ),
    [ 1, 3, 2, 7, 8, 6, 4, 5 ] ), Transformation( Group( (1,2,3,4),
    (2,4) ), [ 1, 3, 5, 4, 8, 6, 7, 2 ] ),
  Transformation( Group( (1,2,3,4), (2,4) ), [ 1, 5, 3, 7, 2, 6, 4, 8
     ] ), Transformation( Group( (1,2,3,4), (2,4) ),
    [ 1, 5, 8, 4, 2, 6, 7, 3 ] ), Transformation( Group( (1,2,3,4),
    (2,4) ), [ 1, 8, 2, 4, 3, 6, 7, 5 ] ),
  Transformation( Group( (1,2,3,4), (2,4) ), [ 1, 8, 5, 7, 3, 6, 4, 2
     ] ), Transformation( Group( (1,2,3,4), (2,4) ),
    [ 1, 2, 3, 4, 5, 6, 7, 8 ] ) ]
```

63.73 InnerAutomorphisms

`InnerAutomorphisms(` *group* `)`

`InnerAutomorphisms` computes all the inner automorphisms of the group *group*. The inner automorphisms are returned as a list of transformations s.t. the identity automorphism is always the last entry in this list. For each transformation in this list the record components `isInnerAutomorphism`, `isGroupEndomorphism`, and `isGroupAutomorphism` are all set to `true`.

```
gap> i := InnerAutomorphisms( d8 );
[ Transformation( Group( (1,2,3,4), (2,4) ), [ 1, 2, 8, 7, 5, 6, 4, 3
     ] ), Transformation( Group( (1,2,3,4), (2,4) ),
    [ 1, 5, 3, 7, 2, 6, 4, 8 ] ), Transformation( Group( (1,2,3,4),
    (2,4) ), [ 1, 5, 8, 4, 2, 6, 7, 3 ] ),
  Transformation( Group( (1,2,3,4), (2,4) ), [ 1, 2, 3, 4, 5, 6, 7, 8
     ] ) ]
```

63.74 SmallestGeneratingSystem

`SmallestGeneratingSystem(` *group* `)`

`SmallestGeneratingSystem` computes a smallest generating system of the group *group* i.e. a smallest subset of the elements of the group s.t. the group is generated by this subset.

Remark: there is a GAP3 standard library function `SmallestGenerators` for permutation groups. This function computes a generating set of a given group such that its elements are smallest possible permutations (w.r.t. the GAP3 internal sorting of permutations).

```
gap> s5 := SymmetricGroup( 5 );
Group( (1,5), (2,5), (3,5), (4,5) )
gap> SmallestGenerators( s5 );
[ (4,5), (3,4), (2,3), (1,2) ]
gap> SmallestGeneratingSystem( s5 );
[ (1,3,5)(2,4), (1,2)(3,4,5) ]
```

63.75 IsIsomorphicGroup

`IsIsomorphicGroup(` *g1* `,` *g2* `)`

`IsIsomorphicGroup` determines if the groups *g1* and *g2* are isomorphic and if so, returns a group homomorphism that is an isomorphism between these two groups and `false` otherwise.

```
gap> g1 := Group( (1,2,3) );
Group( (1,2,3) )
gap> g2 := Group( (7,8,9) );
Group( (7,8,9) )
gap> g1 = g2;
false
gap> IsIsomorphicGroup( g1, g2 );
GroupHomomorphismByImages( Group( (1,2,3) ), Group( (7,8,9) ),
[ (1,2,3) ], [ (7,8,9) ] )
```

63.76 Predefined groups

The following variables are predefined as according permutation groups with a default smallest set of generators: `C2`, `C3`, `C4`, `V4`, `C5`, `C6`, `S3`, `C7`, `C8`, `C2xC4`, `C2xC2xC2`, `D8`, `Q8`, `C9`, `C3xC3`, `C10`, `D10`, `C11`, `C12`, `C2xC6`, `D12`, `A4`, `T`, `C13`, `C14`, `D14`, `C15`.

```
gap> S3;
S3
gap> Elements( S3 );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
gap> IsPermGroup( S3 );
true
gap> S3.generators;
[ (1,2), (1,2,3) ]
```

63.77 How to find near-rings with certain properties

The near-ring library files can be used to systematically search for near-rings with (or without) certain properties.

For instance, the function `LibraryNearring` can be integrated into a loop or occur as a part of the `Filtered` or the `First` function to get all numbers of classes (or just the first class) of near-rings on a specified group which have the specified properties.

In what follows, we shall give a few examples how this can be accomplished:

To get the number of the first class of near-rings on the group C_6 which have an identity, one could use a command like:

```
gap> First( [1..60], i ->
>           Identity( LibraryNearring( "C6", i ) ) <> [ ] );
28
```

If we try the same with S_3, we will get an error message, indicating that there is no near-ring with identity on S_3:

```
gap> First( [1..39], i ->
>           Identity( LibraryNearring( "S3", i ) ) <> [ ] );
Error, at least one element of <list> must fulfill <func> in
First( [ 1 .. 39 ], function ( i ) ... end ) called from
main loop
brk>
gap>
```

To get all seven classes of near-rings with identity on the dihedral group D_8 of order 8, something like the following will work fine:

```
gap> l := Filtered( [1..1447], i ->
>           Identity( LibraryNearring( "D8", i ) ) <> [ ] );
[ 842, 844, 848, 849, 1094, 1096, 1097 ]
gap> time;
122490
```

Note that a search like this may take a few minutes.

Here is another example that provides the class numbers of the four boolean near-rings on D_8:

```
gap> l := Filtered( [1..1447], i ->
>           IsBooleanNearring( LibraryNearring( "D8", i ) ) );
[ 1314, 1380, 1446, 1447 ]
```

The search for class numbers of near-rings can also be accomplished in a loop like the following:

```
gap> l:=[ ];;
gap> for i in [1..1447] do
>      n := LibraryNearring( "D8", i );
>      if IsDgNearring( n ) and
>        not IsDistributiveNearring( n ) then
>          Add( l, i );
>      fi;
>    od;
gap> time;
261580
gap> l;
[ 765, 1094, 1098, 1306 ]
```

This provides a list of those class numbers of near-rings on D_8 which are distributively generated but not distributive.

Two or more conditions for library near-rings can also be combined with `or`:

```
gap> l := [ ];;
gap> for i in [1..1447] do
>      n := LibraryNearring( "D8", i );
>      if Size( ZeroSymmetricElements( n ) ) < 8 or
>         Identity( n ) <> [ ] or
>         IsIntegralNearring( n ) then
>           Add( l, i );
>      fi;
>    od;
gap> time;
124480
gap> l;
[ 842, 844, 848, 849, 1094, 1096, 1097, 1314, 1315, 1316, 1317, 1318,
  1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329,
  1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340,
  1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351,
  1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362,
  1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, 1372, 1373,
  1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384,
  1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395,
  1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406,
  1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417,
  1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428,
  1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439,
```

```
  1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447 ]
gap> Length( l );
141
```

This provides a list of all 141 class numbers of near-rings on D_8 which are non-zerosymmetric or have an identity or are integral. (cf. [Pil83], pp. 416ff).

The following loop does the same for the near-rings on the quaternion group of order 8, Q_8:

```
gap> l := [ ];;
gap> for i in [1..281] do
>      n := LibraryNearring( "Q8", i );
>      if Size( ZeroSymmetricElements( n ) ) < 8 or
>        Identity( n ) <> [ ] or
>          IsIntegralNearring( n ) then
>          Add( l, i );
>      fi;
>    od;
gap> time;
17740
gap> l;
[ 280, 281 ]
```

Once a list of class numbers has been computed (in this case here: `l`), e.g. the function `LibraryNearringInfo` can be used to print some information about these near-rings:

```
gap> LibraryNearringInfo( "Q8", l );
----------------------------------------------------------------------
>>> GROUP: Q8
elements: [ n0, n1, n2, n3, n4, n5, n6, n7 ]
addition table:

  +  | n0 n1 n2 n3 n4 n5 n6 n7
 ------------------------------
  n0 | n0 n1 n2 n3 n4 n5 n6 n7
  n1 | n1 n2 n3 n0 n7 n4 n5 n6
  n2 | n2 n3 n0 n1 n6 n7 n4 n5
  n3 | n3 n0 n1 n2 n5 n6 n7 n4
  n4 | n4 n5 n6 n7 n2 n3 n0 n1
  n5 | n5 n6 n7 n4 n1 n2 n3 n0
  n6 | n6 n7 n4 n5 n0 n1 n2 n3
  n7 | n7 n4 n5 n6 n3 n0 n1 n2

group endomorphisms:
1:   [ n0, n0, n0, n0, n0, n0, n0, n0 ]
2:   [ n0, n0, n0, n0, n2, n2, n2, n2 ]
3:   [ n0, n1, n2, n3, n5, n6, n7, n4 ]
4:   [ n0, n1, n2, n3, n6, n7, n4, n5 ]
5:   [ n0, n1, n2, n3, n7, n4, n5, n6 ]
6:   [ n0, n2, n0, n2, n0, n2, n0, n2 ]
7:   [ n0, n2, n0, n2, n2, n0, n2, n0 ]
```

```
8:   [ n0, n3, n2, n1, n4, n7, n6, n5 ]
9:   [ n0, n3, n2, n1, n5, n4, n7, n6 ]
10:  [ n0, n3, n2, n1, n6, n5, n4, n7 ]
11:  [ n0, n3, n2, n1, n7, n6, n5, n4 ]
12:  [ n0, n4, n2, n6, n1, n7, n3, n5 ]
13:  [ n0, n4, n2, n6, n3, n5, n1, n7 ]
14:  [ n0, n4, n2, n6, n5, n1, n7, n3 ]
15:  [ n0, n4, n2, n6, n7, n3, n5, n1 ]
16:  [ n0, n5, n2, n7, n1, n4, n3, n6 ]
17:  [ n0, n5, n2, n7, n3, n6, n1, n4 ]
18:  [ n0, n5, n2, n7, n4, n3, n6, n1 ]
19:  [ n0, n5, n2, n7, n6, n1, n4, n3 ]
20:  [ n0, n6, n2, n4, n1, n5, n3, n7 ]
21:  [ n0, n6, n2, n4, n3, n7, n1, n5 ]
22:  [ n0, n6, n2, n4, n5, n3, n7, n1 ]
23:  [ n0, n6, n2, n4, n7, n1, n5, n3 ]
24:  [ n0, n7, n2, n5, n1, n6, n3, n4 ]
25:  [ n0, n7, n2, n5, n3, n4, n1, n6 ]
26:  [ n0, n7, n2, n5, n4, n1, n6, n3 ]
27:  [ n0, n7, n2, n5, n6, n3, n4, n1 ]
28:  [ n0, n1, n2, n3, n4, n5, n6, n7 ]

NEARRINGS:
----------------------------------------------------------------------
280)  phi: [ 1, 28, 28, 28, 28, 28, 28, 28 ];  28;  -B----I--P-RW
----------------------------------------------------------------------
281)  phi: [ 28, 28, 28, 28, 28, 28, 28, 28 ];  28;  -B----I--P-RW
----------------------------------------------------------------------
```

63.78 Defining near-rings with known multiplication table

Suppose that for a given group g the multiplication table of a binary operation $*$ on the elements of g is known such that $*$ is a near-ring multiplication on g. Then the constructor function `Nearring` can be used to input the near-ring specified by g and $*$.

An example shall illustrate a possibility how this could be accomplished: Take the group S_3, which in GAP3 can be defined e.g. by

```
gap> g := Group( (1,2), (1,2,3) );
Group( (1,2), (1,2,3) )
```

This group has the following list of elements:

```
gap> Elements( g );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

Let `1` stand for the first element in this list, `2` for the second, and so on up to `6` which stands for the sixth element in the following multiplication table:

*	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	2	2	6	3	6	3
4	1	1	5	4	5	4
5	1	1	4	5	4	5
6	2	2	3	6	3	6

A near-ring on `g` with this multiplication can be input by first defining a multiplication function, say `m` in the following way:

```
gap> m := function( x, y )
>   local elms, table;
>     elms := Elements( g );
>     table := [ [ 1, 1, 1, 1, 1, 1 ],
>                [ 2, 2, 2, 2, 2, 2 ],
>                [ 2, 2, 6, 3, 6, 3 ],
>                [ 1, 1, 5, 4, 5, 4 ],
>                [ 1, 1, 4, 5, 4, 5 ],
>                [ 2, 2, 3, 6, 3, 6 ] ];
>
>     return elms[ table[Position( elms, x )][Position( elms, y )] ];
>   end;
function ( x, y ) ... end
```

Then the near-ring can be constructed by calling `Nearring` with the parameters `g` and `m`:

```
gap> n := Nearring( g, m );
Nearring( Group( (1,2), (1,2,3) ), function ( x, y )
    local  elms, table;
    elms := Elements( g );
    table := [ [ 1, 1, 1, 1, 1, 1 ], [ 2, 2, 2, 2, 2, 2 ],
        [ 2, 2, 6, 3, 6, 3 ], [ 1, 1, 5, 4, 5, 4 ],
        [ 1, 1, 4, 5, 4, 5 ], [ 2, 2, 3, 6, 3, 6 ] ];
    return elms[table[Position( elms, x )][Position( elms, y )]];
end )
```

Chapter 64

Grape

This chapter describes the main functions of the GRAPE (Version 2.31) share library package for computing with graphs and groups. All functions described here are written entirely in the GAP3 language, except for the automorphism group and isomorphism testing functions, which make use of B. McKay's *nauty* (Version 1.7) package [McK90].

GRAPE is primarily designed for the construction and analysis of graphs related to permutation groups and finite geometries. Special emphasis is placed on the determination of regularity properties and subgraph structure. The GRAPE philosophy is that a graph Γ always comes together with a known subgroup G of $\mathrm{Aut}(\Gamma)$, and that G is used to reduce the storage and CPU-time requirements for calculations with Γ (see [Soi93]). Of course G may be the trivial group, and in this case GRAPE algorithms will perform more slowly than strictly combinatorial algorithms (although this degradation in performance is hopefully never more than a fixed constant factor).

In general GRAPE deals with directed graphs which may have loops but have no multiple edges. However, many GRAPE functions only work for **simple** graphs (i.e. no loops, and whenever $[x, y]$ is an edge then so is $[y, x]$), but these functions will check if an input graph is simple.

In GRAPE, a graph *gamma* is stored as a record, with mandatory components `isGraph`, `order`, `group`, `schreierVector`, `representatives`, and `adjacencies`. Usually, the user need not be aware of this record structure, and is strongly advised only to use GRAPE functions to construct and modify graphs.

The `order` component contains the number of vertices of *gamma*. The vertices of *gamma* are always $1, .., gamma.order$, but they may also be given **names**, either by a user or by a function constructing a graph (e.g. `InducedSubgraph`, `BipartiteDouble`, `QuotientGraph`). The `names` component, if present, records these names. If the `names` component is not present (the user may, for example, choose to unbind it), then the names are taken to be $1, ..., gamma.order$. The `group` component records the the GAP3 permutation group associated with *gamma* (this group must be a subgroup of $\mathrm{Aut}(gamma)$). The `representatives` component records a set of orbit representatives for *gamma.group* on the vertices of *gamma*, with *gamma.adjacencies*[*i*] being the set of vertices adjacent to *gamma.representatives*[*i*]. The only mandatory component which may change once a graph is initially constructed is `adjacencies` (when an edge orbit of *gamma.group* is added to, or removed from, *gamma*).

A graph record may also have some of the optional components `isSimple`, `autGroup`, and `canonicalLabelling`, which record information about that graph.

GRAPE has the ability to make use of certain random group theoretical algorithms, which can result in time and store savings. The use of these random methods may be switched on and off by the global boolean variable `GRAPE_RANDOM`, whose default value is `false` (random methods not used). Even if random methods are used, no function described below depends on the correctness of such a randomly computed result. For these functions the use of these random methods only influences the time and store taken. All global variables used by GRAPE start with `GRAPE_`.

The user who is interested in knowing more about the GRAPE system, and its advanced use, should consult [Soi93] and the GRAPE source code.

Before using any of the functions described in this chapter you must load the package by calling the statement

```
gap> RequirePackage( "grape" );

Loading  GRAPE 2.31  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmw.ac.uk.
```

64.1 Functions to construct and modify graphs

The following sections describe the functions used to construct and modify graphs.

64.2 Graph

`Graph(` G`,` L`,` *act*`,` *rel* `)`
`Graph(` G`,` L`,` *act*`,` *rel*`,` *invt* `)`

This is the most general and useful way of constructing a graph in GRAPE.

First suppose that the optional boolean parameter *invt* is unbound or has value `false`. Then L should be a list of elements of a set S on which the group G acts (**operates** in GAP3 language), with the action given by the function *act*. The parameter *rel* should be a boolean function defining a G-invariant relation on S (so that for g in G, x, y in S, $rel(x, y)$ if and only if $rel(act(x, g), act(y, g))$). Then function `Graph` returns a graph *gamma* which has as vertex names

`Concatenation( Orbits(` G`,` L`,` *act* `) )`

(the concatenation of the distinct orbits of the elements in L under G), and for vertices v, w of *gamma*, $[v, w]$ is an edge if and only if

rel`( VertexName(` *gamma*`,` v `), VertexName(` *gamma*`,` w `) )`

Now if the parameter *invt* exists and has value `true`, then it is assumed that L is invariant under G with respect to action *act*. Then the function `Graph` behaves as above, except that the vertex names of *gamma* become (a copy of) L.

The group associated with the graph *gamma* returned is the image of G acting via *act* on *gamma.names*.

For example, suppose you have an $n \times n$ adjacency matrix A for a graph X, so that the vertices of X are $\{1, \ldots, n\}$, and $[i, j]$ is an edge of X if and only if $A[i][j] = 1$. Suppose also that $G \leq \mathrm{Aut}(X)$ (G may be trivial). Then you can make a GRAPE graph isomorphic to X via `Graph( G, [1..n], OnPoints, function(x,y) return A[x][y]=1; end, true );`

```
gap> A := [[0,1,0],[1,0,0],[0,0,1]];
[ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ]
gap> G := Group( (1,2) );
Group( (1,2) )
gap> Graph( G, [1..3], OnPoints,
>           function(x,y) return A[x][y]=1; end,
>           true );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,2) ),
  schreierVector := [ -1, 1, -2 ],
  adjacencies := [ [ 2 ], [ 3 ] ],
  representatives := [ 1, 3 ],
  names := [ 1 .. 3 ] )
```

We now construct the Petersen graph.

```
gap> Petersen := Graph( SymmetricGroup(5), [[1,2]], OnSets,
>                function(x,y) return Intersection(x,y)=[]; end );
rec(
  isGraph := true,
  order := 10,
  group := Group( ( 1, 2)( 6, 8)( 7, 9), ( 1, 3)( 4, 8)( 5, 9),
    ( 2, 4)( 3, 6)( 9,10), ( 2, 5)( 3, 7)( 8,10) ),
  schreierVector := [ -1, 1, 2, 3, 4, 3, 4, 2, 2, 4 ],
  adjacencies := [ [ 8, 9, 10 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 2, 5 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 1, 3 ], [ 1, 4 ], [ 3, 5 ], [ 4, 5 ], [ 3, 4 ] ] )
```

64.3 EdgeOrbitsGraph

`EdgeOrbitsGraph(` G, E `)`
`EdgeOrbitsGraph(` G, E, n `)`

This is a common way of constructing a graph in GRAPE.

This function returns the (directed) graph with vertex set $\{1, ..., n\}$, edge set $\cup_{e \in E} e^G$, and associated (permutation) group G, which must act naturally on $\{1, ..., n\}$. The parameter E should be a list of edges (lists of length 2 of vertices), although a singleton edge will be understood as an edge list of length 1. The parameter n may be omitted, in which case the number of vertices is the largest point moved by a generator of G.

Note that G may be the trivial permutation group (`Group( () )` in GAP3 notation), in which case the (directed) edges of *gamma* are simply those in the list E.

```
gap> EdgeOrbitsGraph( Group((1,3),(1,2)(3,4)), [[1,2],[4,5]], 5 );
rec(
  isGraph := true,
  order := 5,
  group := Group( (1,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2, -2 ],
  adjacencies := [ [ 2, 4, 5 ], [  ] ],
  representatives := [ 1, 5 ],
  isSimple := false )
```

64.4 NullGraph

`NullGraph(` G `)`
`NullGraph(` G, n `)`

This function returns the null graph on n vertices, with associated (permutation) group G. The parameter n may be omitted, in which case the number of vertices is the largest point moved by a generator of G.

See also 64.29.

```
gap> NullGraph( Group( (1,2,3) ), 4 );
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,2,3) ),
  schreierVector := [ -1, 1, 1, -2 ],
  adjacencies := [ [  ], [  ] ],
  representatives := [ 1, 4 ],
  isSimple := true )
```

64.5 CompleteGraph

`CompleteGraph( G )`
`CompleteGraph( G,` n `)`
`CompleteGraph( G,` n, *mustloops* `)`

This function returns a complete graph on n vertices with associated (permutation) group G. The parameter n may be omitted, in which case the number of vertices is the largest point moved by a generator of G. The optional boolean parameter *mustloops* determines whether the complete graph has all loops present or no loops (default: `false` (no loops)).

See also 64.30.

```
gap> CompleteGraph( Group( (1,2,3), (1,2) ) );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,2,3), (1,2) ),
  schreierVector := [ -1, 1, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

64.6 JohnsonGraph

`JohnsonGraph(` *n*, *e* `)`

This function returns a graph *gamma* isomorphic to the Johnson graph, whose vertices are the e-subsets of $\{1, ..., n\}$, with x joined to y if and only if $|x \cap y| = e - 1$. The group associated with *gamma* is image of the the symmetric group S_n acting on the e-subsets of $\{1, \ldots, n\}$.

```
gap> JohnsonGraph(5,3);
rec(
  isGraph := true,
  order := 10,
  group := Group( ( 1, 8)( 2, 9)( 4,10), ( 1, 5)( 2, 6)( 7,10),
    ( 1, 3)( 4, 6)( 7, 9), ( 2, 3)( 4, 5)( 7, 8) ),
  schreierVector := [ -1, 4, 3, 4, 2, 3, 4, 1, 3, 2 ],
  adjacencies := [ [ 2, 3, 4, 5, 7, 8 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ],
      [ 1, 3, 5 ], [ 1, 4, 5 ], [ 2, 3, 4 ], [ 2, 3, 5 ],
      [ 2, 4, 5 ], [ 3, 4, 5 ] ],
  isSimple := true )
```

64.7 AddEdgeOrbit

`AddEdgeOrbit(` *gamma*, *e* `)`
`AddEdgeOrbit(` *gamma*, *e*, *H* `)`

This procedure adds the edge orbit $e^{gamma.group}$ to the edge set of graph *gamma*. The parameter e must be a sequence of length 2 of vertices of *gamma*. If the optional third parameter H is given then it is assumed that $e[2]$ has the same orbit under H as it does under the stabilizer in *gamma.group* of $e[1]$, and this knowledge can greatly speed up the procedure.

Note that if *gamma.group* is trivial then this procedure simply adds the single edge e to *gamma*.

```
gap> gamma := NullGraph( Group( (1,3), (1,2)(3,4) ) );
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [  ] ],
  representatives := [ 1 ],
  isSimple := true )
gap> AddEdgeOrbit( gamma, [4,3] );
gap> gamma;
rec(
  isGraph := true,
  order := 4,
```

```
  group := Group( (1,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 2, 4 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

64.8 RemoveEdgeOrbit

`RemoveEdgeOrbit(` *gamma*, *e* `)`
`RemoveEdgeOrbit(` *gamma*, *e*, *H* `)`

This procedure removes the edge orbit $e^{gamma.group}$ from the edge set of the graph *gamma*. The parameter e must be a sequence of length 2 of vertices of *gamma*, but if e is not an edge of *gamma* then this procedure has no effect. If the optional third parameter H is given then it is assumed that $e[2]$ has the same orbit under H as it does under the stabilizer in *gamma.group* of $e[1]$, and this knowledge can greatly speed up the procedure.

```
gap> gamma := CompleteGraph( Group( (1,3), (1,2)(3,4) ) );
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 2, 3, 4 ] ],
  representatives := [ 1 ],
  isSimple := true )
gap> RemoveEdgeOrbit( gamma, [4,3] );
gap> gamma;
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 3 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

64.9 AssignVertexNames

`AssignVertexNames(` *gamma*, *names* `)`

This function allows the user to give new names to the vertices of *gamma*, by specifying a list *names* of vertex names for the vertices of *gamma*, such that *names*$[i]$ contains the user's name for the i-th vertex of *gamma*.

A copy of *names* is assigned to *gamma.names*. See also 64.14.

```
gap> gamma := NullGraph( Group(()), 3 );
rec(
  isGraph := true,
  order := 3,
```

```
  group := Group( () ),
  schreierVector := [ -1, -2, -3 ],
  adjacencies := [ [  ], [  ], [  ] ],
  representatives := [ 1, 2, 3 ],
  isSimple := true )
gap> AssignVertexNames( gamma, ["a","b","c"] );
gap> gamma;
rec(
  isGraph := true,
  order := 3,
  group := Group( () ),
  schreierVector := [ -1, -2, -3 ],
  adjacencies := [ [  ], [  ], [  ] ],
  representatives := [ 1, 2, 3 ],
  isSimple := true,
  names := [ "a", "b", "c" ] )
```

64.10 Functions to inspect graphs, vertices and edges

The next sections describe functions to inspect graphs, vertices and edges.

64.11 IsGraph

`IsGraph(` *obj* `)`

This boolean function returns `true` if and only if *obj*, which can be an object of arbitrary type, is a graph.

```
gap> IsGraph( 1 );
false
gap> IsGraph( JohnsonGraph( 3, 2 ) );
true
```

64.12 OrderGraph

`OrderGraph(` *gamma* `)`

This function returns the number of vertices (order) of the graph *gamma*.

```
gap> OrderGraph( JohnsonGraph( 4, 2 ) );
6
```

64.13 IsVertex

`IsVertex(` *gamma*, *v* `)`

This boolean function returns `true` if and only if *v* is vertex of *gamma*.

```
gap> gamma := JohnsonGraph( 3, 2 );;
gap> IsVertex( gamma, 1 );
true
gap> IsVertex( gamma, 4 );
false
```

64.14 VertexName

`VertexName(` *gamma*`,` *v* `)`

This function returns (a copy of) the name of the vertex v of *gamma*.

See also 64.9.

```
gap> VertexName( JohnsonGraph(4,2), 6 );
[ 3, 4 ]
```

64.15 Vertices

`Vertices(` *gamma* `)`

This function returns the vertex set $\{1, ..., gamma.order\}$ of the graph *gamma*.

```
gap> Vertices( JohnsonGraph( 4, 2 ) );
[ 1 .. 6 ]
```

64.16 VertexDegree

`VertexDegree(` *gamma*`,` *v* `)`

This function returns the (out)degree of the vertex v of the graph *gamma*.

```
gap> VertexDegree( JohnsonGraph( 3, 2 ), 1 );
2
```

64.17 VertexDegrees

`VertexDegrees(` *gamma* `)`

This function returns the set of vertex (out)degrees for the graph *gamma*.

```
gap> VertexDegrees( JohnsonGraph( 4, 2 ) );
[ 4 ]
```

64.18 IsLoopy

`IsLoopy(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma* has a **loop**, that is, an edge of the form $[x, x]$.

```
gap> IsLoopy( JohnsonGraph( 4, 2 ) );
false
gap> IsLoopy( CompleteGraph( Group( (1,2,3), (1,2) ), 3 ) );
false
gap> IsLoopy( CompleteGraph( Group( (1,2,3), (1,2) ), 3, true ) );
true
```

64.19 IsSimpleGraph

`IsSimpleGraph(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma* is **simple**, that is, has no loops and whenever $[x, y]$ is an edge then so is $[y, x]$.

```
gap> IsSimpleGraph( CompleteGraph( Group( (1,2,3) ), 3 ) );
true
gap> IsSimpleGraph( CompleteGraph( Group( (1,2,3) ), 3, true ) );
false
```

64.20 Adjacency

`Adjacency(` *gamma*`,` *v* `)`

This function returns (a copy of) the set of vertices of *gamma* adjacent to vertex v. A vertex w is **adjacent** to v if and only if $[v, w]$ is an edge.

```
gap> Adjacency( JohnsonGraph( 4, 2 ), 1 );
[ 2, 3, 4, 5 ]
gap> Adjacency( JohnsonGraph( 4, 2 ), 6 );
[ 2, 3, 4, 5 ]
```

64.21 IsEdge

`IsEdge(` *gamma*`,` *e* `)`

This boolean function returns `true` if and only if e is an edge of *gamma*.

```
gap> IsEdge( JohnsonGraph( 4, 2 ), [ 1, 2 ] );
true
gap> IsEdge( JohnsonGraph( 4, 2 ), [ 1, 6 ] );
false
```

64.22 DirectedEdges

`DirectedEdges(` *gamma* `)`

This function returns the set of directed (ordered) edges of the graph *gamma*.

See also 64.23.

```
gap> gamma := JohnsonGraph( 3, 2 );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,3), (1,2) ),
  schreierVector := [ -1, 2, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ],
  isSimple := true )
```

```
gap> DirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ] ]
gap> UndirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
```

64.23 UndirectedEdges

`UndirectedEdges(` *gamma* `)`

This function returns the set of undirected (unordered) edges of *gamma*, which must be a simple graph.

See also 64.22.

```
gap> gamma := JohnsonGraph( 3, 2 );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,3), (1,2) ),
  schreierVector := [ -1, 2, 1 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ],
  isSimple := true )
gap> DirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ] ]
gap> UndirectedEdges( gamma );
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
```

64.24 Distance

`Distance(` *gamma*, *X*, *Y* `)`
`Distance(` *gamma*, *X*, *Y*, *G* `)`

This function returns the distance from X to Y in *gamma*. The parameters X and Y may be vertices or vertex sets. We define the **distance** $d(X, Y)$ from X to Y to be the minimum length of a (directed) path joining a vertex of X to a vertex of Y if such a path exists, and -1 otherwise.

The optional parameter G, if present, is assumed to be a subgroup of $\mathrm{Aut}\,(gamma)$ fixing X setwise. Including such a G can speed up the function.

```
gap> Distance( JohnsonGraph(4,2), 1, 6 );
2
gap> Distance( JohnsonGraph(4,2), 1, 5 );
1
```

64.25 Diameter

`Diameter(` *gamma* `)`

This function returns the diameter of *gamma*. A diameter of -1 is returned if *gamma* is not (strongly) connected.

```
gap> Diameter( JohnsonGraph( 5, 3 ) );
2
gap> Diameter( JohnsonGraph( 5, 4 ) );
1
```

64.26 Girth

`Girth(` *gamma* `)`

This function returns the girth of *gamma*, which must be a simple graph. A girth of -1 is returned if *gamma* is a forest.

```
gap> Girth( JohnsonGraph( 4, 2 ) );
3
```

64.27 IsConnectedGraph

`IsConnectedGraph(` *gamma* `)`

This boolean function returns `true` if and only if *gamma* is (strongly) **connected**, i.e. if there is a (directed) path from x to y for every pair of vertices x, y of *gamma*.

```
gap> IsConnectedGraph( JohnsonGraph(4,2) );
true
gap> IsConnectedGraph( NullGraph(SymmetricGroup(4)) );
false
```

64.28 IsBipartite

`IsBipartite(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma*, which must be simple, is **bipartite**, i.e. if the vertex set can be partitioned into two null graphs (which are called **bicomponents** or **parts** of *gamma*).

See also 64.41, 64.51, and 64.54.

```
gap> gamma := JohnsonGraph(4,2);
rec(
  isGraph := true,
  order := 6,
  group := Group( (1,5)(2,6), (1,3)(4,6), (2,3)(4,5) ),
  schreierVector := [ -1, 3, 2, 3, 1, 2 ],
  adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ],
      [ 3, 4 ] ],
  isSimple := true )
gap> IsBipartite(gamma);
false
gap> delta := BipartiteDouble(gamma);
rec(
```

```
    isGraph := true,
    order := 12,
    group := Group( ( 1, 5)( 2, 6)( 7,11)( 8,12), ( 1, 3)( 4, 6)( 7, 9)
      (10,12), ( 2, 3)( 4, 5)( 8, 9)(10,11), ( 1, 7)( 2, 8)( 3, 9)
      ( 4,10)( 5,11)( 6,12) ),
    schreierVector := [ -1, 3, 2, 3, 1, 2, 4, 4, 4, 4, 4, 4 ],
    adjacencies := [ [ 8, 9, 10, 11 ] ],
    representatives := [ 1 ],
    isSimple := true,
    names := [ [ [ 1, 2 ], "+" ], [ [ 1, 3 ], "+" ], [ [ 1, 4 ], "+" ],
        [ [ 2, 3 ], "+" ], [ [ 2, 4 ], "+" ], [ [ 3, 4 ], "+" ],
        [ [ 1, 2 ], "-" ], [ [ 1, 3 ], "-" ], [ [ 1, 4 ], "-" ],
        [ [ 2, 3 ], "-" ], [ [ 2, 4 ], "-" ], [ [ 3, 4 ], "-" ] ] )
gap> IsBipartite(delta);
true
```

64.29 IsNullGraph

`IsNullGraph(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma* has no edges.

See also 64.4.

```
gap> IsNullGraph( CompleteGraph( Group(()), 3 ) );
false
gap> IsNullGraph( CompleteGraph( Group(()), 1 ) );
true
```

64.30 IsCompleteGraph

`IsCompleteGraph(` *gamma* `)`
`IsCompleteGraph(` *gamma*`,` *mustloops* `)`

This boolean function returns `true` if and only if the graph *gamma* is a complete graph. The optional boolean parameter *mustloops* determines whether all loops must be present for *gamma* to be considered a complete graph (default: `false` (loops are ignored)).

See also 64.5.

```
gap> IsCompleteGraph( NullGraph( Group(()), 3 ) );
false
gap> IsCompleteGraph( NullGraph( Group(()), 1 ) );
true
gap> IsCompleteGraph( CompleteGraph(SymmetricGroup(3)), true );
false
```

64.31 Functions to determine regularity properties of graphs

The following sections describe functions to determine regularity properties of graphs.

64.32 IsRegularGraph

`IsRegularGraph(` *gamma* `)`

This boolean function returns `true` if and only if the graph *gamma* is (out)regular.

```
gap> IsRegularGraph( JohnsonGraph(4,2) );
true
gap> IsRegularGraph( EdgeOrbitsGraph(Group(()),[[1,2]],2) );
false
```

64.33 LocalParameters

`LocalParameters(` *gamma*`,` *V* `)`
`LocalParameters(` *gamma*`,` *V*`,` *G* `)`

This function determines any **local parameters** $c_i(V)$, $a_i(V)$, or $b_i(V)$ that simple, connected *gamma* may have, with respect to the singleton vertex or vertex set V (see [BCN89]). The function returns a list of triples, whose i-th element is $[c_{i-1}(V), a_{i-1}(V), b_{i-1}(V)]$, except that if some local parameter does not exist then a -1 is put in its place. This function can be used to determine whether a given subset of the vertices of a graph is a distance-regular code in that graph.

The optional parameter G, if present, is assumed to be a subgroup of Aut (*gamma*) fixing V (setwise). Including such a G can speed up the function.

```
gap> LocalParameters( JohnsonGraph(4,2), 1 );
[ [ 0, 0, 4 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> LocalParameters( JohnsonGraph(4,2), [1,6] );
[ [ 0, 0, 4 ], [ 2, 2, 0 ] ]
```

64.34 GlobalParameters

`GlobalParameters(` *gamma* `)`

In a similar way to `LocalParameters` (see 64.33), this function determines the **global parameters** c_i, a_i, b_i of simple, connected *gamma* (see [BCN89]). The nonexistence of a global parameter is denoted by -1.

```
gap> gamma := JohnsonGraph(4,2);;
gap> GlobalParameters( gamma );
[ [ 0, 0, 4 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> GlobalParameters( BipartiteDouble(gamma) );
[ [ 0, 0, 4 ], [ 1, 0, 3 ], [ -1, 0, -1 ], [ 4, 0, 0 ] ]
```

64.35 IsDistanceRegular

`IsDistanceRegular(` *gamma* `)`

This boolean function returns `true` if and only if *gamma* is distance-regular, i.e. *gamma* is simple, connected, and all possible global parameters exist.

```
gap> gamma := JohnsonGraph(4,2);;
```

```
gap> IsDistanceRegular( gamma );
true
gap> IsDistanceRegular( BipartiteDouble(gamma) );
false
```

64.36 CollapsedAdjacencyMat

`CollapsedAdjacencyMat(` *G*, *gamma* `)`

This function returns the collapsed adjacency matrix for *gamma*, where the collapsing group is G. It is assumed that G is a subgroup of Aut (*gamma*).

The (i, j)-entry of the collapsed adjacency matrix equals the number of edges in $\{[x, y] | y \in j$-th G-orbit $\}$, where x is a fixed vertex in the i-th G-orbit.

See also 64.37.

```
gap> gamma := JohnsonGraph(4,2);;
gap> G := Stabilizer( gamma.group, 1 );;
gap> CollapsedAdjacencyMat( G, gamma );
[ [ 0, 4, 0 ], [ 1, 2, 1 ], [ 0, 4, 0 ] ]
```

64.37 OrbitalGraphIntersectionMatrices

`OrbitalGraphIntersectionMatrices(` *G* `)`
`OrbitalGraphIntersectionMatrices(` *G*, *H* `)`

This function returns a sequence of intersection matrices corresponding to the orbital graphs for the transitive permutation group G. An intersection matrix for an orbital graph *gamma* for G is a collapsed adjacency matrix of *gamma*, collapsed with respect to the stabilizer in G of a point.

If the optional parameter H is given then it is assumed to be the stabilizer in G of the point 1, and this information can speed up the function.

See also 64.36.

```
gap> OrbitalGraphIntersectionMatrices( SymmetricGroup(7) );
[ [ [ 1, 0 ], [ 0, 1 ] ], [ [ 0, 6 ], [ 1, 5 ] ] ]
```

64.38 Some special vertex subsets of a graph

The following sections describe functions for special vertex subsets of a graph.

64.39 ConnectedComponent

`ConnectedComponent(` *gamma*, *v* `)`

This function returns the set of all vertices in *gamma* which can be reached by a path starting at the vertex v. The graph *gamma* must be simple.

See also 64.40.

```
gap> ConnectedComponent( NullGraph( Group((1,2)) ), 2 );
[ 2 ]
gap> ConnectedComponent( JohnsonGraph(4,2), 2 );
[ 1, 2, 3, 4, 5, 6 ]
```

64.40 ConnectedComponents

`ConnectedComponents(` *gamma* `)`

This function returns a list of the vertex sets of the connected components of *gamma*, which must be a simple graph.

See also 64.39.

```
gap> ConnectedComponents( NullGraph( Group((1,2,3,4)) ) );
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
gap> ConnectedComponents( JohnsonGraph(4,2) );
[ [ 1, 2, 3, 4, 5, 6 ] ]
```

64.41 Bicomponents

`Bicomponents(` *gamma* `)`

If the graph *gamma*, which must be simple, is bipartite, this function returns a length 2 list of bicomponents or parts of *gamma*, otherwise the empty list is returned.

Note: if *gamma* is not connected then its bicomponents are not necessarily uniquely determined. See also 64.28.

```
gap> Bicomponents( NullGraph(SymmetricGroup(4)) );
[ [ 1, 2, 3 ], [ 4 ] ]
gap> Bicomponents( JohnsonGraph(4,2) );
[  ]
```

64.42 DistanceSet

`DistanceSet(` *gamma*`,` *distances*`,` *V* `)`
`DistanceSet(` *gamma*`,` *distances*`,` *V*`,` *G* `)`

This function returns the set of vertices w of *gamma*, such that $d(V, w)$ is in *distances* (a list or singleton distance).

The optional parameter G, if present, is assumed to be a subgroup of Aut (*gamma*) fixing V setwise. Including such a G can speed up the function.

```
gap> DistanceSet( JohnsonGraph(4,2), 1, [1,6] );
[ 2, 3, 4, 5 ]
```

64.43 Layers

`Layers(` *gamma*`,` *V* `)`
`Layers(` *gamma*`,` *V*`,` *G* `)`

This function returns a list whose i-th element is the set of vertices of *gamma* at distance $i-1$ from V, which may be a vertex set or a singleton vertex.

The optional parameter G, if present, is assumed to be a subgroup of Aut (*gamma*) which fixes V setwise. Including such a G can speed up the function.

```
gap> Layers( JohnsonGraph(4,2), 6 );
[ [ 6 ], [ 2, 3, 4, 5 ], [ 1 ] ]
```

64.44 IndependentSet

`IndependentSet(` *gamma* `)`
`IndependentSet(` *gamma*, *indset* `)`
`IndependentSet(` *gamma*, *indset*, *forbidden* `)`

Returns a (hopefully large) independent set (coclique) of the graph *gamma*, which must be simple. At present, a **greedy** algorithm is used. The returned independent set will contain the (assumed) independent set *indset* (default: `[]`), and not contain any element of *forbidden* (default: `[]`, in which case the returned independent set is maximal). An error is signalled if *indset* and *forbidden* have non-trivial intersection.

```
gap> IndependentSet( JohnsonGraph(4,2), [3] );
[ 3, 4 ]
```

64.45 Functions to construct new graphs from old

The following sections describe functions to construct new graphs from old ones.

64.46 InducedSubgraph

`InducedSubgraph(` *gamma*, *V* `)`
`InducedSubgraph(` *gamma*, *V*, *G* `)`

This function returns the subgraph of *gamma* induced on the vertex list V (which must not contain repeated elements). If the optional third parameter G is given, then it is assumed that G fixes V setwise, and is a group of automorphisms of the induced subgraph when restriced to V. This knowledge is then used to give an associated group to the induced subgraph. If no such G is given then the associated group is trivial.

```
gap> gamma := JohnsonGraph(4,2);;
gap> S := [2,3,4,5];;
gap> InducedSubgraph( gamma, S, Stabilizer(gamma.group,S,OnSets) );
rec(
  isGraph := true,
  order := 4,
  group := Group( (2,3), (1,2)(3,4) ),
  schreierVector := [ -1, 2, 1, 2 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
```

64.47 DistanceSetInduced

`DistanceSetInduced(` *gamma*, *distances*, *V* `)`
`DistanceSetInduced(` *gamma*, *distances*, *V*, *G* `)`

This function returns the subgraph of *gamma* induced on the set of vertices w of *gamma* such that $d(V, w)$ is in *distances* (a list or singleton distance).

The optional parameter G, if present, is assumed to be a subgroup of Aut ($gamma$) fixing V setwise. Including such a G can speed up the function.

```
gap> DistanceSetInduced( JohnsonGraph(4,2), [0,1], [1] );
rec(
  isGraph := true,
  order := 5,
  group := Group( (2,3)(4,5), (2,5)(3,4) ),
  schreierVector := [ -1, -2, 1, 2, 2 ],
  adjacencies := [ [ 2, 3, 4, 5 ], [ 1, 3, 4 ] ],
  representatives := [ 1, 2 ],
  isSimple := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
```

64.48 DistanceGraph

`DistanceGraph(` *gamma*, *distances* `)`

This function returns the graph *delta*, with the same vertex set as *gamma*, such that $[x, y]$ is an edge of *delta* if and only if $d(x, y)$ (in *gamma*) is in the list *distances*.

```
gap> DistanceGraph( JohnsonGraph(4,2), [2] );
rec(
  isGraph := true,
  order := 6,
  group := Group( (1,5)(2,6), (1,3)(4,6), (2,3)(4,5) ),
  schreierVector := [ -1, 3, 2, 3, 1, 2 ],
  adjacencies := [ [ 6 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ],
      [ 3, 4 ] ],
  isSimple := true )
gap> ConnectedComponents(last);
[ [ 1, 6 ], [ 2, 5 ], [ 3, 4 ] ]
```

64.49 ComplementGraph

`ComplementGraph(` *gamma* `)`
`ComplementGraph(` *gamma*, *comploops* `)`

This function returns the complement of the graph *gamma*. The optional boolean parameter *comploops* determines whether or not loops/nonloops are complemented (default: `false` (loops/nonloops are not complemented)).

```
gap> ComplementGraph( NullGraph(SymmetricGroup(3)) );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,3), (2,3) ),
  schreierVector := [ -1, 2, 1 ],
  adjacencies := [ [ 2, 3 ] ],
```

```
  representatives := [ 1 ],
  isSimple := true )
gap> IsLoopy(last);
false
gap> IsLoopy(ComplementGraph(NullGraph(SymmetricGroup(3)),true));
true
```

64.50 PointGraph

`PointGraph(` *gamma* `)`
`PointGraph(` *gamma*, *v* `)`

Assuming that *gamma* is simple, connected, and bipartite, this function returns the induced subgraph on the connected component of `DistanceGraph(`*gamma*`,2)` containing the vertex v (default: $v = 1$).

Thus, if *gamma* is the incidence graph of a connected geometry, and v represents a point, then the point graph of the geometry is returned.

```
gap> BipartiteDouble( CompleteGraph(SymmetricGroup(4)) );;
gap> PointGraph(last);
rec(
  isGraph := true,
  order := 4,
  group := Group( (3,4), (2,4), (1,4) ),
  schreierVector := [ -1, 2, 1, 3 ],
  adjacencies := [ [ 2, 3, 4 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ 1, "+" ], [ 2, "+" ], [ 3, "+" ], [ 4, "+" ] ] )
gap> IsCompleteGraph(last);
true
```

64.51 EdgeGraph

`EdgeGraph(` *gamma* `)`

This function returns the edge graph, also called the line graph, of the simple graph *gamma*.

This **edge graph** *delta* has the unordered edges of *gamma* as vertices, and e is joined to f in *delta* precisely when $|e \cap f| = 1$.

```
gap> EdgeGraph( CompleteGraph(SymmetricGroup(5)) );
rec(
  isGraph := true,
  order := 10,
  group := Group( ( 1, 7)( 2, 9)( 3,10), ( 1, 4)( 5, 9)( 6,10),
    ( 2, 4)( 5, 7)( 8,10), ( 3, 4)( 6, 7)( 8, 9) ),
  schreierVector := [ -1, 3, 4, 2, 3, 4, 1, 4, 2, 2 ],
  adjacencies := [ [ 2, 3, 4, 5, 6, 7 ] ],
  representatives := [ 1 ],
  isSimple := true,
```

```
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ],
      [ 2, 4 ], [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ] )
```

64.52 UnderlyingGraph

`UnderlyingGraph(` *gamma* `)`

This function returns the underlying graph *delta* of *gamma*. The graph *delta* has the same vertex set as *gamma*, and has an edge $[x, y]$ precisely when *gamma* has an edge $[x, y]$ or an edge $[y, x]$. This function also sets the `isSimple` components of *gamma* and *delta*.

```
gap> gamma := EdgeOrbitsGraph( Group((1,2,3,4)), [1,2] );
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,2,3,4) ),
  schreierVector := [ -1, 1, 1, 1 ],
  adjacencies := [ [ 2 ] ],
  representatives := [ 1 ],
  isSimple := false )
gap> UnderlyingGraph(gamma);
rec(
  isGraph := true,
  order := 4,
  group := Group( (1,2,3,4) ),
  schreierVector := [ -1, 1, 1, 1 ],
  adjacencies := [ [ 2, 4 ] ],
  representatives := [ 1 ],
  isSimple := true )
```

64.53 QuotientGraph

`QuotientGraph(` *gamma*, *R* `)`

Let S be the smallest *gamma.group*-invariant equivalence relation on the vertices of *gamma*, such that S contains the relation R (which should be a list of ordered pairs (length 2 lists) of vertices of *gamma*). Then this function returns a graph isomorphic to the quotient *delta* of the graph *gamma*, defined as follows. The vertices of *delta* are the equivalence classes of S, and $[X, Y]$ is an edge of *delta* if and only if $[x, y]$ is an edge of *gamma* for some $x \in X$, $y \in Y$.

```
gap> gamma := JohnsonGraph(4,2);;
gap> QuotientGraph( gamma, [[1,6]] );
rec(
  isGraph := true,
  order := 3,
  group := Group( (1,2), (1,3), (2,3) ),
  schreierVector := [ -1, 1, 2 ],
  adjacencies := [ [ 2, 3 ] ],
  representatives := [ 1 ],
```

```
  isSimple := true,
  names := [ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 3 ], [ 2, 4 ] ],
      [ [ 1, 4 ], [ 2, 3 ] ] ] )
```

64.54 BipartiteDouble

`BipartiteDouble(` *gamma* `)`

This function returns the bipartite double of the graph *gamma*, as defined in [BCN89].

```
gap> gamma := JohnsonGraph(4,2);
rec(
  isGraph := true,
  order := 6,
  group := Group( (1,5)(2,6), (1,3)(4,6), (2,3)(4,5) ),
  schreierVector := [ -1, 3, 2, 3, 1, 2 ],
  adjacencies := [ [ 2, 3, 4, 5 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ],
      [ 3, 4 ] ],
  isSimple := true )
gap> IsBipartite(gamma);
false
gap> delta := BipartiteDouble(gamma);
rec(
  isGraph := true,
  order := 12,
  group := Group( ( 1, 5)( 2, 6)( 7,11)( 8,12), ( 1, 3)( 4, 6)( 7, 9)
    (10,12), ( 2, 3)( 4, 5)( 8, 9)(10,11), ( 1, 7)( 2, 8)( 3, 9)
    ( 4,10)( 5,11)( 6,12) ),
  schreierVector := [ -1, 3, 2, 3, 1, 2, 4, 4, 4, 4, 4, 4 ],
  adjacencies := [ [ 8, 9, 10, 11 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ [ 1, 2 ], "+" ], [ [ 1, 3 ], "+" ], [ [ 1, 4 ], "+" ],
      [ [ 2, 3 ], "+" ], [ [ 2, 4 ], "+" ], [ [ 3, 4 ], "+" ],
      [ [ 1, 2 ], "-" ], [ [ 1, 3 ], "-" ], [ [ 1, 4 ], "-" ],
      [ [ 2, 3 ], "-" ], [ [ 2, 4 ], "-" ], [ [ 3, 4 ], "-" ] ] )
gap> IsBipartite(delta);
true
```

64.55 GeodesicsGraph

`GeodesicsGraph(` *gamma*`,` *x*`,` *y* `)`

This function returns the the graph induced on the set of geodesics between vertices x and y, but not including x or y. This function is only for a simple graph *gamma*.

```
gap> GeodesicsGraph( JohnsonGraph(4,2), 1, 6 );
rec(
```

```
    isGraph := true,
    order := 4,
    group := Group( (1,3)(2,4), (1,4)(2,3), (1,3,4,2) ),
    schreierVector := [ -1, 2, 1, 2 ],
    adjacencies := [ [ 2, 3 ] ],
    representatives := [ 1 ],
    isSimple := true,
    names := [ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ] )
  gap> GlobalParameters(last);
  [ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
```

64.56 CollapsedIndependentOrbitsGraph

`CollapsedIndependentOrbitsGraph(` *G*, *gamma* `)`
`CollapsedIndependentOrbitsGraph(` *G*, *gamma*, *N* `)`

Given a subgroup G of the automorphism group of the graph *gamma*, this function returns a graph isomorphic to *delta*, defined as follows. The vertices of *delta* are those G-orbits of the vertices of *gamma* that are independent sets, and x is **not** joined to y in *delta* if and only if $x \cup y$ is an independent set in *gamma*.

If the optional parameter N is given, then it is assumed to be a subgroup of Aut (*gamma*) preserving the set of G-orbits of the vertices of *gamma* (for example, the normalizer in *gamma.group* of G). This information can make the function more efficient.

```
  gap> G := Group( (1,2) );;
  gap> gamma := NullGraph( SymmetricGroup(3) );;
  gap> CollapsedIndependentOrbitsGraph( G, gamma );
  rec(
    isGraph := true,
    order := 2,
    group := Group( () ),
    schreierVector := [ -1, -2 ],
    adjacencies := [ [  ], [  ] ],
    representatives := [ 1, 2 ],
    isSimple := true,
    names := [ [ 1, 2 ], [ 3 ] ] )
```

64.57 CollapsedCompleteOrbitsGraph

`CollapsedCompleteOrbitsGraph(` *G*, *gamma* `)`
`CollapsedCompleteOrbitsGraph(` *G*, *gamma*, *N* `)`

Given a subgroup G of the automorphism group of the simple graph *gamma*, this function returns a graph isomorphic to *delta*, defined as follows. The vertices of *delta* are those G-orbits of the vertices of *gamma* on which complete subgraphs are induced in *gamma*, and x is joined to y in *delta* if and only if $x \neq y$ and the subgraph of *gamma* induced on $x \cup y$ is a complete graph.

If the optional parameter N is given, then it is assumed to be a subgroup of Aut (*gamma*) preserving the set of G-orbits of the vertices of *gamma* (for example, the normalizer in *gamma.group* of G). This information can make the function more efficient.

```
gap> G := Group( (1,2) );;
gap> gamma := NullGraph( SymmetricGroup(3) );;
gap> CollapsedCompleteOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 1,
  group := Group( () ),
  schreierVector := [ -1 ],
  adjacencies := [ [  ] ],
  representatives := [ 1 ],
  names := [ [ 3 ] ],
  isSimple := true )
gap> gamma := CompleteGraph( SymmetricGroup(3) );;
gap> CollapsedCompleteOrbitsGraph( G, gamma );
rec(
  isGraph := true,
  order := 2,
  group := Group( () ),
  schreierVector := [ -1, -2 ],
  adjacencies := [ [ 2 ], [ 1 ] ],
  representatives := [ 1, 2 ],
  names := [ [ 1, 2 ], [ 3 ] ],
  isSimple := true )
```

64.58 NewGroupGraph

`NewGroupGraph(` *G*, *gamma* `)`

This function returns a copy *delta* of *gamma*, except that the group associated with *delta* is G, which is a assumed to be a subgroup of $\mathrm{Aut}\,(delta)$.

Note that the result of some functions of a graph depend on the group associated with that graph (which must always be a subgroup of the automorphism group of the graph).

```
gap> gamma := JohnsonGraph(4,2);;
gap> aut := AutGroupGraph(gamma);
Group( (3,4), (2,3)(4,5), (1,2)(5,6) )
gap> Size(gamma.group);
24
gap> Size(aut);
48
gap> delta := NewGroupGraph( aut, gamma );;
gap> Size(delta.group);
48
gap> IsIsomorphicGraph( gamma, delta );
true
```

64.59 Vertex-Colouring and Complete Subgraphs

The following sections describe functions for vertex-colouring or constructing complete subgraphs of given graphs.

64.60 VertexColouring

`VertexColouring(` *gamma* `)`

This function returns a proper vertex-colouring C for the graph *gamma*, which must be simple.

This **proper vertex-colouring** C is a list of natural numbers, indexed by the vertices of *gamma*, and has the property that $C[i] \neq C[j]$ whenever $[i,j]$ is an edge of *gamma*. At present a **greedy** algorithm is used.

```
gap> VertexColouring( JohnsonGraph(4,2) );
[ 1, 2, 3, 3, 2, 1 ]
```

64.61 CompleteSubgraphs

`CompleteSubgraphs(` *gamma* `)`
`CompleteSubgraphs(` *gamma*`,` *k* `)`
`CompleteSubgraphs(` *gamma*`,` *k*`,` *alls* `)`

This function returns a set K of complete subgraphs of *gamma*, which must be a simple graph. A complete subgraph is represented by its vertex set. If $k > -1$ then the elements of K each have size k, otherwise the elements of K represent maximal complete subgraphs of *gamma*. The default for k is -1, i.e. maximal complete subgraphs.

The optional boolean parameter *alls* controls how many complete subgraphs are returned. If *alls* is `true` (the default), then K will contain (perhaps properly) a set of *gamma.group* orbit-representatives of the size k (if $k > -1$) or maximal (if $k < 0$) complete subgraphs of *gamma*.

If *alls* is `false` then K will contain at most one element. In this case, if $k < 0$ then K will contain just one maximal complete subgraph, and if $k > -1$ then K will contain a complete subgraph of size k if and only if such a subgraph is contained in *gamma*.

```
gap> gamma := JohnsonGraph(5,2);;
gap> CompleteSubgraphs(gamma);
[ [ 1, 2, 3, 4 ], [ 1, 2, 5 ] ]
gap> CompleteSubgraphs(gamma,2,false);
[ [ 1, 2 ] ]
```

64.62 CompleteSubgraphsOfGivenSize

`CompleteSubgraphsOfGivenSize(` *gamma*`,` *k* `)`
`CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls* `)`
`CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls*`,` *maxi* `)`
`CompleteSubgraphsOfGivenSize(` *gamma*`,` *k*`,` *alls*`,` *maxi*`,` *colnum* `)`

Let *gamma* be a simple graph and $k > 0$. This function returns a set K of complete subgraphs of size k of *gamma*, if such subgraphs exist (else the empty set is returned). A complete subgraph is represented by its vertex set. This function is more efficient for its purpose than the more general function `CompleteSubgraphs`.

The boolean parameter *alls* is used to control how many complete subgraphs are returned. If *alls* is true (the default), then K will contain (perhaps properly) a set of *gamma.group* orbit-representatives of the size k complete subgraphs of *gamma*. If *alls* is false then K will contain at most one element, and will contain one element if and only if *gamma* contains a complete subgraph of size k.

If the boolean parameter *maxi* is bound and has value true, then it is assumed that all complete subgraphs of size k of *gamma* are maximal.

If the optional rational parameter *colnum* is given, then a sensible value is

`OrderGraph(`*gamma*`)/Length(Set(VertexColouring(`*gamma*`)))`.

The use of this parameter may speed up the function.

```
gap> gamma := JohnsonGraph(5,2);;
gap> CompleteSubgraphsOfGivenSize(gamma,5);
[  ]
gap> CompleteSubgraphsOfGivenSize(gamma,4,true,true);
[ [ 1, 2, 3, 4 ] ]
gap> gamma := NewGroupGraph( Group(()), gamma );;
gap> CompleteSubgraphsOfGivenSize(gamma,4,true,true);
[ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 2, 5, 8, 9 ], [ 3, 6, 8, 10 ],
  [ 4, 7, 9, 10 ] ]
```

64.63 Functions depending on nauty

For convenience, GRAPE provides a (somewhat primitive) interface to Brendan McKay's *nauty* (Version 1.7) package (see [McK90]) for calculating automorphism groups of vertex-coloured graphs, and for testing graph isomorphism.

64.64 AutGroupGraph

`AutGroupGraph(` *gamma* `)`
`AutGroupGraph(` *gamma*`,` *colouring* `)`

The first version of this function returns the automorphism group of the (directed) graph *gamma*, using *nauty*.

In the second version, *colouring* is a vertex-colouring of *gamma*, and the subgroup of Aut (*gamma*) preserving this colouring is returned. Here, a colouring should be given as a list of sets, forming a partion of the vertices. The set for the last colour may be omitted. Note that we do not require that adjacent vertices have different colours.

```
gap> gamma := JohnsonGraph(4,2);;
gap> Size(AutGroupGraph(gamma));
48
gap> Size(AutGroupGraph(gamma,[[1,6]]));
16
```

64.65 IsIsomorphicGraph

`IsIsomorphicGraph(` *gamma1*`,` *gamma2* `)`

This boolean function uses the *nauty* program to test the isomorphism of *gamma1* with *gamma2*. The value `true` is returned if and only if the graphs are isomorphic (as directed, uncoloured graphs).

This function creates or uses the record component `canonicalLabelling` of a graph. As noted in [McK90], a canonical labelling given by *nauty* can depend on the version of *nauty* (Version 1.7 in GRAPE 2.31), certain parameters of *nauty* (always set the same by GRAPE 2.31), and the compiler and computer used. If you use the `canonicalLabelling` component (say by using `IsIsomorphicGraph`) of a graph stored on a file, then you must be sure that this field was created in the same environment in which you are presently computing. If in doubt, unbind the `canonicalLabelling` component of the graph before testing isomorphism.

```
gap> gamma := JohnsonGraph(7,4);;
gap> delta := JohnsonGraph(7,3);;
gap> IsIsomorphicGraph( gamma, delta );
true
```

64.66 An example

We conclude this chapter with a simple example to illustrate further the use of GRAPE.

In this example we construct the Petersen graph P, and its edge graph (often called line graph) EP. We compute the (global) parameters of EP, and so verify that EP is distance-regular (see [BCN89]). We also show that there is just one orbit of 1-factors of P under the automorphism group of P (but you should read the documentation of the function `CompleteSubgraphsOfGivenSize` to see exactly what that function does).

```
gap> P := Graph( SymmetricGroup(5), [[1,2]], OnSets,
>          function(x,y) return Intersection(x,y)=[]; end );
rec(
  isGraph := true,
  order := 10,
  group := Group( ( 1, 2)( 6, 8)( 7, 9), ( 1, 3)( 4, 8)( 5, 9),
    ( 2, 4)( 3, 6)( 9,10), ( 2, 5)( 3, 7)( 8,10) ),
  schreierVector := [ -1, 1, 2, 3, 4, 3, 4, 2, 2, 4 ],
  adjacencies := [ [ 8, 9, 10 ] ],
  representatives := [ 1 ],
  names := [ [ 1, 2 ], [ 2, 5 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 1, 3 ], [ 1, 4 ], [ 3, 5 ], [ 4, 5 ], [ 3, 4 ] ] )
gap> Diameter(P);
2
gap> Girth(P);
5
gap> EP := EdgeGraph(P);
rec(
  isGraph := true,
  order := 15,
  group := Group( ( 1, 4)( 2, 5)( 3, 6)(10,11)(12,13)(14,15), ( 1, 7)
    ( 2, 8)( 3, 9)(10,15)(11,13)(12,14), ( 2, 3)( 4, 7)( 5,10)( 6,11)
    ( 8,12)( 9,14), ( 1, 3)( 4,12)( 5, 8)( 6,13)( 7,10)( 9,15) ),
```

```
  schreierVector := [ -1, 3, 4, 1, 3, 1, 2, 3, 2, 4, 1, 4, 1, 2, 2 ],
  adjacencies := [ [ 2, 3, 13, 15 ] ],
  representatives := [ 1 ],
  isSimple := true,
  names := [ [ [ 1, 2 ], [ 3, 5 ] ], [ [ 1, 2 ], [ 4, 5 ] ],
      [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 3 ], [ 2, 5 ] ],
      [ [ 1, 4 ], [ 2, 5 ] ], [ [ 2, 5 ], [ 3, 4 ] ],
      [ [ 1, 5 ], [ 2, 3 ] ], [ [ 1, 5 ], [ 2, 4 ] ],
      [ [ 1, 5 ], [ 3, 4 ] ], [ [ 1, 4 ], [ 2, 3 ] ],
      [ [ 2, 3 ], [ 4, 5 ] ], [ [ 1, 3 ], [ 2, 4 ] ],
      [ [ 2, 4 ], [ 3, 5 ] ], [ [ 1, 3 ], [ 4, 5 ] ],
      [ [ 1, 4 ], [ 3, 5 ] ] ] )
gap> GlobalParameters(EP);
[ [ 0, 0, 4 ], [ 1, 1, 2 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> CompleteSubgraphsOfGivenSize(ComplementGraph(EP),5);
[ [ 1, 5, 9, 11, 12 ] ]
```

Chapter 65

GRIM (Groups of Rational and Integer Matrices)

This chapter describes the main functions of the GRIM(Version 1.0) share library package for testing finiteness of rational and integer matrix groups. All functions described here are written entirely in the GAP3 language.

Before using any of the functions described in this chapter you must load the package by calling the statement

```
gap> RequirePackage( "grim" );

Loading  GRIM (Groups of Rational and Integer Matrices) 1.0,
by beals@math.arizona.edu
```

65.1 Functions to test finiteness and integrality

The following sections describe the functions used to test finiteness and integrality of rational matrix groups.

65.2 IsFinite for rational matrix groups

`IsFinite(` G `)`

The group G, which must consist of rational matrices, is tested for finiteness.

A group of rational matrices is finite iff the following two conditions hold: There is a basis with respect to which all elements of G have integer entries, and G preserves a positive definite quadratic form.

If G contains non-integer matrices, then IsFinite first calls InvariantLattice (see 65.3) to find a basis with respect to which all elements of G are integer matrices.

IsFinite then finds a positive definite quadratic form, or determines that none exists. If G is finite, then the quadratic form is stored in G.quadraticForm.

```
gap> a := [[1,1/2],[0,-1]];; G := Group(a);;
```

```
gap> IsFinite(G);
true
gap> L := G.invariantLattice;;
gap> L*a*L^(-1);
[ [ 1, 1 ], [ 0, -1 ] ]
gap> B := G.quadraticForm;
[ [ 4, 1 ], [ 1, 3/2 ] ]
gap> TransposedMat(a)*B*a;
[ [ 4, 1 ], [ 1, 3/2 ] ]
```

This function is Las Vegas: it is randomized, but the randomness only affects the running time, not the correctness of the output. (See 65.4.)

65.3 InvariantLattice for rational matrix groups

`InvariantLattice(` G `)`

This function returns a lattice L (given by a basis) which is G-invariant. That is, for any A in G, LAL^{-1} is an integer matrix.

L is also stored in G.invariantLattice, and the conjugate group LGL^{-1} is stored in G.integerMatrixGroup.

This function finds an L unless G contains elements of non-integer trace (in which case no such L exists, and *false* is returned).

```
gap> a := [[1,1/2],[0,-1]];; G := Group(a);;
gap> L := InvariantLattice(G);;
gap> L*a*L^(-1);
[ [ 1, 1 ], [ 0, -1 ] ]
```

This function is Las Vegas: it is randomized, but the randomization only affects the running time, not the correctness of the output.

65.4 IsFiniteDeterministic for integer matrix groups

`IsFiniteDeterministic(` G `)`

The integer matrix group G is tested for finiteness, using a deterministic algorithm. In most cases, this seems to be less efficient than the Las Vegas IsFinite. However, the number of arithmetic steps of this algorithm does not depend on the size of the entries of G, which is not true of the Las Vegas version.

If G is finite, then a G-invariant positive definite quadratic form is stored in G.quadraticForm.

```
gap> a := [[1,1],[0,-1]];
[ [ 1, 1 ], [ 0, -1 ] ]
gap> G := Group(a);;
gap> IsFiniteDeterministic(G);
true
gap> B := G.quadraticForm;;
gap> B;
[ [ 1, 1/2 ], [ 1/2, 3/2 ] ]
gap> TransposedMat(a)*B*a;
[ [ 1, 1/2 ], [ 1/2, 3/2 ] ]
```

See also (65.2).

Chapter 66

GUAVA

GUAVA is a share library package that implements coding theory algorithms in GAP3. Codes can be created and manipulated and information about codes can be calculated.

GUAVA consists of various files written in the GAP3 language, and an external program from J.S. Leon for dealing with automorphism groups of codes and isomorphism testing functions. Several algorithms that need the speed are integrated in the GAP3 kernel. Please send your bug reports to the gap-forum (`GAP-Forum@Math.RWTH-Aachen.DE`).

GUAVA is written as a final project during our study of Mathematics at the Delft University of Technology, department of Pure Mathematics, and in Aachen, at Lehrstuhl D fuer Mathematik.

We would like to thank the GAP3 people at the RWTH Aachen for their support, A.E. Brouwer for his advice and J. Simonis for his supervision.

Jasper Cramwinckel,
Erik Roijackers, and
Reinald Baart.

Delft University of Technology
Faculty of Technical Mathematics and Informatics
Department of Pure Mathematics

As of version 1.3, new functions are added. These functions are also written in Delft as a final project during my study of Mathematics. For more information, see 66.141.

Eric Minkes.

The following sections describe the functions of the GUAVA (Version 1.3) share libary package for computing with codes. All functions described here are written entirely in the GAP3 language, except for the automorphism group and isomorphism testing functions, which make use of J.S. Leon's partition backtrack programs.

GUAVA is primarily designed for the construction and analysis of codes. The functions can be divided into three subcategories:

Construction of codes
: GUAVA can construct **unrestricted**, **linear** and **cyclic codes**. Information about the code is stored in a record, together with operations applicable to the code.

Manipulations of codes
Manipulation transforms one code into another, or constructs a new code from two codes. The new code can profit from the data in the record of the old code(s), so in these cases calculation time decreases.

Computations of information about codes
GUAVA can calculate important data of codes very fast. The results are stored in the code record.

66.1 Loading GUAVA

After starting up GAP3, the GUAVA package needs to be loaded. Load GUAVA by typing at the GAP3 prompt

```
gap> RequirePackage( "guava" );
```

```
                                 |
   -----------
  /            \           /   --+--  Version 1.3
 /      |    | |\\        //|    |
|    _  |    | | \\      // |
|     \ |    | |--\\    //--|     Jasper Cramwinckel
 \     ||    | |   \\  //   |     Erik Roijackers
  \___/  \___/ |    \\//    |     Reinald Baart
                                  Eric Minkes
```

If GUAVA isn't already in memory, it is loaded and its beautiful banner is displayed.

If you are a frequent user of GUAVA, you might consider putting this line in your `.gaprc` file.

66.2 Codewords

A **codeword** is basically just a vector of finite field elements. In GUAVA, a codeword is a record, with this base vector as its most important element.

Codewords have been implemented in GUAVA mainly because of their easy interfacing with the user. The user can input codewords in different formats, and output information is formatted in a readable way.

Codewords work together with codes (see 66.15), although many operations are available on codewords themselves.

The first sections describe how codewords are constructed (see 66.3 and 66.4).

The next sections describe the operations applicable to codewords (see 66.5 and 66.6).

The next sections describe the functions that convert codewords back to vectors or polynomials (see 66.7 and 66.8), and functions that change the way a codeword is displayed (see 66.9 and 66.10).

The next section describes a single function to generate a null word (see 66.11).

The next sections describe the functions for codewords (see 66.12, 66.13 and 66.14).

66.3 Codeword

`Codeword(` *obj* `[,` *n*`] [,` *F*`] )`

`Codeword` returns a codeword or a list of codewords constructed from *obj*. The object *obj* can be a vector, a string, a polynomial or a codeword. It may also be a list of those (even a mixed list).

If a number n is specified, all constructed codewords have length n. This is the only way to make sure that all elements of *obj* are converted to codewords of the same length. Elements of *obj* that are longer than n are reduced in length by cutting of the last positions. Elements of *obj* that are shorter than n are lengthened by adding zeros at the end. If no n is specified, each constructed codeword is handled individually.

If a Galois field F is specified, all codewords are constructed over this field. This is the only way to make sure that all elements of obj are converted to the same field F (otherwise they are converted one by one). Note that all elements of *obj* must have elements over F or over `Integers`. Converting from one Galois field to another is not allowed. If no F is specified, vectors or strings with integer elements will be converted to the smallest Galois field possible.

Note that a significant speed increase is achieved if F is specified, even when all elements of *obj* already have elements over F.

Every vector in *obj* can be a finite field vector over F or a vector over `Integers`. In the last case, it is converted to F or, if omitted, to the smallest Galois field possible.

Every string in *obj* must be a string of numbers, without spaces, commas or any other characters. These numbers must be from 0 to 9. The string is converted to a codeword over F or, if F is omitted, over the smallest Galois field possible. Note that since all numbers in the string are interpreted as one-digit numbers, Galois fields of size larger than 10 are not properly represented when using strings.

Every polynomial in *obj* is converted to a codeword of length n or, if omitted, of a length dictated by the degree of the polynomial. If F is specified, a polynomial in *obj* must be over F.

Every element of *obj* that is already a codeword is changed to a codeword of length n. If no n was specified, the codeword doesn't change. If F is specified, the codeword must have base field F.

```
gap> c := Codeword([0,1,1,1,0]);
[ 0 1 1 1 0 ]
gap> Field(c);
GF(2)
gap> c2 := Codeword([0,1,1,1,0], GF(3));
[ 0 1 1 1 0 ]
gap> Field(c2);
GF(3)
gap> Codeword([c, c2, "0110"]);
[ [ 0 1 1 1 0 ], [ 0 1 1 1 0 ], [ 0 1 1 0 ] ]
gap> p := Polynomial(GF(2), [Z(2)^0, 0*Z(2), Z(2)^0]);
Z(2)^0*(X(GF(2))^2 + 1)
gap> Codeword(p);
```

```
x^2 + 1
```

`Codeword(` *obj*`,` *C* `)`

In this format, the elements of *obj* are converted to elements of the same vector space as the elements of a code C. This is the same as calling `Codeword` with the word length of C (which is n) and the field of C (which is F).

```
gap> C := WholeSpaceCode(7,GF(5));
a cyclic [7,7,1]0 whole space code over GF(5)
gap> Codeword(["0220110", [1,1,1]], C);
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ]
gap> Codeword(["0220110", [1,1,1]], 7, GF(5));
[ [ 0 2 2 0 1 1 0 ], [ 1 1 1 0 0 0 0 ] ]
```

66.4 IsCodeword

`IsCodeword(` *obj* `)`

`IsCodeword` returns `true` if *obj*, which can be an object of arbitrary type, is of the codeword type and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsCodeword(1);
false
gap> IsCodeword(ReedMullerCode(2,3));
false
gap> IsCodeword("11111");
false
gap> IsCodeword(Codeword("11111"));
true
```

66.5 Comparisons of Codewords

c_1 `=` c_2
c_1 `<>` c_2

The equality operator `=` evaluates to `true` if the codewords c_1 and c_2 are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the codewords c_1 and c_2 are not equal, and to `false` otherwise.

Note that codewords are equal if and only if their base vectors are equal. Whether they are represented as a vector or polynomial has nothing to do with the comparison.

Comparing codewords with objects of other types is not recommended, although it is possible. If c_2 is the codeword, the other object c_1 is first converted to a codeword, after which comparison is possible. This way, a codeword can be compared with a vector, polynomial, or string. If c_1 is the codeword, then problems may arise if c_2 is a polynomial. In that case, the comparison always yields a `false`, because the polynomial comparison is called (see `Comparisons of Polynomials`).

```
gap> P := Polynomial(GF(2), Z(2)*[1,0,0,1]);
Z(2)^0*(X(GF(2))^3 + 1)
gap> c := Codeword(P, GF(2));
x^3 + 1
```

```
gap> P = c;          #  codeword operation
true
gap> c = P;          #  polynomial operation
false
gap> c2 := Codeword("1001", GF(2));
[ 1 0 0 1 ]
gap> c = c2;
true
```

66.6 Operations for Codewords

The following operations are always available for codewords. The operands must have a common base field, and must have the same length. No implicit conversions are performed.

c_1 `+` c_2

The operator `+` evaluates to the sum of the codewords c_1 and c_2.

c_1 `-` c_2

The operator `-` evaluates to the difference of the codewords c_1 and c_2.

C `+` c
c `+` C

The operator `+` evaluates to the coset code of code C after adding a codeword c to all codewords. See 66.101.

In general, the operations just described can also be performed on vectors, strings or polynomials, although this is not recommended. The vector, string or polynomial is first converted to a codeword, after which the normal operation is performed. For this to go right, make sure that at least one of the operands is a codeword. Further more, it will not work when the right operand is a polynomial. In that case, the polynomial operations (`FiniteFieldPolynomialOps`) are called, instead of the codeword operations (`CodewordOps`).

Some other code-oriented operations with codewords are described in 66.20.

66.7 VectorCodeword

`VectorCodeword(` *obj* `[,` n`] [,` F`] )`
`VectorCodeword(` *obj*`,` C `)`

`VectorCodeword` returns a vector or a list of vectors of elements of a Galois field, converted from *obj*. The object *obj* can be a vector, a string, a polynomial or a codeword. It may also be a list of those (even a mixed list).

In fact, the object *obj* is treated the same as in the function `Codeword` (see 66.3).

```
gap> a := Codeword("011011");; VectorCodeword(a);
[ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> VectorCodeword( [ 0, 1, 2, 1, 2, 1 ] );
[ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3), Z(3)^0 ]
gap> VectorCodeword( [ 0, 0, 0, 0], GF(9) );
[ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
```

66.8 PolyCodeword

`PolyCodeword(` *obj* `[,` *n*`] [,` *F*`] )`
`PolyCodeword(` *obj*`,` *C* `)`

`PolyCodeword` returns a polynomial or a list of polynomials over a Galois field, converted from *obj*. The object *obj* can be a vector, a string, a polynomial or a codeword. It may also be a list of those (even a mixed list).

In fact, the object *obj* is treated the same as in the function `Codeword` (see 66.3).

```
gap> a := Codeword("011011");; PolyCodeword(a);
Z(2)^0*(X(GF(2))^5 + X(GF(2))^4 + X(GF(2))^2 + X(GF(2)))
gap> PolyCodeword( [ 0, 1, 2, 1, 2 ] );
Z(3)^0*(2*X(GF(3))^4 + X(GF(3))^3 + 2*X(GF(3))^2 + X(GF(3)))
gap> PolyCodeword( [ 0, 0, 0, 0], GF(9) );
0*X(GF(3^2))^0
```

66.9 TreatAsVector

`TreatAsVector(` *obj* `)`

`TreatAsVector` adapts the codewords in *obj* to make sure they are printed as vectors. *obj* may be a codeword or a list of codewords. Elements of *obj* that are not codewords are ignored. After this function is called, the codewords will be treated as vectors. The vector representation is obtained by using the coefficient list of the polynomial.

Note that this only changes the way a codeword is printed. `TreatAsVector` returns nothing, it is called only for its side effect. The function `VectorCodeword` converts codewords to vectors (see 66.7).

```
gap> B := BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> c := CodewordNr(B, 4);
x^22 + x^20 + x^17 + x^14 + x^13 + x^12 + x^11 + x^10
gap> TreatAsVector(c);
gap> c;
[ 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 ]
```

66.10 TreatAsPoly

`TreatAsPoly(` *obj* `)`

`TreatAsPoly` adapts the codewords in *obj* to make sure they are printed as polynomials. *obj* may be a codeword or a list of codewords. Elements of *obj* that are not codewords are ignored. After this function is called, the codewords will be treated as polynomials. The finite field vector that defines the codeword is used as a coefficient list of the polynomial representation, where the first element of the vector is the coefficient of degree zero, the second element is the coefficient of degree one, etc, until the last element, which is the coefficient of highest degree.

Note that this only changes the way a codeword is printed. `TreatAsPoly` returns nothing, it is called only for its side effect. The function `PolyCodeword` converts codewords to polynomials (see 66.8).

```
gap> a := Codeword("00001",GF(2));
[ 0 0 0 0 1 ]
gap> TreatAsPoly(a); a;
x^4
gap> b := NullWord(6,GF(4));
[ 0 0 0 0 0 0 ]
gap> TreatAsPoly(b); b;
0
```

66.11 NullWord

`NullWord(` n `)`
`NullWord(` n`,` F `)`
`NullWord(` C `)`

`NullWord` returns a codeword of length n over the field F of only zeros. The default for F is `GF(2)`. n must be greater then zero. If only a code C is specified, `NullWord` will return a null word with the word length and the Galois field of C.

```
gap> NullWord(8);
[ 0 0 0 0 0 0 0 0 ]
gap> Codeword("0000") = NullWord(4);
true
gap> NullWord(5,GF(16));
[ 0 0 0 0 0 ]
gap> NullWord(ExtendedTernaryGolayCode());
[ 0 0 0 0 0 0 0 0 0 0 0 0 ]
```

66.12 DistanceCodeword

`DistanceCodeword(` c_1`,` c_2 `)`

`DistanceCodeword` returns the Hamming distance from c_1 to c_2. Both variables must be codewords with equal word length over the same Galois field. The Hamming distance between two words is the number of places in which they differ. As a result, `DistanceCodeword` always returns an integer between zero and the word length of the codewords.

```
gap> a := Codeword([0, 1, 2, 0, 1, 2]);; b := NullWord(6, GF(3));;
gap> DistanceCodeword(a, b);
4
gap> DistanceCodeword(b, a);
4
gap> DistanceCodeword(a, a);
0
```

66.13 Support

`Support(` c `)`

`Support` returns a set of integers indicating the positions of the non-zero entries in a codeword c.

```
gap> a := Codeword("012320023002");; Support(a);
[ 2, 3, 4, 5, 8, 9, 12 ]
gap> Support(NullWord(7));
[  ]
```

The support of a list with codewords can be calculated by taking the union of the individual supports. The weight of the support is the length of the set.

```
gap> L := Codeword(["000000", "101010", "222000"], GF(3));;
gap> S := Union(List(L, i -> Support(i)));
[ 1, 2, 3, 5 ]
gap> Length(S);
4
```

66.14 WeightCodeword

`WeightCodeword(` c `)`

`WeightCodeword` returns the weight of a codeword c, the number of non-zero entries in c. As a result, `WeightCodeword` always returns an integer between zero and the word length of the codeword.

```
gap> WeightCodeword(Codeword("22222"));
5
gap> WeightCodeword(NullWord(3));
0
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> Minimum(List(Elements(C){[2..Size(C)]}, WeightCodeword ) );
3
```

66.15 Codes

A **code** basically is nothing more than a set of **codewords**. We call these the **elements** of the code. A codeword is a sequence of elements of a finite field `GF(`q`)` where q is a prime power. Depending on the type of code, a codeword can be interpreted as a vector or as a polynomial. This will be explained in more detail in 66.2.

In GUAVA, codes can be defined by their elements (this will be called an **unrestricted code**), by a generator matrix (a **linear code**) or by a generator polynomial (a **cyclic code**).

Any code can be defined by its elements. If you like, you can give the code a name.

```
gap> C := ElementsCode(["1100", "1010", "0001"], "example code",
>                       GF(2) );
a (4,3,1..4)2..4 example code over GF(2)
```

An (n, M, d) code is a code with **word length** n, **size** M and **minimum distance** d. If the minimum distance has not yet been calculated, the lower bound and upper bound are printed. So

```
a (4,3,1..4)2..4 code over GF(2)
```

means a binary unrestricted code of length 4, with 3 elements and the minimum distance is greater than or equal to 1 and less than or equal to 4 and the **covering radius** is greater than or equal to 2 and less than or equal to 4.

```
gap> MinimumDistance(C);
2
gap> C;
a (4,3,2)2..4 example code over GF(2)
```

If the set of elements is a linear subspace of $GF(q)^n$, the code is called **linear**. If a code is linear, it can be defined by its **generator matrix** or **parity check matrix**. The generator matrix is a basis for the elements of a code, the parity check matrix is a basis for the nullspace of the code.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,2]], "demo code", GF(3) );
a linear [3,2,1..2]1 demo code over GF(3)
```

So a linear $[n, k, d]r$ code is a code with **word length** n, **dimension** k, **minimum distance** d and **covering radius** r.

If the code is linear and all cyclic shifts of its elements are again codewords, the code is called **cyclic**. A cyclic code is defined by its **generator polynomial** or **check polynomial**. All elements are multiples of the generator polynomial modulo a polynomial $x^n - 1$ where n is the word length of the code. Multiplying a code element with the check polynomial yields zero (modulo the polynomial $x^n - 1$).

```
gap> G := GeneratorPolCode(X(GF(2))+Z(2)^0, 7, GF(2) );
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
```

It is possible that GUAVA does not know that an unrestricted code is linear. This situation occurs for example when a code is generated from a list of elements with the function `ElementsCode`. By calling the function `IsLinearCode`, GUAVA tests if the code can be represented by a generator matrix. If so, the code record and the operations are converted accordingly.

```
gap> L := Z(2)*[ [0,0,0], [1,0,0], [0,1,1], [1,1,1] ];;
gap> C := ElementsCode( L, GF(2) );
a (3,4,1..3)1 user defined unrestricted code over GF(2)
# so far, GUAVA does not know what kind of code this is
gap> IsLinearCode( C );
true                         # it is linear
gap> C;
a linear [3,2,1]1 user defined unrestricted code over GF(2)
```

Of course the same holds for unrestricted codes that in fact are cyclic, or codes, defined by a generator matrix, that in fact are cyclic.

Codes are printed simply by giving a small description of their parameters, the word length, size or dimension and minimum distance, followed by a short description and the base field of the code. The function `Display` gives a more detailed description, showing the construction history of the code.

GUAVA doesn't place much emphasis on the actual encoding and decoding processes; some algorithms have been included though. Encoding works simply by multiplying an information vector with a code, decoding is done by the function `Decode`. For more information about encoding and decoding, see sections 66.20 and 66.43.

```
gap> R := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> w := [ 1, 1, 1, 1 ] * R;
[ 1 0 0 1 0 1 1 0 ]
gap> Decode( R, w );
[ 1 1 1 1 ]
gap> Decode( R, w + "10000000" ); # One error at the first position
[ 1 1 1 1 ]                       # Corrected by Guava
```

The next sections describes the functions that tests whether an object is a code and what kind of code it is (see 66.16, 66.17 and 66.18).

The following sections describe the operations that are available for codes (see 66.19 and 66.20).

The next sections describe basic functions for codes, e.g. `MinimumDistance` (see 66.21).

The following sections describe functions that generate codes (see 66.49, 66.58 and 66.72).

The next sections describe functions which manipulate codes (see 66.86).

The last part tells more about the implementation of codes. It describes the format of code records (see 66.109).

66.16 IsCode

`IsCode(` *obj* `)`

`IsCode` returns `true` if *obj*, which can be an object of arbitrary type, is a code and `false` otherwise. Will cause an error if *obj* is an unbound variable.

```
gap> IsCode( 1 );
false
gap> IsCode( ReedMullerCode( 2,3 ) );
true
gap> IsCode( This_object_is_unbound );
Error, Variable: 'This_object_is_unbound' must have a value
```

66.17 IsLinearCode

`IsLinearCode(` *obj* `)`

`IsLinearCode` checks if object *obj* (not necessarily a code) is a linear code. If a code has already been marked as linear or cyclic, the function automatically returns `true`. Otherwise, the function checks if a basis G of the elements of *obj* exists that generates the elements of *obj*. If so, G is a generator matrix of *obj* and the function returns `true`. If not, the function returns `false`.

```
gap> C := ElementsCode( [ [0,0,0],[1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
gap> IsLinearCode( C );
true
gap> IsLinearCode( ElementsCode( [ [1,1,1] ], GF(2) ) );
false
gap> IsLinearCode( 1 );
false
```

66.18 IsCyclicCode

`IsCyclicCode(` *obj* `)`

`IsCyclicCode` checks if the object *obj* is a cyclic code. If a code has already been marked as cyclic, the function automatically returns `true`. Otherwise, the function checks if a polynomial g exists that generates the elements of *obj*. If so, g is a generator polynomial of *obj* and the function returns `true`. If not, the function returns `false`.

```
gap> C := ElementsCode( [ [0,0,0], [1,1,1] ], GF(2) );
a (3,2,1..3)1 user defined unrestricted code over GF(2)
# GUAVA does not know the code is cyclic
gap> IsCyclicCode( C );        # this command tells GUAVA to find out
true
gap> IsCyclicCode( HammingCode( 4, GF(2) ) );
false
gap> IsCyclicCode( 1 );
false
```

66.19 Comparisons of Codes

C_1 `=` C_2
C_1 `<>` C_2

The equality operator `=` evaluates to `true` if the codes C_1 and C_2 are equal, and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the codes C_1 and C_2 are not equal, and to `false` otherwise.

Note that codes are equal if and only if their elements are equal. Codes can also be compared with objects of other types. Of course they are never equal.

```
gap> M := [ [0, 0], [1, 0], [0, 1], [1, 1] ];;
gap> C1 := ElementsCode( M, GF(2) );
a (2,4,1..2)0 user defined unrestricted code over GF(2)
gap> M = C1;
false
gap> C2 := GeneratorMatCode( [ [1, 0], [0, 1] ], GF(2) );
a linear [2,2,1]0 code defined by generator matrix over GF(2)
gap> C1 = C2;
true
gap> ReedMullerCode( 1, 3 ) = HadamardCode( 8 );
true
gap> WholeSpaceCode( 5, GF(4) ) = WholeSpaceCode( 5, GF(2) );
false
```

Another way of comparing codes is `IsEquivalent`, which checks if two codes are equivalent (see 66.40).

66.20 Operations for Codes

C_1 `+` C_2

The operator + evaluates to the direct sum of the codes C_1 and C_2. See 66.104.

C + c
c + C

The operator + evaluates to the coset code of code C after adding c to all elements of C. See 66.101.

C_1 * C_2

The operator * evaluates to the direct product of the codes C_1 and C_2. See 66.106.

x * C

The operator * evaluates to the element of C belonging to information word x. x may be a vector, polynomial, string or codeword or a list of those. This is the way to do encoding in GUAVA. C must be linear, because in GUAVA, encoding by multiplication is only defined for linear codes. If C is a cyclic code, this multiplication is the same as multiplying an information polynomial x by the generator polynomial of C (except for the result not being a codeword type). If C is a linear code, it is equal to the multiplication of an information vector x by the generator matrix of C (again, the result then is not a codeword type).

To decode, use the function `Decode` (see 66.43).

c `in` C

The `in` operator evaluates to `true` if C contains the codeword or list of codewords specified by c. Of course, c and C must have the same word lengths and base fields.

```
gap> C := HammingCode( 2 );; Elements( C );
[ [ 0 0 0 ], [ 1 1 1 ] ]
gap> [ [ 0, 0, 0, ], [ 1, 1, 1, ] ] in C;
true
gap> [ 0 ] in C;
false
```

C_1 `in` C_2

The `in` operator evaluates to `true` if C_1 is a subcode of C_2, i.e. if C_2 contains at least all the elements of C_1.

```
gap> RepetitionCode( 7 ) in HammingCode( 3 );
true
gap> HammingCode( 3 ) in RepetitionCode( 7 );
false
gap> HammingCode( 3 ) in WholeSpaceCode( 7 );
true
gap> AreEqualCodes := function(C1, C2)
> return (C1 in C2) and (C2 in C1);
> end;      # this is a slow implementation of the function =
function ( C1, C2 ) ... end
gap> AreEqualCodes( HammingCode(2), RepetitionCode(3) );
true
```

66.21 Basic Functions for Codes

A few sections now follow that describe GUAVA's basic functions on codes.

The first section describes GAP3 functions that work on **Domains** (see `Domains`), but are also applicable for codes (see 66.22).

The next section describes three GAP3 input/output functions (see 66.23).

The next sections describe functions that return the matrices and polynomials that define a code (see 66.24, 66.25, 66.26, 66.27, 66.28).

The next sections describe function that return the basic parameters of codes (see 66.29, 66.30 and 66.31).

The next sections describe functions that return distance and weight distributions (see 66.32, 66.33, 66.34 and 66.35).

The next sections describe boolean functions on codes (see 66.17, 66.18, 66.36, 66.38, 66.39, and 66.37).

The next sections describe functions about equivalence of codes (see 66.40, 66.41 and 66.42).

The next sections describe functions related to decoding (see 66.43, 66.44, 66.45 and 66.46).

The next section describes a function that displays a code (see 66.47).

The next section describes the function `CodewordNr` (see 66.48).

The next sections describe extensions that have been added in version 1.3 of GUAVA (see 66.141).

66.22 Domain Functions for Codes

These are some GAP3 functions that work on `Domains` in general. Their specific effect on `Codes` is explained here.

`IsFinite(` C `)`

`IsFinite` is an implementation of the GAP3 domain function `IsFinite`. It returns true for a code C.

```
gap> IsFinite( RepetitionCode( 1000, GF(11) ) );
true
```

`Size(` C `)`

`Size` returns the size of C, the number of elements of the code. If the code is linear, the size of the code is equal to q^k, where q is the size of the base field of C and k is the dimension.

```
gap> Size( RepetitionCode( 1000, GF(11) ) );
11
gap> Size( NordstromRobinsonCode() );
256
```

`Field(` C `)`

`Field` returns the base field of a code C. Each element of C consists of elements of this base field. If the base field is F, and the word length of the code is n, then the codewords are elements of F^n. If C is a cyclic code, its elements are interpreted as polynomials with coefficients over F.

```
gap> C1 := ElementsCode([[0,0,0], [1,0,1], [0,1,0]], GF(4));
a (3,3,1..3)2..3 user defined unrestricted code over GF(4)
gap> Field( C1 );
GF(2^2)
gap> Field( HammingCode( 3, GF(9) ) );
GF(3^2)
```

`Dimension(` C `)`

`Dimension` returns the parameter k of C, the dimension of the code, or the number of information symbols in each codeword. The dimension is not defined for non-linear codes; `Dimension` then returns an error.

```
gap> Dimension( NordstromRobinsonCode() );
Error, dimension is only defined for linear codes
gap> Dimension( NullCode( 5, GF(5) ) );
0
gap> C := BCHCode( 15, 4, GF(4) );
a cyclic [15,7,5]4..8 BCH code, delta=5, b=1 over GF(4)
gap> Dimension( C );
7
gap> Size( C ) = Size( Field( C ) ) ^ Dimension( C );
true
```

`Elements(` C `)`

`Elements` returns a list of the elements of C. These elements are of the codeword type (see 66.2). Note that for large codes, generating the elements may be very time- and memory-consuming. For generating a specific element or a subset of the elements, use `CodewordNr` (see 66.48).

```
gap> C := ConferenceCode( 5 );
a (5,12,2)1..4 conference code over GF(2)
gap> Elements( C );
[ [ 0 0 0 0 0 ], [ 1 1 0 1 0 ], [ 1 1 1 0 0 ], [ 0 1 1 0 1 ],
  [ 1 0 0 1 1 ], [ 0 0 1 1 1 ], [ 1 0 1 0 1 ], [ 0 1 0 1 1 ],
  [ 1 0 1 1 0 ], [ 0 1 1 1 0 ], [ 1 1 0 0 1 ], [ 1 1 1 1 1 ] ]
gap> CodewordNr( C, [ 1, 2 ] );
[ [ 0 0 0 0 0 ], [ 1 1 0 1 0 ] ]
```

66.23 Printing and Saving Codes

`Print(` C `)`

`Print` prints information about C. This is the same as typing the identifier C at the GAP3-prompt.

If the argument is an unrestricted code, information in the form

`a (`n`,`M`,`d`)`r `... code over GF(q)`

is printed, where n is the word length, M the number of elements of the code, d the minimum distance and r the covering radius.

If the argument is a linear code, information in the form

`a linear [`*n*`,`*k*`,`*d*`]`*r* `... code over GF(q)`

is printed, where n is the word length, k the dimension of the code, d the minimum distance and r the covering radius.

In all cases, if d is not yet known, it is displayed in the form

lowerbound `..` *upperbound*

and if r is not yet known, it is displayed in the same way.

The function `Display` gives more information. See 66.47.

```
gap> C1 := ExtendedCode( HammingCode( 3, GF(2) ) );
a linear [8,4,4]2 extended code
gap> Print( "This is ", NordstromRobinsonCode(), ". \n");
This is a (16,256,6)4 Nordstrom-Robinson code over GF(2).
```

`String(` C `)`

`String` returns information about C in a string. This function is used by `Print` (see `Print`).

`Save(` *filename*`,` C`,` *varname* `)`

`Save` prints the code C to a file with file name *filename*. If the file does not exist, it is created. If it does exist, the previous contents are erased, so be careful. The code is saved with variable name *varname*. The code can be read back by calling `Read(filename)`. The code then has name *varname*. Note that *filename* and *varname* are strings.

```
gap> C1 := HammingCode( 4, GF(3) );
a linear [40,36,3]1 Hamming (4,3) code over GF(3)
gap> Save( "mycodes.lib", C1, "Ham_4_3");
gap> Read( "mycodes.lib" ); Ham_4_3;
a linear [40,36,3]1 Hamming (4,3) code over GF(3)
gap> Ham_4_3 = C1;
true
```

66.24 GeneratorMat

`GeneratorMat(` C `)`

`GeneratorMat` returns a generator matrix of C. The code consists of all linear combinations of the rows of this matrix.

If until now no generator matrix of C was determined, it is computed from either the parity check matrix, the generator polynomial, the check polynomial or the elements (if possible), whichever is available.

If C is a non-linear code, the function returns an error.

```
gap> GeneratorMat( HammingCode( 3, GF(2) ) );
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> GeneratorMat( RepetitionCode( 5, GF(25) ) );
```

```
[ [ Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0, Z(5)^0 ] ]
gap> GeneratorMat( NullCode( 14, GF(4) ) );
[  ]
gap> GeneratorMat( ElementsCode( [[0, 0, 1 ], [1, 1, 0 ]], GF(2) ));
Error, non-linear codes don't have a generator matrix
```

66.25 CheckMat

CheckMat(C)

CheckMat returns a parity check matrix of C. The code consists of all words orthogonal to each of the rows of this matrix. The transpose of the matrix is a right inverse of the generator matrix. The parity check matrix is computed from either the generator matrix, the generator polynomial, the check polynomial or the elements of C (if possible), whichever is available.

If C is a non-linear code, the function returns an error.

```
gap> CheckMat( HammingCode(3, GF(2) ) );
[ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ]
gap> CheckMat( RepetitionCode( 5, GF(25) ) );
[ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2, 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^2 ] ]
gap> CheckMat( WholeSpaceCode( 12, GF(4) ) );
[  ]
```

66.26 GeneratorPol

GeneratorPol(C)

GeneratorPol returns the generator polynomial of C. The code consists of all multiples of the generator polynomial modulo $x^n - 1$ where n is the word length of C. The generator polynomial is determined from either the check polynomial, the generator or check matrix or the elements of C (if possible), whichever is available.

If C is not a cyclic code, the function returns false.

```
gap> GeneratorPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0*(X(GF(2)) + 1)
gap> GeneratorPol( WholeSpaceCode( 4, GF(2) ) );
X(GF(2))^0
gap> GeneratorPol( NullCode( 7, GF(3) ) );
Z(3)^0*(X(GF(3))^7 + 2)
```

66.27 CheckPol

CheckPol(C)

`CheckPol` returns the check polynomial of C. The code consists of all polynomials f with $f * h = 0 \pmod{x^n - 1}$, where h is the check polynomial, and n is the word length of C. The check polynomial is computed from the generator polynomial, the generator or parity check matrix or the elements of C (if possible), whichever is available.

If C if not a cyclic code, the function returns an error.

```
gap> CheckPol(GeneratorMatCode([[1, 1, 0], [0, 1, 1]], GF(2)));
Z(2)^0*(X(GF(2))^2 + X(GF(2)) + 1)
gap> CheckPol(WholeSpaceCode(4, GF(2)));
Z(2)^0*(X(GF(2))^4 + 1)
gap> CheckPol(NullCode(7,GF(3)));
X(GF(3))^0
gap> CheckPol(ElementsCode( [ [0, 0, 1 ], [1, 1, 0 ] ], GF(2) ) );
Error, generator polynomial is only defined for cyclic codes
```

66.28 RootsOfCode

`RootsOfCode(` C `)`

`RootsOfCode` returns a list of all zeros of the generator polynomial of a cyclic code C. These are finite field elements in the splitting field of the generator polynomial, $GF(q^m)$, m is the multiplicative order of the size of the base field of the code, modulo the word length.

The reverse proces, constructing a code from a set of roots, can be carried out by the function `RootsCode` (see 66.77).

```
gap> C1 := ReedSolomonCode( 16, 5 );
a cyclic [16,12,5]3..4 Reed-Solomon code over GF(17)
gap> RootsOfCode( C1 );
[ Z(17), Z(17)^2, Z(17)^3, Z(17)^4 ]
gap> C2 := RootsCode( 16, last );
a cyclic [16,12,5]3..4 code defined by roots over GF(17)
gap> C1 = C2;
true
```

66.29 WordLength

`WordLength(` C `)`

`WordLength` returns the parameter n of C, the word length of the elements. Elements of cyclic codes are polynomials of maximum degree $n-1$, as calculations are carried out modulo $x^n - 1$.

```
gap> WordLength( NordstromRobinsonCode() );
16
gap> WordLength( PuncturedCode( WholeSpaceCode(7) ) );
6
gap> WordLength( UUVCode( WholeSpaceCode(7), RepetitionCode(7) ) );
14
```

66.30 Redundancy

`Redundancy(` C `)`

`Redundancy` returns the redundancy r of C, which is equal to the number of check symbols in each element. If C is not a linear code the redundancy is not defined and `Redundancy` returns an error.

If a linear code C has dimension k and word length n, it has redundancy $r = n - k$.

```
gap> C := TernaryGolayCode();
a cyclic [11,6,5]2 ternary Golay code over GF(3)
gap> Redundancy(C);
5
gap> Redundancy( DualCode(C) );
6
```

66.31 MinimumDistance

`MinimumDistance(` C `)`

`MinimumDistance` returns the minimum distance of C, the largest integer d with the property that every element of C has at least a Hamming distance d (see 66.12) to any other element of C. For linear codes, the minimum distance is equal to the minimum weight. This means that d is also the smallest positive value with $w[d+1] \neq 0$, where w is the weight distribution of C (see 66.32). For unrestricted codes, d is the smallest positive value with $w[d+1] \neq 0$, where w is the inner distribution of C (see 66.33).

For codes with only one element, the minimum distance is defined to be equal to the word length.

```
gap> C := MOLSCode(7);; MinimumDistance(C);
3
gap> WeightDistribution(C);
[ 1, 0, 0, 24, 24 ]
gap> MinimumDistance( WholeSpaceCode( 5, GF(3) ) );
1
gap> MinimumDistance( NullCode( 4, GF(2) ) );
4
gap> C := ConferenceCode(9);; MinimumDistance(C);
4
gap> InnerDistribution(C);
[ 1, 0, 0, 0, 63/5, 9/5, 18/5, 0, 9/10, 1/10 ]
```

`MinimumDistance(` C`,` w `)`

In this form, `MinimumDistance` returns the minimum distance of a codeword w to the code C, also called the **distance to** C. This is the smallest value d for which there is an element c of the code C which is at distance d from w. So d is also the minimum value for which $D[d+1] \neq 0$, where D is the distance distribution of w to C (see 66.35).

Note that w must be an element of the same vector space as the elements of C. w does not necessarily belong to the code (if it does, the minimum distance is zero).

```
gap> C := MOLSCode(7);; w := CodewordNr( C, 17 );
[ 2 2 4 6 ]
gap> MinimumDistance( C, w );
0
gap> C := RemovedElementsCode( C, w );; MinimumDistance( C, w );
3                                # so w no longer belongs to C
```

66.32 WeightDistribution

`WeightDistribution(` *C* `)`

`WeightDistribution` returns the weight distribution of C, as a vector. The i^{th} element of this vector contains the number of elements of C with weight $i-1$. For linear codes, the weight distribution is equal to the inner distribution (see 66.33).

Suppose w is the weight distribution of C. If C is linear, it must have the zero codeword, so $w[1] = 1$ (one word of weight 0).

```
gap> WeightDistribution( ConferenceCode(9) );
[ 1, 0, 0, 0, 0, 18, 0, 0, 0, 1 ]
gap> WeightDistribution( RepetitionCode( 7, GF(4) ) );
[ 1, 0, 0, 0, 0, 0, 0, 3 ]
gap> WeightDistribution( WholeSpaceCode( 5, GF(2) ) );
[ 1, 5, 10, 10, 5, 1 ]
```

66.33 InnerDistribution

`InnerDistribution(` *C* `)`

`InnerDistribution` returns the inner distribution of C. The i^{th} element of the vector contains the average number of elements of C at distance $i-1$ to an element of C. For linear codes, the inner distribution is equal to the weight distribution (see 66.32).

Suppose w is the inner distribution of C. Then $w[1] = 1$, because each element of C has exactly one element at distance zero (the element itself). The minimum distance of C is the smallest value $d > 0$ with $w[d+1] \neq 0$, because a distance between zero and d never occurs. See 66.31.

```
gap> InnerDistribution( ConferenceCode(9) );
[ 1, 0, 0, 0, 63/5, 9/5, 18/5, 0, 9/10, 1/10 ]
gap> InnerDistribution( RepetitionCode( 7, GF(4) ) );
[ 1, 0, 0, 0, 0, 0, 0, 3 ]
```

66.34 OuterDistribution

`OuterDistribution(` *C* `)`

The function `OuterDistribution` returns a list of length q^n, where q is the size of the base field of C and n is the word length. The elements of the list consist of an element of $(GF(q))^n$ (this is a codeword type) and the distribution of distances to the code (a list of integers). This table is **very** large, and for $n > 20$ it will not fit in the memory of most computers. The function `DistancesDistribution` (see 66.35) can be used to calculate one entry of the list.

```
gap> C := RepetitionCode( 3, GF(2) );
a cyclic [3,1,3]1 repetition code over GF(2)
gap> OD := OuterDistribution(C);
[ [ [ 0 0 0 ], [ 1, 0, 0, 1 ] ], [ [ 1 1 1 ], [ 1, 0, 0, 1 ] ],
  [ [ 0 0 1 ], [ 0, 1, 1, 0 ] ], [ [ 1 1 0 ], [ 0, 1, 1, 0 ] ],
  [ [ 1 0 0 ], [ 0, 1, 1, 0 ] ], [ [ 0 1 1 ], [ 0, 1, 1, 0 ] ],
  [ [ 0 1 0 ], [ 0, 1, 1, 0 ] ], [ [ 1 0 1 ], [ 0, 1, 1, 0 ] ] ]
gap> WeightDistribution(C) = OD[1][2];
true
gap> DistancesDistribution( C, Codeword("110") ) = OD[4][2];
true
```

66.35 DistancesDistribution

`DistancesDistribution(` C, w `)`

`DistancesDistribution` returns a distribution of the distances of all elements of C to a codeword w in the same vector space. The i^{th} element of the distance distribution is the number of codewords of C that have distance $i-1$ to w. The smallest value d with $w[d+1] \neq 0$ is defined as the **distance to** C (see 66.31).

```
gap> H := HadamardCode(20);
a (20,40,10)6..8 Hadamard code of order 20 over GF(2)
gap> c := Codeword("10110101101010010101", H);
[ 1 0 1 1 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 ]
gap> DistancesDistribution(H, c);
[ 0, 0, 0, 0, 0, 1, 0, 7, 0, 12, 0, 12, 0, 7, 0, 1, 0, 0, 0, 0, 0 ]
gap> MinimumDistance(H, c);
5                           # distance to H
```

66.36 IsPerfectCode

`IsPerfectCode(` C `)`

`IsPerfectCode` returns `true` if C is a perfect code. For a code with odd minimum distance $d = 2t + 1$, this is the case when every word of the vector space of C is at distance at most t from exactly one element of C. Codes with even minimum distance are never perfect.

In fact, a code that is not **trivial perfect** (the binary repetition codes of odd length, the codes consisting of one word, and the codes consisting of the whole vector space), and does not have the parameters of a Hamming- or Golay-code, cannot be perfect.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> IsPerfectCode( H );
true
gap> IsPerfectCode( ElementsCode( [ [1,1,0], [0,0,1] ], GF(2) ) );
true
gap> IsPerfectCode( ReedSolomonCode( 6, 3 ) );
false
gap> IsPerfectCode(BinaryGolayCode());
true
```

66.37 IsMDSCode

`IsMDSCode(` C `)`

`IsMDSCode` returns true if C is a **Maximum Distance Seperable code**, or MDS code for short. A linear $[n, k, d]$-code of length n, dimension k and minimum distance d is an MDS code if $k = n - d + 1$, in other words if C meets the Singleton bound (see 66.111). An unrestricted (n, M, d) code is called MDS if $k = n - d + 1$, with k equal to the largest integer less than or equal to the logarithm of M with base q, the size of the base field of C.

Well known MDS codes include the repetition codes, the whole space codes, the even weight codes (these are the only **binary** MDS Codes) and the Reed-Solomon codes.

```
gap> C1 := ReedSolomonCode( 6, 3 );
a cyclic [6,4,3]2 Reed-Solomon code over GF(7)
gap> IsMDSCode( C1 );
true    # 6-3+1 = 4
gap> IsMDSCode( QRCode( 23, GF(2) ) );
false
```

66.38 IsSelfDualCode

`IsSelfDualCode(` C `)`

`IsSelfDualCode` returns `true` if C is self-dual, i.e. when C is equal to its dual code (see also 66.99). If a code is self-dual, it automatically is self-orthogonal (see 66.39).

If C is a non-linear code, it cannot be self-dual, so `false` is returned. A linear code can only be self-dual when its dimension k is equal to the redundancy r.

```
gap> IsSelfDualCode( ExtendedBinaryGolayCode() );
true
gap> C := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> DualCode( C ) = C;
true
```

66.39 IsSelfOrthogonalCode

`IsSelfOrthogonalCode(` C `)`

`IsSelfOrthogonalCode` returns `true` if C is **self-orthogonal**. A code is self-orthogonal if every element of C is orthogonal to all elements of C, including itself. In the linear case, this simply means that the generator matrix of C multiplied with its transpose yields a null matrix.

In addition, a code is **self-dual** if it contains all vectors that its elements are orthogonal to (see 66.38).

```
gap> R := ReedMullerCode(1,4);
a linear [16,5,8]6 Reed-Muller (1,4) code over GF(2)
gap> IsSelfOrthogonalCode(R);
true
gap> IsSelfDualCode(R);
false
```

66.40 IsEquivalent

`IsEquivalent(` C_1`,` C_2 `)`

`IsEquivalent` returns true if C_1 and C_2 are equivalent codes. This is the case if C_1 can be obtained from C_2 by carrying out column permutations. GUAVA only handles binary codes. The external program `desauto` from **J.S. Leon** is used to compute the isomorphism between the codes. If C_1 and C_2 are equal, they are also equivalent.

Note that the algorithm is **very** slow for non-linear codes.

```
gap> H := GeneratorPolCode([1,1,0,1]*Z(2), 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> H = HammingCode(3, GF(2));
false
gap> IsEquivalent(H, HammingCode(3, GF(2)));
true                          # H is equivalent to a Hamming code
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
```

66.41 CodeIsomorphism

`CodeIsomorphism(` C_1`,` C_2 `)`

If the two codes C_1 and C_2 are equivalent codes (see 66.40), `CodeIsomorphism` returns the permutation that transforms C_1 into C_2. If the codes are not equivalent, it returns `false`.

```
gap> H := GeneratorPolCode([1,1,0,1]*Z(2), 7, GF(2));
a cyclic [7,4,1..3]1 code defined by generator polynomial over GF(2)
gap> CodeIsomorphism(H, HammingCode(3, GF(2)));
(3,4)(5,6,7)
gap> PermutedCode(H, (3,4)(5,6,7)) = HammingCode(3, GF(2));
true
```

66.42 AutomorphismGroup

`AutomorphismGroup(` C `)`

`AutomorphismGroup` returns the **automorphism group** of a binary code C. This is the largest permutation group of degree n such that each permutation applied to the columns of C again yields C. GUAVA uses the external program `desauto` from **J.S. Leon** to compute the automorphism group. The function `PermutedCode` permutes the columns of a code (see 66.90).

```
gap> R := RepetitionCode(7,GF(2));
a cyclic [7,1,7]3 repetition code over GF(2)
gap> AutomorphismGroup(R);
Group( (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) )
                         # every permutation keeps R identical
gap> C := CordaroWagnerCode(7);
a linear [7,2,4]3 Cordaro-Wagner code over GF(2)
gap> Elements(C);
```

```
[ [ 0 0 0 0 0 0 0 ], [ 1 1 1 1 1 0 0 ], [ 0 0 1 1 1 1 1 ],
  [ 1 1 0 0 0 1 1 ] ]
gap> AutomorphismGroup(C);
Group( (3,4), (4,5), (1,6)(2,7), (1,2), (6,7) )
gap> C2 :=  PermutedCode(C, (1,6)(2,7));
a linear [7,2,4]3 permuted code
gap> Elements(C2);
[ [ 0 0 0 0 0 0 0 ], [ 0 0 1 1 1 1 1 ], [ 1 1 1 1 1 0 0 ],
  [ 1 1 0 0 0 1 1 ] ]
gap> C2 = C;
true
```

66.43 Decode

`Decode(` C, c `)`

`Decode` decodes c with respect to code C. c is a codeword or a list of codewords. First, possible errors in c are corrected, then the codeword is decoded to an information codeword x. If the code record has a field `specialDecoder`, this special algorithm is used to decode the vector. Hamming codes and BCH codes have such a special algorithm. Otherwise, syndrome decoding is used. Encoding is done by multiplying the information vector with the code (see 66.20).

A special decoder can be created by defining a function

```
C.specialDecoder := function(C, c) ... end;
```

The function uses the arguments C, the code record itself, and c, a vector of the codeword type, to decode c to an information word. A normal decoder would take a codeword c of the same word length and field as C, and would return a information word of length k, the dimension of C. The user is not restricted to these normal demands though, and can for instance define a decoder for non-linear codes.

```
gap> C := HammingCode(3);
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := "1010"*C;                    # encoding
[ 1 0 1 0 1 0 1 ]
gap> Decode(C, c);                     # decoding
[ 1 0 1 0 ]
gap> Decode(C, Codeword("0010101"));
[ 1 0 1 0 ]                            # one error corrected
gap> C.specialDecoder := function(C, c)
> return NullWord(Dimension(C));
> end;
function ( C, c ) ... end
gap> Decode(C, c);
[ 0 0 0 0 ]              # new decoder always returns null word
```

66.44 Syndrome

`Syndrome(` C, c `)`

`Syndrome` returns the syndrome of word c with respect to a code C. c is a word of the vector space of C. If c is an element of C, the syndrome is a zero vector. The syndrome can be used for looking up an error vector in the syndrome table (see 66.45) that is needed to correct an error in c.

A syndrome is not defined for non-linear codes. `Syndrome` then returns an error.

```
gap> C := HammingCode(4);
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> v := CodewordNr( C, 7 );
[ 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 ]
gap> Syndrome( C, v );
[ 0 0 0 0 ]
gap> Syndrome( C, "000000001100111" );
[ 1 1 1 1 ]
gap> Syndrome( C, "000000000000001" );
[ 1 1 1 1 ]     # the same syndrome:both codewords are in the same
                # coset of C
```

66.45 SyndromeTable

`SyndromeTable(` C `)`

`SyndromeTable` returns a **syndrome table** of a linear code C, consisting of two columns. The first column consists of the error vectors that correspond to the syndrome vectors in the second column. These vectors both are of the codeword type. After calculating the syndrome of a word c with `Syndrome` (see 66.44), the error vector needed to correct c can be found in the syndrome table. Subtracting this vector from c yields an element of C. To make the search for the syndrome as fast as possible, the syndrome table is sorted according to the syndrome vectors.

```
gap> H := HammingCode(2);
a linear [3,1,3]1 Hamming (2,2) code over GF(2)
gap> SyndromeTable(H);
[ [ [ 0 0 0 ], [ 0 0 ] ], [ [ 1 0 0 ], [ 0 1 ] ],
  [ [ 0 1 0 ], [ 1 0 ] ], [ [ 0 0 1 ], [ 1 1 ] ] ]
gap> c := Codeword("101");
[ 1 0 1 ]
gap> c in H;
false           # c is not an element of H
gap> Syndrome(H,c);
[ 1 0 ]         # according to the syndrome table,
                # the error vector [ 0 1 0 ] belongs to this syndrome
gap> c - Codeword("010") in H;
true            # so the corrected codeword is
                # [ 1 0 1 ] - [ 0 1 0 ] = [ 1 1 1 ],
                # this is an element of H
```

66.46 StandardArray

`StandardArray(` C `)`

`StandardArray` returns the standard array of a code C. This is a matrix with elements of the codeword type. It has q^r rows and q^k columns, where q is the size of the base field of C, r is the redundancy of C, and k is the dimension of C. The first row contains all the elements of C. Each other row contains words that do not belong to the code, with in the first column their syndrome vector (see 66.44).

A non-linear code does not have a standard array. `StandardArray` then returns an error.

Note that calculating a standard array can be very time- and memory- consuming.

```
gap> StandardArray(RepetitionCode(3,GF(3)));
[ [ [ 0 0 0 ], [ 1 1 1 ], [ 2 2 2 ] ],
  [ [ 0 0 1 ], [ 1 1 2 ], [ 2 2 0 ] ],
  [ [ 0 0 2 ], [ 1 1 0 ], [ 2 2 1 ] ],
  [ [ 0 1 0 ], [ 1 2 1 ], [ 2 0 2 ] ],
  [ [ 0 2 0 ], [ 1 0 1 ], [ 2 1 2 ] ],
  [ [ 1 0 0 ], [ 2 1 1 ], [ 0 2 2 ] ],
  [ [ 1 2 0 ], [ 2 0 1 ], [ 0 1 2 ] ],
  [ [ 2 0 0 ], [ 0 1 1 ], [ 1 2 2 ] ],
  [ [ 2 1 0 ], [ 0 2 1 ], [ 1 0 2 ] ] ]
```

66.47 Display

`Display(` C `)`

`Display` prints the method of construction of code C. With this history, in most cases an equal or equivalent code can be reconstructed. If C is an unmanipulated code, the result is equal to output of the function `Print` (see 3.14).

```
gap> Display( RepetitionCode( 6, GF(3) ) );
a cyclic [6,1,6]4 repetition code over GF(3)
gap> C1 := ExtendedCode( HammingCode(2) );;
gap> C2 := PuncturedCode( ReedMullerCode( 2, 3 ) );;
gap> Display( LengthenedCode( UUVCode( C1, C2 ) ) );
a linear [12,8,2]2..4 code, lengtened with 1 column(s) of
a linear [11,8,1]1..2 U|U+V construction code of
U: a linear [4,1,4]2 extended code of
   a cyclic [3,1,3]1 Hamming (2,2) code over GF(2)
V: a linear [7,7,1]0 punctured code of
   a cyclic [8,7,2]1 Reed-Muller (2,3) code over GF(2)
```

66.48 CodewordNr

`CodewordNr(` C`,` *list* `)`

`CodewordNr` returns a list of codewords of C. *list* may be a list of integers or a single integer. For each integer of *list*, the corresponding codeword of C is returned. The correspondence of a number i with a codeword is determined as follows: if a list of elements of C is available, the i^{th} element is taken. Otherwise, it is calculated by multiplication of the i^{th} information vector by the generator matrix or generator polynomial, where the information vectors are ordered lexicographically.

So `CodewordNr(`C`, `i`)` is equal to `Elements(`C`)[`i`]`. The latter function first calculates the set of all the elements of C and then returns the i^{th} element of that set, whereas the former only calculates the i^{th} codeword.

```
gap> R := ReedSolomonCode(2,2);
a cyclic [2,1,2]1 Reed-Solomon code over GF(3)
gap> Elements(R);
[ 0, x + 1, 2x + 2 ]
gap> CodewordNr(R, [1,3]);
[ 0, 2x + 2 ]
gap> C := HadamardCode( 16 );
a (16,32,8)5..6 Hadamard code of order 16 over GF(2)
gap> Elements(C)[17] = CodewordNr( C, 17 );
true
```

66.49 Generating Unrestricted Codes

The following sections start with the description of creating codes from user defined matrices or special matrices (see 66.50, 66.51, 66.52 and 66.53). These codes are unrestricted codes; they may later be discovered to be linear or cyclic.

The next section describes a function for generating random codes (see 66.54).

The next section describes the Nordstrom-Robinson code (see 66.55).

The last sections describe two functions for generating Greedy codes. These are codes that contructed by gathering codewords from a space (see 66.56 and 66.57).

66.50 ElementsCode

`ElementsCode(` L `[,` *Name* `],` F `)`

`ElementsCode` creates an unrestricted code of the list of elements L, in the field F. L must be a list of vectors, strings, polynomials or codewords. *Name* can contain a short description of the code.

If L contains a codeword more than once, it is removed from the list and a GAP3 set is returned.

```
gap> M := Z(3)^0 * [ [1, 0, 1, 1], [2, 2, 0, 0], [0, 1, 2, 2] ];;
gap> C := ElementsCode( M, "example code", GF(3) );
a (4,3,1..4)2 example code over GF(3)
gap> MinimumDistance( C );
4
gap> Elements( C );
[ [ 1 0 1 1 ], [ 2 2 0 0 ], [ 0 1 2 2 ] ]
gap> last = M;
true     # Note that the elements are of codeword type
```

66.51 HadamardCode

`HadamardCode(` H`,` t `)`
`HadamardCode(` H `)`

In the first form HadamardCode returns a Hadamard code from the Hadamard matrix H, of the t^{th} kind. In the second form, $t = 3$ is used.

A Hadamard matrix is a square matrix H with $H * H^T = -n * I_n$, where n is the size of H. The entries of H are either 1 or -1.

The matrix H is first transformed into a binary matrix A_n (by replacing the 1's by 0's and the -1's by 1's).

The first kind ($t = 1$) is created by using the rows of A_n as elements, after deleting the first column. This is a $(n-1, n, n/2)$ code. We use this code for creating the Hadamard code of the second kind ($t = 2$), by adding all the complements of the already existing codewords. This results in a $(n-1, 2n, n/2-1)$ code. The third code ($t = 3$) is created by using the rows of A_n (without cutting a column) and their complements as elements. This way, we have an $(n, 2n, n/2)$ code. The returned code is generally an unrestricted code, but for $n = 2^r$, the code is linear.

```
gap> H4 := [[1,1,1,1],[1,-1,1,-1],[1,1,-1,-1],[1,-1,-1,1]];;
gap> HadamardCode( H4, 1 );
a (3,4,2)1 Hadamard code of order 4 over GF(2)
gap> HadamardCode( H4, 2 );
a (3,8,1)0 Hadamard code of order 4 over GF(2)
gap> HadamardCode( H4 );
a (4,8,2)1 Hadamard code of order 4 over GF(2)
```

HadamardCode(*n*, *t*)
HadamardCode(*n*)

In the first form HadamardCode returns a Hadamard code with parameter n of the t^{th} kind. In the second form, $t = 3$ is used.

When called in these forms, HadamardCode first creates a Hadamard matrix (see 66.125), of size n and then follows the same procedure as described above. Therefore the same restrictions with respect to n as for Hadamard matrices hold.

```
gap> C1 := HadamardCode( 4 );
a (4,8,2)1 Hadamard code of order 4 over GF(2)
gap> C1 = HadamardCode( H4 );
true
```

66.52 ConferenceCode

ConferenceCode(*H*)

ConferenceCode returns a code of length $n - 1$ constructed from a symmetric **conference matrix** H. H must be a symmetric matrix of order n, which satisfies $H * H^T = ((n-1) * I$. $n = 2 \pmod 4$. The rows of $1/2(H + I + J), 1/2(-H + I + J)$, plus the zero and all-ones vectors form the elements of a binary non-linear $(n-1, 2*n, 1/2*(n-2))$ code.

```
gap> H6 := [[0,1,1,1,1,1],[1,0,1,-1,-1,1],[1,1,0,1,-1,-1],
> [1,-1,1,0,1,-1],[1,-1,-1,1,0,1],[1,1,-1,-1,1,0]];;
gap> C1 := ConferenceCode( H6 );
a (5,12,2)1..4 conference code over GF(2)
gap> IsLinearCode( C1 );
```

```
false
```

`ConferenceCode(` n `)`

GUAVA constructs a symmetric conference matrix of order $n+1$ ($n = 1 \pmod 4$) and uses the rows of that matrix, plus the zero and all-ones vectors, to construct a binary non-linear $(n, 2*(n+1), 1/2*(n-1))$ code.

```
gap> C2 := ConferenceCode( 5 );
a (5,12,2)1..4 conference code over GF(2)
gap> Elements( C2 );
[ [ 0 0 0 0 0 ], [ 1 1 0 1 0 ], [ 1 1 1 0 0 ], [ 0 1 1 0 1 ],
  [ 1 0 0 1 1 ], [ 0 0 1 1 1 ], [ 1 0 1 0 1 ], [ 0 1 0 1 1 ],
  [ 1 0 1 1 0 ], [ 0 1 1 1 0 ], [ 1 1 0 0 1 ], [ 1 1 1 1 1 ] ]
```

66.53 MOLSCode

`MOLSCode(` n`,` q `)`
`MOLSCode(` q `)`

`MOLSCode` returns an $(n, q^2, n-1)$ code over `GF(`q`)`. The code is created from $n-2$ **Mutually Orthogonal Latin Squares** (MOLS) of size $q*q$. The default for n is 4. GUAVA can construct a MOLS code for $n-2 \leq q$; q must be a prime power, $q > 2$. If there are no $n-2$ MOLS, an error is signalled.

Since each of the $n-2$ MOLS is a $q*q$ matrix, we can create a code of size q^2 by listing in each code element the entries that are in the same position in each of the MOLS. We precede each of these lists with the two coordinates that specify this position, making the word length become n.

The MOLS codes are MDS codes (see 66.37).

```
gap> C1 := MOLSCode( 6, 5 );
a (6,25,5)3..4 code generated by 4 MOLS of order 5 over GF(5)
gap> mols := List( [1 .. WordLength(C1) - 2 ], function( nr )
>       local ls, el;
>       ls := NullMat( Size(Field(C1)), Size(Field(C1)) );
>       for el in VectorCodeword( Elements( C1 ) ) do
>          ls[IntFFE(el[1])+1][IntFFE(el[2])+1] := el[nr + 2];
>       od;
>       return ls;
>    end );;
gap> AreMOLS( mols );
true
gap> C2 := MOLSCode( 11 );
a (4,121,3)2 code generated by 2 MOLS of order 11 over GF(11)
```

66.54 RandomCode

`RandomCode(` n`,` M`,` F `)`

`RandomCode` returns a random unrestricted code of size M with word length n over F. M must be less than or equal to the number of elements in the space $GF(q)^n$.

The function RandomLinearCode returns a random linear code (see 66.70).

```
gap> C1 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> MinimumDistance(C1);
3
gap> C2 := RandomCode( 6, 10, GF(8) );
a (6,10,1..6)4..6 random unrestricted code over GF(8)
gap> C1 = C2;
false
```

66.55 NordstromRobinsonCode

NordstromRobinsonCode()

NordstromRobinsonCode returns a Nordstrom-Robinson code, the best code with word length $n = 16$ and minimum distance $d = 6$ over GF(2). This is a non-linear $(16, 256, 6)$ code.

```
gap> C := NordstromRobinsonCode();
a (16,256,6)4 Nordstrom-Robinson code over GF(2)
gap> OptimalityCode( C );
0
```

66.56 GreedyCode

GreedyCode(L, d, F)

GreedyCode returns a Greedy code with design distance d over F. The code is constructed using the Greedy algorithm on the list of vectors L. This algorithm checks each vector in L and adds it to the code if its distance to the current code is greater than or equal to d. It is obvious that the resulting code has a minimum distance of at least d.

Note that Greedy codes are often linear codes.

The function LexiCode creates a Greedy code from a basis instead of an enumerated list (see 66.57).

```
gap> C1 := GreedyCode( Tuples( Elements( GF(2) ), 5 ), 3, GF(2) );
a (5,4,3..5)2 Greedy code, user defined basis over GF(2)
gap> C2 := GreedyCode( Permuted( Tuples( Elements( GF(2) ), 5 ),
>                         (1,4) ), 3, GF(2) );
a (5,4,3..5)2 Greedy code, user defined basis over GF(2)
gap> C1 = C2;
false
```

66.57 LexiCode

LexiCode(n, d, F)

In this format, Lexicode returns a Lexicode with word length n, design distance d over F. The code is constructed using the Greedy algorithm on the lexicographically ordered list of all vectors of length n over F. Every time a vector is found that has a distance to the current code of at least d, it is added to the code. This results, obviously, in a code with minimum distance greater than or equal to d.

```
gap> C := LexiCode( 4, 3, GF(5) );
a (4,17,3..4)2..4 lexicode over GF(5)
```

`LexiCode(` B`,` d`,` F `)`

When called in this format, `LexiCode` uses the basis B instead of the standard basis. B is a matrix of vectors over F. The code is constructed using the Greedy algorithm on the list of vectors spanned by B, ordered lexicographically with respect to B.

```
gap> B := [ [Z(2)^0, 0*Z(2), 0*Z(2)], [Z(2)^0, Z(2)^0, 0*Z(2)] ];;
gap> C := LexiCode( B, 2, GF(2) );
a linear [3,1,2]1..2 lexicode over GF(2)
```

Note that binary Lexicodes are always linear.

The function `GreedyCode` creates a Greedy code that is not restricted to a lexicographical order (see 66.56).

66.58 Generating Linear Codes

The following sections describe functions for constructing linear codes. A linear code always has a generator or check matrix.

The first two sections describe functions that generate linear codes from the generator matrix (66.59) or check matrix (66.60). All linear codes can be constructed with these functions.

The next sections describes some well known codes, like Hamming codes (66.61), Reed-Muller codes (66.62) and the extended Golay codes (66.63 and 66.64).

A large and powerful family of codes are alternant codes. They are obtained by a small modification of the parity check matrix of a BCH code. See sections 66.65, 66.66, 66.67 and 66.68.

The next section describes a function for generating random linear codes (see 66.70).

66.59 GeneratorMatCode

`GeneratorMatCode(` G `[,` *Name* `],` F `)`

`GeneratorMatCode` returns a linear code with generator matrix G. G must be a matrix over Galois field F. *Name* can contain a short description of the code. The generator matrix is the basis of the elements of the code. The resulting code has word length n, dimension k if G is a $k*n$-matrix. If $GF(q)$ is the field of the code, the size of the code will be q^k.

If the generator matrix does not have full row rank, the linearly dependent rows are removed. This is done by the function `BaseMat` (see 34.19) and results in an equal code. The generator matrix can be retrieved with the function `GeneratorMat` (see 66.24).

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := GeneratorMatCode( G, GF(3) );
a linear [5,3,1..2]1..2 code defined by generator matrix over GF(3)
gap> C2 := GeneratorMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a linear [5,5,1]0 code defined by generator matrix over GF(2)
gap> GeneratorMatCode( Elements( NordstromRobinsonCode() ), GF(2) );
a linear [16,11,1..4]2 code defined by generator matrix over GF(2)
# This is the smallest linear code that contains the N-R code
```

66.60 CheckMatCode

`CheckMatCode(` *H* `[,` *Name* `],` *F* `)`

`CheckMatCode` returns a linear code with check matrix H. H must be a matrix over Galois field F. *Name* can contain a short description of the code. The parity check matrix is the transposed of the nullmatrix of the generator matrix of the code. Therefore, $c * H^T = 0$ where c is an element of the code. If H is a $r * n$-matrix, the code has word length n, redundancy r and dimension $n - r$.

If the check matrix does not have full row rank, the linearly dependent rows are removed. This is done by the function `BaseMat` (see 34.19) and results in an equal code. The check matrix can be retrieved with the function `CheckMat` (see 66.25).

```
gap> G := Z(3)^0 * [[1,0,1,2,0],[0,1,2,1,1],[0,0,1,2,1]];;
gap> C1 := CheckMatCode( G, GF(3) );
a linear [5,2,1..2]2..3 code defined by check matrix over GF(3)
gap> CheckMat(C1);
[ [ Z(3)^0, 0*Z(3), Z(3)^0, Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3), Z(3)^0 ] ]
gap> C2 := CheckMatCode( IdentityMat( 5, GF(2) ), GF(2) );
a linear [5,0,5]5 code defined by check matrix over GF(2)
```

66.61 HammingCode

`HammingCode(` *r*`,` *F* `)`

`HammingCode` returns a Hamming code with redundancy r over F. A Hamming code is a single-error-correcting code. The parity check matrix of a Hamming code has all nonzero vectors of length r in its columns, except for a multiplication factor. The decoding algorithm of the Hamming code (see 66.43) makes use of this property.

If q is the size of its field F, the returned Hamming code is a linear $[(q^r - 1)/(q - 1), (q^r - 1)/(q - 1) - r, 3]$ code.

```
gap> C1 := HammingCode( 4, GF(2) );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := HammingCode( 3, GF(9) );
a linear [91,88,3]1 Hamming (3,9) code over GF(9)
```

66.62 ReedMullerCode

`ReedMullerCode(` *r*`,` *k* `)`

`ReedMullerCode` returns a binary **Reed-Muller code** $R(r, k)$ with dimension k and order r. This is a code with length 2^k and minimum distance 2^{k-r}. By definition, the r^{th} order binary Reed-Muller code of length $n = 2^m$, for $0 \leq r \leq m$, is the set of all vectors f, where f is a Boolean function which is a polynomial of degree at most r.

```
gap> ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

66.63 ExtendedBinaryGolayCode

`ExtendedBinaryGolayCode( )`

`ExtendedBinaryGolayCode` returns an extended binary Golay code. This is a $[24, 12, 8]$ code. Puncturing in the last position results in a perfect binary Golay code (see 66.75). The code is self-dual (see 66.38).

```
gap> C := ExtendedBinaryGolayCode();
a linear [24,12,8]4 extended binary Golay code over GF(2)
gap> P := PuncturedCode(C);
a linear [23,12,7]3 punctured code
gap> P = BinaryGolayCode();
true
```

66.64 ExtendedTernaryGolayCode

`ExtendedTernaryGolayCode( )`

`ExtendedTernaryGolayCode` returns an extended ternary Golay code. This is a $[12, 6, 6]$ code. Puncturing this code results in a perfect ternary Golay code (see 66.76). The code is self-dual (see 66.38).

```
gap> C := ExtendedTernaryGolayCode();
a linear [12,6,6]3 extended ternary Golay code over GF(3)
gap> P := PuncturedCode(C);
a linear [11,6,5]2 punctured code
gap> P = TernaryGolayCode();
true
```

66.65 AlternantCode

`AlternantCode(` r, Y, F `)`
`AlternantCode(` r, Y, *alpha*, F `)`

`AlternantCode` returns an **alternant code**, with parameters r, Y and *alpha* (optional). r is the design redundancy of the code. Y and *alpha* are both vectors of length n from which the parity check matrix is constructed. The check matrix has entries of the form $a_i^j y_i$. If no *alpha* is specified, the vector $[1, a, a^2, .., a^{n-1}]$ is used, where a is a primitive element of a Galois field F.

```
gap> Y := [ 1, 1, 1, 1, 1, 1, 1];; a := PrimitiveUnityRoot( 2, 7 );;
gap> alpha := List( [0..6], i -> a^i );;
gap> C := AlternantCode( 2, Y, alpha, GF(8) );
a linear [7,3,3..4]3..4 alternant code over GF(8)
```

66.66 GoppaCode

`GoppaCode(` G, L `)`

`GoppaCode` returns a Goppa code from Goppa polynomial G, having coefficients in a Galois Field $GF(q^m)$. L must be a list of elements in $GF(q^m)$, that are not roots of G. The word length of the code is equal to the length of L. The parity check matrix contains entries of the form $a_i^j G(a_i)$, a_i in L. The function `VerticalConversionFieldMat` converts this matrix to a matrix with entries in $GF(q)$ (see 66.130).

```
gap> x := Indeterminate( GF(2) );; x.name := "x";;
gap> G := x^2 + x + 1;; L := Elements( GF(8) );;
gap> C := GoppaCode( G, L );
a linear [8,2,5]3 Goppa code over GF(2)
```

`GoppaCode(` G`,` n `)`

When called with parameter n, GUAVA constructs a list L of length n, such that no element of L is a root of G.

```
gap> x := Indeterminate( GF(2) );; x.name := "x";;
gap> G := x^2 + x + 1;;
gap> C := GoppaCode( G, 8 );
a linear [8,2,5]3 Goppa code over GF(2)
```

66.67 GeneralizedSrivastavaCode

`GeneralizedSrivastavaCode(` a`,` w`,` z`,` F `)`
`GeneralizedSrivastavaCode(` a`,` w`,` z`,` t`,` F `)`

`GeneralizedSrivastavaCode` returns a generalized Srivastava code with parameters a, w, z, t. $a = a_1, ..., a_n$ and $w = w_1, ..., w_s$ are lists of $n + s$ distinct elements of $F = GF(q^m)$, z is a list of length n of nonzero elements of $GF(q^m)$. The parameter t determines the designed distance: $d \geq st + 1$. The parity check matrix of this code has entries of the form

$$\frac{z_i}{(a_i - w_l)^k}$$

`VerticalConversionFieldMat` converts this matrix to a matrix with entries in $GF(q)$ (see 66.130). The default for t is 1. The original Srivastava codes (see 66.68) are a special case $t = 1$, $z_i = a_i^{\mu}$ for some μ.

```
gap> a := Filtered( Elements( GF(2^6) ), e -> e in GF(2^3) );;
gap> w := [ Z(2^6) ];; z := List( [1..8], e -> 1 );;
gap> C := GeneralizedSrivastavaCode( a, w, z, 1, GF(64) );
a linear [8,2,2..5]3..4 generalized Srivastava code over GF(2)
```

66.68 SrivastavaCode

`SrivastavaCode(` a`,` w`,` F `)`
`SrivastavaCode(` a`,` w`,` mu`,` F `)`

`SrivastavaCode` returns a Srivastava code with parameters a, w, mu. $a = a_1, ..., a_n$ and $w = w_1, ..., w_s$ are lists of $n + s$ distinct elements of $F = GF(q^m)$. The default for mu is 1. The Srivastava code is a generalized Srivastava code (see 66.67), in which $z_i = a_i^{mu}$ for some mu and $t = 1$.

```
gap> a := Elements( GF(11) ){[2..8]};;
gap> w := Elements( GF(11) ){[9..10]};;
gap> C := SrivastavaCode( a, w, 2, GF(11) );
```

```
a linear [7,5,3]2 Srivastava code over GF(11)
gap> IsMDSCode( C );
true     # Always true if F is a prime field
```

66.69 CordaroWagnerCode

`CordaroWagnerCode(` *n* `)`

`CordaroWagnerCode` returns a binary Cordaro-Wagner code. This is a code of length n and dimension 2 having the best possible minimum distance d. This code is just a little bit less trivial than `RepetitionCode` (see 66.84).

```
gap> C := CordaroWagnerCode( 11 );
a linear [11,2,7]5 Cordaro-Wagner code over GF(2)
gap> Elements(C);
[ [ 0 0 0 0 0 0 0 0 0 0 0 ], [ 1 1 1 1 1 1 1 0 0 0 0 ],
  [ 0 0 0 0 1 1 1 1 1 1 1 ], [ 1 1 1 1 0 0 0 1 1 1 1 ] ]
```

66.70 RandomLinearCode

`RandomLinearCode(` *n*`,` *k* `,` *F* `)`

`RandomLinearCode` returns a random linear code with word length n, dimension k over field F.

To create a random unrestricted code, use `RandomCode` (see 66.54).

```
gap> C := RandomLinearCode( 15, 4, GF(3) );
a linear [15,4,1..4]6..10 random linear code over GF(3)
gap> RandomSeed( 13 ); C1 := RandomLinearCode( 12, 5, GF(5) );
a linear [12,5,1..5]4..7 random linear code over GF(5)
gap> RandomSeed( 13 ); C2 := RandomLinearCode( 12, 5, GF(5) );
a linear [12,5,1..5]4..7 random linear code over GF(5)
gap> C1 = C2;
true     # Thanks to RandomSeed
```

66.71 BestKnownLinearCode

`BestKnownLinearCode(` *n*`,` *k* `,` *F* `)`

`BestKnownLinearCode` returns the best known linear code of length n, dimension k over field F. The function uses the tables described in section 66.120 to construct this code.

```
gap> C1 := BestKnownLinearCode( 23, 12, GF(2) );
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> C1 = BinaryGolayCode();
true
gap> Display( BestKnownLinearCode( 8, 4, GF(4) ) );
a linear [8,4,4]2..3 U|U+V construction code of
U: a cyclic [4,3,2]1 dual code of
   a cyclic [4,1,4]3 repetition code over GF(4)
V: a cyclic [4,1,4]3 repetition code over GF(4)
```

```
gap> C := BestKnownLinearCode(131,47);
a linear [131,47,28..32]23..68 shortened code
```

`BestKnownLinearCode(` *rec* `)`

In this form, *rec* must be a record containing the fields `lowerBound`, `upperBound` and `construction`. It uses the information in this field to construct a code. This form is meant to be used together with the function `BoundsMinimumDistance` (see 66.120), if the bounds are already calculated.

```
gap> bounds := BoundsMinimumDistance( 20, 17, GF(4) );
an optimal linear [20,17,d] code over GF(4) has d=3
gap> C := BestKnownLinearCode( bounds );
a linear [20,17,3]2 shortened code
gap> C = BestKnownLinearCode( 20, 17, GF(4) );
true
```

66.72 Generating Cyclic Codes

The elements of a cyclic code C are all multiples of a polynomial $g(x)$, where calculations are carried out modulo $x^n - 1$. Therefore, the elements always have a degree less than n. A cyclic code is an ideal in the ring of polynomials modulo $x^n - 1$. The polynomial $g(x)$ is called the **generator polynomial** of C. This is the unique monic polynomial of least degree that generates C. It is a divisor of the polynomial $x^n - 1$.

The **check polynomial** is the polynomial $h(x)$ with $g(x) * h(x) = x^n - 1$. Therefore it is also a divisor of $x^n - 1$. The check polynomial has the property that $c(x) * h(x) = 0 \pmod{(x^n - 1)}$ for every codeword $c(x)$.

The first two sections describe functions that generate cyclic codes from a given generator or check polynomial. All cyclic codes can be constructed using these functions.

The next sections describe the two cyclic Golay codes (see 66.75 and 66.76).

The next sections describe functions that generate cyclic codes from a prescribed set of roots of the generator polynomial, among which the BCH codes. See 66.77, 66.78, 66.79 and 66.80.

The next sections describe the trivial codes (see 66.82, 66.83, 66.84).

66.73 GeneratorPolCode

`GeneratorPolCode(` g`,` n `[,` *Name* `],` F `)`

`GeneratorPolCode` creates a cyclic code with a generator polynomial g, word length n, over F. g can be entered as a polynomial over F, or as a list of coefficients over F or `Integers`. If g is a list of integers, these are first converted to F. *Name* can contain a short description of the code.

If g is not a divisor of $x^n - 1$, it cannot be a generator polynomial. In that case, a code is created with generator polynomial $gcd(g, x^n - 1)$, i.e. the greatest common divisor of g and $x^n - 1$. This is a valid generator polynomial that generates the ideal $< g$. See 66.72.

```
gap> P := Polynomial(GF(2), Z(2)*[1,0,1]);
Z(2)^0*(X(GF(2))^2 + 1)
```

```
gap> G := GeneratorPolCode(P, 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol(G);
Z(2)^0*(X(GF(2)) + 1)
gap> G2 := GeneratorPolCode([1,1], 7, GF(2));
a cyclic [7,6,1..2]1 code defined by generator polynomial over GF(2)
gap> GeneratorPol(G2);
Z(2)^0*(X(GF(2)) + 1)
```

66.74 CheckPolCode

`CheckPolCode(` *h*, *n* `[,` *Name* `],` *F* `)`

`CheckPolCode` creates a cyclic code with a check polynomial h, word length n, over F. h can be entered as a polynomial over F, or as a list of coefficients over F or `Integers`. If h is a list of integers, these are first converted to F. *Name* can contain a short description of the code.

If h is not a divisor of $x^n - 1$, it cannot be a check polynomial. In that case, a code is created with check polynomial $gcd(h, x^n - 1)$, i.e. the greatest common divisor of h and $x^n - 1$. This is a valid check polynomial that yields the same elements as the ideal $< h$. See 66.72.

```
gap> P := Polynomial(GF(3), Z(3)*[1,0,2]);
Z(3)^0*(X(GF(3))^2 + 2)
gap> H := CheckPolCode(P, 7, GF(3));
a cyclic [7,1,7]4 code defined by check polynomial over GF(3)
gap> CheckPol(H);
Z(3)^0*(X(GF(3)) + 2)
gap> Gcd(P, X(GF(3))^7-1);
Z(3)^0*(X(GF(3)) + 2)
```

66.75 BinaryGolayCode

`BinaryGolayCode()`

`BinaryGolayCode` returns a binary Golay code. This is a perfect [23,12,7] code. It is also cyclic, and has generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it results in an extended Golay code (see 66.63). There's also the ternary Golay code (see 66.76).

```
gap> BinaryGolayCode();
a cyclic [23,12,7]3 binary Golay code over GF(2)
gap> ExtendedBinaryGolayCode() = ExtendedCode(BinaryGolayCode());
true
gap> IsPerfectCode(BinaryGolayCode());
true
```

66.76 TernaryGolayCode

`TernaryGolayCode()`

`TernaryGolayCode` returns a ternary Golay code. This is a perfect [11,6,5] code. It is also cyclic, and has generator polynomial $g(x) = 2 + x^2 + 2x^3 + x^4 + x^5$. Extending it results in an extended Golay code (see 66.64). There's also the binary Golay code (see 66.75).

```
gap> TernaryGolayCode();
a cyclic [11,6,5]2 ternary Golay code over GF(3)
gap> ExtendedTernaryGolayCode() = ExtendedCode(TernaryGolayCode());
true
```

66.77 RootsCode

`RootsCode(` *n*, *list* `)`

This is the generalization of the BCH, Reed-Solomon and quadratic residue codes (see 66.78, 66.79 and 66.80). The user can give a length of the code n and a prescribed set of zeros. The argument *list* must be a valid list of primitive n^{th} roots of unity in a splitting field $GF(q^m)$. The resulting code will be over the field $GF(q)$. The function will return the largest possible cyclic code for which the list *list* is a subset of the roots of the code. From this list, GUAVA calculates the entire set of roots.

```
gap> a := PrimitiveUnityRoot( 3, 14 );
Z(3^6)^52
gap> C1 := RootsCode( 14, [ a^0, a, a^3 ] );
a cyclic [14,7,3..6]3..7 code defined by roots over GF(3)
gap> MinimumDistance( C1 );
4
gap> b := PrimitiveUnityRoot( 2, 15 );
Z(2^4)
gap> C2 := RootsCode( 15, [ b, b^2, b^3, b^4 ] );
a cyclic [15,7,5]3..5 code defined by roots over GF(2)
gap> C2 = BCHCode( 15, 5, GF(2) );
true
```

`RootsCode(` *n*, *list*, *F* `)`

In this second form, the second argument is a list of integers, ranging from 0 to n-1. The resulting code will be over a field F. GUAVA calculates a primitive n^{th} root of unity, α, in the extension field of F. It uses the set of the powers of α in the list as a prescribed set of zeros.

```
gap> C := RootsCode( 4, [ 1, 2 ], GF(5) );
a cyclic [4,2,3]2 code defined by roots over GF(5)
gap> RootsOfCode( C );
[ Z(5), Z(5)^2 ]
gap> C = ReedSolomonCode( 4, 3 );
true
```

66.78 BCHCode

`BCHCode(` *n*, *d* , *F* `)`
`BCHCode(` *n*, *b*, *d*, *F* `)`

The function BCHCode returns a **Bose-Chaudhuri-Hockenghem code** (or BCH code for short). This is the largest possible cyclic code of length n over field F, whose generator polynomial has zeros

$$a^b, a^{b+1}, ..., a^{b+d-2},$$

where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer > 1 and m is the multiplicative order of q modulo n. Default value for b is 1. The length n of the code and the size q of the field must be relatively prime. The generator polynomial is equal to the product of the minimal polynomials of $X^b, X^{b+1}, ..., X^{b+d-2}$.

Special cases are $b = 1$ (resulting codes are called **narrow-sense** BCH codes), and $n = q^m - 1$ (known as **primitive** BCH codes). GUAVA calculates the largest value of d' for which the BCH code with designed distance d' coincides with the BCH code with designed distance d. This distance is called the **Bose distance** of the code. The true minimum distance of the code is greater than or equal to the Bose distance.

Printed are the designed distance (to be precise, the Bose distance) delta, and the starting power b.

```
gap> C1 := BCHCode( 15, 3, 5, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C1.designedDistance;
7
gap> C2 := BCHCode( 23, 2, GF(2) );
a cyclic [23,12,5..7]3 BCH code, delta=5, b=1 over GF(2)
gap> C2.designedDistance;
5
gap> MinimumDistance(C2);
7
```

66.79 ReedSolomonCode

ReedSolomonCode(n, d)

ReedSolomonCode returns a **Reed-Solomon code** of length n, designed distance d. This code is a primitive narrow-sense BCH code over the field $GF(q)$, where $q = n + 1$. The dimension of an RS code is $n - d + 1$. According to the Singleton bound (see 66.111) the dimension cannot be greater than this, so the true minimum distance of an RS code is equal to d and the code is maximum distance separable (see 66.37).

```
gap> C1 := ReedSolomonCode( 3, 2 );
a cyclic [3,2,2]1 Reed-Solomon code over GF(4)
gap> C2 := ReedSolomonCode( 4, 3 );
a cyclic [4,2,3]2 Reed-Solomon code over GF(5)
gap> RootsOfCode( C2 );
[ Z(5), Z(5)^2 ]
gap> IsMDSCode(C2);
true
```

66.80 QRCode

QRCode(n, F)

QRCode returns a quadratic residue code. If F is a field $GF(q)$, then q must be a quadratic residue modulo n. That is, an x exists with $x^2 = q \pmod n$. Both n and q must be primes. Its generator polynomial is the product of the polynomials $x - a^i$. a is a primitive n^{th} root of unity, and i is an integer in the set of quadratic residues modulo n.

```
gap> C1 := QRCode( 7, GF(2) );
a cyclic [7,4,3]1 quadratic residue code over GF(2)
gap> IsEquivalent( C1, HammingCode( 3, GF(2) ) );
true
gap> C2 := QRCode( 11, GF(3) );
a cyclic [11,6,4..5]2 quadratic residue code over GF(3)
gap> C2 = TernaryGolayCode();
true
```

66.81 FireCode

FireCode(G, b)

FireCode constructs a (binary) Fire code. G is a primitive polynomial of degree m, factor of $x^r - 1$. b an integer $0 \leq b \leq m$ not divisible by r, that determines the burst length of a single error burst that can be corrected. The argument G can be a polynomial with base ring $GF(2)$, or a list of coefficients in $GF(2)$. The generator polynomial is defined as the product of G and $x^{2b-1} + 1$.

```
gap> G := Polynomial( GF(2), Z(2)^0 * [ 1, 0, 1, 1 ] );
Z(2)^0*(X(GF(2))^3 + X(GF(2))^2 + 1)
gap> Factors( G );
[ Z(2)^0*(X(GF(2))^3 + X(GF(2))^2 + 1) ]     # So it is primitive
gap> C := FireCode( G, 3 );
a cyclic [35,27,1..4]2..6 3 burst error correcting fire code over GF(2)
gap> MinimumDistance( C );
4     # Still it can correct bursts of length 3
```

66.82 WholeSpaceCode

WholeSpaceCode(n, F)

WholeSpaceCode returns the cyclic whole space code of length n over F. This code consists of all polynomials of degree less than n and coefficients in F.

```
gap> C := WholeSpaceCode( 5, GF(3) );
a cyclic [5,5,1]0 whole space code over GF(3)
```

66.83 NullCode

NullCode(n, F)

NullCode returns the zero-dimensional nullcode with length n over F. This code has only one word: the all zero word. It is cyclic though!

```
gap> C := NullCode( 5, GF(3) );
a cyclic [5,0,5]5 nullcode over GF(3)
```

```
gap> Elements( C );
[ 0 ]                 # this is the polynomial 0
gap> TreatAsVector( Elements( C ) ); Elements( C );
[ [ 0 0 0 0 0 ] ]     # this is the vector 0
```

66.84 RepetitionCode

`RepetitionCode( n, F )`

`RepetitionCode` returns the cyclic repetition code of length n over F. The code has as many elements as F, because each codeword consists of a repetition of one of these elements.

```
gap> C := RepetitionCode( 3, GF(5) );
a cyclic [3,1,3]2 repetition code over GF(5)
gap> Elements( C );
[ 0, x^2 + x + 1, 2x^2 + 2x + 2, 4x^2 + 4x + 4, 3x^2 + 3x + 3 ]
gap> IsPerfectCode( C );
false
gap> IsMDSCode( C );
true
```

66.85 CyclicCodes

`CyclicCodes( n, F )`

`CyclicCodes` returns a list of all cyclic codes of length n over F. It constructs all possible generator polynomials from the factors of $x^n - 1$. Each combination of these factors yields a generator polynomial after multiplication.

`NrCyclicCodes( n, F )`

The function `NrCyclicCodes` calculates the number of cyclic codes of length n over field F.

```
gap> NrCyclicCodes( 23, GF(2) );
8
gap> codelist := CyclicCodes( 23, GF(2) );
[ a cyclic [23,23,1]0 enumerated code over GF(2),
  a cyclic [23,22,1..2]1 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,0,23]23 enumerated code over GF(2),
  a cyclic [23,11,1..8]4..7 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2),
  a cyclic [23,1,23]11 enumerated code over GF(2),
  a cyclic [23,12,1..7]3 enumerated code over GF(2) ]
gap> BinaryGolayCode() in codelist;
true
gap> RepetitionCode( 23, GF(2) ) in codelist;
true
gap> CordaroWagnerCode( 23 ) in codelist;
false      # This code is not cyclic
```

66.86 Manipulating Codes

This section describes several functions GUAVA uses to manipulate codes. Some of the best codes are obtained by starting with for example a BCH code, and manipulating it.

In some cases, it is faster to perform calculations with a manipulated code than to use the original code. For example, if the dimension of the code is larger than half the word length, it is generally faster to compute the weight distribution by first calculating the weight distribution of the dual code than by directly calculating the weight distribution of the original code. The size of the dual code is smaller in these cases.

Because GUAVA keeps all information in a code record, in some cases the information can be preserved after manipulations. Therefore, computations do not always have to start from scratch.

The next sections describe manipulating function that take a code with certain parameters, modify it in some way and return a different code. See 66.87, 66.88, 66.89, 66.90, 66.91, 66.92, 66.93, 66.94, 66.95, 66.96, 66.97, 66.98, 66.99, 66.100, 66.102, 66.103 and 66.101.

The next sections describe functions that generate a new code out of two codes. See 66.104, 66.105, 66.106, 66.107 and 66.108.

66.87 ExtendedCode

`ExtendedCode(` C `[,` i `] )`

`ExtendedCode` **extends** the code C i times and returns the result. i is equal to 1 by default. Extending is done by adding a parity check bit after the last coordinate. The coordinates of all codewords now add up to zero. In the binary case, each codeword has even weight.

The word length increases by i. The size of the code remains the same. In the binary case, the minimum distance increases by one if it was odd. In other cases, that is not always true.

A cyclic code in general is no longer cyclic after extending.

```
gap> C1 := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> C2 := ExtendedCode( C1 );
a linear [8,4,4]2 extended code
gap> IsEquivalent( C2, ReedMullerCode( 1, 3 ) );
true
gap> List( Elements( C2 ), WeightCodeword );
[ 0, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8 ]
gap> PuncturedCode( C2 ) = C1;
true
gap> C3 := EvenWeightSubcode( C1 );
a linear [7,3,4]2..3 even weight subcode
```

To undo extending, call `PuncturedCode` (see 66.88). The function `EvenWeightSubcode` (see 66.89) also returns a related code with only even weights, but without changing its word length.

66.88 PuncturedCode

`PuncturedCode(` C `)`

`PuncturedCode` **punctures** C in the last column, and returns the result. Puncturing is done simply by cutting off the last column from each codeword. This means the word length decreases by one. The minimum distance in general also decrease by one.

`PuncturedCode(` C`,` L `)`

`PuncturedCode` punctures C in the columns specified by L, a list of integers. All columns specified by L are omitted from each codeword. If l is the length of L (so the number of removed columns), the word length decreases by l. The minimum distance can also decrease by l or less.

Puncturing a cyclic code in general results in a non-cyclic code. If the code is punctured in all the columns where a word of minimal weight is unequal to zero, the dimension of the resulting code decreases.

```
gap> C1 := BCHCode( 15, 5, GF(2) );
a cyclic [15,7,5]3..5 BCH code, delta=5, b=1 over GF(2)
gap> C2 := PuncturedCode( C1 );
a linear [14,7,4]3..5 punctured code
gap> ExtendedCode( C2 ) = C1;
false
gap> PuncturedCode( C1, [1,2,3,4,5,6,7] );
a linear [8,7,1..2]1 punctured code
gap> PuncturedCode( WholeSpaceCode( 4, GF(5) ) );
a linear [3,3,1]0 punctured code  # The dimension decreased from 4 to 3
```

`ExtendedCode` extends the code again (see 66.87) although in general this does not result in the old code.

66.89 EvenWeightSubcode

`EvenWeightSubcode(` C `)`

`EvenWeightSubcode` returns the **even weight subcode** of C, consisting of all codewords of C with even weight. If C is a linear code and contains words of odd weight, the resulting code has a dimension of one less. The minimum distance always increases with one if it was odd. If C is a binary cyclic code, and $g(x)$ is its generator polynomial, the even weight subcode either has generator polynomial $g(x)$ (if $g(x)$ is divisible by $x-1$) or $g(x)*(x-1)$ (if no factor $x-1$ was present in $g(x)$). So the even weight subcode is again cyclic.

Of course, if all codewords of C are already of even weight, the returned code is equal to C.

```
gap> C1 := EvenWeightSubcode( BCHCode( 8, 4, GF(3) ) );
an (8,33,4..8)3..8 even weight subcode
gap> List( Elements( C1 ), WeightCodeword );
[ 0, 4, 4, 4, 4, 4, 6, 4, 4, 4, 6, 4, 4, 4, 8, 6, 8, 4, 6, 4, 4, 6,
  4, 4, 6, 8, 4, 4, 6, 4, 8, 4, 6 ]
gap> EvenWeightSubcode( ReedMullerCode( 1, 3 ) );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
```

`ExtendedCode` also returns a related code of only even weights, but without reducing its dimension (see 66.87).

66.90 PermutedCode

`PermutedCode(` C, L `)`

`PermutedCode` returns C after column permutations. L is the permutation to be executed on the columns of C. If C is cyclic, the result in general is no longer cyclic. If a permutation results in the same code as C, this permutation belongs to the **automorphism group** of C (see 66.42). In any case, the returned code is **equivalent** to C (see 66.40).

```
gap> C1 := PuncturedCode( ReedMullerCode( 1, 4 ) );
a linear [15,5,7]5 punctured code
gap> C2 := BCHCode( 15, 7, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C2 = C1;
false
gap> p := CodeIsomorphism( C1, C2 );
( 2,13, 7,10, 8, 3, 5, 4,14)(12,15)
gap> C3 := PermutedCode( C1, p );
a linear [15,5,7]5 permuted code
gap> C2 = C3;
true
```

66.91 ExpurgatedCode

`ExpurgatedCode(` C, L `)`

`ExpurgatedCode` **expurgates** code C by throwing away codewords in list L. C must be a linear code. L must be a list of codeword input. The generator matrix of the new code no longer is a basis for the codewords specified by L. Since the returned code is still linear, it is very likely that, besides the words of L, more codewords of C are no longer in the new code.

```
gap> C1 := HammingCode( 4 );; WeightDistribution( C1 );
[ 1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> L := Filtered( Elements(C1), i -> WeightCodeword(i) = 3 );;
gap> C2 := ExpurgatedCode( C1, L );
a linear [15,4,3..4]5..11 code, expurgated with 7 word(s)
gap> WeightDistribution( C2 );
[ 1, 0, 0, 0, 14, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ]
```

This function does not work on non-linear codes. For removing words from a non-linear code, use `RemovedElementsCode` (see 66.93). For expurgating a code of all words of odd weight, use `EvenWeightSubcode` (see 66.89).

66.92 AugmentedCode

`AugmentedCode(` C, L `)`

`AugmentedCode` returns C after **augmenting**. C must be a linear code, L must be a list of codeword input. The generator matrix of the new code is a basis for the codewords specified by L as well as the words that were already in code C. Note that the new code in general will consist of more words than only the codewords of C and the words L. The returned code is also a linear code.

```
gap> C31 := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> C32 := AugmentedCode(C31,["00000011","00000101","00010001"]);
a linear [8,7,1..2]1 code, augmented with 3 word(s)
gap> C32 = ReedMullerCode( 2, 3 );
true
```

`AugmentedCode(` C `)`

When called without a list of codewords, `AugmentedCode` returns C after adding the all-ones vector to the generator matrix. C must be a linear code. If the all-ones vector was already in the code, nothing happens and a copy of the argument is returned. If C is a binary code which does not contain the all-ones vector, the complement of all codewords is added.

```
gap> C1 := CordaroWagnerCode(6);
a linear [6,2,4]2..3 Cordaro-Wagner code over GF(2)
gap> [0,0,1,1,1,1] in C1;
true
gap> C2 := AugmentedCode( C1 );
a linear [6,3,1..2]2..3 code, augmented with 1 word(s)
gap> [1,1,0,0,0,0] in C2;     # the complement of [001111]
true
```

The function `AddedElementsCode` adds elements to the codewords instead of adding them to the basis (see 66.94).

66.93 RemovedElementsCode

`RemovedElementsCode(` C`,` L `)`

`RemovedElementsCode` returns code C after removing a list of codewords L from its elements. L must be a list of codeword input. The result is an unrestricted code.

```
gap> C1 := HammingCode( 4 );; WeightDistribution( C1 );
[ 1, 0, 0, 35, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> L := Filtered( Elements(C1), i -> WeightCodeword(i) = 3 );;
gap> C2 := RemovedElementsCode( C1, L );
a (15,2013,3..15)2..15 code with 35 word(s) removed
gap> WeightDistribution( C2 );
[ 1, 0, 0, 0, 105, 168, 280, 435, 435, 280, 168, 105, 35, 0, 0, 1 ]
gap> MinimumDistance( C2 );
3         # C2 is not linear, so the minimum weight does not have to
          # be equal to the minimum distance
```

Adding elements to a code is done by the function `AddedElementsCode` (see 66.94). To remove codewords from the base of a linear code, use `ExpurgatedCode` (see 66.91).

66.94 AddedElementsCode

`AddedElementsCode(` C`,` L `)`

`AddedElementsCode` returns code C after adding a list of codewords L to its elements. L must be a list of codeword input. The result is an unrestricted code.

```
gap> C1 := NullCode( 6, GF(2) );
a cyclic [6,0,6]6 nullcode over GF(2)
gap> C2 := AddedElementsCode( C1, "111111" );
a (6,2,1..6)3 code with 1 word(s) added
gap> IsCyclicCode( C2 );
true
gap> C3 := AddedElementsCode( C2, [ "101010", "010101" ] );
a (6,4,1..6)2 code with 2 word(s) added
gap> IsCyclicCode( C3 );
true
```

To remove elements from a code, use `RemovedElementsCode` (see 66.93). To add elements to the base of a linear code, use `AugmentedCode` (see 66.92).

66.95 ShortenedCode

`ShortenedCode(` C `)`

`ShortenedCode` returns code C shortened by taking a cross section. If C is a linear code, this is done by removing all codewords that start with a non-zero entry, after which the first column is cut off. If C was a $[n, k, d]$ code, the shortened code generally is a $[n-1, k-1, d]$ code. It is possible that the dimension remains the same; it is also possible that the minimum distance increases.

```
gap> C1 := HammingCode( 4 );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> C2 := ShortenedCode( C1 );
a linear [14,10,3]2 shortened code
```

If C is a non-linear code, `ShortenedCode` first checks which finite field element occurs most often in the first column of the codewords. The codewords not starting with this element are removed from the code, after which the first column is cut off. The resulting shortened code has at least the same minimum distance as C.

```
gap> C1 := ElementsCode( ["1000", "1101", "0011" ], GF(2) );
a (4,3,1..4)2 user defined unrestricted code over GF(2)
gap> MinimumDistance( C1 );
2
gap> C2 := ShortenedCode( C1 );
a (3,2,2..3)1..2 shortened code
gap> Elements( C2 );
[ [ 0 0 0 ], [ 1 0 1 ] ]
```

`ShortenedCode(` C`,` L `)`

When called in this format, `ShortenedCode` repeats the shortening process on each of the columns specified by L. L therefore is a list of integers. The column numbers in L are the numbers as they are **before** the shortening process. If L has l entries, the returned code has a word length of l positions shorter than C.

```
gap> C1 := HammingCode( 5, GF(2) );
```

```
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
gap> C2 := ShortenedCode( C1, [ 1, 2, 3 ] );
a linear [28,23,3]2 shortened code
gap> OptimalityLinearCode( C2 );
0
```

The function `LengthenedCode` lengthens the code again (only for linear codes), see 66.96. In general, this is not exactly the inverse function.

66.96 LengthenedCode

`LengthenedCode(` C `[,` i `] )`

`LengtenedCode` returns code C lengthened. C must be a linear code. First, the all-ones vector is added to the generator matrix (see 66.92). If the all-ones vector was already a codeword, nothing happens to the code. Then, the code is extended i times (see 66.87). i is equal to 1 by default. If C was an $[n, k]$ code, the new code generally is a $[n + i, k + 1]$ code.

```
gap> C1 := CordaroWagnerCode( 5 );
a linear [5,2,3]2 Cordaro-Wagner code over GF(2)
gap> C2 := LengthenedCode( C1 );
a linear [6,3,2]2..3 code, lengtened with 1 column(s)
```

`ShortenedCode` shortens the code, see 66.95. In general, this is not exactly the inverse function.

66.97 ResidueCode

`ResidueCode(` C `[,` w `] )`

The function `ResidueCode` takes a codeword c of C of weight w (if w is omitted, a codeword of minimal weight is used). C must be a linear code and w must be greater than zero. It removes this word and all its linear combinations from the code and then punctures the code in the coordinates where c is unequal to zero. The resulting code is an $[n - w, k - 1, d - \lfloor w * (q - 1)/q \rfloor]$ code.

```
gap> C1 := BCHCode( 15, 7 );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> C2 := ResidueCode( C1 );
a linear [8,4,4]2 residue code
gap> c := Codeword( [ 0,0,0,1,0,0,1,1,0,1,0,1,1,1,1 ], C1);;
gap> C3 := ResidueCode( C1, c );
a linear [7,4,3]1 residue code
```

66.98 ConstructionBCode

`ConstructionBCode(` C `)`

The function `ConstructionBCode` takes a binary linear code C and calculates the minimum distance of the dual of C (see 66.99). It then removes the columns of the parity check matrix of C where a codeword of the dual code of minimal weight has coordinates unequal to zero.

the resulting matrix is a parity check matrix for an $[n - dd, k - dd + 1, \geq d]$ code, where dd is the minimum distance of the dual of C.

```
gap> C1 := ReedMullerCode( 2, 5 );
a linear [32,16,8]6 Reed-Muller (2,5) code over GF(2)
gap> C2 := ConstructionBCode( C1 );
a linear [24,9,8]5..10 Construction B (8 coordinates)
gap> BoundsMinimumDistance( 24, 9, GF(2) );
an optimal linear [24,9,d] code over GF(2) has d=8  # so C2 is optimal
```

66.99 DualCode

`DualCode(` C `)`

`DualCode` returns the dual code of C. The dual code consists of all codewords that are orthogonal to the codewords of C. If C is a linear code with generator matrix G, the dual code has parity check matrix G (or if C has parity check matrix H, the dual code has generator matrix H). So if C is a linear [n, k] code, the dual code of C is a linear [n, n-k] code. If C is a cyclic code with generator polynomial $g(x)$, the dual code has the reciprocal polynomial of $g(x)$ as check polynomial.

The dual code is always a linear code, even if C is non-linear.

If a code C is equal to its dual code, it is called **self-dual**.

```
gap> R := ReedMullerCode( 1, 3 );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> RD := DualCode( R );
a linear [8,4,4]2 Reed-Muller (1,3) code over GF(2)
gap> R = RD;
true
gap> N := WholeSpaceCode( 7, GF(4) );
a cyclic [7,7,1]0 whole space code over GF(4)
gap> DualCode( N ) = NullCode( 7, GF(4) );
true
```

66.100 ConversionFieldCode

`ConversionFieldCode(` C `)`

`ConversionFieldCode` returns code C after converting its field. If the field of C is `GF(`q^m`)`, the returned code has field `GF(`q`)`. Each symbol of every codeword is replaced by a concatenation of m symbols from `GF(`q`)`. If C is an (n, M, d_1) code, the returned code is a $(n * m, M, d_2)$ code, where $d_2 > d_1$.

See also 66.131.

```
gap> R := RepetitionCode( 4, GF(4) );
a cyclic [4,1,4]3 repetition code over GF(4)
gap> R2 := ConversionFieldCode( R );
a linear [8,2,4]3..4 code, converted to basefield GF(2)
gap> Size( R ) = Size( R2 );
true
```

```
gap> GeneratorMat( R );
[ [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> GeneratorMat( R2 );
[ [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ]
```

66.101 CosetCode

`CosetCode(` C`,` w `)`

`CosetCode` returns the coset of a code C with respect to word w. w must be of the codeword type. Then, w is added to each codeword of C, yielding the elements of the new code. If C is linear and w is an element of C, the new code is equal to C, otherwise the new code is an unrestricted code.

Generating a coset is also possible by simply adding the word w to C. See 66.20.

```
gap> H := HammingCode(3, GF(2));
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> c := Codeword("1011011");; c in H;
false
gap> C := CosetCode(H, c);
a (7,16,3)1 coset code
gap> List(Elements(C), el-> Syndrome(H, el));
[ [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ],
  [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ], [ 1 1 1 ] ]
# All elements of the coset have the same syndrome in H
```

66.102 ConstantWeightSubcode

`ConstantWeightSubcode(` C`,` w `)`

`ConstantWeightSubcode` returns the subcode of C that only has codewords of weight w. The resulting code is a non-linear code, because it does not contain the all-zero vector.

```
gap> N := NordstromRobinsonCode();; WeightDistribution(N);
[ 1, 0, 0, 0, 0, 0, 112, 0, 30, 0, 112, 0, 0, 0, 0, 0, 1 ]
gap> C := ConstantWeightSubcode(N, 8);
a (16,30,6..16)5..8 code with codewords of weight 8
gap> WeightDistribution(C);
[ 0, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

`ConstantWeightSubcode(` C `)`

In this format, `ConstantWeightSubcode` returns the subcode of C consisting of all minimum weight codewords of C.

```
gap> E := ExtendedTernaryGolayCode();; WeightDistribution(E);
[ 1, 0, 0, 0, 0, 0, 264, 0, 0, 440, 0, 0, 24 ]
gap> C := ConstantWeightSubcode(E);
a (12,264,6..12)3..6 code with codewords of weight 6
gap> WeightDistribution(C);
[ 0, 0, 0, 0, 0, 0, 264, 0, 0, 0, 0, 0, 0 ]
```

66.103 StandardFormCode

`StandardFormCode(` C `)`

`StandardFormCode` returns C after putting it in standard form. If C is a non-linear code, this means the elements are organized using lexicographical order. This means they form a legal GAP3 `Set`.

If C is a linear code, the generator matrix and parity check matrix are put in standard form. The generator matrix then has an identity matrix in its left part, the parity check matrix has an identity matrix in its right part. Although GUAVA always puts both matrices in a standard form using `BaseMat`, this never alters the code. `StandardFormCode` even applies column permutations if unavoidable, and thereby changes the code. The column permutations are recorded in the construction history of the new code (see 66.47). C and the new code are of course equivalent.

If C is a cyclic code, its generator matrix cannot be put in the usual upper triangular form, because then it would be inconsistent with the generator polynomial. The reason is that generating the elements from the generator matrix would result in a different order than generating the elements from the generator polynomial. This is an unwanted effect, and therefore `StandardFormCode` just returns a copy of C for cyclic codes.

```
gap> G := GeneratorMatCode( Z(2) * [ [0,1,1,0], [0,1,0,1], [0,0,1,1] ],
> "random form code", GF(2) );
a linear [4,2,1..2]1..2 random form code over GF(2)
gap> Codeword( GeneratorMat( G ) );
[ [ 0 1 0 1 ], [ 0 0 1 1 ] ]
gap> Codeword( GeneratorMat( StandardFormCode( G ) ) );
[ [ 1 0 0 1 ], [ 0 1 0 1 ] ]
```

66.104 DirectSumCode

`DirectSumCode(` C_1, C_2 `)`

`DirectSumCode` returns the direct sum of codes C_1 and C_2. The direct sum code consists of every codeword of C_1 concatenated by every codeword of C_2. Therefore, if C_i was a (n_i, M_i, d_i) code, the result is a $(n_1 + n_2, M_1 * M_2, min(d_1, d_2))$ code.

If both C_1 and C_2 are linear codes, the result is also a linear code. If one of them is non-linear, the direct sum is non-linear too. In general, a direct sum code is not cyclic.

Performing a direct sum can also be done by adding two codes (see 66.20). Another often used method is the "u, u+v"-construction, described in 66.105.

```
gap> C1 := ElementsCode( [ [1,0], [4,5] ], GF(7) );;
gap> C2 := ElementsCode( [ [0,0,0], [3,3,3] ], GF(7) );;
gap> D := DirectSumCode(C1, C2);;
gap> Elements(D);
[ [ 1 0 0 0 0 ], [ 1 0 3 3 3 ], [ 4 5 0 0 0 ], [ 4 5 3 3 3 ] ]
gap> D = C1 + C2;    # addition = direct sum
true
```

66.105 UUVCode

UUVCode(C_1, C_2)

UUVCode returns the so-called $(u|u+v)$ construction applied to C_1 and C_2. The resulting code consists of every codeword u of C_1 concatenated by the sum of u and every codeword v of C_2. If C_1 and C_2 have different word lengths, sufficient zeros are added to the shorter code to make this sum possible. If C_i is a (n_i, M_i, d_i) code, the result is a $(n_1+max(n_1, n_2), M_1 * M_2, min(2*d_1, d_2))$ code.

If both C_1 and C_2 are linear codes, the result is also a linear code. If one of them is non-linear, the UUV sum is non-linear too. In general, a UUV sum code is not cyclic.

The function DirectSumCode returns another sum of codes (see 66.104).

```
gap> C1 := EvenWeightSubcode(WholeSpaceCode(4, GF(2)));
a cyclic [4,3,2]1 even weight subcode
gap> C2 := RepetitionCode(4, GF(2));
a cyclic [4,1,4]2 repetition code over GF(2)
gap> R := UUVCode(C1, C2);
a linear [8,4,4]2 U|U+V construction code
gap> R = ReedMullerCode(1,3);
true
```

66.106 DirectProductCode

DirectProductCode(C_1, C_2)

DirectProductCode returns the direct product of codes C_1 and C_2. Both must be linear codes. Suppose C_i has generator matrix G_i. The direct product of C_1 and C_2 then has the Kronecker product of G_1 and G_2 as the generator matrix (see KroneckerProduct).

If C_i is a $[n_i, k_i, d_i]$ code, the direct product then is a $[n_1 * n_2, k_1 * k_2, d_1 * d_2]$ code.

```
gap> L1 := LexiCode(10, 4, GF(2));
a linear [10,5,4]2..4 lexicode over GF(2)
gap> L2 := LexiCode(8, 3, GF(2));
a linear [8,4,3]2..3 lexicode over GF(2)
gap> D := DirectProductCode(L1, L2);
a linear [80,20,12]20..45 direct product code
```

66.107 IntersectionCode

IntersectionCode(C_1, C_2)

IntersectionCode returns the intersection of codes C_1 and C_2. This code consists of all codewords that are both in C_1 and C_2. If both codes are linear, the result is also linear. If both are cyclic, the result is also cyclic.

```
gap> C := CyclicCodes(7, GF(2));
[ a cyclic [7,7,1]0 enumerated code over GF(2),
  a cyclic [7,6,1..2]1 enumerated code over GF(2),
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,0,7]7 enumerated code over GF(2),
```

```
  a cyclic [7,3,1..4]2..3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2),
  a cyclic [7,1,7]3 enumerated code over GF(2),
  a cyclic [7,4,1..3]1 enumerated code over GF(2) ]
gap> IntersectionCode(C[6], C[8]) = C[7];
true
```

66.108 UnionCode

`UnionCode(` C_1, C_2 `)`

`UnionCode` returns the union of codes C_1 and C_2. This code consists of the union of all codewords of C_1 and C_2 and all linear combinations. Therefore this function works only for linear codes. The function `AddedElementsCode` can be used for non-linear codes, or if the resulting code should not include linear combinations. See 66.94. If both arguments are cyclic, the result is also cyclic.

```
gap> G := GeneratorMatCode([[1,0,1],[0,1,1]]*Z(2)^0, GF(2));
a linear [3,2,1..2]1 code defined by generator matrix over GF(2)
gap> H := GeneratorMatCode([[1,1,1]]*Z(2)^0, GF(2));
a linear [3,1,3]1 code defined by generator matrix over GF(2)
gap> U := UnionCode(G, H);
a linear [3,3,1]0 union code
gap> c := Codeword("010");; c in G;
false
gap> c in H;
false
gap> c in U;
true
```

66.109 Code Records

Like other GAP3 structures, codes are represented by records that contain important information about them. Creating such a code record is done by the code generating functions described in 66.49, 66.58 and 66.72. It is possible to create one by hand, though this is not recommended.

Once a code record is created you may add record components to it but it is not advisable to alter information already present, because that may make the code record inconsistent.

Code records must always contain the components `isCode`, `isDomain`, `operations` and one of the identification components `elements`, `generatorMat`, `checkMat`, `generatorPol`, `checkPol`. The contents of all components of a code C are described below.

The following two components are the so-called **category components** used to identify the category this domain belongs to.

`isDomain`
: is always `true` as a code is a domain.

`isCode`
: is always `true` as a code is a code is a code...

The following components determine a code domain. These are the so-called **identification components**.

`elements`
: a list of elements of the code of type codeword. The field must be present for non-linear codes.

`generatorMat` and `checkMat`
: a matrix of full rank over a finite field. Neither can exist for non-linear codes. Either one or both must be present for linear codes.

`generatorPol` and `checkPol`
: a polynomial with coefficients in a finite field. Neither can exist for non-cyclic codes. Either one or both must be present for cyclic codes.

The following components contain **basic information** about the code.

`name`
: contains a short description of the code. See 3.14 and 30.1.

`history`
: is a list of strings, containing the history of the code. The current name of the code is excluded in the list, so that if the minimum distance is calculated, it can be included in the history. Each time the code is altered by a manipulating function, one or more strings are added to this list. See 66.47.

`baseField`
: the finite field of the codewords of C. See 6.2.

`wordLength`
: is an integer specifying the word length of each codeword of C. See 66.29.

`size`
: is an integer specifying the size of C, being the number of codewords that C has. See 4.10.

The following components contain **knowledge** about the code C.

`dimension`
: is an integer specifying the dimension of C. The dimension is equal to the number of rows of the generator matrix. The field is invalid for unrestricted codes. See 9.8.

`redundancy`
: is an integer specifying the redundancy of C. The redundancy is equal to the number of rows of the parity check matrix. The field is invalid for unrestricted codes. See 66.30.

`lowerBoundMinimumDistance` and `upperBoundMinimumDistance`
: contains a lower and upper bound on the minimum distance of the code. The exact minimum distance is known if the two values are equal. See 66.31.

`upperBoundOptimalMinimumDistance`
: contains an upper bound for the minimum distance of an optimal code with the same parameters.

`minimumWeightOfGenerators`
: contains the minimum weight of the words in the generator matrix (if the code is

linear) or in the generator polynomial (if the code is cyclic). The field is invalid for unrestricted codes.

`designedDistance`
: is an integer specifying the designed distance of a BCH code. See 66.78.

`weightDistribution`
: is a list of integers containing the weight distribution of C. See 66.32.

`innerDistribution`
: is a list of integers containing the inner distribution of C. This component may only be present if C is an unrestricted code. See 66.33.

`outerDistribution`
: is a matrix containing the outer distribution, in which the first element of each row is an element of type codeword, and the second a list of integers. See 66.34.

`syndromeTable`
: is a matrix containing the syndrome table, in which the first element of each row consists of two elements of type codeword. This component is invalid for unrestricted codes. See 66.45.

`boundsCoveringRadius`
: is a list of integers specifying possible values for the covering radius. See 66.143.

`codeNorm`
: is an integer specifying the norm of C. See 66.170.

The following components are `true` if the code C has the property, `false` if not, and are not present if it is unknown whether the code has the property or not.

`isLinearCode`
: is `true` if the code is linear. See 66.17.

`isCyclicCode`
: is `true` if the code is cyclic. See 66.18.

`isPerfectCode`
: is `true` if C is a perfect code. See 66.36.

`isSelfDualCode`
: is `true` if C is equal to its dual code. See 66.38.

`isNormalCode`
: is `true` if C is a normal code. See 66.173.

`isSelfComplementaryCode`
: is `true` if C is a self complementary code. See 66.179.

`isAffineCode`
: is `true` if C is an affine code. See 66.180.

`isAlmostAffineCode`
: is `true` if C is an almost affine code. See 66.181.

The component `specialDecoder` contains a function that implements a for C specific algorithm for decoding. See 66.43.

The component `operations` contains the **operations record** (see `Domain Records` and `Dispatchers`).

66.110 Bounds on codes

This section describes the functions that calculate estimates for upper bounds on the size and minimum distance of codes. Several algorithms are known to compute a largest number of words a code can have with given length and minimum distance. It is important however to understand that in some cases the true upper bound is unknown. A code which has a size equal to the calculated upper bound may not have been found. However, codes that have a larger size do not exist.

A second way to obtain bounds is a table. In GUAVA, an extensive table is implemented for linear codes over GF(2), GF(3) and GF(4). It contains bounds on the minimum distance for given word length and dimension. For binary codes, it contains entries for word length less than or equal to 257. For codes over $GF(3)$ and $GF(4)$, it contains entries for word length less than or equal to 130.

The next sections describe functions that compute specific upper bounds on the code size (see 66.111, 66.112, 66.113, 66.114, 66.115 and 66.116).

The next section describes a function that computes GUAVA's best upper bound on the code size (see 66.117).

The next sections describe two function that compute a lower and upper bound on the minimum distance of a code (see 66.118 and 66.119).

The last section describes a function that returns a lower and upper bound on the minimum distance with given parameters and a description how the bounds were obtained (see 66.120).

66.111 UpperBoundSingleton

`UpperBoundSingleton(` n`,` d`,` q `)`

`UpperBoundSingleton` returns the Singleton bound for a code of length n, minimum distance d over a field of size q. This bound is based on the shortening of codes. By shortening an (n,M,d) code $d-1$ times, an $(n-d+1,M,1)$ code results, with $M \leq q^{n-d+1}$ (see 66.95). Thus

$$M \leq q^{n-d+1}$$

Codes that meet this bound are called **maximum distance separable** (see 66.37).

```
gap> UpperBoundSingleton(4, 3, 5);
25
gap> C := ReedSolomonCode(4,3);; Size(C);
25
gap> IsMDSCode(C);
true
```

66.112 UpperBoundHamming

`UpperBoundHamming(` n`,` d`,` q `)`

The Hamming bound (also known as **sphere packing bound**) returns an upper bound on the size of a code of length n, minimum distance d, over a field of size q. The Hamming bound is obtained by dividing the contents of the entire space $GF(q)^n$ by the contents of a ball with radius $\lfloor (d-1)/2 \rfloor$. As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)}$$

where M is the maxmimum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e (see 66.135). This bound is useful for small values of d. Codes for which equality holds are called **perfect** (see 66.36).

```
gap> UpperBoundHamming( 15, 3, 2 );
2048
gap> C := HammingCode( 4, GF(2) );
a linear [15,11,3]1 Hamming (4,2) code over GF(2)
gap> Size( C );
2048
```

66.113 UpperBoundJohnson

`UpperBoundJohnson(` n, d `)`

The Johnson bound is an improved version of the Hamming bound (see 66.112). In addition to the Hamming bound, it takes into account the elements of the space outside the balls of radius e around the elements of the code. The Johnson bound only works for binary codes.

```
gap> UpperBoundJohnson( 13, 5 );
77
gap> UpperBoundHamming( 13, 5, 2);
89    # in this case the Johnson bound is better
```

66.114 UpperBoundPlotkin

`UpperBoundPlotkin(` n, d, q `)`

The function `UpperBoundPlotkin` calculates the sum of the distances of all ordered pairs of different codewords. It is based on the fact that the minimum distance is at most equal to the average distance. It is a good bound if the weights of the codewords do not differ much. It results in:

$$M \leq \frac{d}{d-(1-1/q)n}$$

M is the maximum number of codewords. In this case, d must be larger than $(1-1/q)n$, but by shortening the code, the case $d < (1-1/q)n$ is covered.

```
gap> UpperBoundPlotkin( 15, 7, 2 );
32
gap> C := BCHCode( 15, 7, GF(2) );
a cyclic [15,5,7]5 BCH code, delta=7, b=1 over GF(2)
gap> Size(C);
32
gap> WeightDistribution(C);
[ 1, 0, 0, 0, 0, 0, 0, 15, 15, 0, 0, 0, 0, 0, 0, 1 ]
```

66.115 UpperBoundElias

`UpperBoundElias(` n, d, q `)`

The Elias bound is an improvement of the Plotkin bound (see 66.114) for large codes. Subcodes are used to decrease the size of the code, in this case the subcode of all codewords within a certain ball. This bound is useful for large codes with relatively small minimum distances.

```
gap> UpperBoundPlotkin( 16, 3, 2 );
12288
gap> UpperBoundElias( 16, 3, 2 );
10280
```

66.116 UpperBoundGriesmer

`UpperBoundGriesmer(` n, d, q `)`

The Griesmer bound is valid only for linear codes. It is obtained by counting the number of equal symbols in each row of the generator matrix of the code. By omitting the coordinates in which all rows have a zero, a smaller code results. The Griesmer bound is obtained by repeating this proces until a trivial code is left in the end.

```
gap> UpperBoundGriesmer( 13, 5, 2 );
64
gap> UpperBoundGriesmer( 18, 9, 2 );
8          # the maximum number of words for a linear code is 8
gap> Size( PuncturedCode( HadamardCode( 20, 1 ) ) );
20         # this non-linear code has 20 elements
```

66.117 UpperBound

`UpperBound(` n, d, q `)`

`UpperBound` returns the best known upper bound $A(n,d)$ for the size of a code of length n, minimum distance d over a field of size q. The function `UpperBound` first checks for trivial cases (like $d = 1$ or $n = d$) and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the methods of Singleton (see 66.111), Hamming (see 66.112), Johnson (see 66.113), Plotkin (see 66.114) and Elias (see 66.115). If the code is binary, $A(n, 2*l-1) = A(n+1, 2*l)$, so the `UpperBound` takes the minimum of the values obtained from all methods for the parameters $(n, 2*l-1)$ and $(n+1, 2*l)$.

```
gap> UpperBound( 10, 3, 2 );
85
gap> UpperBound( 25, 9, 8 );
1211778792827540
```

66.118 LowerBoundMinimumDistance

`LowerBoundMinimumDistance(` C `)`

In this form, `LowerBoundMinimumDistance` returns a lower bound for the minimum distance of code C.

```
gap> C := BCHCode( 45, 7 );
a cyclic [45,23,7..9]6..16 BCH code, delta=7, b=1 over GF(2)
gap> LowerBoundMinimumDistance( C );
7       # designed distance is lower bound for minimum distance
```

`LowerBoundMinimumDistance( n, k, F )`

In this form, `LowerBoundMinimumDistance` returns a lower bound for the minimum distance of the best known linear code of length n, dimension k over field F. It uses the mechanism explained in section 66.120.

```
gap> LowerBoundMinimumDistance( 45, 23, GF(2) );
10
```

66.119 UpperBoundMinimumDistance

`UpperBoundMinimumDistance( C )`

In this form, `UpperBoundMinimumDistance` returns an upper bound for the minimum distance of code C. For unrestricted codes, it just returns the word length. For linear codes, it takes the minimum of the possibly known value from the method of construction, the weight of the generators, and the value from the table (see 66.120).

```
gap> C := BCHCode( 45, 7 );;
gap> UpperBoundMinimumDistance( C );
9
```

`UpperBoundMinimumDistance( n, k, F )`

In this form, `UpperBoundMinimumDistance` returns an upper bound for the minimum distance of the best known linear code of length n, dimension k over field F. It uses the mechanism explained in section 66.120.

```
gap> UpperBoundMinimumDistance( 45, 23, GF(2) );
11
```

66.120 BoundsMinimumDistance

`BoundsMinimumDistance( n, k, F )`

The function `BoundsMinimumDistance` calculates a lower and upper bound for the minimum distance of an optimal linear code with word length n, dimension k over field F. The function returns a record with the two bounds and an explenation for each bound. The function `Display` can be used to show the explanations.

The values for the lower and upper bound are obtained from a table. GUAVA has tables containing lower and upper bounds for $q = 2$ ($n \leq 257$), 3 and 4 ($n \leq 130$). These tables were derived from the table of Brouwer & Verhoeff. For codes over other fields and for larger word lengths, trivial bounds are used.

The resulting record can be used in the function `BestKnownLinearCode` (see 66.71) to construct a code with minimum distance equal to the lower bound.

```
gap> bounds := BoundsMinimumDistance( 7, 3 );; Display( bounds );
an optimal linear [7,3,d] code over GF(2) has d=4
----------------------------------------------------------------------
```

```
Lb(7,3)=4, by shortening of:
Lb(8,4)=4, u|u+v construction of C1 and C2:
C1: Lb(4,3)=2, dual of the repetition code
C2: Lb(4,1)=4, repetition code
----------------------------------------------------------------------
Ub(7,3)=4, Griesmer bound
# The lower bound is equal to the upper bound, so a code with
# these parameters is optimal.
gap> C := BestKnownLinearCode( bounds );; Display( C );
a linear [7,3,4]2..3 shortened code of
a linear [8,4,4]2 U|U+V construction code of
U: a cyclic [4,3,2]1 dual code of
   a cyclic [4,1,4]2 repetition code over GF(2)
V: a cyclic [4,1,4]2 repetition code over GF(2)
```

66.121 Special matrices in GUAVA

This section explains functions that work with special matrices GUAVA needs for several codes.

The next sections describe some matrix generating functions (see 66.122, 66.123, 66.124, 66.125 and 66.126).

The next sections describe two functions about a standard form of matrices (see 66.127 and 66.128).

The next sections describe functions that return a matrix after a manipulation (see 66.129, 66.130 and 66.131).

The last sections describe functions that do some tests on matrices (see 66.132 and 66.133).

66.122 KrawtchoukMat

`KrawtchoukMat(` n `,` q `)`

`KrawtchoukMat` returns the $n+1$ by $n+1$ matrix $K = (k_{ij})$ defined by $k_{ij} = K_i(j)$ for $i, j = 0, ..., n$. $K_i(j)$ is the Krawtchouk number (see 66.136). n must be a positive integer and q a prime power. The Krawtchouk matrix is used in the **MacWilliams identities**, defining the relation between the weight distribution of a code of length n over a field of size q, and its dual code. Each call to `KrawtchoukMat` returns a new matrix, so it is safe to modify the result.

```
gap> PrintArray( KrawtchoukMat( 3, 2 ) );
[ [   1,   1,   1,   1 ],
  [   3,   1,  -1,  -3 ],
  [   3,  -1,  -1,   3 ],
  [   1,  -1,   1,  -1 ] ]
gap> C := HammingCode( 3 );; a := WeightDistribution( C );
[ 1, 0, 0, 7, 7, 0, 0, 1 ]
gap> n := WordLength( C );; q := Size( Field( C ) );;
gap> k := Dimension( C );;
gap> q^( -k ) * KrawtchoukMat( n, q ) * a;
```

```
[ 1, 0, 0, 0, 7, 0, 0, 0 ]
gap> WeightDistribution( DualCode( C ) );
[ 1, 0, 0, 0, 7, 0, 0, 0 ]
```

66.123 GrayMat

`GrayMat(` n`,` F `)`

`GrayMat` returns a list of all different vectors (see `Vectors`) of length n over the field F, using Gray ordening. n must be a positive integer. This order has the property that subsequent vectors differ in exactly one coordinate. The first vector is always the null vector. Each call to `GrayMat` returns a new matrix, so it is safe to modify the result.

```
gap> GrayMat(3);
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> G := GrayMat( 4, GF(4) );; Length(G);
256          # the length of a GrayMat is always q^n
gap> G[101] - G[100];
[ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]
```

66.124 SylvesterMat

`SylvesterMat(` n `)`

`SylvesterMat` returns the n by n Sylvester matrix of order n. This is a special case of the Hadamard matrices (see 66.125). For this construction, n must be a power of 2. Each call to `SylvesterMat` returns a new matrix, so it is safe to modify the result.

```
gap> PrintArray(SylvesterMat(2));
[ [   1,   1 ],
  [   1,  -1 ] ]
gap> PrintArray( SylvesterMat(4) );
[ [   1,   1,   1,   1 ],
  [   1,  -1,   1,  -1 ],
  [   1,   1,  -1,  -1 ],
  [   1,  -1,  -1,   1 ] ]
```

66.125 HadamardMat

`HadamardMat(` n `)`

`HadamardMat` returns a Hadamard matrix of order n. This is an n by n matrix with the property that the matrix multiplied by its transpose returns n times the identity matrix. This is only possible for $n = 1, n = 2$ or in cases where n is a multiple of 4. If the matrix does not exist or is not known, `HadamardMat` returns an error. A large number of construction methods is known to create these matrices for different orders. `HadamardMat` makes use of two construction methods (among which the Sylvester construction (see 66.124)).

These methods cover most of the possible Hadamard matrices, although some special algorithms have not been implemented yet. The following orders less than 100 do not have an implementation for a Hadamard matrix in GUAVA: $28, 36, 52, 76, 92$.

```
gap> C := HadamardMat(8);; PrintArray(C);
[ [   1,   1,   1,   1,   1,   1,   1,   1 ],
  [   1,  -1,   1,  -1,   1,  -1,   1,  -1 ],
  [   1,   1,  -1,  -1,   1,   1,  -1,  -1 ],
  [   1,  -1,  -1,   1,   1,  -1,  -1,   1 ],
  [   1,   1,   1,   1,  -1,  -1,  -1,  -1 ],
  [   1,  -1,   1,  -1,  -1,   1,  -1,   1 ],
  [   1,   1,  -1,  -1,  -1,  -1,   1,   1 ],
  [   1,  -1,  -1,   1,  -1,   1,   1,  -1 ] ]
gap> C * TransposedMat(C) = 8 * IdentityMat( 8, 8 );
true
```

66.126 MOLS

`MOLS(` q `)`
`MOLS(` q`,` n `)`

`MOLS` returns a list of n **Mutually Orthogonal Latin Squares** (**MOLS**). A **Latin square** of order q is a q by q matrix whose entries are from a set F_q of q distinct symbols (GUAVA uses the integers from 0 to q) such that each row and each column of the matrix contains each symbol exactly once.

A set of Latin squares is a set of MOLS if and only if for each pair of Latin squares in this set, every ordered pair of elements that are in the same position in these matrices occurs exactly once.

n must be less than q. If n is omitted, two MOLS are returned. If q is not a prime power, at most 2 MOLS can be created. For all values of q with $q > 2$ and $q \neq 6$, a list of MOLS can be constructed. GUAVA however does not yet construct MOLS for $q \bmod 4 = 2$. If it is not possible to construct n MOLS, the function returns `false`.

MOLS are used to create q-ary codes (see 66.53).

```
gap> M := MOLS( 4, 3 );;PrintArray( M[1] );
[ [  0,  1,  2,  3 ],
  [  1,  0,  3,  2 ],
  [  2,  3,  0,  1 ],
  [  3,  2,  1,  0 ] ]
gap> PrintArray( M[2] );
[ [  0,  2,  3,  1 ],
  [  1,  3,  2,  0 ],
  [  2,  0,  1,  3 ],
  [  3,  1,  0,  2 ] ]
gap> PrintArray( M[3] );
[ [  0,  3,  1,  2 ],
  [  1,  2,  0,  3 ],
  [  2,  1,  3,  0 ],
  [  3,  0,  2,  1 ] ]
```

```
gap> MOLS( 12, 3 );
false
```

66.127 PutStandardForm

`PutStandardForm(` *M* `)`
`PutStandardForm(` *M* `,` *idleft* `)`

`PutStandardForm` puts a matrix M in standard form, and returns the permutation needed to do so. *idleft* is a boolean that sets the position of the identity matrix in M. If *idleft* is set to `true`, the identity matrix is put in the left side of M. Otherwise, it is put at the right side. The default for *idleft* is `true`.

The function `BaseMat` also returns a similar standard form, but does not apply column permutations. The rows of the matrix still span the same vector space after `BaseMat`, but after calling `PutStandardForm`, this is not necessarily true.

```
gap> M := Z(2)*[[1,0,0,1],[0,0,1,1]];; PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  0*Z(2),  Z(2)^0,  Z(2)^0 ] ]
gap> PutStandardForm(M);                    # identity at the left side
(2,3)
gap> PrintArray(M);
[ [  Z(2)^0,  0*Z(2),  0*Z(2),  Z(2)^0 ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
gap> PutStandardForm(M, false);             # identity at the right side
(1,4,3)
gap> PrintArray(M);
[ [  0*Z(2),  Z(2)^0,  Z(2)^0,  0*Z(2) ],
  [  0*Z(2),  Z(2)^0,  0*Z(2),  Z(2)^0 ] ]
```

66.128 IsInStandardForm

`IsInStandardForm(` *M* `)`
`IsInStandardForm(` *M* `,` *idleft* `)`

`IsInStandardForm` determines if M is in standard form. *idleft* is a boolean that indicates the position of the identity matrix in M. If *idleft* is `true`, `IsInStandardForm` checks if the identity matrix is at the left side of M, otherwise if it is at the right side. The default for *idleft* is `true`. The elements of M may be elements of any field. To put a matrix in standard form, use `PutStandardForm` (see 66.127).

```
gap> IsInStandardForm(IdentityMat(7, GF(2)));
true
gap> IsInStandardForm([[1, 1, 0], [1, 0, 1]], false);
true
gap> IsInStandardForm([[1, 3, 2, 7]]);
true
gap> IsInStandardForm(HadamardMat(4));
false
```

66.129 PermutedCols

`PermutedCols(` M `,` P `)`

`PermutedCols` returns a matrix M with a permutation P applied to its columns.

```
gap> M := [[1,2,3,4],[1,2,3,4]];; PrintArray(M);
[ [  1,  2,  3,  4 ],
  [  1,  2,  3,  4 ] ]
gap> PrintArray(PermutedCols(M, (1,2,3)));
[ [  3,  1,  2,  4 ],
  [  3,  1,  2,  4 ] ]
```

66.130 VerticalConversionFieldMat

`VerticalConversionFieldMat(` M `,` F `)`

`VerticalConversionFieldMat` returns the matrix M with its elements converted from a field $F = GF(q^m)$, q prime, to a field $GF(q)$. Each element is replaced by its representation over the latter field, placed vertically in the matrix.

If M is a k by n matrix, the result is a $k * m$ by n matrix, since each element of $GF(q^m)$ can be represented in $GF(q)$ using m elements.

```
gap> M := Z(9)*[[1,2],[2,1]];; PrintArray(M);
[ [    Z(3^2),  Z(3^2)^5 ],
  [  Z(3^2)^5,    Z(3^2) ] ]
gap> DefaultField( Flat(M) );
GF(3^2)
gap> VCFM := VerticalConversionFieldMat( M, GF(9) );; PrintArray(VCFM);
[ [  0*Z(3),  0*Z(3) ],
  [  Z(3)^0,    Z(3) ],
  [  0*Z(3),  0*Z(3) ],
  [    Z(3),  Z(3)^0 ] ]
gap> DefaultField( Flat(VCFM) );
GF(3)
```

A similar function is `HorizontalConversionFieldMat` (see 66.131).

66.131 HorizontalConversionFieldMat

`HorizontalConversionFieldMat(` M `,` F `)`

`HorizontalConversionFieldMat` returns the matrix M with its elements converted from a field $F = GF(q^m)$, q prime, to a field $GF(q)$. Each element is replaced by its representation over the latter field, placed horizontally in the matrix.

If M is a k by n matrix, the result is a $k * m$ by $n * m$ matrix. The new word length of the resulting code is equal to $n * m$, because each element of $GF(q^m)$ can be represented in $GF(q)$ using m elements. The new dimension is equal to $k * m$ because the new matrix should be a basis for the same number of vectors as the old one.

`ConversionFieldCode` uses horizontal conversion to convert a code (see 66.100).

```
gap> M := Z(9)*[[1,2],[2,1]];; PrintArray(M);
[ [    Z(3^2),  Z(3^2)^5 ],
  [  Z(3^2)^5,    Z(3^2) ] ]
gap> DefaultField( Flat(M) );
GF(3^2)
gap> HCFM := HorizontalConversionFieldMat(M, GF(9));; PrintArray(HCFM);
[ [  0*Z(3),  Z(3)^0,  0*Z(3),    Z(3) ],
  [  Z(3)^0,  Z(3)^0,    Z(3),    Z(3) ],
  [  0*Z(3),    Z(3),  0*Z(3),  Z(3)^0 ],
  [    Z(3),    Z(3),  Z(3)^0,  Z(3)^0 ] ]
gap> DefaultField( Flat(HCFM) );
GF(3)
```

A similar function is `VerticalConversionFieldMat` (see 66.130).

66.132 IsLatinSquare

`IsLatinSquare(` M `)`

`IsLatinSquare` determines if a matrix M is a latin square. For a latin square of size n by n, each row and each column contains all the integers $1..n$ exactly once.

```
gap> IsLatinSquare([[1,2],[2,1]]);
true
gap> IsLatinSquare([[1,2,3],[2,3,1],[1,3,2]]);
false
```

66.133 AreMOLS

`AreMOLS(` L `)`

`AreMOLS` determines if L is a list of mutually orthogonal latin squares (MOLS). For each pair of latin squares in this list, the function checks if each ordered pair of elements that are in the same position in these matrices occurs exactly once. The function `MOLS` creates MOLS (see 66.126).

```
gap> M := MOLS(4,2);
[ [ [ 0, 1, 2, 3 ], [ 1, 0, 3, 2 ], [ 2, 3, 0, 1 ], [ 3, 2, 1, 0 ] ],
  [ [ 0, 2, 3, 1 ], [ 1, 3, 2, 0 ], [ 2, 0, 1, 3 ], [ 3, 1, 0, 2 ] ] ]
gap> AreMOLS(M);
true
```

66.134 Miscellaneous functions

The following sections describe several functions GUAVA uses for constructing codes or performing calculations with codes.

66.135 SphereContent

`SphereContent(` n, t, F `)`

`SphereContent` returns the content of a ball of radius t around an arbitrary element of the vectorspace F^n. This is the cardinality of the set of all elements of F^n that are at distance (see 66.12) less than or equal to t from an element of F^n.

In the context of codes, the function is used to determine if a code is perfect. A code is perfect if spheres of radius t around all codewords contain exactly the whole vectorspace, where t is the number of errors the code can correct.

```
gap> SphereContent( 15, 0, GF(2) );
1     # Only one word with distance 0, which is the word itself
gap> SphereContent( 11, 3, GF(4) );
4984
gap> C := HammingCode(5);
a linear [31,26,3]1 Hamming (5,2) code over GF(2)
# the minimum distance is 3, so the code can correct one error
gap> ( SphereContent( 31, 1, GF(2) ) * Size(C) ) = 2 ^ 31;
true
```

66.136 Krawtchouk

`Krawtchouk(` k `,` i `,` n `,` q `)`

`Krawtchouk` returns the Krawtchouk number $K_k(i)$. q must be a primepower, n must be a positive integer, k must be a non-negative integer less then or equal to n and i can be any integer. (See 66.122).

```
gap> Krawtchouk( 2, 0, 3, 2);
3
```

66.137 PrimitiveUnityRoot

`PrimitiveUnityRoot(` F `,` n `)`

`PrimitiveUnityRoot` returns a **primitive n'th root of unity** in an extension field of F. This is a finite field element a with the property $a^n = 1 \bmod n$, and n is the smallest integer such that this equality holds.

```
gap> PrimitiveUnityRoot( GF(2), 15 );
Z(2^4)
gap> last^15;
Z(2)^0
gap> PrimitiveUnityRoot( GF(8), 21 );
Z(2^6)^3
```

66.138 ReciprocalPolynomial

`ReciprocalPolynomial(` P `)`

`ReciprocalPolynomial` returns the **reciprocal** of polynomial P. This is a polynomial with coefficients of P in the reverse order. So if $P = a_0 + a_1X + ... + a_nX^n$, the reciprocal polynomial is $P' = a_n + a_{n-1}X + ... + a_0X^n$.

```
gap> P := Polynomial( GF(3), Z(3)^0 * [1,0,1,2] );
```

```
Z(3)^0*(2*X(GF(3))^3 + X(GF(3))^2 + 1)
gap> RecP := ReciprocalPolynomial( P );
Z(3)^0*(X(GF(3))^3 + X(GF(3)) + 2)
gap> ReciprocalPolynomial( RecP ) = P;
true
```

`ReciprocalPolynomial(` P `,` n `)`

In this form, the number of coefficients of P is considered to be at least n (possibly with zero coefficients at the highest degrees). Therefore, the reciprocal polynomial P' also has degree at least n.

```
gap> P := Polynomial( GF(3), Z(3)^0 * [1,0,1,2] );
Z(3)^0*(2*X(GF(3))^3 + X(GF(3))^2 + 1)
gap> ReciprocalPolynomial( P, 6 );
Z(3)^0*(X(GF(3))^6 + X(GF(3))^4 + 2*X(GF(3))^3)
```

In this form, the degree of P is considered to be at least n (if not, zero coefficients are added). Therefore, the reciprocal polynomial P' also has degree at least n.

66.139 CyclotomicCosets

`CyclotomicCosets(` q, n `)`

`CyclotomicCosets` returns the cyclotomic cosets of q modulo n. q and n must be relatively prime. Each of the elements of the returned list is a list of integers that belong to one cyclotomic coset. Each coset contains all multiplications of the **coset representative** by q, modulo n. The coset representative is the smallest integer that isn't in the previous cosets.

```
gap> CyclotomicCosets( 2, 15 );
[ [ 0 ], [ 1, 2, 4, 8 ], [ 3, 6, 12, 9 ], [ 5, 10 ],
  [ 7, 14, 13, 11 ] ]
gap> CyclotomicCosets( 7, 6 );
[ [ 0 ], [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
```

66.140 WeightHistogram

`WeightHistogram(` C `)`
`WeightHistogram(` C, h `)`

The function `WeightHistogram` plots a histogram of weights in code C. The maximum length of a column is h. Default value for h is 1/3 of the size of the screen. The number that appears at the top of the histogram is the maximum value of the list of weights.

```
gap> H := HammingCode(2, GF(5));
a linear [6,4,3]1 Hamming (2,5) code over GF(5)
gap> WeightDistribution(H);
[ 1, 0, 0, 80, 120, 264, 160 ]
gap> WeightHistogram(H);
264----------------
               *
               *
```

```
                 *
                 *
                 *  *
              *  *  *
           *  *  *  *
           *  *  *  *
+--------+--+--+--+--
0  1  2  3  4  5  6
```

66.141 Extensions to GUAVA

In this section and the following sections some extensions to GUAVA will be discussed. The most important extensions are new code constructions and new algorithms and bounds for the covering radius. Another important function is the implementation of the algorithm of Leon for finding the minimum distance.

66.142 Some functions for the covering radius

Together with the new code constructions, the new functions for computing (the bounds of) the covering radius are the most important additions to GUAVA.

These additions required a change in the fields of a code record. In previous versions of GUAVA, the covering radius field was an integer field, called `coveringRadius`. To allow the code-record to contain more information about the covering radius, this field has been replaced by a field called `boundsCoveringRadius`. This field contains a vector of possible values of the covering radius of the code. If the value of the covering radius is known, then the length of this vector is one.

This means that every instance of `coveringRadius` in the previous version had to be changed to `boundsCoveringRadius`. There is also an advantage to this: if bounds for a specific type of code are known, they can be implemented (and they have been). This has been especially useful for the Reed-Muller codes.

Of course, the main covering radius function dispatcher, `CoveringRadius`, had to be changed to incorporate these changes. Previously, this dispatcher called `code.operations.CoveringRadius`. Problem with these functions is that they only work if the redundancy is not too large. Another problem is that the algorithm for linear and cyclic codes is written in `C` (in the kernel of GAP3). This does not allow the user to interrupt the function, except by pressing `ctrl-C` twice, which exits GAP3 altogether. For more information, check the section on the (new) `CoveringRadius` (66.143) function.

Perhaps the most interesting new covering radius function is `IncreaseCoveringRadiusLowerBound` (66.146). It uses a probabilistic algorithm that tries to find better lower bounds of the covering radius of a code. It works best when the dimension is low, thereby giving a sort of complement function to `CoveringRadius`. When the dimension is about half the length of a code, neither algorithm will work, although `IncreaseCoveringRadiusLowerBound` is specifically designed to avoid memory problems, unlike `CoveringRadius`.

The function `ExhaustiveSearchCoveringRadius` (66.147) tries to find the covering radius of a code by exhaustively searching the space in which the code lies for coset leaders.

A number of bounds for the covering radius in general have been implemented, including some well known bounds like the sphere-covering bound, the redundancy bound and the Delsarte bound. These function all start with `LowerBoundCoveringRadius` (sections 66.150, 66.151, 66.152, 66.153, 66.154, 66.155, 66.156, 66.150) or `UpperBoundCoveringRadius` (sections 66.157, 66.158, 66.159, 66.160, 66.161).

The functions `GeneralLowerBoundCoveringRadius` (66.148) and `GeneralUpperBoundCoveringRadius` (66.149) try to find the best known bounds for a given code. `BoundsCoveringRadius` (66.144) uses these functions to return a vector of possible values for the covering radius.

To allow the user to enter values in the `.boundsCoveringRadius` record herself, the function `SetCoveringRadius` is provided.

66.143 CoveringRadius

`CoveringRadius(` *code* `)`

`CoveringRadius` is a function that already appeared in earlier versions of GUAVA, but it is changed to reflect the implementation of new functions for the covering radius.

If there exists a function called `SpecialCoveringRadius` in the `operations` field of the code, then this function will be called to compute the covering radius of the code. At the moment, no code-specific functions are implemented.

If the length of `BoundsCoveringRadius` (see 66.144), is 1, then the value in `code.boundsCoveringRadius` is returned. Otherwise, the function `code.operations.CoveringRadius` is executed, unless the redundancy of *code* is too large. In the last case, a warning is issued.

If you want to overrule this restriction, you might want to execute `code.operations.CoveringRadius` yourself. However, this algorithm might also issue a warning that it cannot be executed, but this warning is sometimes issued too late, resulting in GAP3 exiting with an memory error. If it does run, it can only be stopped by pressing `ctrl-C` twice, thereby quitting GAP3. It will not enter the usual break-loop. Therefore it is recommendable to save your work before trying `code.operations.CoveringRadius`.

```
gap> CoveringRadius( BCHCode( 17, 3, GF(2) ) );
3
gap> CoveringRadius( HammingCode( 5, GF(2) ) );
1
gap> code := ReedMullerCode( 1, 9 );;
gap> CoveringRadius( code );
CoveringRadius: warning, the covering radius of
this code cannot be computed straightforward.
Try to use IncreaseCoveringRadiusLowerBound( <code> ).
(see the manual for more details).
The covering radius of <code> lies in the interval:
[ 240 .. 248 ]
gap> code.operations.CoveringRadius( code );
Error, CosetLeaderMatFFE: sorry, no hope to finish
```

66.144 BoundsCoveringRadius

`BoundsCoveringRadius(` *code* `)`

`BoundsCoveringRadius` returns a list of integers. The first entry of this list is the maximum of some lower bounds for the covering radius of *code*, the last entry the minimum of some upper bounds of *code*.

If the covering radius of *code* is known, a list of length 1 is returned.

`BoundsCoveringRadius` makes use of the functions `GeneralLowerBoundCoveringRadius` and `GeneralUpperBoundCoveringRadius`.

```
gap> BoundsCoveringRadius( BCHCode( 17, 3, GF(2) ) );
[ 3 .. 4 ]
gap> BoundsCoveringRadius( HammingCode( 5, GF(2) ) );
[ 1 ]
```

66.145 SetCoveringRadius

`SetCoveringRadius(` *code*`,` *intlist* `)`

`SetCoveringRadius` enables the user to set the covering radius herself, instead of letting GUAVA compute it. If *intlist* is an integer, GUAVA will simply put it in the `boundsCoveringRadius` field. If it is a list of integers, however, it will intersect this list with the `boundsCoveringRadius` field, thus taking the best of both lists. If this would leave an empty list, the field is set to *intlist*.

Because some other computations use the covering radius of the code, it is important that the entered value is not wrong, otherwise new results may be invalid.

```
gap> code := BCHCode( 17, 3, GF(2) );;
gap> BoundsCoveringRadius( code );
[ 3 .. 4 ]
gap> SetCoveringRadius( code, [ 2 .. 3 ] );
gap> BoundsCoveringRadius( code );
[ 3 ]
```

66.146 IncreaseCoveringRadiusLowerBound

`IncreaseCoveringRadiusLowerBound(` *code* `[,` *stopdistance* `] [,` *startword* `] )`

`IncreaseCoveringRadiusLowerBound` tries to increase the lower bound of the covering radius of *code*. It does this by means of a probabilistic algorithm. This algorithm takes a random word in $GF(q)^n$ (or *startword* if it is specified), and, by changing random coordinates, tries to get as far from *code* as possible. If changing a coordinate finds a word that has a larger distance to the code than the previous one, the change is made permanent, and the algorithm starts all over again. If changing a coordinate does not find a coset leader that is further away from the code, then the change is made permanent with a chance of 1 in 100, if it gets the word closer to the code, or with a chance of 1 in 10, if the word stays at the same distance. Otherwise, the algorithm starts again with the same word as before.

If the algorithm did not allow changes that decrease the distance to the code, it might get stuck in a sub-optimal situation (the coset leader corresponding to such a situation (i.e. no coordinate of this coset leader can be changed in such a way that we get at a larger distance from the code) is called an orphan).

If the algorithm finds a word that has distance *stopdistance* to the code, it ends and returns that word, which can be used for further investigations.

The variable `InfoCoveringRadius` can be set to `Print` to print the maximum distance reached so far every 1000 runs. The alogrithm can be interrupted with `ctrl-C`, allowing the user to look at the word that is currently being examined (called `current`), or to change the chances that the new word is made permanent (these are called `staychance` and `downchance`). If one of these variables is i, then it corresponds with a i in 100 chance.

At the moment, the algorithm is only useful for codes with small dimension, where small means that the elements of the code fit in the memory. It works with larger codes, however, but when you use it for codes with large dimension, you should be **very** patient. If running the algorithm quits GAP3 (due to memory problems), you can change the global variable `CRMemSize` to a lower value. This might cause the algorithm to run slower, but without quitting GAP3. The only way to find out the best value of `CRMemSize` is by experimenting.

66.147 ExhaustiveSearchCoveringRadius

`ExhaustiveSearchCoveringRadius(` *code* `)`

`ExhaustiveSearchCoveringRadius` does an exhaustive search to find the covering radius of *code*. Every time a coset leader of a coset with weight w is found, the function tries to find a coset leader of a coset with weight $w+1$. It does this by enumerating all words of weight $w+1$, and checking whether a word is a coset leader. The start weight is the current known lower bound on the covering radius.

66.148 GeneralLowerBoundCoveringRadius

`GeneralLowerBoundCoveringRadius(` *code* `)`

`GeneralLowerBoundCoveringRadius` returns a lower bound on the covering radius of *code*. It uses as many functions which names start with `LowerBoundCoveringRadius` as possible to find the best known lower bound (at least that GUAVA knows of) together with tables for the covering radius of binary linear codes with length not greater than 64.

66.149 GeneralUpperBoundCoveringRadius

`GeneralUpperBoundCoveringRadius(` *code* `)`

`GeneralUpperBoundCoveringRadius` returns an upper bound on the covering radius of *code*. It uses as many functions which names start with `UpperBoundCoveringRadius` as possible to find the best known upper bound (at least that GUAVA knows of).

66.150 LowerBoundCoveringRadiusSphereCovering

`LowerBoundCoveringRadiusSphereCovering(` *n*`,` *M* `[,` *F* `], false )`

`LowerBoundCoveringRadiusSphereCovering(` *n*`,` *r* `[,` *F* `] [, true ] )`

If the last argument of `LowerBoundCoveringRadiusSphereCovering` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

F is the field over which the code is defined. If F is omitted, it is assumed that the code is over `GF(2)`.

The bound is computed according to the sphere covering bound

$$MV_q(n,r) \geq q^n \tag{66.1}$$

where $V_q(n,r)$ is the size of a sphere of radius r in $\mathtt{GF}(q)^n$.

66.151 LowerBoundCoveringRadiusVanWee1

`LowerBoundCoveringRadiusVanWee1(` n`,` M `[,` F `], false )`

`LowerBoundCoveringRadiusVanWee1(` n`,` r `[,` F `] [, true ] )`

If the last argument of `LowerBoundCoveringRadiusVanWee1` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

F is the field over which the code is defined. If F is omitted, it is assumed that the code is over `GF(2)`.

The Van Wee bound is an improvement of the sphere covering bound

$$M\left\{V_q(n,r) - \frac{\binom{n}{r}}{\lceil\frac{n-r}{r+1}\rceil}\left(\left\lceil\frac{n+1}{r+1}\right\rceil - \frac{n+1}{r+1}\right)\right\} \geq q^n \tag{66.2}$$

66.152 LowerBoundCoveringRadiusVanWee2

`LowerBoundCoveringRadiusVanWee2(` n`,` M`, false )`

`LowerBoundCoveringRadiusVanWee2(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusVanWee2` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality

$$M\frac{\left(\left(V_2(n,2) - \frac{1}{2}(r+2)(r-1)\right)V_2(n,r) + \varepsilon V_2(n,r-2)\right)}{\left(V_2(n,2) - \frac{1}{2}(r+2)(r-1) + \varepsilon\right)} \geq 2^n, \tag{66.3}$$

where

$$\varepsilon = \binom{r+2}{2}\left\lceil\binom{n-r+1}{2}\Big/\binom{r+2}{2}\right\rceil - \binom{n-r+1}{2}. \tag{66.4}$$

66.153 LowerBoundCoveringRadiusCountingExcess

`LowerBoundCoveringRadiusCountingExcess(` n`,` M`, false )`

`LowerBoundCoveringRadiusCountingExcess(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusCountingExcess` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality

$$M\left(\rho V_2(n,r) + \varepsilon V_2(n,r-1)\right) \geq (\rho+\varepsilon)2^n, \tag{66.5}$$

where

$$\varepsilon = (r+1)\left\lceil \frac{n+1}{r+1} \right\rceil - (n+1) \tag{66.6}$$

and

$$\rho = \begin{cases} n-3+\frac{2}{n} & \text{if } r=2 \\ n-r-1 & \text{if } r \geq 3 \end{cases} \tag{66.7}$$

66.154 LowerBoundCoveringRadiusEmbedded1

`LowerBoundCoveringRadiusEmbedded1(` n`,` M`, false )`

`LowerBoundCoveringRadiusEmbedded1(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusEmbedded1` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality

$$M\left(V_2(n,r) - \binom{2r}{r}\right) \geq 2^n - A(n,2r+1)\binom{2r}{r}, \tag{66.8}$$

where $A(n,d)$ denotes the maximal cardinality of a (binary) code of length n and minimum distance d. The function `UpperBound` is used to compute this value.

Sometimes `LowerBoundCoveringRadiusEmbedded1` is better than `LowerBoundCoveringRadiusEmbedded2`, sometimes it is the other way around.

66.155 LowerBoundCoveringRadiusEmbedded2

`LowerBoundCoveringRadiusEmbedded2(` n`,` M`, false )`

`LowerBoundCoveringRadiusEmbedded2(` n`,` r `[, true ] )`

If the last argument of `LowerBoundCoveringRadiusEmbedded2` is `false`, then it returns a lower bound for the covering radius of a code of size M and length n. Otherwise, it returns a lower bound for the size of a code of length n and covering radius r.

This bound only works for binary codes. It is based on the following inequality

$$M\left(V_2(n,r) - \frac{3}{2}\binom{2r}{r}\right) \geq 2^n - 2A(n,2r+1)\binom{2r}{r}, \tag{66.9}$$

where $A(n,d)$ denotes the maximal cardinality of a (binary) code of length n and minimum distance d. The function `UpperBound` is used to compute this value.

Sometimes `LowerBoundCoveringRadiusEmbedded1` is better than `LowerBoundCoveringRadiusEmbedded2`, sometimes it is the other way around.

66.156 LowerBoundCoveringRadiusInduction

`LowerBoundCoveringRadiusInduction(` *n*`,` *r* `)`

`LowerBoundCoveringRadiusInduction` returns a lower bound for the size of a code with length n and covering radius r.

If $n = 2r+2$ and $r \geq 1$, the returned value is 4.
If $n = 2r+3$ and $r \geq 1$, the returned value is 7.
If $n = 2r+4$ and $r \geq 4$, the returned value is 8.
Otherwise, 0 is returned.

66.157 UpperBoundCoveringRadiusRedundancy

`UpperBoundCoveringRadiusRedundancy(` *code* `)`

`UpperBoundCoveringRadiusRedundancy` returns the redundancy of *code* as an upper bound for the covering radius of *code*. *code* must be a linear code.

66.158 UpperBoundCoveringRadiusDelsarte

`UpperBoundCoveringRadiusDelsarte(` *code* `)`

`UpperBoundCoveringRadiusDelsarte` returns an upper bound for the covering radius of *code*. This upperbound is equal to the *external distance* of *code*, this is the minimum distance of the dual code, if *code* is a linear code.

66.159 UpperBoundCoveringRadiusStrength

`UpperBoundCoveringRadiusStrength(` *code* `)`

`UpperBoundCoveringRadiusStrength` returns an upper bound for the covering radius of *code*.

First the code is punctured at the zero coordinates (i.e. the coordinates where all codewords have a zero). If the remaining code has strength 1 (i.e. each coordinate contains each element of the field an equal number of times), then it returns $\frac{q-1}{q}m + (n-m)$ (where q is the size of the field and m is the length of punctured code), otherwise it returns n. This bound works for all codes.

66.160 UpperBoundCoveringRadiusGriesmerLike

`UpperBoundCoveringRadiusGriesmerLike(` *code* `)`

This function returns an upper bound for the covering radius of *code*, which must be linear, in a Griesmer-like fashion. It returns

$$n - \sum_{i=1}^{k} \left\lceil \frac{d}{q^i} \right\rceil \tag{66.10}$$

66.161 UpperBoundCoveringRadiusCyclicCode

`UpperBoundCoveringRadiusCyclicCode(` *code* `)`

This function returns an upper bound for the covering radius of *code*, which must be a cyclic code. It returns

$$n - k + 1 - \left\lceil \frac{w(g(x))}{2} \right\rceil, \qquad (66.11)$$

where $g(x)$ is the generator polynomial of *code*.

66.162 New code constructions

The next sections describe some new constructions for codes. The first constructions are variations on the direct sum construction, most of the time resulting in better codes than the direct sum.

The piecewise constant code construction stands on its own. Using this construction, some good codes have been obtained.

The last five constructions yield linear binary codes with fixed minimum distances and covering radii. These codes can be arbitrary long.

66.163 ExtendedDirectSumCode

`ExtendedDirectSumCode(` *L*`,` *B*`,` *m* `)`

The extended direct sum construction is described in an article by Graham and Sloane. The resulting code consists of m copies of L, extended by repeating the codewords of B m times.

Suppose L is an $[n_L, k_L]r_L$ code, and B is an $[n_L, k_B]r_B$ code (non-linear codes are also permitted). The length of B must be equal to the length of L. The length of the new code is $n = mn_L$, the dimension (in the case of linear codes) is $k \leq mk_L + k_B$, and the covering radius is $r \leq \lfloor m\Psi(L, B) \rfloor$, with

$$\Psi(L, B) = \max_{u \in F_2^{n_L}} \frac{1}{2^{k_B}} \sum_{v \in B} \mathrm{d}(L, v + u). \qquad (66.12)$$

However, this computation will not be executed, because it may be too time consuming for large codes.

If $L \subseteq B$, and L and B are linear codes, the last copy of L is omitted. In this case the dimension is $k = mk_L + (k_B - k_L)$.

```
gap> c := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> d := WholeSpaceCode( 7, GF(2) );
a cyclic [7,7,1]0 whole space code over GF(2)
gap> e := ExtendedDirectSumCode( c, d, 3 );
a linear [21,15,1..3]2 3-fold extended direct sum code
```

66.164 AmalgatedDirectSumCode

`AmalgatedDirectSumCode(` c_1, c_2 `[,` *check* `] )`

`AmalgatedDirectSumCode` returns the amalgated direct sum of the codes c_1 and c_2. The amalgated direct sum code consists of all codewords of the form $(u \,|\, 0 \,|\, v)$ if $(u \,|\, 0) \in c_1$ and $(0 \,|\, v) \in c_2$ and all codewords of the form $(u \,|\, 1 \,|\, v)$ if $(u \,|\, 1) \in c_1$ and $(1 \,|\, v) \in c_2$. The result is a code with length $n = n_1 + n_2 - 1$ and size $M <= M_1 \cdot M_2/2$.

If both codes are linear, they will first be standardized, with information symbols in the last and first coordinates of the first and second code, respectively.

If c_1 is a normal code with the last coordinate acceptable, and c_2 is a normal code with the first coordinate acceptable, then the covering radius of the new code is $r <= r_1 + r_2$. However, checking whether a code is normal or not is a lot of work, and almost all codes seem to be normal. Therefore, an option *check* can be supplied. If *check* is true, then the codes will be checked for normality. If *check* is false or omitted, then the codes will not be checked. In this case it is assumed that they are normal. Acceptability of the last and first coordinate of the first and second code, respectively, is in the last case also assumed to be done by the user.

```
gap> c := HammingCode( 3, GF(2) );
a linear [7,4,3]1 Hamming (3,2) code over GF(2)
gap> d := ReedMullerCode( 1, 4 );
a linear [16,5,8]6 Reed-Muller (1,4) code over GF(2)
gap> e := DirectSumCode( c, d );
a linear [23,9,3]7 direct sum code
gap> f := AmalgatedDirectSumCode( c, d );;
gap> MinimumDistance( f );;
gap> CoveringRadius( f );; # takes some time
gap> f;
a linear [22,8,3]7 amalgated direct sum code
```

66.165 BlockwiseDirectSumCode

`BlockwiseDirectSumCode(` c_1, l_1, c_2, l_2 `)`

`BlockwiseDirectSumCode` returns a subcode of the direct sum of c_1 and c_2. The fields of c_1 and c_2 should be same. l_1 and l_2 are two equally long lists with elements from the spaces where c_1 and c_2 are in, respectively, *or* l_1 and l_2 are two equally long lists containing codes. The union of the codes in l_1 and l_2 must be c_1 and c_2, respectively.

In the first case, the blockwise direct sum code is defined as

$$bds = \bigcup_{1 \le i \le l} (c_1 + (l_1)_i) \oplus (c_2 + (l_2)_i),$$

where l is the length of l_1 and l_2, and $\oplus$ is the direct sum.

In the second case, it is defined as

$$bds = \bigcup_{1 \le i \le l} ((l_1)_i \oplus (l_2)_i).$$

The length of the new code is $n = n_1 + n_2$.

```
gap> c := HammingCode( 3, GF(2) );;
gap> d := EvenWeightSubcode( WholeSpaceCode( 6, GF(2) ) );;
gap> BlockwiseDirectSumCode( c, [[ 0,0,0,0,0,0,0 ],[ 1,0,1,0,1,0,0 ]],
> d, [[ 0,0,0,0,0,0 ],[ 1,0,1,0,1,0 ]] );
a (13,1024,1..13)1..2 blockwise direct sum code
```

66.166 PiecewiseConstantCode

`PiecewiseConstantCode(` *part*, *weights* `[,` *field* `] )`

`PiecewiseConstantCode` returns a code with length $n = \sum n_i$, where *part* $= [n_1, ..., n_k]$. *weights* is a list of constraints, each of length k. The default field is `GF(2)`.

A constraint is a list of integers, and a word $c = (c_1, ..., c_k)$ (according to *part*) is in the resulting code if and only if $|c_i| = w_i^{(l)}$ for all $1 \leq i \leq k$ for some constraint $w^{(l)} \in$ *constraints*.

An example might be more clear

```
gap> PiecewiseConstantCode( [ 2, 3 ],
> [ [ 0, 0 ], [ 0, 3 ], [ 1, 0 ], [ 2, 2 ] ],
> GF(2) );
a (5,7,1..5)1..5 piecewise constant code over GF(2)
gap> Elements(last);
[ [ 0 0 0 0 0 ], [ 0 0 1 1 1 ], [ 0 1 0 0 0 ], [ 1 0 0 0 0 ],
  [ 1 1 0 1 1 ], [ 1 1 1 0 1 ], [ 1 1 1 1 0 ] ]
```

The first constraint is satisfied by codeword 1, the second by codeword 2, the third by codewords 3 and 4, and the fourth by codewords 5, 6 and 7.

66.167 Gabidulin codes

These five codes are derived from an article by Gabidulin, Davydov and Tombak. These five codes are defined by check matrices. Exact definitions can be found in the article.

The Gabidulin code, the enlarged Gabidulin code, the Davydov code, the Tombak code, and the enlarged Tombak code, correspond with theorem 1, 2, 3, 4, and 5, respectively in the article.

These codes have fixed minimum distance and covering radius, but can be arbitrarily long. They are defined through check matrices.

`GabidulinCode(` *m*, *w1*, *w2* `)`

`GabidulinCode` yields a code of length $5 \cdot 2^{m-2} - 1$, redundancy $2m - 1$, minimum distance 3 and covering radius 2. *w1* and *w2* should be elements of `GF(`2^{m-2}`)`.

`EnlargedGabidulinCode(` *m*, *w1*, *w2*, *e* `)`

`EnlargedGabidulinCode` yields a code of length $7 \cdot 2^{m-2} - 2$, redundancy $2m$, minimum distance 3 and covering radius 2. *w1* and *w2* are elements of `GF(`2^{m-2}`)`. *e* is an element of `GF(`2^m`)`. The core of an enlarged Gabidulin code consists of a Gabidulin code.

`DavydovCode(` *r*, *v*, *ei*, *ej* `)`

DavydovCode yields a code of length $2^v + 2^{r-v} - 3$, redundancy r, minimum distance 4 and covering radius 2. v is an integer between 2 and $r-2$. *ei* and *ej* are elements of GF(2^v) and GF(2^{r-v}), respectively.

TombakCode(*m*, *e*, *beta*, *gamma*, *w1*, *w2*)

TombakCode yields a code of length $15 \cdot 2^{m-3} - 3$, redundancy $2m$, minimum distance 4 and covering radius 2. *e* is an element of GF(2^m). *beta* and *gamma* are elements of GF(2^{m-1}). *w1* and *w2* are elements of GF(2^{m-3}).

EnlargedTombakCode(*m*, *e*, *beta*, *gamma*, *w1*, *w2*, *u*)

EnlargedTombakCode yields a code of length $23 \cdot 2^{m-4} - 3$, redundancy $2m-1$, minimum distance 4 and covering radius 2. The parameters *m*, *e*, *beta*, *gamma*, *w1* and *w2* are defined as in TombakCode. *u* is an element of GF(2^{m-1}). The code of an enlarged Tombak code consists of a Tombak code.

```
gap> GabidulinCode( 4, Z(4)^0, Z(4)^1 );
a linear [19,12,3]2 Gabidulin code (m=4) over GF(2)
gap> EnlargedGabidulinCode( 4, Z(4)^0, Z(4)^1, Z(16)^11 );
a linear [26,18,3]2 enlarged Gabidulin code (m=4) over GF(2)
gap> DavydovCode( 6, 3, Z(8)^1, Z(8)^5 );
a linear [13,7,4]2 Davydov code (r=6, v=3) over GF(2)
gap> TombakCode( 5, Z(32)^6, Z(16)^14, Z(16)^10, Z(4)^0, Z(4)^1 );
a linear [57,47,4]2 Tombak code (m=5) over GF(2)
gap> EnlargedTombakCode( 6, Z(32)^6, Z(16)^14, Z(16)^10,
> Z(4)^0, Z(4)^0, Z(32)^23 );
a linear [89,78,4]2 enlarged Tombak code (m=6) over GF(2)
```

66.168 Some functions related to the norm of a code

In the next sections, some functions that can be used to compute the norm of a code and to decide upon its normality are discussed.

66.169 CoordinateNorm

CoordinateNorm(*code*, *coord*)

CoordinateNorm returns the norm of *code* with respect to coordinate *coord*. If $C_a = \{c \in code | c_{coord} = a\}$, then the norm of *code* with respect to *coord* is defined as

$$\max_{v \in GF(q)^n} \sum_{a=1}^{q} d(x, C_a), \tag{66.13}$$

with the convention that $d(x, C_a) = n$ if C_a is empty.

```
gap> CoordinateNorm( HammingCode( 3, GF(2) ), 3 );
3
```

66.170 CodeNorm

CodeNorm(*code*)

CodeNorm returns the norm of *code*. The norm of a code is defined as the minimum of the norms for the respective coordinates of the code. In effect, for each coordinate CoordinateNorm is called, and the minimum of the calculated numbers is returned.

```
gap> CodeNorm( HammingCode( 3, GF(2) ) );
3
```

66.171 IsCoordinateAcceptable

IsCoordinateAcceptable(*code*, *coord*)

IsCoordinateAcceptable returns true if coordinate *coord* of *code* is acceptable. A coordinate is called acceptable if the norm of the code with respect to that coordinate is not more than two times the covering radius of the code plus one.

```
gap> IsCoordinateAcceptable( HammingCode( 3, GF(2) ), 3 );
true
```

66.172 GeneralizedCodeNorm

GeneralizedCodeNorm(*code*, *subcode1*, *subcode2*, ..., *subcodek*)

GeneralizedCodeNorm returns the k-norm of *code* with respect to k subcodes.

```
gap> c := RepetitionCode( 7, GF(2) );;
gap> ham := HammingCode( 3, GF(2) );;
gap> d := EvenWeightSubcode( ham );;
gap> e := ConstantWeightSubcode( ham, 3 );;
gap> GeneralizedCodeNorm( ham, c, d, e );
4
```

66.173 IsNormalCode

IsNormalCode(*code*)

IsNormalCode returns true if *code* is normal. A code is called normal if the norm of the code is not more than two times the covering radius of the code plus one. Almost all codes are normal, however some (non-linear) abnormal codes have been found.

Often, it is difficult to find out whether a code is normal, because it involves computing the covering radius. However, IsNormalCode uses much information from the literature about normality for certain code parameters.

```
gap> IsNormalCode( HammingCode( 3, GF(2) ) );
true
```

66.174 DecreaseMinimumDistanceLowerBound

DecreaseMinimumDistanceLowerBound(*code*, *s*, *iterations*)

DecreaseMinimumDistanceLowerBound is an implementation of the algorithm for the minimum distance by Leon. It is described in full detail in J.S. Leon, *A Probabilistic Algorithm for Computing Minimum Weights of Large Error-Correcting Codes*, IEEE Trans. Inform. Theory, vol. 34, September 1988.

This algorithm tries to find codewords with small minimum weights. The parameter s is described in the article, the best results are obtained if it is close to the dimension of the code. The parameter *iterations* gives the number of runs that the algorithm will perform.

The result returned is a record with two fields; the first, `mindist`, gives the lowest weight found, and `word` gives the corresponding codeword.

66.175 New miscellaneous functions

In this section, some new miscellaneous functions are described, including weight enumerators, the MacWilliams-transform and affinity and almost affinity of codes.

66.176 CodeWeightEnumerator

`CodeWeightEnumerator(` *code* `)`

`CodeWeightEnumerator` returns a polynomial of the following form

$$f(x) = \sum_{i=0}^{n} A_i x^i, \tag{66.14}$$

where A_i is the number of codewords in *code* with weight i.

```
gap> CodeWeightEnumerator( ElementsCode( [ [ 0,0,0 ], [ 0,0,1 ],
> [ 0,1,1 ], [ 1,1,1 ] ], GF(2) ) );
x^3 + x^2 + x + 1
gap> CodeWeightEnumerator( HammingCode( 3, GF(2) ) );
x^7 + 7*x^4 + 7*x^3 + 1
```

66.177 CodeDistanceEnumerator

`CodeDistanceEnumerator(` *code*`,` *word* `)`

`CodeDistanceEnumerator` returns a polynomial of the following form

$$f(x) = \sum_{i=0}^{n} B_i x^i, \tag{66.15}$$

where B_i is the number of codewords with distance i to *word*.

If *word* is a codeword, then `CodeDistanceEnumerator` returns the same polynomial as `CodeWeightEnumerator`.

```
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[0,0,0,0,0,0,1] );
x^6 + 3*x^5 + 4*x^4 + 4*x^3 + 3*x^2 + x
gap> CodeDistanceEnumerator( HammingCode( 3, GF(2) ),[1,1,1,1,1,1,1] );
x^7 + 7*x^4 + 7*x^3 + 1 # [1,1,1,1,1,1,1] ∈ HammingCode( 3, GF(2 ) )
```

66.178 CodeMacWilliamsTransform

`CodeMacWilliamsTransform(` *code* `)`

`CodeMacWilliamsTransform` returns a polynomial of the following form

$$f(x) = \sum_{i=0}^{n} C_i x^i, \tag{66.16}$$

where C_i is the number of codewords with weight i in the dual code of *code*.

```
gap> CodeMacWilliamsTransform( HammingCode( 3, GF(2) ) );
7*x^4 + 1
```

66.179 IsSelfComplementaryCode

`IsSelfComplementaryCode(` *code* `)`

`IsSelfComplementaryCode` returns *true* if

$$v \in code \Rightarrow 1 - v \in code, \tag{66.17}$$

where 1 is the all-one word of length n.

```
gap> IsSelfComplementaryCode( HammingCode( 3, GF(2) ) );
true
gap> IsSelfComplementaryCode( EvenWeightSubcode(
> HammingCode( 3, GF(2) ) ) );
false
```

66.180 IsAffineCode

`IsAffineCode(` *code* `)`

`IsAffineCode` returns `true` if *code* is an affine code. A code is called *affine* if it is an affine space. In other words, a code is affine if it is a coset of a linear code.

```
gap> IsAffineCode( HammingCode( 3, GF(2) ) );
true
gap> IsAffineCode( CosetCode( HammingCode( 3, GF(2) ),
> [ 1, 0, 0, 0, 0, 0, 0 ] ) );
true
gap> IsAffineCode( NordstromRobinsonCode() );
false
```

66.181 IsAlmostAffineCode

`IsAlmostAffineCode(` *code* `)`

`IsAlmostAffineCode` returns `true` if *code* is an almost affine code. A code is called *almost affine* if the size of any punctured code of *code* is q^r for some r, where q is the size of the alphabet of the code. Every affine code is also almost affine, and every code over `GF(2)` and `GF(3)` that is almost affine is also affine.

```
gap> code := ElementsCode( [ [0,0,0], [0,1,1], [0,2,2], [0,3,3],
>                            [1,0,1], [1,1,0], [1,2,3], [1,3,2],
>                            [2,0,2], [2,1,3], [2,2,0], [2,3,1],
>                            [3,0,3], [3,1,2], [3,2,1], [3,3,0] ],
>                            GF(4) );;
gap> IsAlmostAffineCode( code );
true
gap> IsAlmostAffineCode( NordstromRobinsonCode() );
false
```

66.182 IsGriesmerCode

`IsGriesmerCode(` *code* `)`

`IsGriesmerCode` returns `true` if *code*, which must be a linear code, is Griesmer code, and `false` otherwise.

A code is called Griesmer if its length satisfies

$$n = g[k,d] = \sum_{i=0}^{k-1} \lceil \frac{d}{q^i} \rceil. \tag{66.18}$$

```
gap> IsGriesmerCode( HammingCode( 3, GF(2) ) );
true
gap> IsGriesmerCode( BCHCode( 17, 2, GF(2) ) );
false
```

66.183 CodeDensity

`CodeDensity(` *code* `)`

`CodeDensity` returns the *density* of *code*. The density of a code is defined as

$$\frac{M \cdot V_q(n,t)}{q^n}, \tag{66.19}$$

where M is the size of the code, $V_q(n,t)$ is the size of a sphere of radius t in q^n, t is the covering radius of the code and n is the length of the code.

```
gap> CodeDensity( HammingCode( 3, GF(2) ) );
1
gap> CodeDensity( ReedMullerCode( 1, 4 ) );
14893/2048
```

Chapter 67

KBMAG

KBMAG (pronounced "Kay-bee-mag") stands for **Knuth–Bendix on Monoids, and Automatic Groups**. It is a stand-alone package written in C, for use under UNIX, with an interface to GAP3. This chapter describes its use as an external share library from within GAP3. The current interface is restricted to finitely presented groups. Interfaces for the use of KBMAG with monoids and semigroups will be released as soon as these categories exist as established types inGAP3.

To use this package effectively, some knowledge of the underlying theory and algorithms is advisable. The Knuth-Bendix algorithm is described in various places in the literature. Good general references that deal with the applications to groups and monoids are [LeC86] and the first few chapters of [Sim94]. For the theory of automatic groups see the multi-author book [ECH$^+$92]. The algorithms employed by KBMAG are described more specifically in [HER91] and [Holar].

The manual for the stand-alone KBMAG package (which can be found in the `doc` directory of the package) provides more detailed information on the external C programs that are called from GAP3. The stand-alone also includes a number of general programs for manipulating finite state automata, which could easily be made accessible from GAP3. This, and other possible extensions, such as the provision of more orderings on words, may be made in the future, depending to some extent on user-demand.

The overall objective of KBMAG is to construct a normal form for the elements of a finitely presented group G in terms of the given generators, together with a word reduction algorithm for calculating the normal form representation of an element in G, given as a word in the generators. If this can be achieved, then it is also possible to enumerate the words in normal form up to a given length, and to determine the order of the group, by counting the number of words in normal form. In most serious applications, this will be infinite, since finite groups are (with some exceptions) usually handled better by Todd-Coxeter related methods. In fact a finite state automaton W is calculated that accepts precisely the language of words in the group generators that are in normal form, and W is used for the enumeration and counting functions. It is possible to inspect W directly if required; for example, it is often possible to use W to determine whether an element in G has finite or infinite order. (See Example 4 below.)

The normal form for an element $g \in G$ is defined to be the least word in the group generators (and their inverses) that represents G, with respect to a specified ordering on the set of all words in the group generators. The available orderings are described in 67.3 below.

KBMAG offers two possible means of achieving these objectives. The first is to apply the Knuth-Bendix algorithm to the group presentation, with one of the available orderings on words, and hope that the algorithm will complete with a finite confluent presentation. (If the group is finite, then it is guaranteed to complete eventually but, like the Todd-Coxeter procedure, it may take a long time, or require more space than is available.) The second is to use the automatic group program. This also uses the Knuth-Bendix procedure as one component of the algorithm, but it aims to compute certain finite state automata rather than to obtain a finite confluent rewriting system, and it completes successfully on many examples for which such a finite system does not exist. In the current implementation, its use is restricted to the shortlex ordering on words. That is, words are ordered first by increasing length, and then words of equal length are ordered lexicographically, using the specified ordering of the generators.

For both of the above procedures, the first step is to create a GAP3 record known as a *rewriting system* R from the finitely presented group G. Some of the fields of this record can be used to specify the input parameters for the external programs, such as the ordering on words to be used by the Knuth-Bendix procedure. One of the two external programs is then run on R. If successful, it updates some of the fields of R, which can then be used to reduce words in the group generators to normal form, and to count and enumerate the words in normal form.

In fact, the relationship of a rewriting system to that of the group from which it is constructed is in many ways similar to that between a Presentation Record and its associated finitely presented group, as described in 23.8. In particular, the rewriting rules, which can be thought of as (a highly redundant) set of defining relations for the group, can be changed, whereas the defining relators of a finitely presented group cannot be altered without effectively changing the group itself.

In the descriptions of the functions that follow, it is important to distinguish between irreducible words, and words in normal form. As already stated, a word is in normal form if it is the least word under the ordering of the rewriting system that defines a particular group element. So there is always a unique word in normal form for each group element, and it is determined by the group generators and the ordering on words in the group generators. A word in a rewriting system is said to be irreducible if it does not contain the left hand side of any of the reduction rules in the system as a subword. Words in normal form are always irreducible, but the converse is true if and only if the rewriting system is confluent. The automatic groups programs provide a method of reducing words to normal form without obtaining a finite confluent rewriting system (which may not even exist).

Diagnostic output from the GAP3 procedures can be turned on by setting the global variable `InfoRWS` to `Print`. Diagnostic output from the external programs is controlled by setting the `silent`, `verbose` or `veryVerbose` flags of the rewriting system. See 67.4 below.

67.1 Creating a rewriting system

`FpGroupToRWS(`G `[,`*case_change*`])`

`FpGroupToRWS` constructs and returns a rewriting system R from a finitely presented group G. The generators of R are the generators of G together with inverses for those generators which are not obviously involutory. Normally, if a generator of G prints as a, say, then its inverse will print, as might be expected, as a^{-1}. However, if the optional argument *case_change* is set to `true`, then its printing string will be derived by changing the case of the letters in the original generator; so, the inverse of a will print as A. One advantage of this is that it can save space in the temporary files used by the external programs.

R is a GAP3 record. However, its internal storage does not correspond precisely to the way in which it is displayed, and so the user is strongly advised not to attempt to modify its fields directly. (To convince yourself of this, try examining some of the fields individually.) In particular, the ordering on words to be used by the Knuth-Bendix procedure should be changed, if desired, by using the functions `SetOrderingRWS` and `ReorderGeneratorsRWS` described in 67.3 below. However, the control fields that are described in 67.4 below are designed to be set directly.

67.2 Elementary functions on rewriting systems

`IsRWS(`*rws*`)`

Returns true if *rws* is a rewriting system.

`IsConfluentRWS(`*rws*`)`

Returns true if *rws* is a rewriting system that is known to be confluent.

`IsAvailableNormalForm(`*rws*`)`

Returns true if *rws* is a rewriting system with an associated normal form. When this is the case, the word-reduction, counting and enumeration functions may be applied to *rws* and are guaranteed to give the correct answer.

The normal form can only be created by applying one of the two functions `KB` or `Automata` to *rws*.

`IsAvailableReductionRWS(`*rws*`)`

Returns true if *rws* is a rewriting system for which words can be reduced. When this is the case, the word-reduction, counting and enumeration functions may be applied to *rws*, but are NOT guaranteed to give the correct answer.

The result of `ReduceWordRWS` will always be equal to its argument in the underlying group of *rws*, but it may not be the correct normal form. The counting and enumeration algorithms may return answers that are too large (never too small). This situation results when `KB` is run and exits, for some reason, with a non-confluent system of equations.

`ResetRWS(`*rws*`)`

This function resets the rewriting system *rws* back to its form as it was before the application of `KB` or `Automata`. However, the current ordering and values of control parameters will not be changed. The normal form and reduction algorithms will be unavailable after this call.

`AddOriginalEqnsRWS(`*rws*`)`

Occasionally, as a result of a call of `KB` on the rewriting system *rws*, some rewriting rules can be lost, which means that the underlying group of *rws* is changed. This function appends the original defining relations of the group to the rewriting system, which ensures that the underlying group is made correct again. It is advisable to call this function in between two calls of `KB` on the same rewriting system.

67.3 Setting the ordering

`SetOrderingRWS(`*rws*`,` *ordering* `[,`*list*`])`
`ReorderGeneratorsRWS(`*rws*`,` *p*`)`

`SetOrderingRWS` changes the ordering on the words of the rewriting system *rws* to *ordering*, which must be one of the strings "shortlex", "recursive", "wtlex" and "wreathprod". The default is "shortlex", and this is the ordering of rewriting systems returned by `FpGroupToRWS`. The orderings "wtlex" and "wreathprod" require the third parameter, *list*, which must be a list of non-negative integers in one-one correspondence with the generators of *rws*, in the order that they are displayed in the `generatorOrder` field. They have the effect of attaching weights or levels to the generators, in the cases "wtlex" and "wreathprod", respectively.

Each of these orderings depends on the order of the generators, The current ordering of generators is displayed under the `generatorOrder` field when *rws* is printed. This ordering can be changed by the function `ReorderGeneratorsRWS`. The second parameter p to this function should be a permutation that moves at most ng points, where ng is the number of generators. This permutation is applied to the current list of generators.

In the "shortlex" ordering, shorter words come before longer ones, and, for words of equal length, the lexicographically smaller word comes first, using the ordering of generators specified by the `generatorOrder` field. The "wtlex" ordering is similar, but instead of using the length of the word as the first criterion, the total weight of the word is used; this is defined as the sum of the weights of the generators in the word. So "shortlex" is the special case of "wtlex" in which all generators have the same nonzero weight.

The "recursive" ordering is the special case of "wreathprod" in which the levels of the ng generators are $1, 2, \ldots, ng$, in the order defined by the `generatorOrder` field. We shall not attempt to give a complete definition of these orderings here, but refer the reader instead to pages 46–50 of [Sim94]. The "recursive" ordering is the one appropriate for a power-conjugate presentation of a polycyclic group, but where the generators are ordered in the reverse order from the usual convention for polycyclic groups. The confluent presentation will then be the same as the power-conjugate presentation. For example, for the Heisenberg group $\langle x, y, z \mid [x, z] = [y, z] = 1, [y, x] = z\rangle$, a good ordering is "recursive" with the order of generators $[z^{-1}, z, y^{-1}, y, x^{-1}, x]$. This example is included in 68.17 below.

67.4 Control parameters

The Knuth-Bendix procedure is unusually sensitive to the settings of a number of parameters that control its operation. In some examples, a small change in one of these parameters can mean the difference between obtaining a confluent rewriting system fairly quickly on the one hand, and the procedure running on until it uses all available memory on the other hand.

Unfortunately, it is almost impossible to give even very general guidelines on these settings, although the "wreathproduct" orderings appear to be more sensitive than the "shortlex" and "wtlex" orderings. The user can only acquire a feeling for the influence of these parameters by experimentation on a large number of examples.

The control parameters are defined by the user by setting values of certain fields of a rewriting system *rws* directly. These fields will now be listed.

rws.`maxeqns`
A positive integer specifying the maximum number of rewriting rules allowed in *rws*. The default is 32767. If this number is exceeded, then `KB` or `Automata` will abort.

rws.`tidyint`
A positive integer, 100 by default. During the Knuth-Bendix procedure, the search for overlaps is interrupted periodically to tidy up the existing system by removing and/or simplifying rewriting rules that have become redundant. This tidying is done after finding *rws*.`tidyint` rules since the last tidying.

rws.`confnum`
A positive integer, 500 by default. If *rws*.`confnum` overlaps are processed in the Knuth-Bendix procedure but no new rules are found, then a fast test for confluence is carried out. This saves a lot of time if the system really is confluent, but usually wastes time if it is not.

rws.`maxstoredlen`
This is a list of two positive integers, *maxlhs* and *maxrhs*; the default is that both are infinite. Only those rewriting rules for which the left hand side has length at most *maxlhs* and the right hand side has length at most *maxrhs* are stored; longer rules are discarded. In some examples it is essential to impose such limits in order to obtain a confluent rewriting system. Of course, if the Knuth-Bendix procedure halts with such limits imposed, then the resulting system need not be confluent. However, the confluence can then be tested be re-running `KB` with the limits removed. (To remove the limits, unbind the field.) It is advisable to call `AddOriginalEqnsRWS` on *rws* before re-running `KB`.

rws.`maxoverlaplen`
This is an integer, which is infinite by default (when not set). Only those overlaps of total length *rws*.`maxoverlaplen` are processed. Similar remarks apply to those for `maxstoredlen`.

rws.`sorteqns`
This should be true or false, and false is the default. When it is true, the rewriting rules are output in order of increasing length of left hand side. (The default is that they are output in the order that they were found).

rws.`maxoplen`
This is an integer, which is infinite by default (when not set). When it is set, the rewriting rules are output in order of increasing length of left hand side (as if *rws*.`sorteqns` were true), and only those rules having left hand sides of length up to *rws*.`maxoplen` are output at all. Again, similar remarks apply to those for `maxstoredlen`.

rws.`maxreducelen`
A positive integer, 32767 by default. This is the maximum length that a word is allowed to have during the reduction process. It is only likely to be exceeded when using the “wreathproduct” or “recursive” ordering.

rws.`silent`, *rws*.`verbose`, *rws*.`veryVerbose`
These should be true or false, and are false by default. It only makes sense to set one of them to be true. They control the amount of diagnostic output that is printed by `KB` and `Automata`. By default there is a small amount of such output.

rws.`maxstates`, *rws*.`maxwdiffs`

These are positive integers, controlling the maximum number of states of the word-reduction automaton used by `KB`, and the maximum number of word-differences allowed when running `Automata`, respectively. These numbers are normally increased automatically when required, so it unusual to want to set these flags. They can be set when either it is desired to limit these parameters (and prevent them being increased automatically), or (as occasionally happens), the number of word-differences increases too rapidly for the program to cope - when this happens, the run is usually doomed to failure anyway.

67.5 The Knuth-Bendix program

`KB(`*rws*`)`

Run the external Knuth-Bendix program on the rewriting system *rws*. `KB` returns true if it finds a confluent rewriting system and otherwise false. In either case, if it halts normally, then it will update *rws* by changing the `equations` field, which contains a list of the rewriting rules, and by appending a finite state automaton *rws*.`reductionFSA` that can be used for word reduction, and the counting and enumeration of irreducible words.

All control parameters (as defined in the preceding section) should be set before calling `KB`. In the author's experience, it is usually most helpful to run `KB` with the verbose flag *rws*.`verbose` set, in order to follow what is happening. `KB` will halt either when it finds a finite confluent system of rewriting rules, or when one of the control parameters (such as *rws*.`maxeqns`) requires it to stop. The program can also be made to halt and output manually at any time by hitting the interrupt key (normally *ctr*-`C`) once. (Hitting it twice has unpredictable consequences, since GAP3 may intercept the signal.)

If `KB` halts without finding a confluent system, but still manages to output the current system and update *rws*, then it is possible to use the resulting rewriting system to reduce words, and count and enumerate the irreducible words; it cannot be guaranteed that the irreducible words are all in normal form, however. It is also possible to re-run `KB` on the current system, usually after altering some of the control parameters. In fact, is some more difficult examples, this seems to be the only means of finding a finite confluent system.

67.6 The automatic groups program

`Automata(`*rws*`, [`*large*`], [`*filestore*`], [`*diff1*`])`

Run the external automatic groups program on the rewriting system *rws*. `Autgroup` returns true if successful and false otherwise. If successful, it appends two finite state automata *rws*.`diff1c` and *rws*.`wa` to *rws*. The first of these can be used for word-reduction, and the second for counting and enumeration of irreducible words (i.e. words in normal form). In fact, the second is the word-acceptor of the automatic structure. (The multiplier automata of the automatic structure are not currently saved when using the GAP3 interface. To access these, the user should either use KBMAG as a stand-alone, or complain to the author.)

The three optional parameters to `Automata` are all boolean, and false by default. Setting *large* true results in some of the control parameters (such as *rws*.`maxeqns` and *rws*.`tidyint`) being set larger than they would be otherwise. This is necessary for examples that require a large amount of space. Setting *filestore* true results in more use being made of temporary files than would be otherwise. This makes the program run slower, but it may be necessary if you are short of core memory. Setting *diff1* to be true is a more technical option, which is explained more fully in the documentation for the stand-alone KBMAG package. It is not usually necessary or helpful, but it enables one or two examples to complete that would otherwise run out of space.

The ordering field of *rws* must currently be equal to "shortlex" for `Automata` to be applicable. The control parameters for *rws* that are likely to be relevant are `maxeqns` and `maxwdiffs`.

67.7 Word reduction

`IsReducedWordRWS(`*rws*`,`*w*`)`

Test whether the word w in the generators of the rewriting system *rws* (or, equivalently, in the generators of the underlying group of *rws*) is reduced or not, and return true or false.

`IsReducedWordRWS` can only be used after `KB` or `Automata` has been run successfully on *rws*. In the former case, if `KB` halted without a confluent set of rules, then irreducible words are not necessarily in normal form (but reducible words are definitely not in normal form). If `KB` completes with a confluent rewriting system or `Automata` completes successfully, then it is guaranteed that all irreducible words are in normal form.

`ReduceWordRWS(`*rws*`,`*w*`)`

Reduce the word w in the generators of the rewriting system *rws* (or, equivalently, in the generators of the underlying group of *rws*), and return the result.

`ReduceWordRWS` can only be used after `KB` or `Automata` has been run successfully on *rws*. In the former case, if `KB` halted without a confluent set of rules, then the irreducible word returned is not necessarily in normal form. If `KB` completes with a confluent rewriting system or `Automata` completes successfully, then it is guaranteed that all irreducible words are in normal form.

67.8 Counting and enumerating irreducible words

`SizeRWS(`*rws*`)`

Returns the number of irreducible words in the rewriting system *rws*. If this is infinite, then the string "infinite" is returned.

`SizeRWS` can only be used after `KB` or `Automata` has been run successfully on *rws*. In the former case, if `KB` halted without a confluent set of rules, then the number of irreducible words may be greater than the number of words in normal form (which is equal to the order of the underlying group of *rws*). If `KB` completes with a confluent rewriting system or `Automata` completes successfully, then it is guaranteed that `SizeRWS` will return the correct order of the underlying group.

`EnumerateRWS(`*rws*`, `*min*`, `*max*`)`

Enumerate all irreducible words in the rewriting system *rws* that have lengths between *min* and *max* (inclusive), which should be non-negative integers. The result is returned as a list of words. The enumeration is by depth-first search of a finite state automaton, and so the words in the list returned are ordered lexicographically (not by shortlex).

`EnumerateRWS` can only be used after `KB` or `Automata` has been run successfully on *rws*. In the former case, if `KB` halted without a confluent set of rules, then not all irreducible words in the list returned will necessarily be in normal form. If `KB` completes with a confluent rewriting system or `Automata` completes successfully, then it is guaranteed that all words in the list will be in normal form.

`SortEnumerateRWS(`*rws*`,` *min*`,` *max*`)`

This is the same as `EnumerateRWS` but the list returned contains the words in shortlex order; so shorter words come before longer ones. It is slightly slower than `EnumerateRWS`.

`SizeEnumerateRWS(`*rws*`,` *min*`,` *max*`)`

This returns the length of the list that would be returned by `EnumerateRWS(`*rws*`,` *min*`,` *max*`)`; that is, the number of irreducible words of *rws* that have lengths between *min* and *max* inclusive. It is faster than `EnumerateRWS`, since it does not need to store the words enumerated.

67.9 Rewriting System Examples

`Example 1`

We start with a easy example - the alternating group A_4.

```
gap> G:=FreeGroup("a","b");;
gap> a:=G.1;; b:=G.2;;
gap> G.relators:=[a^2, b^3, (a*b)^3];;
gap> R:=FpGroupToRWS(G);
rec(
           isRWS := true,
  generatorOrder := [a,b,b^-1],
        inverses := [a,b^-1,b],
        ordering := "shortlex",
       equations := [
         [b^2,b^-1],
         [a*b*a,b^-1*a*b^-1]
       ]
)
gap> KB(R);
# System is confluent.
# Halting with 11 equations.
true
gap> R;
rec(
           isRWS := true,
     isConfluent := true,
  generatorOrder := [a,b,b^-1],
```

```
          inverses := [a,b^-1,b],
          ordering := "shortlex",
         equations := [
           [a^2,IdWord],
           [b*b^-1,IdWord],
           [b^-1*b,IdWord],
           [b^2,b^-1],
           [b^-1*a*b^-1,a*b*a],
           [b^-2,b],
           [b*a*b,a*b^-1*a],
           [b^-1*a*b*a,b*a*b^-1],
           [a*b*a*b^-1,b^-1*a*b],
           [b*a*b^-1*a,b^-1*a*b],
           [a*b^-1*a*b,b*a*b^-1]
         ]
)
```

The equations field of R is now a complete system of rewriting rules

```
gap> SizeRWS(R);
12
gap> SortEnumerateRWS(R,0,12);
[ IdWord, a, b, b^-1, a*b, a*b^-1, b*a, b^-1*a, a*b*a, a*b^-1*a,
  b*a*b^-1, b^-1*a*b ]
```

We have enumerated all of the elements of the group

Example 2

The Heisenberg group - that is, the free 2-generator nilpotent group of class 2. For this to complete, we need to use the recursive ordering, and reverse our initial order of generators. (Alternatively, we could avoid this reversal, by using a wreathproduct ordering, and setting the levels of the generators to be 6,5,4,3,2,1.)

```
gap> G:=FreeGroup("x","y","z");;
gap> x:=G.1;; y:=G.2;; z:=G.3;;
gap> G.relators:=[Comm(y,x)*z^-1, Comm(z,x), Comm(z,y)];;
gap> R:=FpGroupToRWS(G);
rec(
             isRWS := true,
  generatorOrder := [x,x^-1,y,y^-1,z,z^-1],
          inverses := [x^-1,x,y^-1,y,z^-1,z],
          ordering := "shortlex",
         equations := [
           [y^-1*x^-1*y,z*x^-1],
           [z^-1*x^-1,x^-1*z^-1],
           [z^-1*y^-1,y^-1*z^-1]
         ]
)
gap> SetOrderingRWS(R,"recursive");
gap> ReorderGeneratorsRWS(R,(1,6)(2,5)(3,4));
gap> R;
rec(
```

```
           isRWS := true,
  generatorOrder := [z^-1,z,y^-1,y,x^-1,x],
        inverses := [z,z^-1,y,y^-1,x,x^-1],
        ordering := "recursive",
       equations := [
         [y^-1*x^-1*y,z*x^-1],
         [z^-1*x^-1,x^-1*z^-1],
         [z^-1*y^-1,y^-1*z^-1]
       ]
)
gap> KB(R);
# System is confluent.
# Halting with 18 equations.
true
gap> R;
rec(
           isRWS := true,
     isConfluent := true,
  generatorOrder := [z^-1,z,y^-1,y,x^-1,x],
        inverses := [z,z^-1,y,y^-1,x,x^-1],
        ordering := "recursive",
       equations := [
         [z^-1*z,IdWord],
         [z*z^-1,IdWord],
         [y^-1*y,IdWord],
         [y*y^-1,IdWord],
         [x^-1*x,IdWord],
         [x*x^-1,IdWord],
         [z^-1*x^-1,x^-1*z^-1],
         [z^-1*y^-1,y^-1*z^-1],
         [y^-1*x^-1,x^-1*y^-1*z],
         [z*x^-1,x^-1*z],
         [z^-1*x,x*z^-1],
         [z*y^-1,y^-1*z],
         [z^-1*y,y*z^-1],
         [y*x,x*y*z],
         [y^-1*x,x*y^-1*z^-1],
         [y*x^-1,x^-1*y*z^-1],
         [z*x,x*z],
         [z*y,y*z]
       ]
)
gap> SizeRWS(R);
"infinity"
gap> IsReducedWordRWS(R,z*y*x);
false
gap> ReduceWordRWS(R,z*y*x);
x*y*z^2
```

```
gap> IsReducedWordRWS(R,x*y*z^2);
true
```

`Example 3`

This is an example of the use of the Knuth-Bendix algorithm to prove the nilpotence of a finitely presented group. (The method is due to Sims, and is described in Chapter 11.8 of [Sim94].) This example is of intermediate difficulty, and demonstrates the necessity of using the `maxstoredlen` control parameter.

The group is

$$\langle a, b \mid [b, a, b], [b, a, a, a, a], [b, a, a, a, b, a, a] \rangle$$

with left-normed commutators. The first step in the method is to check that there is a maximal nilpotent quotient of the group, for which we could use, for example, the GAP3 `NilpotentQuotient` command, from the shared-library "nq". We find that there is a maximal such quotient, and it has class 7, and the layers going down the lower central series have the abelian structures [0,0], [0], [0], [0], [0], [2], [2].

By using the stand-alone `C` nilpotent quotient program, it is possible to find a power-commutator presentation of this maximal quotient. We now construct a new presentation of the same group, by introducing the generators in this power-commutator presentation, together with their definitions as powers or commutators of earlier generators. It is this new presentation that we use as input for the Knuth-Bendix program. Again we use the recursive ordering, but this time we will be careful to introduce the generators in the correct order in the first place!

```
gap> G:=FreeGroup("h","g","f","e","d","c","b","a");;
gap> h:=G.1;;g:=G.2;;f:=G.3;;e:=G.4;;d:=G.5;;c:=G.6;;b:=G.7;;a:=G.8;;
gap> G.relators:=[Comm(b,a)*c^-1, Comm(c,a)*d^-1, Comm(d,a)*e^-1,
> Comm(e,b)*f^-1, Comm(f,a)*g^-1, Comm(g,b)*h^-1,
> Comm(g,a), Comm(c,b), Comm(e,a)];;
gap> R:=FpGroupToRWS(G);
rec(
           isRWS := true,
  generatorOrder := [h,h^-1,g,g^-1,f,f^-1,e,e^-1,d,d^-1,c,c^-1,
                                                b,b^-1,a,a^-1],
        inverses := [h^-1,h,g^-1,g,f^-1,f,e^-1,e,d^-1,d,c^-1,c,
                                                b^-1,b,a^-1,a],
        ordering := "shortlex",
       equations := [
         [b^-1*a^-1*b,c*a^-1],
         [c^-1*a^-1*c,d*a^-1],
         [d^-1*a^-1*d,e*a^-1],
         [e^-1*b^-1*e,f*b^-1],
         [f^-1*a^-1*f,g*a^-1],
         [g^-1*b^-1*g,h*b^-1],
         [g^-1*a^-1,a^-1*g^-1],
         [c^-1*b^-1,b^-1*c^-1],
         [e^-1*a^-1,a^-1*e^-1]
       ]
)
```

```
gap> SetOrderingRWS(R,"recursive");
```

A little experimentation reveals that this example works best when only those equations with left and right hand sides of lengths at most 10 are kept.

```
gap> R.maxstoredlen:=[10,10];;
gap> R.verbose:=true;;
gap> KB(R);
  # 60 eqns; total len: lhs, rhs = 129, 143; 25 states; 0 secs.
  # 68 eqns; total len: lhs, rhs = 364, 326; 28 states; 0 secs.
  # 77 eqns; total len: lhs, rhs = 918, 486; 45 states; 0 secs.
  # 91 eqns; total len: lhs, rhs = 728, 683; 58 states; 0 secs.
  # 102 eqns; total len: lhs, rhs = 1385, 1479; 89 states; 0 secs.
  . . . .
  # 310 eqns; total len: lhs, rhs = 4095, 4313; 489 states; 1 secs.
  # 200 eqns; total len: lhs, rhs = 2214, 2433; 292 states; 1 secs.
  # 194 eqns; total len: lhs, rhs = 835, 922; 204 states; 1 secs.
  # 157 eqns; total len: lhs, rhs = 702, 723; 126 states; 1 secs.
  # 151 eqns; total len: lhs, rhs = 553, 444; 107 states; 1 secs.
  # 101 eqns; total len: lhs, rhs = 204, 236; 19 states; 1 secs.
  # No new eqns for some time - testing for confluence
  # System is not confluent.
  # 172 eqns; total len: lhs, rhs = 616, 473; 156 states; 1 secs.
  # 171 eqns; total len: lhs, rhs = 606, 472; 156 states; 1 secs.
  # No new eqns for some time - testing for confluence
  # System is not confluent.
  # 151 eqns; total len: lhs, rhs = 452, 453; 92 states; 1 secs.
  # 151 eqns; total len: lhs, rhs = 452, 453; 92 states; 1 secs.
  # No new eqns for some time - testing for confluence
  # System is not confluent.
  # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 1 secs.
  # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 1 secs.
  # No new eqns for some time - testing for confluence
# System is confluent.
# Halting with 101 equations.
WARNING: The monoid defined by the presentation may have changed,
         since equations have been discarded.
         If you re-run, include the original equations.
true
# We re-run with the maxstoredlen limit removed and the original
# equations added, to check that the system really is confluent.
gap> Unbind(R.maxstoredlen);
gap> AddOriginalEqnsRWS(R);
gap> KB(R);
  # 101 eqns; total len: lhs, rhs = 200, 239; 15 states; 0 secs.
  # No new eqns for some time - testing for confluence
# System is confluent.
# Halting with 101 equations.
true
```

In fact, in this case, we did have a confluent set already.

Inspection of the confluent set now reveals it to be precisely a power-commutator presentation of a nilpotent group, and so we have proved that the group we started with really is nilpotent. Of course, this means also that it is equal to its largest nilpotent quotient, of which we already know the structure.

`Example 4`

Our final example illustrates the use of the `Automata` command, which runs the automatic groups programs. The group has a balanced symmetrical presentation with 3 generators and 3 relators, and was originally proposed by Heineken as a possible example of a finite group with such a presentation. In fact, the `Automata` command proves it to be infinite.

This example is of intermediate difficulty, but there is no need to use any special options. It takes about 20 minutes to run on a fast WorkStation.

We will not attempt to explain all of the output in detail here; the interested user should consult the documentation for the stand-alone KBMAG package. Roughly speaking, it first runs the Knuth-Bendix program, which does not halt with a confluent rewriting system, but is used instead to construct a word-difference finite state automaton. This in turn is used to construct the word-acceptor and multiplier automata for the group. Sometimes the initial constructions are incorrect, and part of the procedure consists in checking for this, and making corrections. In fact, in this example, the correct automata are considerably smaller than the ones first constructed. The final stage is to run an axiom-checking program, which essentially checks that the automata satisfy the group relations. If this completes successfully, then the correctness of the automata has been proved, and they can be used for correct word-reduction and enumeration in the group.

```
gap> G:=FreeGroup("a","b","c");;
gap> a:=G.1;;b:=G.2;;c:=G.3;;
gap> G.relators:=[Comm(a,Comm(a,b))*c^-1, Comm(b,Comm(b,c))*a^-1,
>                  Comm(c,Comm(c,a))*b^-1];
[ a^-1*b^-1*a^-1*b*a*b^-1*a*b*c^-1, b^-1*c^-1*b^-1*c*b*c^-1*b*c*a^-1,
  c^-1*a^-1*c^-1*a*c*a^-1*c*a*b^-1 ]
gap> R:=FpGroupToRWS(G);
rec(
           isRWS := true,
  generatorOrder := [a,a^-1,b,b^-1,c,c^-1],
        inverses := [a^-1,a,b^-1,b,c^-1,c],
        ordering := "shortlex",
       equations := [
         [a^-1*b^-1*a^-1*b*a,c*b^-1*a^-1*b],
         [b^-1*c^-1*b^-1*c*b,a*c^-1*b^-1*c],
         [c^-1*a^-1*c^-1*a*c,b*a^-1*c^-1*a]
       ]
)
gap> Automata(R);
# Running Knuth-Bendix Program
# Maximum number of equations exceeded.
# Halting with 200 equations.
# First word-difference machine with 161 states computed.
```

```
# Second word-difference machine with 175 states computed.
# Re-running Knuth-Bendix Program
# Halting with 7672 equations.
# First word-difference machine with 259 states computed.
# Second word-difference machine with 269 states computed.
# System is confluent, or halting factor condition holds.
# Running program to construct word-acceptor and multiplier automata
# Word-acceptor with 5488 states computed.
# General multiplier with 6806 states computed.
# Multiplier incorrect with generator number 2.
# Equation found between two words accepted by word-acceptor.
# Word-acceptor with 1106 states computed.
# General multiplier with 2428 states computed.
# Validity test on general multiplier succeeded.
# Running program to verify axioms on the automatic structure
# General length-2 multiplier with 2820 states computed.
# Checking inverse and short relations.
# Checking relation: a^-1*b^-1*a^-1*b*a = c*b^-1*a^-1*b
# Checking relation: b^-1*c^-1*b^-1*c*b = a*c^-1*b^-1*c
# Checking relation: c^-1*a^-1*c^-1*a*c = b*a^-1*c^-1*a
# Axiom checking succeeded.
# Minimal reducible word acceptor with 1058 states computed.
# Minimal Knuth-Bendix equation fsa with 1891 states computed.
# Minimal diff1 fsa with 271 states computed.
true
gap> SizeRWS(R);
"infinity"
```

We have proved that the group is infinite, but it would also be interesting to know whether the group generators have infinite order. This can often be shown by inspecting the word-acceptor automaton directly.

The GAP3 interface to KBMAG includes a number of (currently undocumented) functions for manipulating finite state automata. The calculation below illustrates the use of one or two of these. In this example, it turns out that all powers of the generators are accepted by the word-acceptor automaton `R.wa`. The accepted language of this automaton is precisely the set of words in normal form, and so this proves that each of these powers is in normal form - so, in particular, no such power is equal to the identity, and the generators have infinite order.

The comments in the example below were added after the run.

```
gap> IsFSA(R.wa);
true   # R.wa is a finite-state automaton.
gap> RecFields(R.wa);
[ "isFSA", "alphabet", "states", "flags", "initial",
  "accepting", "table", "denseDTable", "operations",
  "isInitializedFSA" ]
gap> R.wa.states.size;
1106 # The number of states of the automaton R.wa
gap> R.wa.initial;
```

```
[ 1 ]    # The initial state of R.wa is state number 1.
gap> R.wa.flags;
[ "BFS", "DFA", "accessible", "minimized", "trim" ]
# The flags fields list properties that are known to be true in the
# automaton. For example, "DFA" means that it is deterministic.
# The alphabet of the automaton is the set of integers {1,...,6},
# the integer i in this set corresponds to the i-th generator of
# R, as listed in R.generatorOrder.
# To inspect transitions, we use the function TargetDFA.
gap> TargetDFA(R.wa,1,1);
2    #  The first generator, a, maps the initial state 1 to state 2.
gap> TargetDFA(R.wa,1,2);
8    # It maps state 2 to state 8 -
gap> TargetDFA(R.wa,1,8);
8    # and state 8 to itself.
gap> 8 in R.wa.accepting;
true
```

We now know that all powers of the first generator, a, map the initial state of the word-acceptor to an accepting state, which establishes our claim that all powers of a are in normal form. In fact, the same can be verified for all 6 generators.

Chapter 68

The Matrix Package

This chapter describes functions which may be used to construct and investigate the structure of matrix groups defined over finite fields.

68.1 Aim of the matrix package

The aim of the matrix package is to provide integrated and comprehensive access to a collection of algorithms, developed primarily over the past decade, for investigating the structure of matrix groups defined over finite fields. We sought to design a package which provides easy access to existing algorithms and implementations, and which both allows new algorithms to be developed easily using existing components, and to update existing ones readily.

Some of the facilities provided are necessarily limited, both on theoretical and practical grounds; others are **experimental** and developmental in nature; we welcome criticism of their performance. One motivation for its release is to encourage input from others.

68.2 Contents of the matrix package

We summarise the contents of the package and provide references for the relevant algorithms.

(a) Irreducibility and absolutely irreducibility for G-modules; isomorphism testing for irreducible G-modules; see Holt and Rees [5]. The corresponding functions are described in 68.8, 68.9, 68.14, 68.15, 68.16, 68.25, 68.26.

(b) Decide whether a matrix group has certain decompositions with respect to a normal subgroup; see Holt, Leedham-Green, O'Brien and Rees [6]. The corresponding functions are described in 68.10, 68.13, 68.28, 68.29, 68.30, and 68.31.

(c) Decide whether a matrix group is primitive; see Holt, Leedham-Green, O'Brien and Rees [7]. The corresponding functions are described in 68.11, 68.32.

(d) Decide whether a given group contains a classical group in its natural representation. Here we provide access to the algorithms of Celler and Leedham-Green [3] and those of Niemeyer and Praeger [11, 12]. The corresponding function is described in 68.19, the associated lower-level functions in 68.22 and 68.23.

(e) A constructive recognition process for the special linear group developed by Celler and Leedham-Green [4] and described in 68.20.

(e) Random element selection; see Celler, Leedham-Green, Murray, Niemeyer and O'Brien [1]. The corresponding functions are described in 68.48, 68.49.

(f) Matrix order calculation; see Celler and Leedham-Green [2]. The corresponding functions are described in 68.47.

(g) Base point selection for the Random Schreier-Sims algorithm for matrix groups; see Murray and O'Brien [10]. The corresponding function is described in 68.45.

(h) Decide whether a matrix group preserves a tensor decomposition; see Leedham-Green and O'Brien [8, 9]. The corresponding function is described in 68.12.

(i) Recursive exploration of reducible groups; see Pye [13]. The corresponding function is described in 68.21.

The algorithms make extensive use of Aschbacher's classification of the maximal subgroups of the general linear group. Possible classes of subgroups mentioned below refer to this classification; see [14, 15] for further details.

In order to access the functions, you must use the command `RequirePackage` to load them.

```
gap> RequirePackage("matrix");
```

68.3 The Developers of the matrix package

The development and organisation of this package was carried out in Aachen by Frank Celler, Eamonn O'Brien and Anthony Pye.

In addition to the new material, this package combines, updates, and replaces material from various contributing sources. These include:

1. Classic package – originally developed by Celler;
2. Smash package – originally developed by Holt, Leedham-Green, O'Brien, and Rees;
3. Niemeyer/Praeger classical recognition algorithm – originally developed by Niemeyer;
4. Recursive code – originally developed by Pye.

As part of the preparation of this package, much of the contributed code was revised (sometimes significantly) and streamlined, in cooperation with the original developers.

Comments and criticisms are welcome and should be directed to:

Eamonn O'Brien
obrien@math.auckland.ac.nz

68.4 Basic conventions employed in matrix package

A G-module is defined by the action of a group G, generated by a set of matrices, on a d-dimensional vector space over a field, $F = GF(q)$.

The function `GModule` returns a G-module record, where the component `.field` is set to F, `.dimension` to d, `.generators` to the set of generating matrices for G, and `.isGModule` to true. These components are set for every G-module record constructed using `GModule`.

Many of the functions described below return or update a G-module record. Additional components which describe the nature of the action of the underlying group G on the G-module are set by these functions. Some of these carry information which may be of general use. These components are described briefly in 68.34. They need not appear in a G-module record, or may have the value `"unknown"`.

A component `.component` of a G-module record is accessed by `ComponentFlag` and its value is set by `SetComponentFlag`, where the first letter of the component is capitalised in the function names. For example, the component `.tensorBasis` of *module* is set by `SetTensorBasisFlag(` *module*`,` *boolean* `)` and its value accessed by `TensorBasisFlag(` *module* `)`. Such access functions and conventions also apply to other records constructed by all of these functions.

If a function listed below takes as input a matrix group G, it also **usually** accepts a G-module.

68.5 Organisation of this manual

Sections 68.6 and 68.7 describe how to construct a G-module from a set of matrices or a group and how to test for a G-module.

Sections 68.8, 68.9, 68.10, 68.11, and 68.12 describe high-level functions which provide access to some of the algorithms mentioned in 68.2; these are tests for reducibility, semi-linearity, primitivity, and tensor decomposition, respectively.

Section 68.13 describes `SmashGModule` which can be used to explore whether a matrix group preserves certain decompositions with respect to a normal subgroup.

Sections 68.14, 68.15, and 68.16 consider homomorphisms between and composition factors of G-modules.

Sections 68.18, 68.19, and 68.20 describe functions for exploring classical groups.

Section 68.21 describes the **experimental** function `RecogniseMatrixGroup`.

Sections 68.22 and 68.23 describe the low-level classical recognition functions.

Sections 68.24, 68.25, 68.26, and 68.27 describe the low-level Meataxe functions.

Sections 68.28, 68.29, 68.30, 68.31, 68.32, 68.33, and 68.34 describe the low-level `SmashGModule` functions.

Sections 68.35, 68.36, 68.37, and 68.38 describe the low-level functions for the function `RecogniseMatrixGroup`.

Sections 68.39, 68.40, 68.41, 68.42, 68.43, and 68.44 describe functions to construct new G-modules from given ones.

Sections 68.45 to 68.52 describe functions which are somewhat independent of G-modules; these include functions to compute the order of a matrix, construct a permutation representation for a matrix group, and construct pseudo-random elements of a group.

Section 68.53 provides a bibliography.

68.6 GModule

`GModule(` *matrices*`, [`F`] )`
`GModule(` G`, [`F`] )`

`GModule` constructs a G-module record from a list *matrices* of matrices or from a matrix group G. The underlying field F may be specified as an optional argument; otherwise, it is taken to be the field generated by the entries of the given matrices.

The G-module record returned contains components `.field`, `.dimension`, `.generators` and `.isGModule`.

In using many of the functions described in this chapter, other components of the G-module record may be set, which describe the nature of the action of the group on the module. A description of these components is given in 68.34.

68.7 IsGModule

`IsGModule(` *module* `)`

If *module* is a record with component `.isGModule` set to `true`, `IsGModule` returns `true`, otherwise `false` .

68.8 IsIrreducible for GModules

`IsIrreducible(` *module* `)`

module is a G-module. `IsIrreducible` tests *module* for irreducibility, and returns `true` or `false`. If *module* is reducible, a sub- and quotient-module can be constructed using `InducedAction` (see 68.24).

The algorithm is described in [5].

68.9 IsAbsolutelyIrreducible

`IsAbsolutelyIrreducible(` *module* `)`

The G-module *module* is tested for absolute irreducibility, and `true` or `false` is returned. If the result is `false`, then the dimension e of the centralising field of *module* can be accessed by `DegreeFieldExtFlag(`*module*`)`. A matrix which centralises *module* (that is, it centralises the generating matrices `GeneratorsFlag(`*module*`)`) and which has minimal polynomial of degree e over the ground field can be accessed as `CentMatFlag(`*module*`)`. If such a matrix is required with multiplicative order $q^e - 1$, where q is the order of the ground field, then `FieldGenCentMat` (see 68.25) can be called.

The algorithm is described in [5].

68.10 IsSemiLinear

`IsSemiLinear(` G `)`

`IsSemiLinear` takes as input a matrix group G over a finite field and seeks to decide whether or not G acts semilinearly.

The function returns a list containing two values: a boolean and a G-module record, *module*, for G. If the boolean is `true`, then G is semilinear and information about the decomposition can be obtained using `SemiLinearPartFlag (`*module*`)`, `LinearPartFlag (`*module*`)`, and `FrobeniusAutomorphismsFlag (`*module*`)`. See 68.34 for an explanation of these.

If `IsSemiLinear` discovers that G acts imprimitively, it cannot decide whether or not G acts semilinearly and returns `"unknown"`.

`SmashGModule` is called by `IsSemiLinear`.

The algorithm is described in [6].

68.11 IsPrimitive for GModules

`IsPrimitive(` G `[,` *factorisations*`] )`

`IsPrimitive` takes as input a matrix group G over a finite field and seeks to decide whether or not G acts primitively. The function returns a list containing two values: a boolean and a G-module record, *module*, for G. If the boolean is `false`, then G is imprimitive and `BlockSystemFlag (`*module*`)` returns a block system (described in 68.32).

If `IsPrimitive` discovers that G acts semilinearly, then it cannot decide whether or not G acts primitively and returns `"unknown"`.

The second optional argument is a list of possible factorisations of d, the dimension of G. For each $[r, s]$ in this list where $rs = d$, the function seeks to decide whether G preserves a non-trivial system of imprimitivity having r blocks of size s.

`SmashGModule` is called by `IsPrimitive`.

The algorithm is described in [7].

68.12 IsTensor

`IsTensor(` G `[,` *factorisations*`] )`

`IsTensor` takes as input a matrix group G and seeks to decide whether or not G preserves a non-trivial tensor decomposition of the underlying vector space.

The implementation currently demands that G acts irreducibly, although this is not an inherent requirement of the algorithm.

The second optional argument is a list of possible factorisations of d, the dimension of G. For each $[r, s]$ in this list where $rs = d$, the function seeks to decide whether G preserves a non-trivial tensor decomposition of the underlying space as the tensor product of two spaces of dimensions r and s.

The function returns a list containing three values: a boolean, a G-module record, *module*, for G, and a change-of-basis matrix which exhibits the decomposition (if one is found). If the boolean is `false`, then G is not a tensor product. If the boolean is `true`, then G is a tensor product and the second argument in the list are the two tensor factors.

If `IsTensor` cannot decide whether G or not preserves a tensor decomposition, then it returns `"unknown"`. The second entry returned is now the list of unresolved tensor factorisations.

```
gap> ReadDataPkg ("matrix", "data", "a5xa5d25.gap");
```

```
gap> x:=IsTensor (G);;
gap> x[1];
true
gap> # Hence we have found a tensor decomposition.

gap> # Set up the two factors
gap> U := x[2][1];;
gap> W := x[2][2];;

gap> DisplayMat (GeneratorsFlag (U));
 4 1 5 2 4
 5 4 3 6 2
 2 2 4 5 6
 . 1 5 6 4
 5 2 6 3 .

 . 5 1 4 2
 1 4 4 5 .
 3 3 6 5 4
 6 5 6 3 3
 . 4 1 2 1

 3 1 3 2 6
 1 4 2 6 3
 . . 4 . .
 5 4 2 3 2
 4 1 6 4 4

 6 3 1 6 6
 6 3 5 1 4
 3 3 5 1 .
 2 6 2 1 2
 4 4 . 4 6

gap> ReadDataPkg ("matrix", "data", "a5d4.gap");

gap> x := IsTensor (G);
[ false, [  ], "undefined" ]
gap> # Hence not a tensor product
```

The algorithm is described in [8, 9]. Since a complete implementation requires basic tools which are not yet available in GAP3, the performance of this function is **currently seriously limited**.

`KroneckerFactors(` *g*`,` *d1*`,` *d2* `[,`*F*`] )`

`KroneckerFactors` decides whether or not a matrix *g* can be written as the Kronecker product of two matrices *A* and *B* of dimension *d1* and *d2*, respectively. If the field *F* is not supplied, it is taken to be `Field (Flat (`*g*`))`. The function returns either the pair [*A*, *B*] or `false`.

68.13 SmashGModule

`SmashGModule(` *module*`,` *S* `[,`*flag*`] )`

`SmashGModule` seeks to find a decomposition of a G-module with respect to a normal subgroup of G.

module is a module for a finite group G of matrices over a finite field and S is a set of matrices, generating a subgroup of G.

`SmashGModule` attempts to find some way of decomposing the module with respect to the normal subgroup $\langle S\rangle^G$. It returns `true` if some decomposition is found, `false` otherwise.

It first ensures that G acts absolutely irreducibly and that S contain at least one non-scalar matrix. If either of these conditions fails, then it returns `false`. The function returns `true` if it succeeds in verifying that either G acts imprimitively, or semilinearly, or preserves a tensor product, or preserves a symmetric tensor product (that is, permutes the tensor factors) or G normalises a group which is extraspecial or a 2-group of symplectic type. Each of these decompositions, if found, involves $\langle S\rangle^G$ in a natural way. Components are added to the record *module* which indicate the nature of a decomposition. Details of these components can be found in 68.34. If no decomposition is found, the function returns `false`. In general, the answer `false` indicates that there is no such decomposition with respect to $\langle S\rangle^G$. However, `SmashGModule` may fail to find a symmetric tensor product decomposition, since the detection of such a decomposition relies on the choice of random elements.

`SmashGModule` adds conjugates to S until a decomposition of the underlying vector space as a sum of irreducible $\langle S\rangle$-modules is found. The functions `SemiLinearDecomposition`, `TensorProductDecomposition`, `SymTensorProductDecomposition`, and `ExtraSpecialDecomposition` now search for decompositions.

At the end of the call to `SmashGModule`, S may be larger than at the start (but its normal closure has not changed).

The only permitted value for the optional parameter *flag* is the string `"PartialSmash"`. If `"PartialSmash"` is supplied, `SmashGModule` returns `false` as soon as it is clear that G is not the normaliser of a p-group nor does it preserve a symmetric tensor product decomposition with respect to $\langle S\rangle^G$.

The algorithm is described in [6].

68.14 HomGModule

`HomGModule(` *module1*`,` *module2* `)`

This function can only be run if `IsIrreducible(`*module1*`)` has returned `true`. *module1* and *module2* are assumed to be G-modules for the same group G, and a basis of the space of G-homomorphisms from *module1* to *module2* is calculated and returned. Each homomorphism in this list is given as a $d_1 \times d_2$ matrix, where d_1 and d_2 are the dimensions of *module1* and *module2*; the rows of the matrix are the images of the standard basis of *module1* in *module2* under the homomorphism.

68.15 IsomorphismGModule

`IsomorphismGModule(` *module1*`,` *module2* `)`

This function tests the G-modules *module1* and *module2* for isomorphism. Both G-modules must be defined over the same field with the same number of defining matrices, and at least one of them must be known to be irreducible (that is, `IsIrreducible(`*module*`)` has returned `true`). Otherwise the function will exit with an error. If they are not isomorphic, then `false` is returned. If they are isomorphic, then a $d \times d$ matrix is returned (where d is the dimension of the modules) whose rows give the images of the standard basis vectors of *module1* in an isomorphism to *module2*.

The algorithm is described in [5].

68.16 CompositionFactors

`CompositionFactors(` *module* `)`

module is a G-module. This function returns a list, each element of which is itself a 2-element list [*mod*, *int*], where *mod* is an irreducible composition factor of *module*, and *int* is the multiplicity of this factor in *module*. The elements *mod* correspond to non-isomorphic irreducible modules.

68.17 Examples

Example 1

```
gap> # First set up the natural permutation module for the
gap> # alternating group A5 over the field GF(2).
gap> P := Group ((1,2,3), (3,4,5));;
gap> M := PermGModule (P, GF(2));
rec(
  field := GF(2),
  dimension := 5,
  generators := [ [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ],
      [ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ] ],
  isGModule := true )
gap> # Now test for irreducibility, and calculate a proper submodule.
gap> IsIrreducible (M);
false
gap> SM := SubGModule (M, SubbasisFlag (M));;
gap> DimensionFlag (SM);
4
gap> DSM := DualGModule (SM);;
gap> # Test to see if SM is self-dual. We must prove irreducibility first.
gap> IsIrreducible (SM);
```

```
true
gap> IsAbsolutelyIrreducible (SM);
true
gap> IsomorphismGModule (SM, DSM);
[ [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> # This is an explicit isomorphism.
gap> # Now form a tensor product and decompose it into composition factors.
gap> TM := TensorProductGModule (SM, SM);;
gap> cf := CompositionFactors (TM);;
gap> Length (cf);
3
gap> DimensionFlag(cf[1][1]); cf[1][2];
1
4
gap> DimensionFlag(cf[2][1]); cf[2][2];
4
2
gap> DimensionFlag(cf[3][1]); cf[3][2];
4
1
gap> # This tells us that TM has three composition factors, of dimensions
gap> # 1, 4 and 4, with multiplicities 4, 2 and 1, respectively.
gap> # Is one of the 4-dimensional factors isomorphic to TM?
gap> IsomorphismGModule (SM, cf[2][1]);
false
gap> IsomorphismGModule (SM, cf[3][1]);
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> IsAbsolutelyIrreducible (cf[2][1]);
false
gap> DegreeFieldExtFlag(cf[2][1]);
2
gap> # If we extend the field of cf[2][1] to GF(4), it should
gap> # become reducible.
gap> MM := GModule (GeneratorsFlag (cf[2][1]), GF(4));;
gap> CF2 := CompositionFactors (MM);;
gap> Length (CF2);
2
gap> DimensionFlag (CF2[1][1]); CF2[1][2];
2
1
gap> DimensionFlag (CF2[2][1]); CF2[2][2];
2
```

```
1
gap> # It reduces into two non-isomorphic 2-dimensional factors.
```

In the next example, we investigate the structure of a matrix group using `SmashGModule` and access some of the stored information about the computed decomposition.

Example 2

```
gap> a := [
> [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]] * Z(2)^0;;
gap> b := [
> [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
> [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
> [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]] * Z(2)^0;;
gap> c := [
> [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]] * Z(2)^0;;
gap> gens := [a, b, c];;
gap> # Next we define the module.
gap> M := GModule (gens);;
```

```
gap> # So far only the basic components have been set.
gap> RecFields (M);
[ "field", "dimension", "generators"", "isGModule" ]
gap>
gap> # First we check for irreducibility and absolute irreducibility.
gap> IsIrreducible (M);
true
gap> IsAbsolutelyIrreducible (M);
true
gap> # A few more components have been set during these two function calls.
gap> RecFields(M);
[ "field", "dimension", "generators"", "isGModule", "algEl", "algElMat",
  "algElCharPol", "algElCharPolFac", "algElNullspaceVec",
  "algElNullspaceDim",
  "reducible", "degreeFieldExt", "absolutelyReducible" ]
gap> # The function Commutators forms the list of commutators of generators.
gap> S := Commutators(gens);;
gap> InfoSmash := Print;;
gap> # Setting InfoSmash to Print means that SmashGModule prints out
gap> # intermediate output to tell us what it is doing. If we
gap> # read this output it tells us what kind of decomposition SmashGModule
gap> # has found. Otherwise the output is only a true or false.
gap> # All the relevant information is contained in the components of M.
gap> SmashGModule (M, S);
Starting call to SmashGModule.
At top of main SmashGModule loop, S has 2 elements.
Translates of W are not modules.
At top of main SmashGModule loop, S has 3 elements.
Translates of W are not modules.
At top of main SmashGModule loop, S has 4 elements.
Translates of W are not modules.
At top of main SmashGModule loop, S has 5 elements.
Group embeds in GammaL(4, GF(2)^3).
SmashGModule returns true.
true
gap> # Additional components are set during the call to SmashGModule.
gap> RecFields(M);
[ "field", "dimension", "generators", "isGModule", "algEl", "algElMat",
  "algElCharPol", "algElCharPolFac", "algElNullspaceVec",
  "algElNullspaceDim",
  "reducible", "degreeFieldExt", "absolutelyReducible",
  "semiLinear", "linearPart",
  "centMat", "frobeniusAutomorphisms" ]
gap> SemiLinearFlag (M);
true
gap> # This flag tells us G that acts semilinearly.
gap> DegreeFieldExtFlag (M);
3
```

```
gap> # This flag tells us the relevant extension field is GF(2^3)
gap> Length (LinearPartFlag (M));
5
gap> #  LinearPartFlag (M) is a set of normal subgroup generators for the
gap> #  intersection of G with GL(4, GF(2^3)). It is also the contents of S
gap> #  at the end of the call to SmashGModule and is bigger than the set S
gap> #  which was input since conjugates have been added.
gap> FrobeniusAutomorphismsFlag (M);
[ 0, 0, 1 ]
gap> #  The first two generators of G act linearly, the last induces the field
gap> #  automorphism which maps x to x^2 (= x^(2^1)) on GF(2^3)
```

In our final example, we demonstrate how to test whether a matrix group is primitive and also how to select pseudo-random elements.

Example 3

```
 gap> #  Read in 18-dimensional representation of L(2, 17) over GF(41).
 gap> ReadDataPkg ("matrix", "data", "l217.gap");
 gap> #  Initialise a seed for random element generation.
 gap> InitPseudoRandom (G, 10, 100);;
 gap> #  Now select a pseudo-random element.
 gap> g := PseudoRandom (G);;
 gap> OrderMat (g);
 3
 gap> h := ElementOfOrder (G, 8, 10);;
 gap> OrderMat (h);
 8
 gap> # Is the group primitive?
 gap> R := IsPrimitive(G);;
 gap> # Examine the boolean returned.
 gap> R[1];
 false
 gap> M := R[2];;
 gap> # What is the block system found?
 gap> BlockSystemFlag (M);
 rec(
   nmrBlocks := 18,
   block :=
    [ [ 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
  0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
  0*Z(41), Z(41)^0, 0*Z(41), 0*Z(41) ] ],
   maps := [ 1, 2, 3 ],
   permGroup := Group( ( 1, 2)( 3, 7)( 5,11)( 6,12)( 8,10)(13,14)(15,17)
(16,18), ( 1, 3, 8,11,15, 9,13, 7,12,16, 6, 2, 5, 4,10,14,17),
( 1, 4, 2, 6, 3, 9, 7,12)( 5, 8,10,11,13,17,15,14) ),
   isBlockSystem := true )
 gap>  v :=
 > [ 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
 >   0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
```

```
>   0*Z(41), Z(41)^0, 0*Z(41), 0*Z(41) ];;
gap> # Illustrate use of MinBlocks
gap> B := MinBlocks (M, [v]);;
gap> B;
rec(
  nmrBlocks := 18,
  block :=
   [ [ 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41), 0*Z(41),
0*Z(41), Z(41)^0, 0*Z(41), 0*Z(41) ] ],
  maps := [ 1, 2, 3 ],
  permGroup := Group( ( 1, 2)( 3, 7)( 5,11)( 6,12)( 8,10)(13,14)(15,17)
(16,18), ( 1, 3, 8,11,15, 9,13, 7,12,16, 6, 2, 5, 4,10,14,17),
( 1, 4, 2, 6, 3, 9, 7,12)( 5, 8,10,11,13,17,15,14) ),
  isBlockSystem := true )
```

68.18 ClassicalForms

`ClassicalForms(` G `)`

Given as input, a classical, irreducible group G, `ClassicalForms` will try to find an invariant classical form for G (that is, an invariant symplectic or unitary bilinear form or an invariant symmetric bilinear form together with an invariant quadratic form, invariant modulo scalars in each case) or try to prove that no such form exists. The dimension of the underlying vector space must be at least 3.

The function may find a form even if G is a proper subgroup of a classical group; however, it is likely to fail for subgroups of ΓL. In these cases "unknown"(see below) is returned.

The results "linear", "symplectic", "unitary", "orthogonal..."and "absolutely reducible"are always correct, but "unknown"can either imply that the algorithm failed to find a form and it could not prove the linear case **or** that G is not a classical group.

["unknown"]
: it is not known if G fixes a form.

["unknown", "absolutely reducible"]
: G acts absolutely reducible on the underlying vector space. The function does not apply in this case.

["linear"]
: it is known that G does not fix a classical form modulo scalars.

["symplectic", *form*, *scalars*]
: G fixes a symplectic *form* modulo *scalars*. The form is only unique up to scalar multiplication. In characteristic two this also implies that no quadratic form is fixed.

["unitary", *form*, *scalars*]
: G fixes a unitary *form* modulo *scalars*. The form is only unique up to scalar multiplication.

["orthogonalcircle", *form*, *scalars*, *quadratic*, ...]
: ["orthogonalplus", *form*, *scalars*, *quadratic*, ...]
  ["orthogonalminus", *form*, *scalars*, *quadratic*, ...]

G fixes a orthogonal *form* with corresponding *quadratic* form modulo *scalars*. The forms are only unique up to scalar multiplication.

The function might return **more** than one list. However, in characteristic 2 it will **not** return "symplectic"if G fixes a quadratic form.

A bilinear form is returned as matrix F such that $g\mathbf{F}g^{tr}$ equals F modulo scalars for all elements g of G. A quadratic form is returned as upper triangular matrix Q such that $g\mathbf{Q}g^{tr}$ equals Q modulo scalars **after** $g\mathbf{Q}g^{tr}$ has been normalized into an upper triangular matrix. See the following example.

```
gap> G := O( 0, 9, 9 );
gap> f := ClassicalForms(G);;
gap> Q := f[1][4];;
gap> DisplayMat(Q);
 . 1 . . . . . . .
 . . . . . . . . .
 . . 1 . . . . . .
 . . . 1 . . . . .
 . . . . 1 . . . .
 . . . . . 1 . . .
 . . . . . . 1 . .
 . . . . . . . 1 .
 . . . . . . . . 1
gap> DisplayMat( G.1 * Q * TransposedMat(G.1) );
 . 1 . . . . . . .
 . . . . . . . . .
 . . 1 . . . . . .
 . . . 1 . . . . .
 . . . . 1 . . . .
 . . . . . 1 . . .
 . . . . . . 1 . .
 . . . . . . . 1 .
 . . . . . . . . 1
gap> DisplayMat( G.2 * Q * TransposedMat(G.2) );
 . . . . . . . . .
 1 . . . . . . . 1
 . . 1 . . . . . .
 . . . 1 . . . . .
 . . . . 1 . . . .
 . . . . . 1 . . .
 . . . . . . 1 . .
 . . . . . . . 1 .
 . 2 . . . . . . 1
```

Note that in general `g * Q * TransposedMat(g)` is not equal to `Q` for an element of an orthogonal group because you have to normalise the quadratic form such that it is an upper triangular matrix. In the above example for *G.1* you have to move the 1 in position $(9, 2)$ to position $(2, 9)$ adding it to the 2 which gives a 0, and you have to move the 2 in position $(1, 2)$ to position $(2, 1)$ thus obtaining the original quadratic form.

Examples

```
gap> G := SP( 4, 2 );
SP(4,2)
gap> ClassicalForms( G );
[ [ "symplectic",
  [ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ],
  [ Z(2)^0, Z(2)^0 ] ] ]
```

In this case G leaves a symplectic (and symmetric) form invariant but does not fix a quadratic form.

```
gap> G := O( -1, 4, 2 );
O(-1,4,2)
gap> ClassicalForms( G );
[ [ "orthogonalminus",
    [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ],
    [ Z(2)^0, Z(2)^0 ],
    [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ] ] ]
```

In this case G leaves a symplectic and symmetric form invariant and there exists also an invariant quadratic form.

```
gap> m1 :=
> [ [ Z(2^2), Z(2)^0, 0*Z(2), Z(2^2) ],
>   [ Z(2^2)^2, Z(2^2), Z(2^2)^2, Z(2^2) ],
>   [ 0*Z(2), Z(2^2)^2, Z(2^2)^2, Z(2)^0 ],
>   [ Z(2^2), Z(2^2)^2, Z(2^2), Z(2^2)^2 ] ];;
gap> m2 :=
> [ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2^2) ],
>   [ 0*Z(2), 0*Z(2), Z(2^2)^2, 0*Z(2) ],
>   [ 0*Z(2), Z(2^2)^2, 0*Z(2), Z(2^2) ],
>   [ Z(2^2), 0*Z(2), Z(2^2)^2, 0*Z(2) ] ];;
gap> G := Group( m1, m2 );;
gap> ClassicalForms( G );
[ [ "unknown" ],
  [ "symplectic",
    [ [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2^2)^2 ],
      [ Z(2)^0, 0*Z(2), Z(2^2), Z(2)^0 ],
      [ Z(2)^0, Z(2^2), 0*Z(2), Z(2)^0 ],
      [ Z(2^2)^2, Z(2)^0, Z(2)^0, 0*Z(2) ] ],
    [ Z(2)^0, Z(2)^0 ] ] ]
```

The "symplectic" indicates that an invariant symplectic form exists, the "unknown" indicates that an invariant "unitary" form might exist. Using the test once more, one gets:

```
gap> ClassicalForms( G );
[ [ "symplectic",
    [ [ 0*Z(2), Z(2^2)^2, Z(2^2)^2, Z(2^2) ],
      [ Z(2^2)^2, 0*Z(2), Z(2)^0, Z(2^2)^2 ],
      [ Z(2^2)^2, Z(2)^0, 0*Z(2), Z(2^2)^2 ],
      [ Z(2^2), Z(2^2)^2, Z(2^2)^2, 0*Z(2) ] ],
    [ Z(2)^0, Z(2)^0 ] ],
  [ "unitary",
    [ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ],
    [ Z(2)^0, Z(2)^0 ] ] ]
```

So G indeed fixes both a symplectic and unitary form but no quadratic form.

```
gap> ReadDataPkg ("matrix", "data", "a5d4.gap");
gap> ClassicalForms( G );
[ [ "unknown", "absolutely reducible" ] ]
```

G acts irreducibly, however `ClassicalForms` is not able to check if an invariant bilinear or quadratic form exists.

```
gap> ReadDataPkg ("matrix", "data", "a5d5.gap" );
gap> ClassicalForms( G );
[ [ "unknown" ] ]
gap> IsAbsolutelyIrreducible(GModule(G));
true
```

Although G fixes a symmetric form, `ClassicalForms` is not able to find an invariant form because G is not a classical group.

68.19 RecogniseClassical

`RecogniseClassical(` G `[,`*strategy*`] [,`*case*`] [,`N`] )`

`RecogniseClassical` takes as input a group G, which is a subgroup of $\mathrm{GL}(d, q)$ with $d > 1$, and seeks to decide whether or not G contains a classical group in its natural representation over a finite field.

strategy is one of the following:

"clg"

use the algorithm of Celler and Leedham-Green [3].

"np"

use the algorithm of Niemeyer and Praeger [11, 12].

The default strategy is "clg".

The parameter *case* is used to supply information about the specific non-degenerate bilinear, quadratic or sesquilinear forms on the underlying vector space V preserved by G modulo scalars. The value of *case* must be one of the following:

"all"

`RecogniseClassical` will try to determine the case of G. This is the default.

"linear"
: $G \leq \mathrm{GL}(d,q)$, and preserves no non-degenerate bilinear, quadratic or sesquilinear form on V. Set $\Omega := \mathrm{SL}(d,q)$.

"symplectic"
: $G \leq \mathrm{GSp}(d,q)$, with d even, and if q is also even we assume that G preserves no non-degenerate quadratic form on V. Set $\Omega := \mathrm{Sp}(d,q)$.

"orthogonalplus"
: $G \leq \mathrm{GO}^{+}(d,q)$ and d is even. Set $\Omega := \Omega^{+}(d,q)$.

"orthogonalminus"
: $G \leq \mathrm{GO}^{-}(d,q)$ and d is even. Set $\Omega := \Omega^{-}(d,q)$.

"orthogonalcircle"
: $G \leq \mathrm{GO}^{\circ}(d,q)$ and d is odd. Set $\Omega := \Omega^{\circ}(d,q)$.

"unitary"
: $G \leq \mathrm{GU}(d,q)$, where q is a square. Set $\Omega := \mathrm{SU}(d,q)$.

N is a positive integer which determines the number of random elements selected. Its default value depends on the strategy and case; see 68.22 and 68.23 for additional details.

In summary, the aim of `RecogniseClassical` is to test whether G contains the subgroup Ω corresponding to the value of *case*.

The function returns a record whose contents depends on the strategy chosen. Detailed information about components of this record can be found in 68.22 and 68.23. However, the record has certain **common** components **independent** of the strategy and these can be accessed using the following flag functions.

`ClassicalTypeFlag`
: returns "linear", "symplectic", "orthogonalplus", "orthogonalminus", "orthogonal-circle"or "unitary"if G is known to be a classical group of this type modulo scalars, otherwise "unknown". Note that $\mathrm{Sp}(2,q)$ is isomorphic to $\mathrm{SL}(2,q)$; "linear"not "symplectic"is returned in this case.

`IsSLContainedFlag`
: returns `true` if G contains the special linear group $\mathrm{SL}(d,q)$.

`IsSymplecticGroupFlag`
: returns `true` if G is contained in $\mathrm{GSp}(d,q)$ modulo scalars and contains $\mathrm{Sp}(d,q)$.

`IsOrthogonalGroupFlag`
: returns `true` if G is contained in an orthogonal group modulo scalars and contains the corresponding Ω.

`IsUnitaryGroupFlag`
: returns `true` if G is contained in an unitary group modulo scalars and contains the corresponding Ω.

These last four functions return `true`, `false`, or "unknown". Both `true` and `false` are **conclusive**. The answer "unknown"indicates either that the algorithm **failed** to determine whether or not G is a classical group or that the algorithm is not applicable to the supplied group; see 68.22 and 68.23 for additional details.

If `RecogniseClassical` **failed** to prove that G is a classical group, additional information about the possible Aschbacher categories of G might have been obtained. See 68.22 for details.

Example 1

```
gap> G := SL(7, 5);
SL(7,5)
gap> r := RecogniseClassical( G, "clg" );;
gap> ClassicalTypeFlag(r);
"linear"
gap> IsSLContainedFlag(r);
true
gap> r := RecogniseClassical( G, "np" );;
gap> ClassicalTypeFlag(r);
"linear"
gap> IsSLContainedFlag(r);
true
```

Example 2

```
gap> ReadDataPkg ("matrix", "data", "j1.gap" );
gap> DisplayMat(GeneratorsFlag(G));
  9  1  1  3  1  3  3
  1  1  3  1  3  3  9
  1  3  1  3  3  9  1
  3  1  3  3  9  1  1
  1  3  3  9  1  1  3
  3  3  9  1  1  3  1
  3  9  1  1  3  1  3

  .  1  .  .  .  .  .
  .  .  1  .  .  .  .
  .  .  . 10  .  .  .
  .  .  .  .  1  .  .
  .  .  .  .  . 10  .
  .  .  .  .  .  . 10
 10  .  .  .  .  .  .

gap> r := RecogniseClassical( G, "clg" );;
gap> ClassicalTypeFlag(r);
"unknown"
```

The algorithms are described in [3, 11, 12].

68.20 ConstructivelyRecogniseClassical

In this section, we describe functions developed by Celler and Leedham-Green (see [4] for details) to recognise constructively classical groups in their natural representation over finite fields.

`ConstructivelyRecogniseClassical(` G`, "linear")`

computes both a standard generating set for a matrix group G which contains the **special linear group** and expressions for the new generators in terms of G`.generators`. This generating set will allow you to write an element of G as a word in the **given** generating set of G.

The algorithm is of polynomial complexity in the dimension and field size. However, it is a **Las Vegas algorithm**, i.e. there is a chance that the algorithm fails to complete in the expected time. It will run **indefinitely** if G does not contain the special linear group.

The following functions can be applied to the record `sl` returned.

`SizeFlag(` *sl* `)`

returns the size of G.

`Rewrite(` *sl*`,` *elm* `)`

returns an expression such that `Value( Rewrite(` *sl*`,` *elm* `),` G`.generators )` is equal to the element *elm*.

Example

```
gap> m1 := [ [ 0*Z(17), Z(17), Z(17)^10, Z(17)^12, Z(17)^2 ],
 >  [ Z(17)^13, Z(17)^10, Z(17)^15, Z(17)^8, Z(17)^0 ],
 >  [ Z(17)^10, Z(17)^6, Z(17)^9, Z(17)^8, Z(17)^10 ],
 >  [ Z(17)^13, Z(17)^5, Z(17)^0, Z(17)^12, Z(17)^5 ],
 >  [ Z(17)^14, Z(17)^13, Z(17)^5, Z(17)^10, Z(17)^0 ] ];;
gap> m2 := [ [ 0*Z(17), Z(17)^10, Z(17)^2, 0*Z(17), Z(17)^10 ],
 >  [ 0*Z(17), Z(17)^6, Z(17)^0, Z(17)^4, Z(17)^15 ],
 >  [ Z(17)^7, Z(17)^6, Z(17)^10, Z(17), Z(17)^2 ],
 >  [ Z(17)^3, Z(17)^10, Z(17)^5, Z(17)^4, Z(17)^6 ],
 >  [ Z(17)^0, Z(17)^8, Z(17)^0, Z(17)^5, Z(17) ] ];;
gap> G := Group( m1, m2 );;
gap> sl := ConstructivelyRecogniseClassical( G, "linear" );;
gap> SizeFlag(sl);
338200968038818404584356577280
gap> w := Rewrite( sl, m1^m2 );;
gap> Value( w, [m1,m2] ) = m1^m2;
true
```

The algorithm is described in [4].

68.21 RecogniseMatrixGroup

`RecogniseMatrixGroup(` G `)`

`RecogniseMatrixGroup` attempts to recognise at least one of the Aschbacher categories in which the matrix group G lies. It then attempts to use features of this category to determine the order of G and provide a membership test for G.

The algorithm is described in [13]. This implementation is **experimental** and **limited** in its application; its inclusion in the package at this time is designed primarily to illustrate the basic features of the approach.

Currently the function attempts to recognise groups that are reducible, imprimitive, tensor products or classical in their natural representation.

The function returns a record whose components store detailed information about the decomposition of G that it finds. The record contents can be viewed using `DisplayMatRecord`.

The record consists of **layers** of records which are the kernels at the various stages of the computation. Individual layers are accessed via the component .kernel. We number these layers 1 to n where layer 0 is G. The n-th layer is a p-group generated by lower uni-triangular matrices. Information about this p-group is stored in the component .pGroup. At the i-th layer ($1 \leq i \leq n$) we have a group generated by matrices with at most $i-1$ identity blocks down the diagonal, followed by a non-singular block. Below the blocks we have non-zero entries and above them we have zero entries. Call this group G_i and the group generated by the non-singular block on the diagonal T_i. In the i-th layer we have a component .quotient. If the module for T_i is irreducible, then .quotient is T_i. If the module for T_i is reducible, then it decomposes into an irreducible submodule and a quotient module. In this case .quotient is the restriction of T_i to the submodule.

The central part of `RecogniseMatrixGroup` is the recursive function `GoDownChain` which takes as arguments a record and a list of matrices. `RecogniseMatrixGroup` initialises this record and then calls `GoDownChain` with the record and a list of the generators of G.

Assume we pass `GoDownChain` the i-th layer of our record and a list of matrices (possibly empty) in the form described above.

If the i-th layer is the last, then we construct a power-commutator presentation for the group generated by the list of matrices.

Otherwise, we now check if we have already decomposed T_i. If not, we split the module for T_i using `IsIrreducible`. We set .quotient to be the trivial group of dimension that of the irreducible submodule, and we store the basis-change matrix. We also initialise the next layer of our record, which will correspond to the kernel of the homomorphism from G_i to .quotient. Then we call `GoDownChain` with the layer and the list of matrices we started with.

If we have a decomposition for T_i, then we apply the basis-change stored in our record to the list of matrices and decide whether the new matrices preserve the decomposition. If they do not, then we discard the current decomposition of T_i and all the layers below the i-th, and recall `GoDownChain`.

If the matrices preserve the decomposition, then we extract the blocks in the matrices which correspond to `.quotient`. We decide if these blocks lie in `.quotient`.

If the blocks lie in `.quotient`, then the next step is to construct relations on `.quotient` which we will then evaluate on the generators of G_i to put into the next layer. There are two approaches to constructing relations on `.quotient`. Let F be the free group on the number of generators of `.quotient`. We construct a permutation representation on `.quotient`. The first approach is to take the image of an element of `.quotient` in the permutation group and then pull it back to the permutation group. The second approach is to take a random word in F, map it into the permutation group and then pull the permutation back into F. The relations from approach one are "generator relations"and those from approach two are "random relations". If `.quotient` contains SL, then we use special techniques.

If the list of matrices with which we called `GoDownChain` is empty, then we construct random relations on `.quotient`, evaluate these in G_i to get a new list of matrices and then call `GoDownChain` with this list and the next layer of our record. We use parameters similar to those in the Random Schreier-Sims algorithm to control how hard we work.

If the list of matrices is non-empty, then we take generator relations on the list of blocks and evaluate these in G_i. This gives us a new list of matrices and we call `GoDownChain` with the list and the next layer of our record.

If, in evaluating the relations in G_i, we get a non-identity block, then we deduce that our permutation representation is not faithful. In this case, the next layer corresponds to the kernel of the action that provided the representation.

If these blocks do not lie in `.quotient`, then we have to enlarge it. We then try to find out the Aschbacher category in which `.quotient` lies, and its size. After applying these tests and computing the size we then construct generator relations on the list of generators of `.quotient` and put them into the kernel. We then call `GoDownChain` with our record and an empty list of matrices.

We first test whether `.quotient` is a classical group in its natural representation using `RecogniseClassicalNP`. If `.quotient` contains SL, we use `ConstructivelyRecognise-Classical` to obtain both its size and a membership test; if `.quotient` contains one of the other classical groups, we simply report this. If `.quotient` contains a classical group, we terminate the testing. If `RecogniseClassicalNP` returns `false`, then we call `RecogniseClassicalCLG`. If `.quotient` contains one of the classical groups, then we behave as before. If `.quotient` is not a classical group, then we obtain a list of possibilities for `.quotient`. This list may help to rule out certain Aschbacher categories and will give pointers to the ones which we should investigate further.

If `.quotient` might be imprimitive, then we test this using `IsPrimitive`. If `.quotient` is imprimitive, then we obtain a permutation representation for the action on the blocks and we store this in `.quotient`. We set the size of `.quotient` to be the size of the permutation group. If the action is not faithful, then we compute the kernel of the action at the next layer and then we have the correct size for `.quotient`. If `.quotient` is imprimitive, then the testing ends here. If `IsPrimitive` returns `unknown` or `true`, then we store this in `.quotient`. We then reprocess `.quotient` using `RecogniseClassicalCLG`.

If `.quotient` might be a tensor product, then we test this using `IsTensor`. If `.quotient` is a tensor product, then we store the tensor factors in `.quotient`. Currently, we do not exploit this conclusion . If `IsTensor` returns `unknown` or `false` then we store this in `.quotient`. We then reprocess `.quotient` using `RecogniseClassicalCLG`.

By default, we obtain the size of `.quotient` using `PermGroupRepresentation`. We advise the user if the list returned by `RecogniseClassicalCLG` suggests that the group contains an almost simple group or an alternating group. `PermGroupRepresentation` constructs a faithful permutation representation for `.quotient` and we store this in `.quotient`.

We illustrate some of these features in the following example. Additional examples can be found in `matrix/reduce/examples.tex`.

```
gap> # Construct the group SL(2, 3) x SP(4, 3)
gap> G1 := SL(2, 3);;
gap> G2 := SP(4, 3);;
gap> m1 := DiagonalMat_mtx( GF(3), G1.1, G2.1 );;
gap> m2 := DiagonalMat_mtx( GF(3), G1.2, G2.2 );;
gap> # Put something in the bottom left hand corner to give us a p-group
gap> m1[3][1] := Z(3)^0;;
gap> m2[5][2] := Z(3);;
gap> G := Group( [m1, m2], m1^0 );;
gap> # Apply RecogniseMatrixGroup to G
gap> x := RecogniseMatrixGroup( G );;
```

```
#I  Input group has dimension 6 over GF(3)
#I  Layer number 1: Type = "Unknown"
#I  Size = 1, # of matrices = 2
#I  Computing the next quotient
#I  <new> acts non-trivially on the block of dim 6

#I  Found a quotient of dim 2
#I  Restarting after finding a decomposition
#I  Layer number 1: Type = "Perm"
#I  Size = 1, # of matrices = 2
#I  Submodule is invariant under <new>
#I  Enlarging quotient, old size = 1

#I  Is quotient classical?
#I  Dimension of group is <= 2, you must supply form
#I  The quotient contains SL
#I  New size = 24
#I  Adding generator relations to the kernel
#I  Layer number 2: Type = "Unknown"
#I  Size = 1, # of matrices = 2
#I  Computing the next quotient
#I  <new> acts non-trivially on the block of dim 4

#I  Found a quotient of dim 4
#I  Restarting after finding a decomposition
#I  Layer number 2: Type = "Perm"
#I  Size = 1, # of matrices = 2
#I  Submodule is invariant under <new>
#I  Enlarging quotient, old size = 1

#I  Is quotient classical?
#I  The case is symplectic
#I  This algorithm does not apply in this case.
#I  The quotient contains SP
#W  Applying Size to (matrix group) quotient
#I  New size = 51840
#I  Adding generator relations to the kernel
#I  Restarting after enlarging the quotient
#I  Layer number 2: Type = "Perm"
#I  Size = 51840, # of matrices = 0
#I  Using a permutation representation
#I  Adding random relations at layer number 2
#I  Adding a random relation at layer number 2
#I  Layer number 3: Type = "PGroup"
#I  Size = 1, # of matrices = 3
#I  Reached the p-group case
#I  New size = 27
#I  Adding a random relation at layer number 2
```

```
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 27
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 2
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Kernel is finished, size = 340122240
#I  Restarting after enlarging the quotient
#I  Layer number 1: Type = "SL"
#I  Size = 8162933760, # of matrices = 0
#I  Using the SL recognition
#I  Adding random relations at layer number 1
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
```

```
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
```

```
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Adding a random relation at layer number 1
#I  Layer number 2: Type = "Perm"
#I  Size = 340122240, # of matrices = 3
#I  Submodule is invariant under <new>
#I  Using a permutation representation
#I  Adding generator relations to the kernel
#I  Kernel p-group, old size = 6561
#I  Kernel p-group, new size = 6561
#I  Kernel is finished, size = 8162933760
gap> # Let us look at what we have found
gap> DisplayMatRecord( x );
#I  Matrix group over field GF(3) of dimension 6 has size 8162933760
#I  Number of layers is 3
gap> DisplayMatRecord( x, 1 );
#I  Layer Number = 1
#I  Type = SL
#I  Dimension = 2
#I  Size = 24
gap> # The module for G splits into an irreducible submodule of dimension
gap> # 2 and a quotient module of dimension 4. The restriction of G to
gap> # the submodule contains SL(2, 3). Call this group G1.
gap> DisplayMatRecord( x, 2 );
#I  Layer Number = 2
#I  Type = Perm
#I  Dimension = 4
#I  Size = 51840
gap> # We have now taken relations on G1 and evaluated them in G to get
gap> # a group H, which is the kernel of the homomorphism from G to G1.
gap> # The group generated by the last 4x4 block on the diagonal of the
gap> # matrices of H has an irreducible module and we have computed
gap> # a permutation representation on it. Call this group H1.
gap> DisplayMatRecord( x, 3 );
#I  Layer Number = 3
```

```
#I  Type = PGroup
#I  Dimension = 6
#I  Size = 6561
gap> # We have now taken relations on H1 and evaluated them in H to get the
gap> # kernel of the homomorphism from H to H1. This kernel consists of
gap> # lower uni-triangular matrices. It is a p-group of size 6561.
```

68.22 RecogniseClassicalCLG

In this section, we describe functions developed by Celler and Leedham-Green (see [3] for details) to recognise classical groups in their natural representation over finite fields.

`RecogniseClassicalCLG(` *G* `[,`*case*`] [,`*N*`] )`

This is the top-level function, taking as input a group G, which is a subgroup of $\mathrm{GL}(d, q)$ with $d > 1$. The other optional arguments have the same meaning as those supplied to `RecogniseClassical`. The default value of N, the number of random elements to consider, depends on the case; it is 40 for small fields and dimensions, but decreases to 10 for larger dimensions.

Constraints

In the case of an orthogonal group, the dimension of the underlying vector space must be at least 7, since there are exceptional isomorphisms between the orthogonal groups in dimensions 6 or less and other classical groups which are not dealt with in `RecogniseClassical-CLG`. In dimension 8, `RecognizeSO` will **not** rule out the possibility of $O_7(q)$ embedded as irreducible subgroup of $O_8^+(q)$. Since G must also act irreducibly, `RecogniseClassicalCLG` does **not** recognise $O_{2n+1}^0(2^k)$.

The record returned by this function is similar to that described in 68.19. In particular, the flag functions described there and below can be applied to the record. You should ignore undocumented record components.

Additional information

`DualFormFlag`
: if G has been proved to be a symplectic or orthogonal group, `DualFormFlag` returns the symplectic or orthogonal form.

`QuadraticFormFlag`
: if G has been proved to be an orthogonal group, `QuadraticFormFlag` returns the quadratic form.

`UnitaryFormFlag`
: if G has been proved to be a unitary group, `DualFormFlag` returns the symplectic or orthogonal form.

If `RecogniseClassical` **failed** to prove that G is a classical group, additional information about the possible Aschbacher categories of G might have been obtained.

In particular, the following flag functions may be applied to the record. If one of these functions returns a list, it has the following meaning: **if** G belongs to the corresponding Aschbacher category, **then** G is determined by one of the possibilities returned; it does **not** imply that G is a member of this category. However, an empty list indicates that G does not belong to this category. Each of these functions may also return "unknown".

A group G is **almost simple** if G contains a non-abelian simple group T and is contained in the automorphism group of T. If G is almost simple, then G is either an almost sporadic group, an almost alternating group, or an almost Chevalley group.

`PossibleAlmostSimpleFlag`
: if G is not a classical group, this function returns a list of possible almost sporadic groups modulo scalars. This function deals only with **sporadic** groups T. The names of the corresponding non-abelian simple groups are returned. Possible names are: "M11", "M12", "M22", "M23", "M24", "J2", "Suz", "HS", "McL", "Co3", "Co2", "Co1", "He", "Fi22", "Fi23", "F3+", "HN", "Th", "B", "M", "J1", "ON", "J3", "Ly", "Ru", "J4".

`PossibleAlternatingGroupsFlag`
: if G is not a classical group, this function returns a list of possible almost alternating groups modulo scalars. This list contains the possible degrees as integers.

`PossibleChevalleyGroupsFlag`
: if G is not a classical group, this function returns a list of possible almost Chevalley groups modulo scalars. The various Chevalley groups are described by tuples $[type, rank, p, k]$, where *type* is a string giving the type (e.g. "2A", see [15, p. 170] for details), $rank$ is the rank of the Chevalley group, and p^k is the size of the underlying field.

`IsPossibleImprimitiveFlag`
: returns `true` if G might be imprimitive.

`PossibleImprimitiveDimensionsFlag`
: returns the possible block dimensions (`IsPossibleImprimitiveFlag` must be `true`).

`IsPossibleTensorProductFlag`
: returns `true` if G might be a tensor product.

`PossibleTensorDimensionsFlag`
: returns the possible tensor product dimensions; note that this entry is only valid if `IsPossibleTensorProductFlag` is `true` **or** `IsPossibleTensorPowerFlag` is true **and** the dimension is a square.

`IsPossibleTensorPowerFlag`
: returns `true` if G might be a tensor power.

`IsPossibleSmallerFieldFlag`
: retuns `true` if G could be defined (modulo scalars) over a **smaller** field.

`PossibleSmallerFieldFlag`
: returns the the least possible field (`IsPossibleSmallerFieldFlag` must be `true`).

`IsPossibleSemiLinearFlag`
: the natural module could be isomorphic to a module of smaller dimension over a **larger** field on which this extension field acts semi-linearly.

`IsPossibleNormalizerPGroupFlag`
: the dimension of the underlying vector space must be r^m for some prime r and G could be an extension of a r-group of symplectic type and exponent $r \cdot \gcd(2, r)$ by a subgroup of $Sp(m, r)$, modulo scalars. A r-group is of **symplectic type** if every characteristic abelian subgroup is cyclic.

Examples

```
gap> m1 :=
> [ [ 0*Z(17), Z(17), Z(17)^10, Z(17)^12, Z(17)^2 ],
>   [ Z(17)^13, Z(17)^10, Z(17)^15, Z(17)^8, Z(17)^0 ],
>   [ Z(17)^10, Z(17)^6, Z(17)^9, Z(17)^8, Z(17)^10 ],
>   [ Z(17)^13, Z(17)^5, Z(17)^0, Z(17)^12, Z(17)^5 ],
>   [ Z(17)^14, Z(17)^13, Z(17)^5, Z(17)^10, Z(17)^0 ] ];;
gap> m2 :=
> [ [ 0*Z(17), Z(17)^10, Z(17)^2, 0*Z(17), Z(17)^10 ],
>   [ 0*Z(17), Z(17)^6, Z(17)^0, Z(17)^4, Z(17)^15 ],
>   [ Z(17)^7, Z(17)^6, Z(17)^10, Z(17), Z(17)^2 ],
>   [ Z(17)^3, Z(17)^10, Z(17)^5, Z(17)^4, Z(17)^6 ],
>   [ Z(17)^0, Z(17)^8, Z(17)^0, Z(17)^5, Z(17) ] ];;
gap> G := Group( m1, m2 );;
gap> sl := RecogniseClassicalCLG( G, "all", 1 );;
gap> IsSLContainedFlag(sl);
"unknown"
```

Since the algorithm has a random component, it may fail to prove that a group contains the special linear group even if the group does. As a reminder, `IsSLContainedFlag` may return `true`, `false`, or `"unknown"`.

Here we chose only one random element. If `RecogniseClassicalCLG` fails but you suspect that the group contains the special linear group, you can restart it using more random elements. You should, however, **not** change the *case*. If you don't already know the case, then call `RecogniseClassicalCLG` either without a case parameter or "all".

```
gap> sl := RecogniseClassicalCLG( G, 5 );;
gap> IsSLContainedFlag(sl);
true
```

The following is an example where G is not an classical group but additional information has been obtained.

```
gap> ReadDataPkg ("matrix", "data", "j1.gap" );
gap> DisplayMat(GeneratorsFlag(G));
  9  1  1  3  1  3  3
  1  1  3  1  3  3  9
  1  3  1  3  3  9  1
  3  1  3  3  9  1  1
  1  3  3  9  1  1  3
  3  3  9  1  1  3  1
  3  9  1  1  3  1  3

  .  1  .  .  .  .  .
  .  .  1  .  .  .  .
  .  .  . 10  .  .  .
  .  .  .  .  1  .  .
  .  .  .  .  . 10  .
  .  .  .  .  .  . 10
 10  .  .  .  .  .  .
```

```
gap> r := RecogniseClassical( G, "clg" );;
gap> ClassicalTypeFlag(r);
"unknown"
gap> IsPossibleImprimitiveFlag(r);
false
gap> IsPossibleTensorProductFlag(r);
false
gap> IsPossibleTensorPowerFlag(r);
false
gap> PossibleAlmostSimpleFlag(r);
[ "J1" ]
gap> PossibleAlternatingGroupsFlag(r);
[  ]
gap> PossibleChevalleyGroupsFlag(r);
[ [ "A", 1, 11, 3 ], [ "A", 2, 11, 2 ], [ "A", 3, 11, 1 ],
  [ "G", 2, 11, 1 ] ]
```

68.23 RecogniseClassicalNP

In this section, we describe functions developed by Niemeyer and Praeger (see [11, 12] for details) to recognise classical groups in their natural representation over finite fields.

`RecogniseClassicalNP(` G `[,`*case*`] [,`N`] )`

This is the top-level function taking as input a group G, which is a subgroup of $\mathrm{GL}(d,q)$ with $d > 2$. The other optional arguments have the same meaning as those supplied to `RecogniseClassical`.

The aim of `RecogniseClassicalNP` is to test whether G contains the subgroup Ω corresponding to the value of *case*. The algorithm employed is Monte-Carlo based on random selections of elements from G. `RecogniseClassicalNP` returns either `true` or `false` or `"does not apply"`. If it returns `true` and G satisfies the constraints listed for *case* (see `RecogniseClassical`) then we know with certainty that G contains the corresponding classical subgroup Ω. It is not checked whether G satisfies all these conditions. If it returns `"does not apply"` then either the theoretical background of this algorithm does not allow us to decide whether or not G contains Ω (because the parameter values are too small) or G is not a group of type *case*. If it returns `false` then there is still a possibility that G contains Ω. The probability that G contains Ω and `RecogniseClassicalNP` returns `false` can be controlled by the parameter N, which is the number of elements selected from G. The larger N is, the smaller this probability becomes. If N is not passed as an argument, the default value for N is 15 if *case* is `"linear"` and 25 otherwise. These values were experimentally determined over a large number of trials. But if d has several distinct prime divisors, larger values of N may be required (see [12]).

The complexity of the function for small fields ($q < 2^{16}$) and for a given value of N is $O(d^3 \log\log d)$ bit operations.

Assume `InfoRecog1` is set to `Print`; if `RecogniseClassicalNP` returns `true`, it prints

```
"Proved that the group contains  a classical group of type <case>
in  <n> selections\",
```

where n is the actual number of elements used; if `RecogniseClassicalNP` returns `false`, it prints `"The group probably does not contain a classical group"` and possibly also a statement suggesting what the group might be.

If *case* is not supplied, then `ClassicalForms` seeks to determine which form is preserved. If `ClassicalForms` fails to find a form, then `RecogniseClassicalNP` returns `false`.

Details of the computation, including the identification of the classical group type, are stored in the component `G.recognise`. Its contents can be accessed using the following flag functions.

`ClassicalTypeFlag`
: returns one of `"linear"`, `"symplectic"`, `"orthogonalplus"`, `"orthogonalminus"`, `"orthogonalcircle"` or `"unitary"` if G is known to be a classical group of this type modulo scalars, otherwise `"unknown"`.

`IsSLContainedFlag`
: returns `true` if G contains the special linear group $\mathrm{SL}(d, q)$.

`IsSymplecticGroupFlag`
: returns `true` if G is contained in $\mathrm{GSp}(d, q)$ modulo scalars and contains $\mathrm{Sp}(d, q)$.

`IsOrthogonalGroupFlag`
: returns `true` if G is contained in an orthogonal group modulo scalars and contains the corresponding Ω.

`IsUnitaryGroupFlag`
: returns `true` if G is contained in an unitary group modulo scalars and contains the corresponding Ω.

These last four functions return `true`, `false`, or "unknown". Both `true` and `false` are **conclusive**. The answer "unknown"indicates either that the algorithm **failed** to determine whether or not G is a classical group or that the algorithm is not applicable to the supplied group.

If `RecogniseClassicalNP` returns `true`, then `G.recognise` contains all the information that proves that G contains the classical group having type `G.recognise.type`. The record components `d`, `p`, `a` and `q` identify G as a subgroup of $GL(d, q)$, where $q = p^a$. For each e in `G.recognise.E` the group G contains a $\mathrm{ppd}(d, q; e)$-element (see `IsPpdElement`) and for each e in `G.recognise.LE` it contains a large $\mathrm{ppd}(d, q; e)$-element. Further, it contains a basic $\mathrm{ppd}(d, q; e)$-element if e is in `G.recognise.basic`. Finally, the component `G.recognise.isReducible` is `false`, indicating that G is now known to act irreducibly.

If `RecogniseClassicalNP` returns `"does not apply"`, then G has no record `G.recognise`.

If `RecogniseClassicalNP` returns `false`, then `G.recognise` gives some indication as to why the algorithm failed to prove that G contains a classical group. Either G could not be shown to be generic and `G.recognise.isGeneric` is `false` and `G.recognise.E`, `G.recognise.LE` and `G.recognise.basic` will indicate which kinds of ppd-elements could not be found; or `G.recognise.isGeneric` is `true` and the algorithm failed to rule out that G preserves an extension field structure and `G.recognise.possibleOverLargerField` is `true`; or `G.isGeneric` is `true` and `G.possibleOverLargerField` is `false` and the possibility that G is nearly simple could not be ruled out and `G.recognise.possibleNearlySimple` contains a list of names of possible nearly simple groups; or `ClassicalForms` failed to find a form and `G.recognise.noFormFound` is `true`; or finally `G.isGeneric` is `true` and

`G.possibleOverLargerField` is `false` and `G.possibleNearlySimple` is empty and G was found to act reducibly and `G.recognise.isReducible` is `true`.

If `RecogniseClassicalNP` returns `false`, then a recall to `RecogniseClassicalNP` for the given group uses the previously computed facts about the group stored in `G.recognise`.

```
gap> RecogniseClassicalNP( GL(10,5), "linear", 10 );
true
gap> RecogniseClassicalNP( SP(6,2), "symplectic", 10 );
# I This algorithm does not apply in this case
"does not apply"
gap> G := SL(20, 5);;
gap> RecogniseClassicalNP( G );
 true
gap> G.recognise;
 rec(
 d := 20,
 p := 5,
 a := 1,
 q := 5,
 E := [ 11, 12, 16, 18 ],
 LE := [ 11, 12, 16, 18 ],
 basic := 12,
 isReducible := false,
 isGeneric := true,
 type := "linear" )
gap> InfoRecog1 := Print;; InfoRecog2 := Print;;
gap> G := GeneralUnitaryMatGroup(7,2);;
gap> RecogniseClassicalNP( G );
# I The case is unitary
# I G acts irreducibly, block criteria failed
# I The group is generic in 4 selections
# I The group is not an extension field group
# I The group does not preserve an extension field
# I The group is not nearly simple
# I The group acts irreducibly
# I Proved that group contains classical group of type unitary
# I in 6 random selections.
true
gap > G.recognise;
rec(
d := 7,
p := 2,
a := 2,
q := 4,
E := [ 5, 7 ],
LE := [ 5, 7 ],
basic := 7,
isReducible := false,
```

```
isGeneric := true,
type := "unitary" )
```

```
gap> InfoRecog1 := Print;; InfoRecog2 := Print;;
gap> G := Group (
> [[0,1,0,0],
>  [1,1,0,0],
>  [0,0,0,1],
>  [0,0,1,1]] * GF(2).one,
> [[0,0,1,0],
>  [0,1,1,0],
>  [1,0,1,1],
>  [0,1,1,1]] * GF(2).one );
gap> RecogniseClassicalNP (G);
# I The case is linear
# I G acts irreducibly, block criteria failed
# I The group is generic in 8 selections
# I The group is not an extension field group
# I The group does not preserve an extension field
# I G' might be A_7;
# I G' is not a Mathieu group;
# I G' is not PSL(2,r)
# I The group could be a nearly simple group.
false
gap> G.recognise;
rec(
d := 4,
p := 2,
a := 1,
q := 2,
E := [ 3, 4 ],
LE := [ 3 ],
basic := 4,
isReducible := false,
isGeneric := true,
possibleNearlySimple := [ "A7" ],
dimsReducible := [ 0, 4 ],
possibleOverLargerField := false )
```

We now describe some of the lower-level functions used.

`GenericParameters(` G, *case* `)`

This function takes as input a subgroup G of $\mathrm{GL}(d, q)$ and a string *case*, one of the list given under `RecogniseClassicalGroup`. The group G has generic parameters if the subgroup Ω of $\mathrm{GL}(d, q)$ contains two different ppd-elements (see `IsPpdElement`), that is a $\mathrm{ppd}(d, q; e_1)$-element and a $\mathrm{ppd}(d, q; e_2)$-element for $d/2 < e_1 < e_2 \leq d$ such that at least one of them is a large ppd-element and one is a basic ppd-element. The function `GenericParameters` returns `true` if G has generic parameters. In this case `RecogniseClassicalNP` can be applied to G and *case*. If G does not have generic parameters, the function returns `false`.

```
gap> GenericParameters( SP(6,2), "symplectic" );
false
gap> GenericParameters( SP(12,2), "symplectic" );
true
```

[Comment: In the near future we propose to extend and modify our algorithm to deal with cases where the group Ω does not contain sufficient ppd-elements.]

`IsGeneric(` G`,` N `)`

This function takes as input a subgroup G of $\mathrm{GL}(d,q)$ and an integer N. The group G is generic if it is irreducible and contains two different ppd-elements (see `IsPpdElement`), that is a $\mathrm{ppd}(d,q;e_1)$-element and a $\mathrm{ppd}(d,q;e_2)$-element for $d/2 < e_1 < e_2 \leq d$ such that at least one of them is a large ppd-element and one is a basic ppd-element. It chooses up to N elements in G and increases G`.recognise.n` by the number of random selections it made. If among these it finds the two required different ppd-elements, which is established by examining the sets G`.recognise.E`, G`.recognise.LE`, and G`.recognise.basic`, then it sets G`.recognise.isGeneric` to `true` and returns `true`. If after N random selections it fails to find two different ppd-elements, the function returns `false`. It is not tested whether G actually is irreducible.

```
gap> IsGeneric( SP(12,2), 20 );
 true
```

`IsExtensionField(` G`,` *case*`,` N `)`

This function takes as input a subgroup G of $\mathrm{GL}(d,q)$, a string *case*, one of the list given under `RecogniseClassicalGroup`, and an integer N. It assumes, but does not test that G is generic (see `IsGeneric`). We say that the group G can be defined over an extension field if there is a prime b dividing d such that G is conjugate to a subgroup of $Z.\mathrm{GL}(d/b,q^b).b$, where Z is the group of scalar matrices in $\mathrm{GL}(d,q)$. Then `IsExtensionField` returns m if certain elements are found in m random selections which together prove that G cannot be a subgroup of an extension field group. In this case G`.recognise.possibleOverLargerField` is set to `false`. If, after N random selections of elements from G, this could not be established, then `IsExtensionField` returns `true`. Hence, if it returns `true`, then either G is an extension field group or one needs to select more elements of G to establish that this is not the case. The component G`.recognise.possibleOverLargerField` is set to `true`.

```
gap> IsExtensionField( GL(12,2), "linear", 30 );
 8
```

`IsGenericNearlySimple(` G`,` *case*`,` N `)`

The subgroup G of $\mathrm{GL}(d,q)$ is said to be **nearly simple** if there is some nonabelian simple group S such that $S \leq G/(G \cap Z) \leq \mathrm{Aut}(S)$, where Z denotes the subgroup of nonsingular scalar matrices of $\mathrm{GL}(d,q)$. This function takes as input a subgroup G of $\mathrm{GL}(d,q)$, a string *case*, one of the list given under `RecogniseClassicalGroup`, and an integer N. It assumes but does not test that G is irreducible on the underlying vector space, is generic (see `IsGeneric`), and is not contained in an extension field group (see `IsExtensionField`). This means that G is known to contain two different ppd-elements (see `IsPpdElement`), that is a $\mathrm{ppd}(d,q;e_1)$-element and a $\mathrm{ppd}(d,q;e_2)$-element for $d/2 < e_1 < e_2 \leq d$ such that at least one of them is a large ppd-element and one is a basic ppd-element, and the values e_1 and e_2 for the elements are stored in G`.recognise.E`. At this stage, the theoretical analysis in [12] tells us that either G contains Ω, or the string *case* is `"linear"` and G is an absolutely irreducible generic nearly simple group. All possibilities for the latter groups are known explicitly, and `IsGenericNearlySimple` tries to establish that G is not one of these groups. Thus it first checks that *case* is `"linear"`, and if so performs further tests.

`IsGenericNearlySimple` returns `false` if certain elements are found which together prove that G cannot be a generic nearly simple group. If, after N random selections of elements from G, this could not be shown, then `IsGenericNearlySimple` returns `true` and G might be a generic nearly simple group. It increases G`.recognise.n` by the number of elements selected. In this case either G is nearly simple or there is a small chance that the output `true` is incorrect. In fact the probability with which the algorithm will return the statement `true` when G is not nearly simple can be made arbitrarily small depending on the number N of random selections performed. The list of irreducible generic nearly simple groups is very short. The name of each nearly simple group which might be isomorphic to G is stored as a string in G`.recognise.possibleNearlySimple`. If `InfoRecog2` is set to `Print`, then in the case that G is such a group `IsGeneric` will print the isomorphism type of the nearly simple group.

```
gap> IsGenericNearlySimple( GL(12,2), "symplectic", 30 );
 11
```

68.24 InducedAction

`InducedAction(` *module*`,` *basis* `)`
`SubGModule(` *module*`,` *basis* `)`
`QuotientGModule(` *module*`,` *basis* `)`

These functions take a G-module *module* as input, together with a basis *basis* for a proper submodule, which is assumed to be normalised, in the weak sense that the first non-zero component of each vector in the basis is 1, and no two vectors in the basis have their first nonzero components in the same position. The basis is given as an $r \times n$ matrix, where r is the length of the basis.

Normally, one runs `IsIrreducible(`*module*`)` first, and (assuming it returns `false`) then runs these functions using `SubbasisFlag(`*module*`)` as the second argument. `InducedAction` returns a 4-element list containing the submodule, the quotient module, the original matrices rewritten with respect to a basis in which a basis for the submodule comes first, and the change-of-basis matrix; `SubGModule` returns the submodule having *basis* as basis; `QuotientGModule` the corresponding quotient module.

`RandomIrreducibleSubGModule(` *module* `)`

Find a basis for an **irreducible** submodule of *module*.

68.25 FieldGenCentMat

`FieldGenCentMat(` *module* `)`

This function should only be applied if the function `IsIrreducible(`*module*`)` has returned `true`, and if `IsAbsolutelyIrreducible(`*module*`)` has returned `false`. A matrix which centralises the G-module *module* and which has multiplicative order $q^e - 1$, where q is the order of the ground field and e is the dimension of the centralising field of *module*, is calculated and stored. It can be accessed as `CentMatFlag(`*module*`)`.

68.26 MinimalSubGModules

`MinimalSubGModules(` *module1* `,` *module2* `[,` *max* `] )`

This function should only be applied if `IsIrreducible(`*module1*`)` has returned `true`. *module1* and *module2* are assumed to be G-modules for the same group G. `MinimalSubGModules` computes and returns a list of the normalised bases of all of the minimal submodules of *module2* that are isomorphic to *module1*. (These can then be constructed as G-modules, if required, by calling `SubGModule(`*module2*`,` *basis*`)` where *basis* is one of these bases.) The optional parameter *max* should be a positive integer. If the number of submodules exceeds *max*, then the procedure is aborted.

68.27 SpinBasis

`SpinBasis(` *vector* `,` *matrices* `)`

The input is a vector, *vector*, and a list of $n \times n$ matrices, *matrices*, where n is the length of the vector. A normalised basis of the submodule generated by the action of the matrices (acting on the right) on the vector is calculated and returned. It is returned as an $r \times n$ matrix, where r is the dimension of this submodule.

`SpinBasis` is called by `IsIrreducible`.

68.28 SemiLinearDecomposition

`SemiLinearDecomposition(` *module* `,` S `,` C `,` e `)`

module is a module for a matrix group G over a finite field $GF(q)$. The function returns `true` if G is found to act semilinearly.

G is assumed to act absolutely irreducibly. S is a set of invertible matrices, generating a subgroup of G, and assumed to act irreducibly but not absolutely irreducibly on the underlying vector space of *module*. The matrix C centralises the action of S on the underlying vector space and so acts as multiplication by a field generator x of $GF(q^e)$ for some embedding of a d/e-dimensional vector space over $GF(q^e)$ in the d-dimensional space. If C centralises the action of the normal subgroup $\langle S\rangle^G$ of G, then $\langle S\rangle^G$ embeds in $GL(d/e, q^e)$, and G embeds in $\Gamma L(d/e, q^e)$, the group of semilinear automorphisms of the d/e-dimensional space. This is verified by the construction of a map from G to $Aut(GF(q^e))$. If the construction is successful, the function returns `true`. Otherwise a conjugate of an element of S is found which does not commute with C. This conjugate is added to S and the function returns `false`.

`SemiLinearDecomposition` is called by `SmashGModule`.

The algorithm is described in [6].

68.29 TensorProductDecomposition

`TensorProductDecomposition(` *module* `,` *basis* `,` *d1* `,` *d2* `)`

module is a module for a matrix group G over a finite field, *basis* is a basis of the underlying vector space, *d1* and *d2* are two integers whose product is the dimension of that space.

`TensorProductDecomposition` returns `true` if *module* can be decomposed as a tensor product of spaces of dimensions *d1* and *d2* with respect to the given basis, and `false` otherwise. The matrices which represent the action of the generators of G with respect to the appropriate basis are computed.

`TensorProductDecomposition` is called by `SmashGModule`.

The algorithm is described in [6].

68.30 SymTensorProductDecomposition

`SymTensorProductDecomposition(` *module*`,` *HM* `)`

module is a module for a matrix group G over a finite field. *HM* is the module corresponding to the action of a subgroup H of G on the same vector space. Both G and H are assumed to act absolutely irreducibly. The function returns `true` if *HM* can be decomposed as a tensor product of two or more H-modules, all of the same dimension, where these tensor factors are permuted by the action of G. In this case, components of *module* record the tensor decomposition and the action of G in permuting the factors. If no such decomposition is found, `SymTensorProductDecomposition` returns `false`.

A negative answer is **not** reliable, since we try to find a decomposition of *HM* as a tensor product only by considering some pseudo-random elements.

`SymTensorProductDecomposition` is called by `SmashGModule`.

The algorithm is described in [6].

68.31 ExtraSpecialDecomposition

`ExtraSpecialDecomposition(` *module*`,` S `)`

module is a module for a matrix group G over a finite field where G is assumed to act absolutely irreducibly.

S is a set of invertible matrices, assumed to act absolutely irreducibly on the underlying vector space of *module*.

`ExtraSpecialDecomposition` returns `true` if (modulo scalars) $\langle S\rangle$ is an extraspecial r-group, for some prime r, or a 2-group of symplectic type (that is, the central product of an extraspecial 2-group with a cyclic group of order 4), normalised by G. Otherwise it returns `false`.

`ExtraSpecialDecomposition` attempts to prove that $\langle S\rangle$ is extraspecial or of symplectic type by construction. It tries to find generators $x_1,\ldots,x_k,y_1,\ldots,y_k,z$ for $\langle S\rangle$, with z central of order r, each x_i commuting with all other generators except y_i, each y_i commuting with all other generators except x_i, and $[x_i,y_i]=z$. If it succeeds, it checks that conjugates of these generators are also in S.

`ExtraSpecialDecomposition` is called by `SmashGModule`.

The algorithm is described in [6].

68.32 MinBlocks

`MinBlocks(` *module*`,` *B* `)`

`MinBlocks` finds the smallest block containing the echelonised basis B under the action of the G-module *module*. The block system record returned has four components: the number of blocks, a block containing the supplied basis B, a permutation group P which describes the action of G on the blocks, and a list which identifies the images of the generators of G as generators of P. For an explanation of this last component, see `ApproximateKernel`.

`MinBlocks` is called by `IsPrimitive`.

The algorithm is described in [7].

68.33 BlockSystemFlag

`BlockSystemFlag(` *module* `)`

If the record for the G-module *module* has a block system component, then `BlockSystemFlag` returns this component, which has the structure described in `MinBlocks`, else it returns `false`.

68.34 Components of a G-module record

The component `.reducible` is set to `true` if *module* is known to be reducible, and to `false` if it is known not to be. This component is set by `IsIrreducible` which may also set the components `.subbasis`, `.algEl`, `.algElMat`, `.algElCharPol`, `.algElCharPolFac`, `.algElNullspaceVec` and `.algElNullspaceDim`. If *module* has been proved reducible, `.subbasis` is a basis for a submodule. Alternatively, if *module* has been proved to be irreducible, `.algEl` is set to the random element *el* of the group algebra which has been successfully used by the algorithm to prove irreducibility, represented abstractly, essentially as a sum of words in the generators, and `.algElMat` to the actual matrix X that represents that element. The component `.algElCharPol` is set to the characteristic polynomial p of X and `.algElCharPolFac` to the factor f of X used by the algorithm. (Essentially irreducibility is proved by applying Norton's irreducibility criterion to the matrix $f(X)$; see [5] for further details.) The component `.algElNullspaceVec` is set to an arbitrary vector of the nullspace N of $f(X)$, and `.algElNullspaceDim` to the dimension of N.

The component `.absolutelyReducible` is set to `false` if *module* is known to be absolutely irreducible, and to `true` if it is known not to be. It is set by `IsAbsolutelyIrreducible`, which also sets the components `.degreeFieldExt`, `.centMat`, `.centMatMinPoly` if *module* is not absolutely irreducible. In that case, `.degreeFieldExt` is set to the dimension e of the centralising field of *module*. The component `.centMat` is set to a matrix C, which both centralises each of the matrices in *module*.generators generating the group action of *module* and has minimal polynomial f of degree e. The component `.centMatMinPoly` is set equal to f.

The component `.semiLinear` is set to `true` in `SemiLinearDecomposition` if G acts absolutely irreducibly on *module* but embeds in a group of semilinear automorphisms over an extension field of degree e over the field F. Otherwise it is not set. At the same time, `.degreeFieldExt` is set to e, `.linearPart` is set to a list of matrices S which are normal subgroup generators for the intersection of G with the general linear group in dimension

d/e over $GF(q^e)$, and `.centMat` is set to a matrix C which commutes with each of those matrices. Here, C corresponds to scalar multiplication in the module by an element of the extension field $GF(q^e)$. The component `.frobeniusAutomorphisms` is set to a list of integers i, one corresponding to each of the generating matrices g for G in the list `.generators`, such that $Cg = gC^{q^{i(g)}}$. Thus the generator g acts on the field $GF(q^e)$ as the Frobenius automorphism $x \to x^{q^{i(g)}}$.

The component `.tensorProduct` is set to `true` in `TensorProductDecomposition` if *module* can be written as a tensor product of two G-modules with respect to an appropriate basis. Otherwise it is not set. At the same time, `.tensorBasis` is set to the appropriate basis of that space, and `.tensorFactors` to the pair of G-modules whose tensor product is isomorphic to *module* written with respect to that basis.

The component `.symTensorProduct` is set to `true` in `SymTensorProductDecomposition` if *module* can be written as a symmetric tensor product whose factors are permuted by the action of G. Otherwise it is not set. At the same time, `.symTensorBasis` is set to the basis with respect to which this decomposition can be found, `.symTensorFactors` to the list of tensor factors, and `.symTensorPerm` to the list of permutations corresponding to the action of each of the generators of G on those tensor factors.

The component `.extraSpecial` is set to `true` in the function `ExtraSpecialDecomposition` if G has been shown to have a normal subgroup H which is an extraspecial r-group for an odd prime r or a 2-group of symplectic type, modulo scalars. Otherwise it is not set. At the same time, `.extraSpecialGroup` is set to the subgroup H, and `.extraSpecialPrime` is set to r.

The component `.imprimitive` is set to `true` if G has been shown to act imprimitively and to `false` if G is primitive. Otherwise it is not set. This component is set in `IsPrimitive`. If G has been shown to act imprimitively, then *module* has a component `.blockSystem` which has the structure described in `BlockSystemFlag`.

68.35 ApproximateKernel

`ApproximateKernel(` G`,` P`,` m`,` n `[,`*maps*`] )`

G is an irreducible matrix group. P is a permutation representation of G.

`ApproximateKernel` returns a generating set for **a subgroup** of the kernel of a homomorphism from G to P. The parameter m is the maximum number of random relations constructed in order to obtain elements of the kernel. If n successive relations provide no **new** elements of the kernel, then we terminate the construction. These two parameters determine the time taken to construct the kernel; n can be used to increase the probability that the whole of the kernel is constructed. The suggested values of m and n are 100 and 30, respectively.

Assume that G has r generators and P has s generators. The optional argument *maps* is a list of length r containing integers between 0 and s. We use *maps* to specify the correspondence between the generators of G and the generators of P. An entry 0 in position i indicates that G.i maps to the identity of P; an entry j in position i indicates that G.i maps to P.j. By default, we assume that G.i maps to P.i.

The function is similar to `RecogniseMatrixGroup` but here we already know `.quotient` is G and we have a permutation representation P for G. The function returns a record containing information about the kernel. The record contents can be viewed using `DisplayMatRecord`.

The algorithm is described in [13]; the implementation is currently **experimental**.

68.36 RandomRelations

`RandomRelation(` *G*`,` *P* `[,`*maps*`] )`

G is a matrix group. P is a permutation representation of G. The optional argument *maps* has the same meaning as in `ApproximateKernel`.

`RandomRelation` returns a relation for G. We set up a free group on the number of generators of G and we also set up a mapping from P to this free group. We then take a random word in the free group and evaluate this in P. Our relation is the product of the original word and the inverse of the image of the permutation under the mapping we have constructed.

`EvaluateRelation(` *reln*`,` *G* `)`

reln is the word returned by an application of `RandomRelation`. `EvaluateRelation` evaluates *reln* on the generators of G.

68.37 DisplayMatRecord

`DisplayMatRecord(` *rec* `[,` *layer*`] )`
`SetPrintLevelFlag(` *rec*`,` *i* `)`
`PrintLevelFlag(` *rec* `)`

rec is the record returned either by `RecogniseMatrixGroup` or `ApproximateKernel`. The optional argument *layer* is an integer between 1 and the last layer reached by the computation and i is an integer between 1 and 3.

`DisplayMatRecord` prints the information contained in *rec* according to three different print level settings. The print level is initially set to 1. This can be changed using `SetPrintLevelFlag`. We can also examine the current print level using `PrintLevelFlag`.

At print level 1, we get basic information about the group; the field over which it is written, its dimension and possibly its size. If *layer* is specified, then we get this basic information about `.quotient` at that *layer*.

At print level 2, we get the same basic information about the group as we did at level 1 along with the basic information about `.quotient` at each level. If *layer* is specified, then we get the same information as we did at level 1.

At print level 3, we print the entire contents of *rec*. If *layer* is specified, then we print the part of *rec* that corresponds to *layer*.

68.38 The record returned by RecogniseMatrixGroup

Both `RecogniseMatrixGroup` and `ApproximateKernel` return a record whose components tell us information about the group and the various kernels which we compute.

Each layer of the record contains basic information about its corresponding group; the field over which it is written, its identity, its dimension and its generators. These are stored in components `.field`, `.identity`, `.dimension` and `.generators` respectively.

Each layer also has components `.layerNumber`, `.type`, `.size` and `.printLevel`. `.layerNumber` is an integer telling us which layer of the record we are in. The top layer is layer 1, `.kernel` is layer 2, etc.

The component `.type` is one of the following strings: "Unknown", "Perm", "SL", "Imprimitive", "Trivial"and "PGroup". If `.type` is "Unknown"then we have not yet computed `.quotient`. If `.type` is "Perm"then we have computed `.quotient`; if `.quotient` does not contain SL then we compute a permutation representation for it. If `.quotient` contains SL then `.type` is "SL". If `.quotient` is imprimitive then `.type` is "Imprimitive". If `.quotient` is trivial then `.type` is "Trivial". If we are in the last layer then `.type` is "PGroup".

The component `.size` is the size of the group generated by `.generators`; `.printLevel` is the current print level (see `DisplayMatRecord`).

All layers except the last have components `.sizeQuotient`, `.dimQuotient`, `.basisSubmodule` and `.basis`. Here `.sizeQuotient` is the size of `.quotient`. If we have a permutation representation for `.quotient` which is not faithful, then `.sizeQuotient` is the size of the permutation group. We compute the kernel of the action in the next layer and thus obtain the correct size of `.quotient`. `.dimQuotient` is the dimension of `.quotient`. The component `.basisSubmodule` is a matrix consisting of standard basis vectors for the quotient module. We use it to check that the `.quotient` block structure is preserved. `.basis` is the basis-change matrix returned when we split the group.

The `.quotient` record may have `.permDomain`, `.permGroupP`, `.fpGroup`, `.abstractGenerators`, `.fpHomomorphism` and `.isFaithful` as components. If we have a permutation representation on the group `.quotient`, then `.permDomain` is either a list of vectors or subspaces on which the group acts to provide a permutation group. `.permGroupP` is the permutation group. `.fpGroup` is a free group on the number of generators of `.quotient`. `.abstractGenerators` is the generators of `.fpGroup`. `.fpHomomorphism` is a mapping from `.permGroupP` to `.fpGroup`. `.isFaithful` tells us whether we have learned that the representation is not faithful.

The `.pGroup` record has components `.field`, `.size`, `.prime`, `.dimension`, `.identity`, `.layers` and `.layersVec`. Here `.field` is the field over which the group is written; `.size` is the size of the group; `.prime` is the characteristic of the field; `.dimension` is the dimension of the group; `.identity` is the identity for the group; `.layers` and `.layersVec` are lists of lists of matrices and vectors respectively which we use to compute the exponents of relations to get the size of the p-group.

68.39 DualGModule

`DualGModule(` *module* `)`

module is a G-module. The dual module (obtained by inverting and transposing the generating matrices) is calculated and returned.

68.40 InducedGModule

`InducedGModule(` G`,` *module* `)`

G is a transitive permutation group , and *module* an H-module, where H is the subgroup of G returned by `Stabilizer(`*group*`, 1)`. (That is, the matrix generators for *module* should correspond to the permutations generators for H returned by this function call.) The induced G-module is calculated and returned. If the action of H on *module* is trivial, then `PermGModule` should be used instead.

68.41 PermGModule

`PermGModule(` *G*`,` *field* `[,` *point*`] )`

G is a permutation group, and *field* a finite field. If *point* is supplied, it should be an integer in the permutation domain of G; by default, it is 1. The permutation module of G on the orbit of *point* over the field *field* is calculated and returned.

68.42 TensorProductGModule

`TensorProductGModule(` *module1* `,` *module2* `)`

The tensor product of the G-modules *module1* and *module2* is calculated and returned.

`WedgeGModule(` *module* `)`

The wedge product of the G-module *module* – that is, the action on anti-symmetric tensors – is calculated and returned.

68.43 ImprimitiveWreathProduct

`ImprimitiveWreathProduct(` *G*`,` *perm-group* `)`

G is a matrix group, a G-module, a list of matrices, a permutation group or a list of permutations, and *perm-group* can be a permutation group or a list of permutations. Let G be the permutation or matrix group specified or generated by the first argument, P the permutation group specified or generated by the second argument. The wreath product of G and P is calculated and returned. This is a matrix group or a permutation group of dimension or degree dt, where d is the dimension or degree of G and t the degree of P. If G is a permutation group, this function has the same effect as `WreathProduct(G, P)`.

68.44 WreathPower

`PowerWreathProduct(` *G*`,` *perm-group* `)`

G is a matrix group, a G-module, a list of matrices, a permutation group or a list of permutations, and *perm-group* can be a permutation group or a list of permutations. Let G be the permutation or matrix group specified or generated by the first argument, and P the permutation group specified or generated by the second argument. The wreath power of G and P is calculated and returned. This is a matrix group or a permutation group of dimension or degree d^t, where d is the dimension or degree of G and t the degree of P.

68.45 PermGroupRepresentation

`PermGroupRepresentation(` *G*`,` *limit* `)`

`PermGroupRepresentation` tries to find a permutation representation of G of degree at most *limit*. The function either returns a permutation group or `false` if no such representation was found.

Note that `false` does **not** imply that no such permutation representation exists. If a permutation representation for G is already known it will be returned **independent** of its degree.

The function tries to find a set of vectors of size at most *limit* closed under the operation of G such that the set spans the whole vector space; it implements parts of the base-point selection algorithm described in [10].

```
gap> m1 := [[0,1],[1,0]] * Z(9);;
gap> m2 := [[1,1],[1,0]] * Z(9);;
gap> G := Group( m1, m2 );;
gap> P := PermGroupRepresentation( G, 100 );
Group( ( 1,15, 4,21, 2,24, 7,30)( 3,18, 5,12, 6,27, 8, 9)
 (10,16,19,22,14,26,29,32)(11,25,20,31,13,17,28,23),
 ( 1,15,19,31)( 2,24,29,23)( 3,18,22,11)( 4,21,14,17)( 5,12,26,20)
 ( 6,27,32,13)( 7,30,10,25)( 8, 9,16,28) )

# note that limit is ignored if a representation is known
gap> P := PermGroupRepresentation( G, 2 );
Group( ( 1,15, 4,21, 2,24, 7,30)( 3,18, 5,12, 6,27, 8, 9)
 (10,16,19,22,14,26,29,32)(11,25,20,31,13,17,28,23),
 ( 1,15,19,31)( 2,24,29,23)( 3,18,22,11)( 4,21,14,17)( 5,12,26,20)
 ( 6,27,32,13)( 7,30,10,25)( 8, 9,16,28) )
```

`OrbitMat(` *G*`,` *vec*`,` *base*`,` *limit* `)`

`OrbitMat` computes the orbit of *vec* under the operation of G. The function will return `false` if this orbit is larger then *limit*. Otherwise the orbit is return as list of vectors and *base*, which must be supplied as an empty list, now contains a list of basis vectors spanning the vector space generated by the orbit.

68.46 GeneralOrthogonalGroup

`GeneralOrthogonalGroup(`*s*`,` *d*`,` *q*`)`
`O(` *s*`,` *d*`,` *q* `)`

This function returns the group of isometries fixing a non-degenerate quadratic form as matrix group. d specifies the dimension, q the size of the finite field, and s the sign of the quadratic form Q. If the dimension is odd, the sign must be 0. If the dimension is even the sign must be -1 or $+1$. The quadratic form Q used is returned in the component `quadraticForm`, the corresponding symmetric form β is returned in the component `symmetricForm`.

Given the standard basis $B = \{e_1, ..., e_d\}$ then `symmetricForm` is the matrix $(f(e_i, e_j))$, `quadraticForm` is an upper triangular matrix (q_{ij}) such that $q_{ij} = f(e_i, e_j)$ for $i < j$, $q_{ii} = Q(e_i)$, and $q_{ij} = 0$ for $j < i$, and the equations $2Q(e_i) = f(e_i, e_i)$ hold.

There are precisely two isometry classes of geometries in each dimension d. If d is even then the geometries are distinguished by the dimension of the maximal totally singular subspaces. If the sign s is $+1$, then the Witt defect of the underlying vector space is 0, i. e. the maximal totally singular subspaces have dimension $d/2$; if the sign is -1, the defect is 1, i.e. the dimension is $d/2 - 1$.

If d is odd then the geometries are distinguished by the **discriminant** of the quadratic form Q which is defined as the determinant of $(f(e_i, e_j))$ modulo $(GF(q)^\star)^2$. The determinant of

$(f(e_i, e_j))$ is not independent of B, whereas modulo squares it is. However, the two geometries are similar and give rise to isomorphic groups of isometries. `GeneralOrthogonalGroup` uses a quadratic form Q with discriminant -2^{d-2} modulo squares.

In case of odd dimension, q must also be odd because the group `O( 0, 2d+1,` 2^k `)` is isomorphic to the symplectic group `Sp( 2d,` 2^k `)` and you can use `SP` to construct it.

```
gap> G := GeneralOrthogonalGroup(0,5,3);
O(0,5,3)
gap> Size( G );
103680
gap> Size( SP(4,3) );
51840
gap> DeterminantMat(G.1);
Z(3)^0
gap> DeterminantMat(G.2);
Z(3)
gap> DisplayMat( G.symmetricForm );
 . 1 . . .
 1 . . . .
 . . 2 . .
 . . . 2 .
 . . . . 2
gap> DisplayMat( G.quadraticForm );
 . 1 . . .
 . . . . .
 . . 1 . .
 . . . 1 .
 . . . . 1
```

You can evaluate the quadratic form on a vector by multiplying it from both sides.

```
gap> v1 := [1,2,0,1,2] * Z(3);
[ Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ]
gap> v1 * G.quadraticForm * v1;
Z(3)^0
gap> v1 * G.symmetricForm * v1;
Z(3)
```

68.47 OrderMat – enhanced

`OrderMat(`g`)`

This function works as described in the GAP3 manual but uses the algorithm of [2] for matrices over finite fields.

```
gap> OrderMat( [ [ Z(17)^4, Z(17)^12, Z(17)^4, Z(17)^7 ],
>   [ Z(17)^10, Z(17), Z(17)^11, 0*Z(17) ],
>   [ Z(17)^8, Z(17)^13, Z(17)^0, Z(17)^14 ],
>   [ Z(17)^14, Z(17)^10, Z(17), Z(17)^10 ] ] );
5220
```

`ProjectiveOrderMat(`g`)`

This function computes the least positive integer n such that g^n is scalar; it returns, as a list, n and z, where g^n is scalar in z.

```
gap> ProjectiveOrderMat( [ [ Z(17)^4, Z(17)^12, Z(17)^4, Z(17)^7 ],
>   [ Z(17)^10, Z(17), Z(17)^11, 0*Z(17) ],
>   [ Z(17)^8, Z(17)^13, Z(17)^0, Z(17)^14 ],
>   [ Z(17)^14, Z(17)^10, Z(17), Z(17)^10 ] ] );
[ 1305, Z(17)^12 ]
```

68.48 PseudoRandom

`PseudoRandom(` *G* `)`
`PseudoRandom(` *module* `)`

It takes as input either a matrix group G, or a G-module *module* and returns a pseudo-random element. If the supplied record has no seed stored as a component, then it constructs one (as in `InitPseudoRandom`).

The algorithm is described in [1].

68.49 InitPseudoRandom

`InitPseudoRandom(` *G*`,` *length*`,` *n* `)`
`InitPseudoRandom(` *module*`,` *length*`,` *n* `)`

`InitPseudoRandom` takes as input either a matrix group G, or a G-module *module*. It constructs a list (or seed) of elements which can be used to generate pseudo-random elements of the matrix group or G-module. This seed is stored as a component of the supplied record and can be accessed using `RandomSeedFlag`.

`InitPseudoRandom` generates a seed of *length* elements containing copies of the generators of G and performs a total of n matrix multiplications to initialise this list.

The quality of the seed is determined by the value of n. For many applications, *length* $=10$ and $n = 100$ seem to suffice; these are the defaults used by `PseudoRandom`.

The algorithm is described in [1].

68.50 IsPpdElement

`IsPpdElement(` *F*`,` *m*`,` *d*`,` *s*`,` *c*`)`

For natural numbers b and e greater than 1 a primitive prime divisor of $b^e - 1$ is a prime dividing $b^e - 1$ but not dividing $b^i - 1$ for any $1 \leq i < e$. If r is a primitive prime divisor of $b^e - 1$ then $r = ce + 1$ for some positive integer c and in particular $r \geq e + 1$. If either $r \geq e + 2$, or $r = e + 1$ and r^2 divides $b^e - 1$ then r is called a large primitive prime divisor of $b^e - 1$.

Let e be a positive integer greater than 1, such that $d/2 < e \leq d$. Let p be a prime and $q = p^a$. An element g of $\mathrm{GL}(d, q)$ whose order is divisible a primitive prime divisor of $q^e - 1$ is a ppd-element, or $\mathrm{ppd}(d, q; e)$-element. An element g of $\mathrm{GL}(d, q)$ whose order is divisible by a primitive prime divisor of $p^{ae} - 1$ is a basic ppd-element, or basic $\mathrm{ppd}(d, q; e)$-element. An element g of $\mathrm{GL}(d, q)$ is called a large ppd-element if there exists a large primitive prime divisor r of $q^e - 1$ such that the order of g is divisible by r, if $r \geq e+2$, or by r^2, if $r = e+1$.

The function `IsPpdElement` takes as input a field F, and a parameter m, and integers d, s and c, where s^c is the size $q = p^a$ of the field F. For the recognition algorithm, (s,c) is either $(q, 1)$ or (p, a). The parameter m is either an element of $\mathrm{GL}(d, F)$ or a characteristic polynomial of such an element. If m is not (the characteristic polynomial of) a ppd(dc, s; ec)-element for some e such that $d/2 < e \leq d$ then `IsPpdElement` returns `false`. Otherwise it returns a list of length 2, whose first entry is the integer e and whose second entry is `true` if m is (the characteristic polynomial of) a large ppd(dc, s; ec)-element or `false` if it is not large. When c is 1 and s is q this function decides whether m is (the characteristic polynomial of) a ppd(d, q; e)-element whereas when s is the characteristic p of F and c is such that a then it decides whether m is (the characteristic polynomial of) a basic ppd(d, q; e)-element.

```
gap> G := GL (6, 3);;
gap> g := [ [ 2, 2, 2, 2, 0, 2 ],
>           [ 1, 0, 0, 0, 0, 1 ],
>           [ 2, 2, 1, 0, 0, 0 ],
>           [ 2, 0, 2, 0, 2, 0 ],
>           [ 1, 2, 0, 1, 1, 0 ],
>           [ 1, 2, 2, 1, 2, 0 ] ] * Z(3)^0;;
gap> IsPpdElement( G.field, g, 6, 3, 1);
[ 5, true ]
gap> Collected( Factors( 3^5 - 1) );
[ [ 2, 1 ], [ 11, 2 ] ]
gap> Order (G, g) mod 11;
0
```

The algorithm is described in [2] and [11].

68.51 SpinorNorm

`SpinorNorm(` *form*`,` *mat* `)`

computes the spinor norm of *mat* with respect to the symmetric bilinear *form*.

The underlying field must have odd characteristic.

```
gap> z  := GF(9).root;;
gap> m1 := [[0,1,0,0,0,0,0,0,0],[1,2,2,0,0,0,0,0,0],
>  [0,0,0,1,0,0,0,0,0],[0,0,0,0,1,0,0,0,0],[0,0,0,0,0,1,0,0,0],
>  [0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,1],
>  [0,2,1,0,0,0,0,0,0]]*z^0;;
gap> m2 := [[z,0,0,0,0,0,0,0,0],[0,z^7,0,0,0,0,0,0,0],
>  [0,0,1,0,0,0,0,0,0],[0,0,0,1,0,0,0,0,0],[0,0,0,0,1,0,0,0,0],
>  [0,0,0,0,0,1,0,0,0],[0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,0],
>  [0,0,0,0,0,0,0,0,1]]*z^0;;
gap> form := IdentityMat( 9, GF(9) );;
gap> form{[1,2]}{[1,2]} := [[0,2],[2,0]] * z^0;;
gap> m1 * form * TransposedMat(m1) = form;
true
gap> m2 * form * TransposedMat(m2) = form;
true
```

```
gap> SpinorNorm( form, m1 );
Z(3)^0
gap> SpinorNorm( form, m2 );
Z(3^2)^5
```

68.52 Other utility functions

`Commutators(` *matrix-list* `)`

It returns a set containing the **non-trivial** commutators of all pairs of matrices in *matrix list*.

`IsDiagonal(` *matrix* `)`

If a matrix, *matrix*, is diagonal, it returns `true`, else `false`.

`IsScalar(` *matrix* `)`

If a matrix, *matrix*, is scalar, it returns `true`, else `false`.

`DisplayMat(` *matrix-list* `)`
`DisplayMat(` *matrix* `)`

It converts the entries of a matrix defined over a finite field into integers and "pretty-prints" the result. All matrices in *matrix list* must be defined over the same finite field.

`ChooseRandomElements(`*G*`,` *NmrElts*`)`
`ChooseRandomElements(`*module*`,` *NmrElts*`)`

It takes as input either a matrix group G, or a G-module *module*, and returns *NmrElts* pseudo-random elements.

`ElementOfOrder(`*G*`,` *RequiredOrder*`,` *NmrTries*`)`
`ElementOfOrder(`*module*`,` *RequiredOrder*`,` *NmrTries*`)`

It takes as input either a matrix group G, or a G-module *module*, and searches for an element of order *RequiredOrder*. It examines at most *NmrTries* elements before returning `false`.

`ElementWithCharPol(`*G*`,` *order*`,` *pol*`,` *NmrTries*`)`
`ElementWithCharPol(`*module*`,` *order*`,` *pol*`,` *NmrTries*`)`

It takes as input either a matrix group G, or a G-module *module*. It searches for an element of order *order* and characteristic polynomial *pol*. It examines at most *NmrTries* pseudo-random elements before returning `false`.

`LargestPrimeOrderElement(`*G*`,` *NmrTries*`)`
`LargestPrimeOrderElement(`*module*`,` *NmrTries*`)`

It takes as input either a matrix group G, or a G-module *module*. It generates *NmrTries* pseudo-random elements and determines which elements of prime order can be obtained from powers of each; it returns the largest found and its order as a list.

`LargestPrimePowerOrderElement(`*G*`,` *NmrTries*`)`
`LargestPrimePowerOrderElement(`*module*`,` *NmrTries*`)`

It takes as input either a matrix group G, or a G-module *module*. It generates *NmrTries* pseudo-random elements and determines which elements of prime-power order can be obtained from powers of each; it returns the largest found and its order as a list.

68.53 References

[1] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E.A. O'Brien, "Generating random elements of a finite group", Comm. Algebra 23, 4931–4948, 1995.

[2] Frank Celler and C.R. Leedham-Green, "Calculating the Order of an Invertible Matrix", "Groups and Computation II", Amer. Math. Soc. DIMACS Series 28, 1997.

[3] Frank Celler and C.R. Leedham-Green, "A Non-Constructive Recognition Algorithm for the Special Linear and Other Classical Groups", "Groups and Computation II", Amer. Math. Soc. DIMACS Series 28, 1997.

[4] Frank Celler and C.R. Leedham-Green, "A constructive recognition algorithm for the special linear group", preprint.

[5] Derek F. Holt and Sarah Rees, "Testing modules for irreducibility", J. Austral. Math. Soc. Ser. A, 57, 1–16, 1994.

[6] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees, "Computing Matrix Group Decompositions with Respect to a Normal Subgroup", J. Algebra 184, 818–838, 1996.

[7] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees, "Testing Matrix Groups for Imprimitivity", J. Algebra 184, 795–817, 1996.

[8] C.R. Leedham-Green and E.A. O'Brien, "Tensor Products are Projective Geometries", to appear J. Algebra.

[9] C.R. Leedham-Green and E.A. O'Brien, "Recognising tensor products of matrix groups", to appear Internat. J. Algebra Comput.

[10] Scott H. Murray and E.A. O'Brien, "Selecting Base Points for the Schreier-Sims Algorithm for Matrix Groups", J. Symbolic Comput. 19, 577–584, 1995.

[11] Alice C. Niemeyer and Cheryl E. Praeger "A Recognition Algorithm for Classical Groups over Finite Fields", submitted to Proceedings of the London Mathematical Society.

[12] Alice C. Niemeyer and Cheryl E. Praeger "Implementing a Recognition Algorithm for Classical Groups", "Groups and Computation II", Amer. Math. Soc. DIMACS Series 28, 1997.

[13] Anthony Pye, "Recognising reducible matrix groups", in preparation.

The following sources provide additional theoretical background to the algorithms.

[14] M. Aschbacher (1984), "On the maximal subgroups of the finite classical groups", Invent. Math. 76, 469–514, 1984.

[15] Peter Kleidman and Martin Liebeck, "The Subgroup Structure of the Finite Classical Groups", Cambridge University Press, London Math. Soc. Lecture Note Series 129, 1990.

Chapter 69

The MeatAxe

This chapter describes the main functions of the MeatAxe (Version 2.0) share library for computing with finite field matrices, permutations, matrix groups, matrix algebras, and their modules.

For the installation of the package, see 57.9.

The chapter consists of seven parts.

First the idea of using the MeatAxe via GAP3 is introduced (see 69.1, 69.2), and an example shows how the programs can be used (see 69.3).

The second part describes functions and operations for single MeatAxe **matrices** (see 69.4, 69.5, 69.6, 69.7, 69.8).

The third part describes functions and operations for single MeatAxe **permutations** (see 69.9, 69.10, 69.11, 69.12).

The fourth part describes functions and operations for **groups** of MeatAxe matrices (see 69.13, 69.14).

(Groups of MeatAxe permutations are not yet supported.)

The fifth part describes functions and operations for **algebras** of MeatAxe matrices (see 69.15, 69.16).

The sixth part describes functions and operations for **modules** for MeatAxe matrix algebras (see 69.17, 69.18, 69.19, 69.20).

The last part describes the data structures (see 69.22).

If you want to use the functions in this package you must load it using

```
gap> RequirePackage( "meataxe" );
#I  The MeatAxe share library functions are available now.
#I  All files will be placed in the directory
#I     '/var/tmp/tmp.017545'
#I  Use 'MeatAxe.SetDirectory( <path> )' if you want to change.
```

69.1 More about the MeatAxe in GAP

The MeatAxe can be used to speed up computations that are possible also using ordinary GAP3 functions. But more interesting are functions that are not (or not yet) available in the GAP3 library itself, such as that for the computation of submodule lattices (see 69.20).

The syntax of the functions is the usual GAP3 syntax, so it might be useful to read the chapters about algebras and modules in GAP3 (see chapters 39, 42) if you want to work with MeatAxe modules.

The main idea is to let the MeatAxe functions do the main work, and use GAP3 as a shell. Since in MeatAxe philosophy, each object is contained in its own file, GAP3's task is mainly to keep track of these files. For example, for GAP3 a MeatAxe matrix is a record containing at least information about the file name, the underlying finite field, and the dimensions of the matrix (see 69.4). Multiplying two such matrices means to invoke the multiplication program of MeatAxe, to store the result in a new file, and notify this to GAP3.

This idea is used not only for basic calculations but also to access elaborate and powerful algorithms, for example the program to compute the composition factors of a module, or the submodule lattice (see 69.20).

In order to avoid conversion overhead the MeatAxe matrices are read into GAP3 only if the user explicitly applies `GapObject` (see 69.2), or applies an operator (like multiplication) to a MeatAxe matrix and an ordinary GAP3 object.

Some of the functions, mainly `CompositionFactors`, print useful information if the variable `InfoMeatAxe` is set to the value `Print`. The default of `InfoMeatAxe` is `Print`, if you want to suppress the information you should set `InfoMeatAxe` to `Ignore`.

For details about the implementation of the standalone functions, see [Rin93].

69.2 GapObject

`GapObject(` *mtxobj* `)`

returns the GAP3 object corresponding to the MeatAxe object *mtxobj* which may be a MeatAxe matrix, a MeatAxe permutation, a MeatAxe matrix algebra, or a MeatAxe module.

Applied to an ordinary GAP3 object, `GapObject` simply returns this object.

```
gap> m:= [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] * GF(2).one;;
gap> mam:= MeatAxeMat( m, "file2" );;
#I  calling 'maketab' for field of size 2
gap> GapObject( mam );
[ [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> map:= MeatAxePerm( (1,2,3), 3 );;
gap> perm:= GapObject( map );
(1,2,3)
gap> GapObject( perm );
(1,2,3)
```

69.3 Using the MeatAxe in GAP. An Example

In this example we compute the 2-modular irreducible representations and Brauer characters of the alternating group A_5. Perhaps it will raise the question whether one uses the MeatAxe in GAP3 or GAP3 for the MeatAxe.

First we take a permutation representation of A_5 and convert the generators into MeatAxe matrices over the field $GF(2)$.

```
gap> a5:= Group( (1,2,3,4,5), (1,2,3) );;
gap> Size( a5 );
60
gap> f:= GF(2);;
gap> m1:= MeatAxeMat( a5.1, f, [5,5] );
MeatAxeMat( "/var/tmp/tmp.017545/a", GF(2), [ 5, 5 ] )
gap> m2:= MeatAxeMat( a5.2, f, [5,5] );;
```

`m1` and `m2` are records that know about the files where the matrices are stored. Let's look at such a matrix (without reading the file into GAP3).

```
gap> Display( m1 );
MeatAxe.Matrix := [
[0,1,0,0,0],
[0,0,1,0,0],
[0,0,0,1,0],
[0,0,0,0,1],
[1,0,0,0,0]
]*Z(2);
```

Next we inspect the 5 dimensional permutation module over $GF(2)$. It contains a trivial submodule `fix`, its quotient is called `quot`.

```
gap> a:= UnitalAlgebra( f, [ m1, m2 ] );;
gap> nat:= NaturalModule( a );;
gap> fix:= FixedSubmodule( nat );;
gap> Dimension( fix );
1
gap> quot:= nat / fix;;
```

The action on `quot` is described by an algebra of 4×4 matrices, the corresponding module turns out to be absolutely irreducible. Of course the action on `fix` would yield 1×1 matrices, the generators being the identity. So we found already two of the four absolutely irreducible representations.

```
gap> op:= Operation( a, quot );
UnitalAlgebra( GF(2),
[ MeatAxeMat( "/var/tmp/tmp.017545/t/g.1", GF(2), [ 4, 4 ], a.1 ),
  MeatAxeMat( "/var/tmp/tmp.017545/t/g.2", GF(2), [ 4, 4 ], a.2 ) ] )
gap> nm:= NaturalModule( op );;
gap> IsIrreducible( nm );
true
gap> IsAbsolutelyIrreducible( nm );
true
```

```
gap> deg4:= nm.ring;;
```

Now we form the tensor product of the 4 dimensional module with itself, and compute the composition factors.

```
gap> tens:= KroneckerProduct( nm, nm );;
gap> comp:= CompositionFactors( tens );;
#I   Name Mult  SF
#I     1a    4   1
#I     4a    1   1
#I     4b    2   2
#I
#I  Ascending composition series:
#I  4a  1a  4b  1a  1a  4b  1a
gap> IsIrreducible( comp[3] );
true
gap> IsAbsolutelyIrreducible( comp[3] );
false
```

The information printed by `CompositionFactors` told that there is an irreducible but not absolutely irreducible factor `4b` of dimension 4, and we will enlarge the field in order to split this module.

```
gap> sf:= SplittingField( comp[3] );
GF(2^2)
gap> new:= UnitalAlgebra( sf, [ comp[3].ring.1, comp[3].ring.2 ] );;
#I  calling 'maketab' for field of size 4
gap> nat:= NaturalModule( new );;
gap> comp:= CompositionFactors( nat );;
#I   Name Mult  SF
#I     2a    1   1
#I     2b    1   1
#I
#I  Ascending composition series:
#I  2a  2b
gap> deg2:= List( comp, x -> x.ring );;
```

Now the representations are known. Let's calculate the Brauer characters. For that, we need representatives of the 2-regular conjugacy classes of A_5.

```
gap> repres:= [ a.1^0, a.1 * a.2 * a.1^3, a.1, a.1^2 ];;
gap> List( repres, OrderMeatAxeMat );
[ 1, 3, 5, 5 ]
```

The expression of the representatives of each irreducible representation in terms of the generators can be got using `MappedExpression`.

```
gap> abstracts:= List( repres, x -> x.abstract );
[ a.one, a.1*a.2*a.1^3, a.1, a.1^2 ]
gap> mapped:= List( [ 1 .. 4 ],
>   x-> MappedExpression( abstracts[x],
>       a.freeAlgebra.generators, deg4.generators ) );;
gap> List( mapped, OrderMeatAxeMat );
```

```
[ 1, 3, 5, 5 ]
gap> List( mapped, BrauerCharacterValue );
[ 4, 1, -1, -1 ]
gap> mapped:= List( [ 1 .. 4 ],
>   x-> MappedExpression( abstracts[x],
>       a.freeAlgebra.generators, deg2[1].generators ) );;
gap> List( mapped, BrauerCharacterValue );
[ 2, -1, E(5)^2+E(5)^3, E(5)+E(5)^4 ]
```

The Brauer character of the trivial module is well-known, and that of the other 2-dimensional module is a Galois conjugate of the computed one, so we computed the 2-modular Brauer character table of A_5.

It is advisable to remove all the MeatAxe files before leaving GAP3. Call `MeatAxe.Unbind();` (see 69.21).

69.4 MeatAxe Matrices

MeatAxe matrices behave similar to lists of lists that are regarded as matrices by GAP3, e.g., there are functions like `Rank` or `Transposed` that work for both types, and one can multiply or add two MeatAxe matrices, the result being again a MeatAxe matrix. But one cannot access rows or single entries of a MeatAxe matrix *mat*, for example *mat*`[1]` will cause an error message.

MeatAxe matrices are constructed or notified by 69.5 `MeatAxeMat`.

`IsMeatAxeMat(` *obj* `)`

returns `true` if *obj* is a MeatAxe matrix, and `false` otherwise.

69.5 MeatAxeMat

`MeatAxeMat(` *mat* `[,` *F* `] [,` *abstract* `] [,` *filename* `] )`

returns a MeatAxe matrix corresponding to the matrix *mat*, viewed over the finite field F, or over the field of all entries of *mat*.

If *mat* is already a MeatAxe matrix then the call means that it shall now be viewed over the field F which may be smaller or larger than the field *mat* was viewed over.

The optional argument *abstract* is an element of a free algebra (see chapter 40) that represents the matrix in terms of generators.

If the optional argument *filename* is given, the MeatAxe matrix is written to the file with this name; a matrix constructed this way will **not** be removed by a call to `MeatAxe.Unbind`. Otherwise GAP3 creates a temporary file under the directory `MeatAxe.direc`.

`MeatAxeMat(` *perm*`,` *F*`,` *dim* `[,`*abstract*`][,`*filename*`] )`

does the same for a permutation *perm* that shall be converted into a permutation matrix over the field F, with *dim*`[1]` rows and *dim*`[2]` columns.

`MeatAxeMat(` *file*`,` *F*`,` *dim* `[,` *abstract* `] )` is the MeatAxe matrix stored on file *file*, viewed over the field F, with dimensions *dim*, and representation *abstract*. This may be used to make a shallow copy of a MeatAxe matrix, or to notify MeatAxe matrices that were not produced by GAP3. Such matrices are **not** removed by calls to `MeatAxe.Unbind`.

Note: No field change is allowed here.

```
gap> f:= GF(2);;
gap> m:= [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] * f.one;;
gap> m1:= MeatAxeMat( m, "file2" );
MeatAxeMat( "/var/tmp/tmp.005046/file2", GF(2), [ 3, 3 ] )
gap> p:= (1,2,3);;
gap> m2:= MeatAxeMat( p, f, [ 3, 3 ], "file" );
MeatAxeMat( "/var/tmp/tmp.005046/file", GF(2), [ 3, 3 ] )
gap> Display( m2 );
MeatAxe.Matrix := [
[0,1,0],
[0,0,1],
[1,0,0]
]*Z(2);
gap> n:= MeatAxeMat( "file", f, [ 3, 3 ] );; # just notify a matrix
```

69.6 Operations for MeatAxe Matrices

Comparisons of MeatAxe Matrices

m1 `=` *m2*

evaluates to `true` if the two MeatAxe matrices have the same entries and are viewed over the same field, and to `false` otherwise. The test for equality uses a shell script that is produced when it is needed for the first time.

m1 `<` *m2*

evaluates to `true` if and only if this relation holds for the file names of the two MeatAxe matrices.

Arithmetic Operations of MeatAxe Matrices

The following arithmetic operations are admissible for MeatAxe matrices.

m1 `+` *m2*

sum of the two MeatAxe matrices *m1*, *m2*

m1 `-` *m2*

difference of the two MeatAxe matrices *m1*, *m2*

m1 `*` *m2*

product of the two MeatAxe matrices *m1*, *m2*

m1 `^` *m2*

conjugation of the MeatAxe matrix *m1* by *m2*

m1 `^` *n*

n-th power of the MeatAxe matrix *m1*, for an integer n

69.7 Functions for MeatAxe Matrices

The following functions that work for ordinary matrices in GAP3 also work for MeatAxe matrices.

`UnitalAlgebra(` *F*`,` *gens* `)`
: returns the unital *F*-algebra generated by the MeatAxe matrices in the list *gens*.

`Base(` *mtxmat* `)`
: returns a MeatAxe matrix whose rows form a vector space basis of the row space; the basis is in semi-echelon form.

`BaseNullspace(` *mtxmat* `)`
: returns a MeatAxe matrix in semi-echelon form whose rows are a basis of the nullspace of the MeatAxe matrix *mtxmat*.

`CharacteristicPolynomial(` *mtxmat* `)`
: returns the characteristic polynomial of the MeatAxe matrix *mtxmat*. The factorization of this polynomial is stored.

`Dimensions(` *mtxmat* `)`
: returns the list `[` *nrows*`,` *ncols* `]` where *nrows* is the number of rows, *ncols* is the number of columns of the MeatAxe matrix *mtxmat*.

`Display(` *mtxmat* `)`
: displays the MeatAxe matrix *mtxmat* (without reading into GAP3).

`Group(` *m1*`,` *m2*`,` ...*mn* `)`
: `Group(` *gens*`,` *id* `)`
: returns the group generated by the MeatAxe matrices *m1*, *m2*, ...*mn*, resp. the group generated by the MeatAxe matrices in the list *gens*, where *id* is the appropriate identity MeatAxe matrix.

`InvariantForm(` *mtxmats* `)`
: returns a MeatAxe matrix M such that $X^{tr}MX = M$ for all MeatAxe matrices in the list *mtxmats* if such a matrix exists, and `false` otherwise. Note that the algebra generated by *mtxmats* must act irreducibly, otherwise an error is signalled.

`KroneckerProduct(` *m1*`,` *m2* `)`
: returns a MeatAxe matrix that is the Kronecker product of the MeatAxe matrices *m1*, *m2*.

`Order( MeatAxeMatrices,` *mtxmat* `)`
: returns the multiplicative order of the MeatAxe matrix *mtxmat*, if this exists. This can be computed also by `OrderMeatAxeMat(` *mtxmat* `)`.

`Rank(` *mtxmat* `)`
: returns the rank of the MeatAxe matrix *mtxmat*.

`SumIntersectionSpaces(` *mtxmat1*`,` *mtxmat2* `)`
: returns a list of two MeatAxe matrices, both in semi-echelon form, whose rows are a basis of the sum resp. the intersection of row spaces generated by the MeatAxe matrices *m1* and *m2*, respectively.

`Trace(` *mtxmat* `)`
: returns the trace of the MeatAxe matrix *mtxmat*.

`Transposed(` *mtxmat* `)`
: returns the transposed matrix of the MeatAxe matrix *mtxmat*.

69.8 BrauerCharacterValue

`BrauerCharacterValue(` *mtxmat* `)`

returns the Brauer character value of the MeatAxe matrix *mtxmat*, which must of course be an invertible matrix of order relatively prime to the characteristic of its entries.

```
gap> g:= MeatAxeMat( (1,2,3,4,5), GF(2), [ 5, 5 ] );;
gap> BrauerCharacterValue( g );
0
```

(This program was originally written by Jürgen Müller.)

69.9 MeatAxe Permutations

MeatAxe permutations behave similar to permutations in GAP3, e.g., one can multiply two MeatAxe permutations, the result being again a MeatAxe permutation. But one cannot map single points by a MeatAxe permutation using the exponentiation operator `^`.

MeatAxe permutations are constructed or notified by 69.10 `MeatAxePerm`.

`IsMeatAxePerm(` *obj* `)`

returns `true` if *obj* is a MeatAxe permutation, and `false` otherwise.

69.10 MeatAxePerm

`MeatAxePerm(` *perm*`,` *maxpoint* `)`
`MeatAxePerm(` *perm*`,` *maxpoint*`,` *filename* `)`

return a MeatAxe permutation corresponding to the permutation *perm*, acting on the points `[ 1 ..` *maxpoint* `]`. If the optional argument *filename* is given, the MeatAxe permutation is written to the file with this name; a permutation constructed this way will **not** be removed by a call to `MeatAxe.Unbind`. Otherwise GAP3 creates a temporary file under the directory `MeatAxe.direc`.

`MeatAxePerm(` *file*`,` *maxpoint* `)`

is the MeatAxe permutation stored on file *file*. This may be used to notify MeatAxe permutations that were not produced by GAP3. Such permutations are **not** removed by calls to `MeatAxe.Unbind`.

```
gap> p1:= MeatAxePerm( (1,2,3), 3 );
MeatAxePerm( "/var/tmp/tmp.005046/a", 3 )
gap> p2:= MeatAxePerm( (1,2), 3, "perm2" );
MeatAxePerm( "/var/tmp/tmp.005046/perm2", 3 );
gap> p:= p1 * p2;
MeatAxePerm( "/var/tmp/tmp.005046/b", 3 )
gap> Display( p );
MeatAxe.Perms := [
    (2,3)
];
```

69.11 Operations for MeatAxe Permutations

Comparisons of MeatAxe Permutations

m1 = *m2*
: evaluates to `true` if the two MeatAxe permutations are equal as permutations, and to `false` otherwise. The test for equality uses a shell script that is produced when it is needed for the first time.

m1 < *m2*
: evaluates to `true` if and only if this relation holds for the file names of the two MeatAxe permutations.

Arithmetic Operations of MeatAxe Permutations

The following arithmetic operations are admissible for MeatAxe permutations.

m1 * *m2*
: product of the two MeatAxe permutations *m1*, *m2*

m1 ^ *m2*
: conjugation of the MeatAxe permutation *m1* by *m2*

m1 ^ *n*
: *n*-th power of the MeatAxe permutation *m1*, for an integer *n*

69.12 Functions for MeatAxe Permutations

The following functions that work for ordinary permutations in GAP3 also work for MeatAxe permutations.

`Display(` *mtxperm* `)`
: displays the MeatAxe permutation *mtxperm* (without reading the file into GAP3).

`Order( MeatAxePermutations,` *mtxperm* `)`
: returns the multiplicative order of the MeatAxe permutation *mtxperm*. This can be computed also by `OrderMeatAxePerm(` *mtxperm* `)`.

69.13 MeatAxe Matrix Groups

Groups of MeatAxe matrices are constructed using the usual `Group` command.

Only very few functions are available for MeatAxe matrix groups. For most of the applications one is interested in matrix algebras, e.g., matrix representations as computed by `Operation` when applied to an algebra and a module. For a permutation representation of a group of MeatAxe matrices, however, it is necessary to call `Operation` with a group as first argument (see 69.14).

69.14 Functions for MeatAxe Matrix Groups

The following functions are overlaid in the operations record of MeatAxe matrix groups.

`Operation(` *G*, *M* `)`
: Let *M* a MeatAxe module acted on by the group *G* of MeatAxe matrices. `Operation(`

G, *M*) returns a permutation group with action on the points equivalent to that of G on the vectors of the module M.

`RandomOrders(` *G* `)`
returns a list with the orders of 120 random elements of the MeatAxe matrix group G.

It should be noted that no set theoretic functions (such as `Size`) are provided for MeatAxe matrix groups, and also group theoretic functions (such as `SylowSubgroup`) will not work.

69.15 MeatAxe Matrix Algebras

Algebras of MeatAxe matrices are constructed using the usual `Algebra` or `UnitalAlgebra` commands.

Note that **all** these algebras are regarded to be unital, that is, also if you construct an algebra by calling `Algebra` you will get a unital algebra.

MeatAxe matrix algebras are used to construct and describe MeatAxe modules and their structure (see 69.17).

For functions for MeatAxe matrix algebras see 69.16.

69.16 Functions for MeatAxe Matrix Algebras

The following functions are overlaid in the operations record of MeatAxe matrix algebras.

`Fingerprint(` *A* `)`
`Fingerprint(` *A*, *list* `)`
returns the fingerprint of A, i.e., a list of nullities of six "standard" words in A (for 2-generator algebras only) or of the words with numbers in *list*.

```
gap> f:= GF(2);;
gap> a:= UnitalAlgebra( f, [ MeatAxeMat( (1,2,3,4,5), f, [5,5] ),
>                            MeatAxeMat( (1,2)      , f, [5,5] ) ] );;
gap> Fingerprint( a );
[ 1, 1, 1, 3, 0, 1 ]
```

`Module(` *matalg*, *gens* `)`
returns the module generated by the rows of the MeatAxe matrix *gens*, and acted on by the MeatAxe matrix algebra *matalg*. Such a module will usually contain the vectors of a basis in the `base` component.

`NaturalModule(` *matalg* `)`
returns the n-dimensional space acted on by the MeatAxe matrix algebra *matalg* which consists of $n \times n$ MeatAxe matrices.

`Operation(` *A*, *M* `)`
Let M be a MeatAxe module acted on by the MeatAxe matrix algebra A. `Operation(` *A*, *M* `)` returns a MeatAxe matrix algebra of $n \times n$ matrices (where n is the dimension of M), with action on its natural module equivalent to that of A on M.
Note: If M is a quotient module, it must be a quotient of the entire space.

`RandomOrders(` *A* `)`
returns a list with the orders of 120 random elements of the MeatAxe matrix algebra A, provided that the generators of A are invertible.

It should be noted that no set theoretic functions (such as `Size`) and vector space functions (such as `Base`) are provided for MeatAxe matrix algebras, and also algebra functions (such as `Centre`) will not work.

69.17 MeatAxe Modules

MeatAxe modules are vector spaces acted on by MeatAxe matrix algebras. In the MeatAxe standalone these modules are described implicitly because the matrices contain all the necessary information there. In GAP3 the modules are the concrete objects whose properties are inspected (see 69.20).

Note that most of the usual set theoreic and vector space functions are not provided for MeatAxe modules (see 69.18, 69.19).

69.18 Set Theoretic Functions for MeatAxe Modules

`Size(` *M* `)`
: returns the size of the MeatAxe module *M*.

`Intersection(` *M1*`,` *M2* `)`
: returns the intersection of the two MeatAxe modules *M1*, *M2* as a MeatAxe module.

69.19 Vector Space Functions for MeatAxe Modules

`Base(` *M* `)`
: returns a MeatAxe matrix in semi-echelon form whose rows are a vector space basis of the MeatAxe module *M*.

`Basis(` *M*`,` *mtxmat* `)`
: returns a basis record for the MeatAxe module *M* with basis vectors equal to the rows of *mtxmat*.

`Dimension(` *M* `)`
: returns the dimension of the MeatAxe module *M*.

`SemiEchelonBasis(` *M* `)`
: returns a basis record of the MeatAxe module *M* that is semi-echelonized (see 33.18).

69.20 Module Functions for MeatAxe Modules

`CompositionFactors(` *M* `)`
: For a MeatAxe module *M* that is acted on by the algebra *A*, this returns a list of MeatAxe modules which are the actions of *A* on the factors of a composition series of *M*. The factors occur with same succession (and multiplicity) as in the composition series. The printed information means the following (for this example, see 69.3).

```
gap> tens:= KroneckerProduct( nm, nm );;
gap> comp:= CompositionFactors( tens );;
#I   Name Mult  SF
#I     1a    4   1
#I     4a    1   1
#I     4b    2   2
```

```
#I
#I  Ascending composition series:
#I  4a  1a  4b  1a  1a  4b  1a
```

The column with header `Name` lists the different composition factors by a name consisting of the dimension and a letter to distinguish different modules of same dimension, the `Mult` columns lists the multiplicities of the composition factor in the module, and the `SF` columns lists the exponential indices of the fields of definition in the splitting fields. In this case there is one 1-dimensional module `1a` with multiplicity 4 that is absolutely irreducible, also one 4-dimensional absolutely irreducible module `4a` of dimension 4, and with multiplicity 2 we have a 4-dimensional module `4b` that is not absolutely irreducible, with splitting field of order p^{2n} when the field of definition had order p^n.

`FixedSubmodule(` *M* `)`
: returns the submodule of fixed points in the MeatAxe module *M* under the action of the generators of *M*`.ring`.

`GeneratorsSubmodule(` *L*`,` *nr* `)`
: returns a MeatAxe matrix whose rows are a vector space basis of the *nr*-th basis of the module with submodule lattice *L*. The lattice can be computed using the `Lattice` command (see below).

`GeneratorsSubmodules(` *M* `)`
: returns a list of MeatAxe matrices, one for each submodule of the MeatAxe module *M*, whose rows are a vector space basis of the submodule. This works only if *M* is a natural module.

`IsAbsolutelyIrreducible(` *M* `)`
: returns `true` if the MeatAxe module *M* is absolutely irreducible, `false` otherwise.

`IsEquivalent(` *M1*`,` *M2* `)`
: returns `true` if the irreducible MeatAxe modules *M1* and *M2* are equivalent, and `false` otherwise. If both *M1* and *M2* are reducible, an error is signalled.

`IsIrreducible(` *M* `)`
: returns `true` if the MeatAxe module *M* is irreducible, `false` otherwise.

`KroneckerProduct(` *M1*`,` *M2* `)`
: returns the Kronecker product of the MeatAxe modules *M1*, *M2*. It is **not** checked that the acting rings are compatible.

`Lattice(` *M* `)`
: returns a list of records, each describing a component of the submodule lattice of *M*; it has the components `dimensions` (a list, at position *i* the dimension of the *i*-th submodule), `maxes` (a list, at position *i* the list of indices of the maximal submodules of submodule no. *i*), `weights` (a list of edge weights), and `XGAP` (a list used to display the submodule lattice in XGAP). **Note** that *M* must be a natural module.

`SplittingField(` *M* `)`
: returns the splitting field of the MeatAxe module *M*.

`StandardBasis(` *M*`,` *seed* `)`
: returns a standard basis record for the MeatAxe module *M*.

69.21 MeatAxe.Unbind

`MeatAxe.Unbind(` *obj1* `,` *obj2* `, ...,` *objn* `)`
`MeatAxe.Unbind(` *listofobjects* `)`

Called without arguments, this removes all files and directories constructed by calls of `MeatAxeMat` and `Group`, provided they are still notified in `MeatAxe.files`, `MeatAxe.dirs` and `MeatAxe.fields`.

Otherwise all those files in `MeatAxe.files`, `MeatAxe.dirs` and `MeatAxe.fields` are removed that are specified in the argument list.

Before leaving GAP3 after using the MeatAxe functions you should always call

```
gap> MeatAxe.Unbind();
```

69.22 MeatAxe Object Records

MeatAxe matrix records

A MeatAxe matrix in GAP3 is a record that has necessarily the components

`isMeatAxeMat`
: always true,

`isMatrix`
: always true,

`domain`
: the record `MeatAxeMatrices`,

`file`
: the name of the file that contains the matrix in MeatAxe format,

`field`
: the (finite) field the matrix is viewed over,

`dimensions`
: list containing the numbers of rows and columns,

`operations`
: the record `MeatAxeMatOps`.

Optional components are

`structure`
: algebra or group that contains the matrix,

`abstract`
: an element of a free algebra (see 40.2) representing the construction of the matrix in terms of generators.

Furthermore the record is used to store information whenever it is computed, e.g., the rank, the multiplicative order, and the inverse of a MeatAxe matrix.

MeatAxe permutation records

A MeatAxe permutation in GAP3 is a record that has necessarily the components

`isMeatAxePerm`
: always true,

`isPermutation`
: always true,

`domain`
: the record `MeatAxePermutations`,

`file`
: the name of the file that contains the permutation in MeatAxe format,

`maxpoint`
: an integer n that means that the permutation acts on the point set `[ 1 ..` n `]`

`operations`
: the record `MeatAxePermOps`.

Optional components are

`structure`
: group that contains the permutation, and

`abstract`
: an element of a free algebra (see 40.2) representing the construction of the permutation in terms of generators.

Furthermore the record is used to store information whenever it is computed, e.g., the multiplicative order, and the inverse of a MeatAxe permutation.

MeatAxe

`MeatAxe` is a record that contains information about the usage of the MeatAxe with GAP3. Currently it has the following components.

`PATH`
: the path name of the directory that contains the MeatAxe executables ,

`fields`
: a list where position i is bound if and only if the field of order i has already been constructed by the `maketab` command; in this case it contains the name of the `pxxx.zzz` file,

`files`
: a list of all file names that were constructed by calls to MeatAxe (for allowing to make clean),

`dirs`
: a list of all directory names that were constructed by calls to MeatAxe (for allowing to make clean),

`gennames`
: list of strings that are used as generator names in `abstract` components of MeatAxe matrices,

`alpha`
: alphabet over which `gennames` entries are formed,

`direc`
: directory that contains all the files that are constructed using MeatAxe functions,

`EXEC`
: function of arbitrary many string arguments that calls `Exec` for the concatenation of these arguments in the directory `MeatAxe.direc`.

`Maketab`
: function that produces field information files,

`SetDirecory`
: function that sets the `direc` component,

`TmpName`
: function of zero arguments that produces file names in the directory `MeatAxe.direc`,

`Unbind`
: function to delete files (see 69.21).

Furthermore some components are bound intermediately when MeatAxe output files are read. So you should better not use the `MeatAxe` record to store your own objects.

Field information

The correspondence between the MeatAxe numbering and the GAP3 numbering of the elements of a finite field F is given by the function `FFList` (see 39.29). The element of F corresponding to MeatAxe number n is `FFList(` F `)[` n`+1 ]`, and the MeatAxe number of the field element z is `Position( FFList(` F `),` z `) -1`.

Chapter 70

The Polycyclic Quotient Algorithm Package

This package is written by Eddie Lo. The original program is available for anonymous ftp at math.rutgers.edu. The program is an implementation of the Baumslag-Cannonito-Miller polycyclic quotient algorithm and is written in C. For more details read [BCM81b],[BCM81a], Section 11.6 of [Sim94]and [Lo96].

This package contains functions to compute the polycyclic quotients which appear in the derived series of a finitely presented group.

Currently, there are five functions implemented in this package
- `CallPCQA` (see 70.3),
- `ExtendPCQA` (see 70.4),
- `AbelianComponent` (see 70.5),
- `HirschLength` (see 70.6),
- `ModuleAction` (see 70.7).

Eddie Lo
email:hlo@math.rutgers.edu

70.1 Installing the PCQA Package

The PCQA is written in C and the package can only be installed under UNIX. It has been tested on SUNs running SunOS and on IBM PCs running FreeBSD 2.1.0. It requires the GNU multiple precision arithmetic. Make sure that this library is installed before trying to install the PCQA.

If you got a complete binary and source distribution for your machine, nothing has to be done if you want to use the PCQA for a single architecture. If you want to use the PCQA for machines with different architectures skip the extraction and compilation part of this section and proceed with the creation of shell scripts described below.

If you got a complete source distribution, skip the extraction part of this section and proceed with the compilation part below.

In the example we will assume that you, as user `gap`, are installing the PCQA package for use by several users on a network of two SUNs, called `bert` and `tiffy`, and a NeXTstation, called `bjerun`. We assume that GAP3 is also installed on these machines following the instructions given in 56.3.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

First of all you have to get the file `pcqa.zoo` (see 56.1). Then you must locate the GAP3 directories containing `lib/` and `doc/`, this is usually `gap3r4p?` where `?` is to be be replaced by the patch level.

```
gap@tiffy:~ > ls -l
drwxr-xr-x  11 gap    gap      1024 Nov  8 15:16 gap3r4p3
-rw-r--r--   1 gap    gap    106307 Jan 24 15:16 pcqa.zoo
```

Unpack the package using `unzoo` (see 56.3). Note that you must be in the directory containing `gap3r4p?` to unpack the files. After you have unpacked the source you may remove the **archive-file**.

```
gap@tiffy:~ > unzoo -x pcqa.zoo
gap@tiffy:~ > ls -l gap3r4p3/pkg/pcqa
-rw-r--r--   1 gap    gap      3697 Dec 14 15:58 Makefile
drwxr-xr-x   2 gap    gap      1024 Dec 14 15:57 bin/
drwxr-xr-x   2 gap    gap      1024 Dec 14 16:12 doc/
drwxr-xr-x   2 gap    gap      1024 Dec 15 18:28 examples/
-rw-r--r--   1 gap    gap     11819 Dec 14 13:31 init.g
drwxr-xr-x   2 gap    gap      3072 Dec 14 16:03 src/
```

Switch into the directory `src/` and type `make` to compile the PCQA. If the header files for the GNU multiple precision arithmetic are **not** in `/usr/local/include` you must set `GNUINC` to the correct directory. If the library for the GNU multiple precision arithmetic is **not** `/usr/local/lib/libgmp.a` you must set `GNULIB`. In our case we first compile the SUN version.

```
gap@tiffy:~ > cd gap3r4p3/pkg/pcqa/src
gap@tiffy:../src > make GNUINC=/usr/gnu/include \
                        GNULIB=/usr/gnu/lib/libmp.a
# you will see a lot of messages
```

If you want to use the PCQA on multiple architectures you have to move the executable to unique name.

```
gap@tiffy:../pcqa > mv bin/pcqa bin/pcqa-sun-sparc-sunos
```

Now repeat the compilation for the NeXTstation. **Do not** forget to clean up.

```
gap@tiffy:../pcqa > rlogin bjerun
gap@bjerun:~ > cd gap3r4p3/pkg/pcqa/src
gap@bjerun:../src > make clean
gap@bjerun:../src > make
# you will see a lot of messages
gap@bjerun:../src > mv bin/pcqa ../bin/pcqa-next-m68k-mach
gap@bjerun:../src > exit
gap@tiffy:../src >
```

Switch into the subdirectory `bin/` and create a script which will call the correct binary for each machine. A skeleton shell script is provided in `bin/pcqa.sh`.

```
gap@tiffy:../src > cd ..
gap@tiffy:../pcqa > cat bin/pcqa.sh
#!/bin/csh
switch ( `hostname` )
  case 'bert':
  case 'tiffy':
    exec $0-dec-mips-ultrix $* ;
    breaksw ;
  case 'bjerun':
    exec $0-next-m68k-mach $* ;
    breaksw ;
  default:
    echo "pcqa: sorry, no executable exists for this machine" ;
    breaksw ;
endsw
ctr-D
gap@tiffy:../pcqa > chmod 755 bin/pcqa
```

Now it is time to test the package.

```
gap> RequirePackage("pcqa");
gap> f := FreeGroup(2);
Group( f.1, f.2 )
gap> g := f/[f.1*f.2*f.1*f.2^-1*f.1^-1*f.2^-1];;
gap> ds := CallPCQA( g, 2 );
rec(
  isDerivedSeries := true,
  DerivedLength := 2,
  QuotientStatus := 0,
  PolycyclicPresentation := rec(
Generators := 3,
ExponentList := [ 0, 0, 0 ],
ppRelations := [ [ [ 0, 1, -1 ], [ 0, 1, 0 ] ],
                 [ [ 0, 0, 1 ] ] ],
pnRelations := [ [ [ 0, -1, 1 ], [ 0, -1, 0 ] ],
                 [ [ 0, 0, -1 ] ] ],
npRelations := [ [ [ 0, 0, 1 ], [ 0, -1, 1 ] ],
                 [ [ 0, 0, 1 ] ] ],
nnRelations := [ [ [ 0, 0, -1 ], [ 0, 1, -1 ] ],
                 [ [ 0, 0, -1 ] ] ],
PowerRelations := [  ] ),
    Homomorphisms := rec(
Epimorphism := [ [ 1, 1, 0 ], [ 1, 0, 0 ] ],
InverseMap := [ [ [ 2, 1 ] ], [ [ 3, -1 ], [ 1, 1 ] ],
                [ [ 1, 1 ], [ 3, -1 ] ] ] ),
    MembershipArray := [ 1, 3 ] )
  gap> ExtendPCQA( g, ds.PolycyclicPresentation, ds.Homomorphisms );
```

```
rec(
    QuotientStatus := 5 )
```

70.2 Input format

This package uses the finitely presented group data structure defined in GAP (see **Finitely Presented Groups**). It also defines and uses two types of data structures. One data structure defines a consistent polycyclic presentation of a polycyclic group and the other defines a homomorphism and an inverse map between the finitely presented group and its quotient.

70.3 CallPCQA

`CallPCQA(` *G, n* `)`

This function attempts to compute the quotient of a finitely presented group G by the $n+1$-st term of its derived series. A record made up of four fields is returned. The fields are **DerivedLength**, **QuotientStatus** , **PolycyclicPresentation** and **Homomorphisms** . If the quotient is not polycyclic then the field **QuotientStatus** will return a positive number. The group element represented by the module element with that positive number generates normally a subgroup which cannot be finitely generated. In this case the field **DerivedLength** will denote the biggest integer k such that the quotient of G by the $k+1$-st term in the derived series is polycyclic. The appropriate polycyclic presentation and maps will be returned. If the field **QuotientStatus** returns -1, then for some number $k < n$, the k-th term of the derived series is the same as the $k+1$-st term of the derived series. In the remaining case **QuotientStatus** returns 0.

The field **PolycyclicPresentation** is a record made up of seven fields. The various conjugacy relations are stored in the fields **ppRelations** , **pnRelations**, **npRelations** and **nnRelations**. Each of these four fields is an array of exponent sequences which correspond to the appropriate left sides of the conjugacy relations . If $a_1, a_2, ..., a_n$ denotes the polycyclic generators and $A_1, A_2, ..., A_n$ their respective inverses, then the field **ppRelations** stores the relations of the form $a_j^{a_i}$ with $i < j$, **pnRelations** stores the relations of the form $A_j^{a_i}$, **npRelations** stores the relations of the form $a_j^{A_i}$ and **nnRelations** stores the relations of the form $A_j^{A_i}$. The positive and negative power relations are stored together similarly in the field **PowerRelations**. The field **Generators** denotes the number of polycyclic generators in the presentation and the field **ExponentList** contains the exponent of the power relations. If there is no power relation involving a generator,then the corresponding entry in the **ExponentList** is equal to 0.

The field **Homomorphisms** consists of a homomorphism from the finitely presented group to the polycyclic group and an inverse map backward. The field **Epimorphism** stores the image of the generators of the finitely presented group as exponent sequences of the polycyclic group . The field **InverseMap** stores a preimage of the polycyclic generators as a word in the finitely presented group.

```
gap> F := FreeGroup(2);
Group( f.1, f.2 )
gap> G := F/[F.1*F.2*F.1*F.2^-1*F.1^-1*F.2^-1];
```

```
Group( f.1, f.2 )
gap> ans := CallPCQA(G,2);
rec(
  DerivedLength := 2,
  QuotientStatus := 0,
  PolycyclicPresentation := rec(
      Generators := 3,
      ExponentList := [ 0, 0, 0 ],
      ppRelations := [ [ [ 0, 1, -1 ], [ 0, 1, 0 ] ],
               [ [ 0, 0, 1 ] ] ],
      pnRelations := [ [ [ 0, -1, 1 ], [ 0, -1, 0 ] ],
               [ [ 0, 0, -1 ] ] ],
      npRelations := [ [ [ 0, 0, 1 ], [ 0, -1, 1 ] ],
               [ [ 0, 0, 1 ] ] ],
      nnRelations := [ [ [ 0, 0, -1 ], [ 0, 1, -1 ] ],
               [ [ 0, 0, -1 ] ] ],
      PowerRelations := [  ] ),
  Homomorphisms := rec(
      Epimorphism := [ [ 1, 1, 0 ], [ 1, 0, 0 ] ],
      InverseMap := [ [ [ 2, 1 ] ], [ [ 3, -1 ], [ 1, 1 ] ],
              [ [ 1, 1 ], [ 3, -1 ] ] ] ),
  MembershipArray := [ 1, 3 ] )
```

70.4 ExtendPCQA

`ExtendPCQA(` *G, CPP, HOM, m, n* `)`

This function takes as input a finitely presented group G, a consistent polycyclic presentation CPP (70.3) of a polycyclic quotient G/N of G, an epimorphism and an inverse map as in the field **Homomorphisms** in 70.3. It determines whether the quotient $G/[N,N]$ is polycyclic and returns the flag **QuotientStatus** . It also returns the polycyclic presentation and the appropriate homomorphism and map if the quotient is polycyclic.

When the parameter m is a positive number the quotient $G/[N,N]N^m$ is computed. When it is a negative number, and if $K/[N,N]$ is the torsion part of $N/[N,N]$, then the quotient $G/[N,N]K$ is computed. The default case is when $m = 0$. If there are only three arguments in the function call, m will be taken to be zero.

When the parameter n is a nonzero number, the quotient $G/[N,G]$ is computed instead. Otherwise the quotient $G/[N,N]$ is computed. If this argument is not assigned by the user, then n is set to zero. Different combinations of m and n give different quotients. For example, when **ExtendPCQA** is called with $m = 6$ and $n = 1$,the quotient $G/[N,G]N^6$ is computed.

```
gap> ExtendPCQA(G,ans.PolycyclicPresentation,ans.Homomorphisms);
rec(
   QuotientStatus := 5 )
gap> ExtendPCQA(G,ans.PolycyclicPresentation,ans.Homomorphisms,6,1);
rec(
  QuotientStatus := 0,
```

```
PolycyclicPresentation := rec(
  Generators := 4,
  ExponentList := [ 0, 0, 0, 6 ],
  ppRelations := [ [[ 0, 1, -1, 0 ],[ 0, 1, 0, 0 ],[ 0, 0, 0, 1 ]],
                   [[ 0, 0, 1, 1 ],[ 0, 0, 0, 1 ]],
                   [[ 0, 0, 0, 1 ]] ],
  pnRelations := [ [[ 0, -1, 1, 5 ],[ 0, -1, 0, 0 ],[ 0, 0, 0, 5]],
                   [[ 0, 0, -1, 5 ],[ 0, 0, 0, 5 ]],
                   [[ 0, 0, 0, 5 ]] ],
  npRelations := [ [[ 0, 0, 1, 0 ],[ 0, -1, 1, 0 ],[ 0, 0, 0, 1 ]],
                   [[ 0, 0, 1, 5 ],[ 0, 0, 0, 1 ]],
                   [[ 0, 0, 0, 1 ]] ],
  nnRelations := [ [[ 0, 0, -1, 0 ],[ 0, 1, -1, 5 ],[ 0, 0, 0, 5]],
                   [[ 0, 0, -1, 1 ],[ 0, 0, 0, 5 ]],
                   [[ 0, 0, 0, 5 ]] ],
  PowerRelations := [ ,,,,,, [ 0, 0, 0, 0 ], [ 0, 0, 0, 5 ] ] ),
Homomorphisms := rec(
  Epimorphism := [ [ 1, 1, 0, 0 ], [ 1, 0, 0, 0 ] ],
  InverseMap :=
    [ [[ 2, 1 ]], [[ 3, -1 ],[ 1, 1 ]], [[ 1, 1 ],[ 3, -1 ]],
      [[ 5, -1 ],[ 4, -1 ],[ 5, 1 ],[ 4, 1 ]] ] ),
Next := 4 )
```

70.5 AbelianComponent

`AbelianComponent(` *QUOT* `)`

This function takes as input the output of a **CallPCQA** function call (see 70.3) or an **ExtendPCQA** function call (see 70.4) and returns the structure of the abelian groups which appear as quotients in the derived series. The structure of each of these quotients is given by an array of nonnegative integers.Read the section on **ElementaryDivisors** for details.

```
gap> F := FreeGroup(3);
Group( f.1, f.2, f.3 )
gap> G := F/[F.1*F.2*F.1*F.2,F.2*F.3^2*F.2*F.3,F.3^6];
Group( f.1, f.2, f.3 )
gap> quot := CallPCQA(G,2);;
gap> AbelianComponent(quot);
[ [ 1, 2, 12 ], [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
```

70.6 HirschLength

`HirschLength(` *CPP* `)`

This function takes as input a consistent polycyclic presentation (see 70.3) and returns the Hirsch length of the group presented.

```
gap> HirschLength(quot.PolycyclicPresentation);
11
```

70.7 ModuleAction

`ModuleAction(` *QUOT* `)`

This function takes as input the output of a **CallPCQA** function call (see 70.3) or an **ExtendPCQA** function call (see 70.4). If the quotient $G/[N,N]$ returned by the function call is polycyclic then **ModuleAction** computes the action of the polycyclic generators corresponding to G/N on the polycyclic generators of $N/[N,N]$. The result is returned as an array of matrices. Notice that the Smith normal form of $G/[N,N]$ is returned by the function **CallPCQA** as part of the polycyclic presentation.

```
gap> ModuleAction(quot);
[ [ [ 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0 ],
    [ 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1 ],
    [ 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0 ],
    [ 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0 ],
    [ 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0 ] ],
  [ [ -1, 0, -1, -1, -1, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
    [ 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1 ],
    [ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ],
    [ 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ] ],
  [ [ 1, -1, 0, 0, 0, 0, 0, 0, 0, 1, 0 ],
    [ 0, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0 ],
    [ 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, -1, -1, -1, -1, -1 ],
    [ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 ] ] ]
```

Chapter 71

Sisyphos

This chapter describes the GAP3 accessible functions of the SISYPHOS (Version 0.6) share library package for computing with modular group algebras of p-groups, namely a function to convert a p-group into SISYPHOS readable format (see 71.2), several functions that compute automorphism groups of p-groups (see 71.4), functions that compute normalized automorphism groups as polycyclically presented groups (see 71.5, 71.6), functions that test two p-groups for isomorphism (see 71.7) and compute isomorphisms between p-groups (see 71.8), and a function to compute the element list of an automorphism group that is given by generators (see 71.10).

The SISYPHOS functions for group rings are not yet available, with the only exception of a function that computed the group of normalized units (see 71.11).

The algorithms require presentations that are compatible with a characteristic series of the group with elementary abelian factors, e.g. the p-central series. If necessary such a presentation is computed secretly using the p-central series, the computations are done using this presentation, and then the results are carried back to the original presentation. The check of compatibility is done by the function `IsCompatiblePCentralSeries` (see 71.3). The component `isCompatiblePCentralSeries` of the group will be either `true` or `false` then. If you know in advance that your group is compatible with a series of the kind required, e.g. the Jennings-series, you can avoid the check by setting this flag to `true` by hand.

Before using any of the functions described in this chapter you must load the package by calling the statement

```
gap> RequirePackage( "sisyphos" );
```

71.1 PrintSISYPHOSWord

`PrintSISYPHOSWord(` *P*`,` *a* `)`

For a polycyclically presented group P and an element a of P, `PrintSISYPHOSWord(` *P* `,`*a* `)` prints a string that encodes a in the input format of the SISYPHOS system.

The string `"1"` means the identity element, the other elements are products of powers of generators, the i-th generator is given the name `g`*i*.

```
gap> g := SolvableGroup ( "D8" );;
```

```
gap> PrintSISYPHOSWord ( g, g.2*g.1 ); Print( "\n" );
g1*g2*g3
```

71.2 PrintSisyphosInputPGroup

`PrintSisyphosInputPGroup(` *P*`,` *name*`,` *type* `)`

prints the presentation of the finite p-group P in a format readable by the SISYPHOS system. P must be a polycyclically or freely presented group.

In SISYPHOS, the group will be named *name*. If P is polycyclically presented the i-th generator gets the name `g`i. In the case of a free presentation the names of the generators are not changed; note that SISYPHOS accepts only generators names beginning with a letter followed by a sequence of letters, digits,underscores and dots.

type must be either `"pcgroup"` or the prime dividing the order of P. In the former case the SISYPHOS object has type `pcgroup`, P must be polycyclically presented for that. In the latter case a SISYPHOS object of type `group` is created. For avoiding computations in freely presented groups, is **neither** checked that the presentation describes a p-group, **nor** that the given prime really divides the group order.

See the SISYPHOS manual [Wur93] for details.

```
gap> g:= SolvableGroup( "D8" );;
gap> PrintSisyphosInputPGroup( g, "d8", "pcgroup" );
d8 = pcgroup(2,
gens(
g1,
g2,
g3),
rels(
g1^2 = 1,
g2^2 = 1,
g3^2 = 1,
[g2,g1] = g3));
gap> q8 := FreeGroup ( 2 );;
gap> q8.relators := [q8.1^4,q8.2^2/q8.1^2,Comm(q8.2,q8.1)/q8.1^2];;
gap> PrintSisyphosInputPGroup ( q8, "q8", 2 );
#I  PQuotient: class 1 : 2
#I  PQuotient: Runtime : 0
q8 = group (minimal,
2,
gens(
f.1,
f.2),
rels(
f.1^4,
f.2^2*f.1^-2,
f.2^-1*f.1^-1*f.2*f.1^-1));
```

71.3 IsCompatiblePCentralSeries

`IsCompatiblePCentralSeries(` G `)`

If the component G`.isCompatiblePCentralSeries` of the polycyclically presented p-group G is bound, its value is returned, otherwise the exponent-p-central series of G is computed and compared to the given presentation. If the generators of each term of this series form a subset of the generators of G the component G`.isCompatiblePCentralSeries` is set to `true`, otherwise to `false`. This value is then returned by the function.

```
gap> g:= SolvableGroup( "D8" );;
gap> IsCompatiblePCentralSeries ( g );
true
gap> a := AbstractGenerators ( "a", 5 );;
gap> h := AgGroupFpGroup ( rec (
> generators := a,
> relators :=
> [a[1]^2/(a[3]*a[5]),a[2]^2/a[3],a[3]^2/(a[4]*a[5]),a[4]^2,a[5]^2]));;
gap> h.name := "H";;
gap> IsCompatiblePCentralSeries ( h );
false
gap> PCentralSeries ( h, 2 );
[ H, Subgroup( H, [ a3, a4, a5 ] ), Subgroup( H, [ a4*a5 ] ),
  Subgroup( H, [  ] ) ]
```

71.4 SAutomorphisms

`SAutomorphisms(` P `)`
`OuterAutomorphisms(` P `)`
`NormalizedAutomorphisms(` P `)`
`NormalizedOuterAutomorphisms(` P `)`

all return a record with components

`sizeOutG`
: the size of the group of outer automorphisms of P,

`sizeInnG`
: the size of the group of inner automorphisms of P,

`sizeAutG`
: the size of the full automorphism group of P,

`generators`
: a list of group automorphisms that generate the group of all, outer, normalized or normalized outer automorphisms of the polycyclically presented p-group P, respectively. In the case of outer or normalized outer automorphisms, this list consists of preimages in $Aut(P)$ of a generating set for $Aut(P)/Inn(P)$ or $Aut_n(P)/Inn(P)$, respectively.

```
gap> g:= SolvableGroup( "Q8" );;
gap> SAutomorphisms( g );
rec(
```

```
sizeAutG := 24,
sizeInnG := 4,
sizeOutG := 6,
generators :=
[ GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ b, a, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a, b*c, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*c, b, c ] ) ] )
gap> OuterAutomorphisms( g );
rec(
sizeAutG := 24,
sizeInnG := 4,
sizeOutG := 6,
generators :=
[ GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ b, a, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b, b, c ] ) ] )
```

Note: If the component P`.isCompatiblePCentralSeries` is not bound it is computed using `IsCompatiblePCentralSeries`.

71.5 AgNormalizedAutomorphisms

`AgNormalizedAutomorphisms(` *P* `)`

returns a polycyclically presented group isomorphic to the group of all normalized automorphisms of the polycyclically presented p-group P.

```
gap> g:= SolvableGroup( "D8" );;
gap> aut:= AgNormalizedAutomorphisms( g );
Group( g0, g1 )
gap> Size( aut );
4
```

Note: If the component P`.isCompatiblePCentralSeries` is not bound it is computed using `IsCompatiblePCentralSeries`.

71.6 AgNormalizedOuterAutomorphisms

`AgNormalizedOuterAutomorphisms(` *P* `)`

returns a polycyclically presented group isomorphic to the group of normalized outer automorphisms of the polycyclically presented p-group P.

```
gap> g:= SolvableGroup( "D8" );;
gap> aut:= AgNormalizedOuterAutomorphisms( g );
Group( IdAgWord )
```

Note: If the component P`.isCompatiblePCentralSeries` is not bound it is computed using `IsCompatiblePCentralSeries`.

71.7 IsIsomorphic

`IsIsomorphic(` *P1*`,` *P2* `)`

returns `true` if the polycyclically or freely presented p-group *P1* and the polycyclically presented p-group *P2* are isomorphic, `false` otherwise.

```
gap> g:= SolvableGroup( "D8" );;
gap> nonab:= AllTwoGroups( Size, 8, IsAbelian, false );
[ Group( a1, a2, a3 ), Group( a1, a2, a3 ) ]
gap> List( nonab, x -> IsIsomorphic( g, x ) );
[ true, false ]
```

(The function `Isomorphisms` returns isomorphisms in case the groups are isomorphic.)

Note: If the component *P2*`.isCompatiblePCentralSeries` is not bound it is computed using `IsCompatiblePCentralSeries`.

71.8 Isomorphisms

`Isomorphisms(` *P1* `,` *P2* `)`

If the polycyclically or freely presented p-groups *P1* and the polycyclically presented p-group *P2* are not isomorphic, `Isomorphisms` returns `false`. Otherwise a record is returned that encodes the isomorphisms from *P1* to *P2*; its components are

`epimorphism`
: a list of images of *P1*`.generators` that defines an isomorphism from *P1* to *P2*,

`generators`
: a list of image lists which encode automorphisms that together with the inner automorphisms generate the full automorphism group of *P2*

`sizeOutG`
: size of the group of outer automorphisms of *P2*,

`sizeInnG`
: size of the group of inner automorphisms of *P2*,

`sizeOutG`
: size of the full automorphism group of *P2*.

```
gap> g:= SolvableGroup( "Q8" );;
gap> nonab:= AllTwoGroups( Size, 8, IsAbelian, false );
[ Group( a1, a2, a3 ), Group( a1, a2, a3 ) ]
gap> nonab[2].name:= "im";;
gap> Isomorphisms( g, nonab[2] );
rec(
sizeAutG := 24,
sizeInnG := 4,
sizeOutG := 6,
epimorphism := [ a1, a2, a3 ],
generators :=
[ GroupHomomorphismByImages( im, im, [ a1, a2, a3 ], [ a2, a1, a3 ] ),
  GroupHomomorphismByImages( im, im, [ a1, a2, a3 ], [ a1*a2, a2, a3
     ] ) ] )
```

(The function `IsIsomorphic` tests for isomorphism of p-groups.)

Note: If the component *P2*`.isCompatiblePCentralSeries` is not bound it is computed using `IsCompatiblePCentralSeries`.

71.9 CorrespondingAutomorphism

`CorrespondingAutomorphism(` G`,` w `)`

If G is a polycyclically presented group of automorphisms of a group P as returned by `AgNormalizedAutomorphisms` (see 71.5) or `AgNormalizedOuterAutomorphisms` (see 71.6), and w is an element of G then the automorphism of P corresponding to w is returned.

```
gap> g:= TwoGroup( 64, 173 );;
gap> g.name := "G173";;
gap> autg := AgNormalizedAutomorphisms ( g );
Group( g0, g1, g2, g3, g4, g5, g6, g7, g8 )
gap> CorrespondingAutomorphism ( autg, autg.2*autg.1^2 );
GroupHomomorphismByImages( G173, G173, [ a1, a2, a3, a4, a5, a6 ],
[ a1, a2*a4, a3*a6, a4*a6, a5, a6 ] )
```

71.10 AutomorphismGroupElements

`AutomorphismGroupElements(` A `)`

A must be an automorphism record as returned by one of the automorphism routines or a list consisting of automorphisms of a p-group P.

In the first case a list of all elements of $Aut(P)$ or $Aut_n(P)$ is returned, if A has been created by `SAutomorphisms` or `NormalizedAutomorphisms` (see 71.4), respectively, or a list of coset representatives of $Aut(P)$ or $Aut_n(P)$ modulo $Inn(P)$, if A has been created by `OuterAutomorphisms` or `NormalizedOuterAutomorphisms` (see 71.4), respectively.

In the second case the list of all elements of the subgroup of $Aut(P)$ generated by A is returned.

```
gap> g:= SolvableGroup( "Q8" );;
gap> outg:= OuterAutomorphisms( g );;
gap> AutomorphismGroupElements( outg );
[ GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ b, a, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b*c, a, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ b, a*b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a, a*b*c, c ] ) ]
gap> l:= [ outg.generators[2] ];
[ GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b, b, c ] ) ]
gap> AutomorphismGroupElements( l );
[ GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*c, b, c ] ),
  GroupHomomorphismByImages( Q8, Q8, [ a, b, c ], [ a*b*c, b, c ] ) ]
```

71.11 NormalizedUnitsGroupRing

`NormalizedUnitsGroupRing(` P `)`
`NormalizedUnitsGroupRing(` P`,` n `)`

When called with a polycyclicly presented p-group P, the group of normalized units of the group ring FP of P over the field F with p elements is returned.

If a second argument n is given, the group of normalized units of FP/I^n is returned, where I denotes the augmentation ideal of FP.

The returned group is represented as polycyclicly presented group.

```
gap> g:= SolvableGroup( "D8" );;
gap> NormalizedUnitsGroupRing( g, 1 );
#D use multiplication table
Group( IdAgWord )
gap> NormalizedUnitsGroupRing( g, 2 );
#D use multiplication table
Group( g1, g2 )
gap> NormalizedUnitsGroupRing( g, 3 );
#D use multiplication table
Group( g1, g2, g3, g4 )
gap> NormalizedUnitsGroupRing( g );
#D use multiplication table
Group( g1, g2, g3, g4, g5, g6, g7 )
```

Chapter 72

Decomposition numbers of Hecke algebras of type A

This package contains functions for computing the decomposition matrices for Iwahori–Hecke algebras of the symmetric groups. As the (modular) representation theory of these algebras closely resembles that of the (modular) representation theory of the symmetric groups — indeed, the later is a special case of the former — many of the combinatorial tools from the representation theory of the symmetric group are included in the package.

These programs grew out of the attempts by Gordon James and myself [JM1] to understand the decomposition matrices of Hecke algebras of type **A** when $q = -1$. The package is now much more general and its highlights include:

1. Specht provides a means of working in the Grothendieck ring of a Hecke algebra H using the three natural bases corresponding to the Specht modules, projective indecomposable modules, and simple modules.

2. For Hecke algebras defined over fields of characteristic zero we have implemented the algorithm of Lascoux, Leclerc, and Thibon [LLT] for computing decomposition numbers and "crystallized decomposition matrices". In principle, this gives all of the decomposition matrices of Hecke algebras defined over fields of characteristic zero.

3. We provide a way of inducing and restricting modules. In addition, it is possible to "induce" decomposition matrices; this function is quite effective in calculating the decomposition matrices of Hecke algebras for small n.

4. The q–analogue of Schaper's theorem [JM2] is included, as is Kleshchev's [K] algorithm of calculating the Mullineux map. Both are used extensively when inducing decomposition matrices.

5. Specht can be used to compute the decomposition numbers of q–Schur algebras (and the general linear groups), although there is less direct support for these algebras. The decomposition matrices for the q–Schur algebras defined over fields of characteristic zero for $n < 11$ and all e are included in Specht.

6. The Littlewood–Richard rule, its inverse, and functions for many of the standard operations on partitions (such as calculating cores, quotients, and adding and removing hooks), are included.

7. The decomposition matrices for the symmetric groups S_n are included for $n < 15$ and for all primes.

The modular representation theory of Hecke algebras

The "modular" representation theory of the Iwahori–Hecke algebras of type **A** was pioneered by Dipper and James [DJ1,DJ2]; here we briefly outline the theory, referring the reader to the references for details. The definition of the Hecke algebra can be found in Chapter 91; see also 90.1.

Given a commutative integral domain R and a non–zero unit q in R, let $H = H_{R,q}$ be the Hecke algebra of the symmetric group S_n on n symbols defined over R and with parameter q. For each partition μ of n, Dipper and James defined a **Specht module** `S`(μ). Let `rad` `S`(μ) be the radical of `S`(μ) and define `D`(μ)=`S`(μ)/`rad` `S`(μ). When R is a field, `D`(μ) is either zero or absolutely irreducible. Henceforth, we will always assume that `R` is a field.

Given a non–negative integer i, let $[i]_q = 1 + q + \ldots + q^{i-1}$. Define e to be the smallest non–negative integer such that $[e]_q = 0$; if no such integer exists, we set e equal to 0. *Many of the functions in this package depend upon e;* the integer e is the Hecke algebras analogue of the characteristic of the field in the modular representation theory of finite groups.

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is e–**singular** if there exists an integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$; otherwise, μ is e–**regular**. Dipper and James [DJ1] showed that `D`$(\nu)\neq(0)$ if and only if ν is e–regular and that the `D`(ν) give a complete set of non–isomorphic irreducible H–modules as ν runs over the e–regular partitions of n. Further, `S`(μ) and `S`(ν) belong to the same block if and only if μ and ν have the same e-core [DJ2,JM2]. Note that these results depend only on e and not directly on R or q.

Given two partitions μ and ν, where ν is e–regular, let $d_{\mu\nu}$ be the composition multiplicity of `D`(ν) in `S`(μ). The matrix $D = (d_{\mu\nu})$ is the **decomposition matrix** of `H`. When the rows and columns are ordered in a way compatible with dominance, D is lower unitriangular.

The indecomposable H-modules `P`(ν) are indexed by e-regular partitions ν. By general arguments, `P`(ν) has the same composition factors as $\sum_\mu d_{\mu\nu}$`S`(μ); so these linear combinations of modules become identified in the Grothendieck ring of `H`. Similarly, `D`$(\nu) = \sum_\mu d^{-1}_{\nu\mu}$`S`$(\mu)$ in the Grothendieck ring. These observations are the basis for many of the computations in SPECHT.

Two small examples

Because of the algorithm of [LLT], in principle, all of decomposition matrices for all Hecke algebras defined over fields of characteristic zero are known and available using SPECHT. The algorithm is recursive; however, it is quite quick and, as with a car, you need never look at the engine:

```
gap> H:=Specht(4);     # e=4, R a field of characteristic 0
Specht(e=4, S(), P(), D(), Pq())
gap> InducedModule(H.P(12,2));
P(13,2)+P(12,3)+P(12,2,1)+P(10,3,2)+P(9,6)
```

The [LLT] algorithm was applied 24 times during this calculation.

For Hecke algebras defined over fields of positive characteristic the major tool provided by SPECHT, apart from the decomposition matrices contained in the libraries, is a way of "inducing" decomposition matrices. This makes it fairly easy to calculate the associated decomposition matrices for "small" n. For example, the SPECHT libraries contain the decomposition matrices for the symmetric groups S_n over fields of characteristic 3 for $n < 15$. These matrices were calculated by SPECHT using the following commands:

```
gap> H:=Specht(3,3);     # e=3, R field of characteristic 3
Specht(e=3, p=3, S(), P(), D())
gap> d:=DecompositionMatrix(H,5);  # known for n < 2e
5     | 1
4,1   | . 1
3,2   | . 1 1
3,1^2 | . . . 1
2^2,1 | 1 . . . 1
2,1^3 | . . . . 1
1^5   | . . 1 . .
gap> for n in [6..14] do
>        d:=InducedDecompositionMatrix(d); SaveDecompositionMatrix(d);
>    od;
```

The function `InducedDecompositionMatrix` contains almost every trick that I know for computing decomposition matrices (except using the spin groups). I would be very happy to hear of any improvements.

SPECHT can also be used to calculate the decomposition numbers of the q–Schur algebras; although, as yet, here no additional routines for calculating the projective indecomposables indexed by e–singular partitions. Such routines will probably be included in a future release, together with the (conjectural) algorithm [LT] for computing the decomposition matrices of the q–Schur algebras over fields of characteristic zero.

In the next release of SPECHT, I will also include functions for computing the decomposition matrices of Hecke algebras of type **B**, and more generally those of the Ariki–Koike algebras. As with the Hecke algebra of type **A**, there is an algorithm for computing the decomposition matrices of these algebras when `R` is a field of characteristic zero [M].

Credits

I would like to thank Gordon James, Johannes Lipp, and Klaus Lux for their comments and suggestions.

If you find SPECHT useful please let me know. I would also appreciate hearing any suggestions, comments, or improvements. In addition, if SPECHT does play a significant role in your research, please send me a copy of the paper(s) and please cite SPECHT in your references.

The lastest version of SPECHT can be obtained from http://maths.usyd.edu.au:8000/u/mathas/specht.

Andrew Mathas[1]
mathas@maths.usyd.edu.au
University of Sydney, 1997.

References

[A] S. Ariki, *On the decomposition numbers of the Hecke algebra of $G(m, 1, n)$*, J. Math. Kyoto Univ., **36** (1996), 789–808.

[1]Supported in part by SERC grant GR/J37690

[B] J. Brundan, *Modular branching rules for quantum GL_n and the Hecke algebra of type* **A**, Proc. London Math. Soc (3), to appear.

[DJ1] R. Dipper and G. James, *Representations of Hecke algebras of general linear groups*, Proc. London Math. Soc. (3), **52** (1986), 20–52.

[DJ2] R. Dipper and G. James, *Blocks and idempotents of Hecke algebras of general linear groups*, Proc. London Math. Soc. (3), **54** (1987), 57–82.

[G] M. Geck, *Brauer trees of Hecke algebras*, Comm. Alg., **20** (1992), 2937–2973.

[Gr] I. Grojnowski, *Affine Hecke algebras (and affine quantum GL_n) at roots of unity*, IMRN **5** (1994), 215–217.

[J] G. James, *The decomposition matrices of $GL_n(q)$ for $n \leq 10$*, Proc. London Math. Soc., **60** (1990), 225–264.

[JK] G. James and A. Kerber, *The representation theory of the symmetric group*, **16**, Encyclopedia of Mathematics, Addison–Wesley, Massachusetts (1981).

[JM1] G. James and A. Mathas, *Hecke algebras of type* **A** *at $q = -1$*, J. Algebra, **184** (1996), 102–158.

[JM2] G. James and A. Mathas, *A q–analogue of the Jantzen–Schaper Theorem*, Proc. London Math. Soc. (3), **74**, 1997, 241–274.

[K] A. Kleshchev, *Branching rules for modular representations III*, J. London Math. Soc., **54**, 1996, 25–38.

[LLT] A. Lascoux, B. Leclerc, and J-Y. Thibon, *Hecke algebras at roots of unity and crystal bases of quantum affine algebras*, Comm. Math. Phys., **181** (1996), 205–263.

[LT] B. Leclerc and J-Y. Thibon, *Canonical bases and q–deformed Fock spaces*, Int. Research Notices **9** (1996), 447–456.

[M] A. Mathas, *Canonical bases and the decomposition matrices of Ariki–Koike algebras*, preprint 1996.

72.1 Specht

`Specht(`e`)`
`Specht(`e`, `p`)`
`Specht(`e`, `p`, `*val* `[,`*HeckeRing*`])`

Let R be a field of characteristic 0, q a non–zero element of R, and let e be the smallest positive integer such that

$$1 + q + \ldots + q^{e-1} = 0$$

(we set $e = 0$ if no such integer exists). The record returned by `Specht(`e`)` allows calculations in the Grothendieck rings of the Hecke algebras `H` of type **A** which are defined over R and have parameter q. (The Hecke algebra is described in Chapter 91; see also `Hecke` 90.1.) Below we also describe how to consider Hecke algebras defined over fields of positive characteristic.

`Specht` returns a record which contains, among other things, functions `S` , `P`, and `D` which correspond to the Specht modules, projective indecomposable modules, and the simple modules for the family of Hecke algebras determined by R and q. SPECHT allows manipulation of arbitrary linear combinations of these "modules", as well as a way of inducing and restricting them, "multiplying" them, and converting between these three natural bases of the Grothendieck ring. Multiplication of modules corresponds to taking a tensor product, and then inducing (thus giving a module for a larger Hecke algebra).

```
gap> RequirePackage("specht"); H:=Specht(5);
Specht(e=5, S(), P(), D(), Pq())
gap> H.D(3,2,1);
D(3,2,1)
gap> H.S( last );
S(6)-S(4,2)+S(3,2,1)
gap> InducedModule(H.P(3,2,1));
P(4,2,1)+P(3,3,1)+P(3,2,2)+2*P(3,2,1,1)
gap> H.S(last);
S(4,2,1)+S(3,3,1)+S(3,2,2)+2*S(3,2,1,1)+S(2,2,2,1)+S(2,2,1,1,1)
gap> H.D(3,1)*H.D(3);
D(7)+2*D(6,1)+D(5,2)+D(5,1,1)+2*D(4,3)+D(4,2,1)+D(3,3,1)
gap> RestrictedModule(last);
4*D(6)+3*D(5,1)+5*D(4,2)+2*D(4,1,1)+2*D(3,3)+2*D(3,2,1)
gap> H.S(last);
S(6)+3*S(5,1)+3*S(4,2)+2*S(4,1,1)+2*S(3,3)+2*S(3,2,1)
gap> H.P(last);
P(6)+3*P(5,1)+2*P(4,2)+2*P(4,1,1)+2*P(3,3)
```

The way in which the partitions indexing the modules are printed can be changed using `SpechtPrettyPrint` 72.57.

There is also a function `Schur` 72.2 for doing calculations with the q–Schur algebra. See `DecompositionMatrix` 72.3, and `CrystalizedDecompositionMatrix` 72.4.

This function requires the package "specht" (see 57.1).

The functions `H.S`, `H.P`, and `H.D`

The functions `H.S`, `H.P`, and `H.D` return records which correspond to Specht modules, projective indecomposable modules, and simple modules respectively. Each of these three functions can be called in four different ways, as we now describe.

`H.S`(μ)   `H.P`(μ)   `H.D`(μ)

In the first form, μ is a partition (either a list, or a sequence of integers), and the corresponding Specht module, PIM, or simple module (respectively), is returned.

```
gap> H.P(4,3,2);
P(4,3,2)
```

`H.S`(x)   `H.P`(x)   `H.D`(x)

Here, x is an H–module. In this form, `H.S` rewrites x as a linear combination of Specht modules, if possible. Similarly, `H.P` and `H.D` rewrite x as a linear combination of PIMs and simple modules respectively. These conversions require knowledge of the relevant decomposition matrix of H; if this is not known then `false` is returned (over fields of characteristic zero, all of the decomposition matrices are known via the algorithm of [LLT]; various other decomposition matrices are included with SPECHT). For example, `H.S(H.P(`μ`))` returns

$$\sum_{\nu} d_{\nu\mu} \mathtt{S}(\nu),$$

or `false` if some of these decomposition multiplicities are not known.

```
gap> H.D( H.P(4,3,2) );
D(5,3,1)+2*D(4,3,2)+D(2,2,2,2,1)
gap> H.S( H.D( H.S(1,1,1,1,1) ) );
-S(5)+S(4,1)-S(3,1,1)+S(2,1,1,1)
```

As the last example shows, SPECHT does not always behave as expected. The reason for this is that Specht modules indexed by e–singular partitions can always be written as a linear combination of Specht modules which involve only e–regular partitions. As such, it is not always clear when two elements are equal in the Grothendieck ring. Consequently, to test whether two modules are equal you should first rewrite both modules in the `D`–basis; this is *not* done by SPECHT because it would be very inefficient.

`H.S(`d, μ`)` `H.P(`d, μ`)` `H.D(`d, μ`)`

In the third form, d is a decomposition matrix and μ is a partition. This is useful when you are trying to calculate a new decomposition matrix d because it allows you to do calculations using the known entries of d to deduce information about the unknown ones. When used in this way, `H.P` and `H.D` use d to rewrite `P(`μ`)` and `D(`μ`)` respectively as a linear combination of Specht modules, and `H.S` uses d to write `S(`μ`)` as a linear combination of simple modules. If the values of the unknown entries in d are needed, `false` is returned.

```
gap> H:=Specht(3,3);    # e = 3, p = 3 = characteristic of R
Specht(e=3, p=3, S(), P(), D())
gap>  d:=InducedDecompositionMatrix(DecompositionMatrix(H,14));;
# Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
gap> H.P(d,4,3,3,2,2,1);
S(4,3,3,2,2,1)+S(4,3,3,2,1,1,1)+S(4,3,2,2,2,1,1)+S(3,3,3,2,2,1,1)
gap> H.S(d,7, 3, 3, 2);
D(11,2,1,1)+D(10,3,1,1)+D(8,5,1,1)+D(8,3,3,1)+D(7,6,1,1)+D(7,3,3,2)
gap> H.D(d,14,1);
false
```

The final example returned `false` because the partitions `(14,1)` and `(15)` have the same 3–core (and `P(15)` is missing from d).

`H.S(`d, x`)` `H.P(`d, x`)` `H.D(`d, x`)`

In the final form, d is a decomposition matrix and x is a module. All three functions rewrite x in their respective basis using d. Again this is only useful when you are trying to calculate a new decomposition matrix because, for any "known" decomposition matrix d, `H.S(`x`)` and `H.S(`d, x`)` are equivalent (and similarly for `H.P` and `H.D`).

```
gap> H.S(d, H.D(d,10,5) );
```

```
-S(13,2)+S(10,5)
```

Decomposition numbers of the symmetric groups

The last example looked at Hecke algebras with parameter q=1 and R a field of characteristic 3 (so e=3); that is, the group algebra of the symmetric group over a field of characteristic 3. More, generally, the command `Specht(`p`, `p`)` can be used to consider the group algebras of the symmetric groups over fields of characteristic p (i.e. e=p, and `R` a field of characteristic p).

For example, the dimensions of the simple modules of S_6 over fields of characteristic 5 can be computed as follows:

```
gap> H:=Specht(5,5);; SimpleDimension(H,6);
6       : 1
5,1     : 5
4,2     : 8
4,1^2   : 10
3^2     : 5
3,2,1   : 8
3,1^3   : 10
2^3     : 5
2^2,1^2 : 1
2,1^4   : 5
```

Hecke algebras over fields of positive characteristic

To consider Hecke algebras defined over arbitrary fields `Specht` must also be supplied with a **valuation map** *val* as an argument. The function *val* is a map from some PID into the natural numbers; at present it is needed only by functions which rely (at least implicitly), upon the q–analogue of Schaper's theorem. In general, *val* depends upon q and the characteristic of R; full details can be found in [JM2].

Over fields of characteristic zero, and in the symmetric group case, the function *val* is automatically defined by `Specht`. When R is a field of characteristic zero, $val([i]_q)$ is 1 if e divides i and 0 otherwise (this is the valuation map associated to the prime ideal in $\mathbf{C}[v]$ generated by the e–th cyclotomic polynomial). When $q = 1$ and R is a field of characteristic p, *val* is the usual p–adic valuation map.

As another example, if $q = 4$ and R is a field of characteristic 5 (so $e = 2$), then the valuation map sends the integer x to $\nu_5([4]_x)$ where $[4]_x$ is interpreted as an integer and ν_5 is the usual 5–adic valuation. To consider this Hecke algebra one could proceed as follows:

```
gap> val:=function(x) local v;
>       x:=Sum([0..x-1],v->4^v);   # x->[x]_q
>       v:=0; while x mod 5=0 do x:=x/5; v:=v+1; od;
>       return v;
>     end;;
gap> H:=Specht(2,5,val,"e2q4");
```

```
Specht(e=2, p=5, S(), P(), D(), HeckeRing="e2q4")
```

Notice the string “e2q4” which was also passed to `Specht` in this example. Although it is not strictly necessary, it is a good idea when using a “non–standard” valuation map *val* to specify the value of `H.HeckeRing`=*HeckeRing*. This string is used for internal bookkeeping by SPECHT; in particular, it is used to determine filenames when reading and saving decomposition matrices. If a “standard” valuation map is used then *HeckeRing* is set to the string “*e<e>p<p>*”; otherwise it defaults to “unknown”. The function `SaveDecompositionMatrix` will not save any decomposition matrix for any Hecke algebra `H` with `H.HeckeRing`=“unknown”.

The Fock space and Hecke algebras over fields of characteristic zero

For Hecke algebras H defined over fields of characteristic zero Lascoux, Leclerc and Thibon [LLT] have described an easy, inductive, algorithm for calculating the decomposition matrices of H. Their algorithm really calculates the **canonical basis**, or (global) **crystal basis** of the Fock space; results of Grojnowski–Lusztig [Gr] show that computing this basis is equivalent to computing the decomposition matrices of H (see also [A]).

The **Fock space** $\mathcal{F}$ is an (integrable) module for the quantum group $U_q(\widehat{sl}_e)$ of the affine special linear group. $\mathcal{F}$ is a free $\mathbf{C}[\mathtt{v}]$–module with basis the set of all Specht modules $\mathtt{S}(\mu)$ for all partitions μ of all integers

$$\mathcal{F} = \bigoplus_{n\geq 0} \bigoplus_{\mu\vdash n} \mathbf{C}[\mathtt{v}]\ \mathtt{S}(\mu);$$

here `v`=`H.info.Indeterminate` is an indeterminate over the integers (or strictly, $\mathbf{C}$). The canonical basis elements $\mathtt{Pq}(\mu)$ for the $U_q(\widehat{sl}_e)$–submodule of $\mathcal{F}$ generated by the 0–partition are indexed by e–regular partitions μ. Moreover, under **specialization**, $\mathtt{Pq}(\mu)$ maps to $\mathtt{P}(\mu)$. An eloquent description of the algorithm for computing $\mathtt{H.Pq}(\mu)$ can be found in [LLT].

To access the elements of the Fock space SPECHT provides the functions:

$\mathtt{H.Pq}(\mu)$ $\quad$ $\mathtt{H.Sq}(\mu)$

Notice that, unlike `H.P` and `H.S` the only arguments which `H.Pq` and `H.Sq` accept are partitions. (Given that our indeterminate is `v` these functions should really be called `H.Pv` and `H.Sv`; here “q” stands for “quantum.)

The function `H.Pq` computes the canonical basis element $\mathtt{Pq}(\mu)$ of the Fock space corresponding to the e–regular partition μ (there is a canonical basis — defined using a larger quantum group — for the whole of the Fock space [LT]; conjecturally, this basis can be used to compute the decomposition matrices for the q–Schur algebra over fields of characteristic zero). The second function returns a standard basis element $\mathtt{S}(\mu)$ of $\mathcal{F}$.

```
gap> H:=Specht(4);
Specht(e=4, S(), P(), D(), Pq())
gap> H.Pq(6,2);
S(6,2)+v*S(5,3)
gap> RestrictedModule(last);
S(6,1)+(v + v^(-1))*S(5,2)+v*S(4,3)
```

```
gap> H.P(last);
P(6,1)+(v + v^(-1))*P(5,2)
gap> Specialized(last);
P(6,1)+2*P(5,2)
gap> H.Sq(5,3,2);
S(5,3,2)
gap> InducedModule(last,0);
v^(-1)*S(5,3,3)
```

The modules returned by `H.Pq` and `H.Sq` behave very much like elements of the Grothendieck ring of H; however, they should be considered as elements of the Fock space. The key difference is that when induced or restricted "quantum" analogues of induction and restriction are used. These analogues correspond to the action of $U_q(\widehat{sl}_e)$ on $\mathcal{F}$ [LLT].

In effect, the functions `H.Pq` and `H.Sq` allow computations in the Fock space, using the functions `InducedModule` 72.6 and `RestrictedModule` 72.8. The functions `H.S`, `H.P`, and `H.D` can also be applied to elements of the Fock space, in which case they have the expected effect. In addition, any element of the Fock space can be specialized to give the corresponding element of the Grothendieck ring of `H` (it is because of this correspondence that we do not make a distinction between elements of the Fock space and the Grothendieck ring of `H`).

When working over fields of characteristic zero SPECHT will automatically calculate any canonical basis elements that it needs for computations in the Grothendieck ring of H. If you are not interested in the canonical basis elements you need never work with them directly. If, for some reason, you do not want SPECHT to use the canonical basis elements to calculate decomposition numbers then all you need to do is `Unbind(H.Pq)`.

72.2 Schur

`Schur(`*e*`)`
`Schur(`*e*`, `*p*`)`
`Schur(`*e*`, `*p*`, `*val* `[,`*HeckeRing*`])`

This function behaves almost identically to the function `Specht` (see 72.1), the only difference being that the three functions in the record `S` returned by `Schur` are called `S.W`, `S.P`, and `S.F` and that they correspond to the q-Weyl modules, the projective decomposable modules, and the simple modules of the q–Schur algebra respectively. Note that our labeling of these modules is non–standard, following that used by James in [J]. The standard labeling can be obtained from ours by replacing all partitions by their conjugates.

Almost all of the functions in SPECHT which accept a `Specht` record H will also accept a record S returned by `Schur`

In the current version of SPECHT the decomposition matrices of q–Schur algebras are not fully supported. The `InducedDecompositionMatrix` function can be applied to these matrices; however there are no additional routines available for calculating the columns corresponding to e–singular partitions. The decomposition matrices for the q–Schur algebras defined over a field of characteristic 0 for $n \leq 10$ are in the SPECHT libraries.

```
gap> S:=Schur(2);
Schur(e=2, W(), P(), F(), Pq())
```

```
gap> InducedDecompositionMatrix(DecompositionMatrix(S,3));
# The following projectives are missing from d
# [ 2, 2 ]
4      | 1                    # DecompositionMatrix(S,4) returns the
3,1    | 1 1                  # full decomposition matrix. The point
2^2    | . 1 .                # of this example is to emphasize the
2,1^2  | 1 1 . 1              # current limitations of Schur.
1^4    | 1 . . 1 1
```

Note that when S is defined over a field of characteristic zero then it contains a function `S.Pq` for calculating canonical basis elements (see `Specht` 72.1); currently `S.Pq(`μ`)` is implemented only for e–regular partitions. There is also a function `H.Wq`.

See also `Specht` 72.1. This function requires the package "specht" (see 57.1).

72.3 DecompositionMatrix

`DecompositionMatrix(`*H*`,` *n* `[,`*Ordering*`])`
`DecompositionMatrix(`*H*`,` *filename* `[,`*Ordering*`])`

The function `DecompositionMatrix` returns the decomposition matrix `D` of `H`(S_n) where `H` is a Hecke algebra record returned by the function `Specht` (or `Schur`). `DecompositionMatrix` first checks to see whether the required decomposition matrix exists as a library file (checking first in the current directory, next in the directory specified by `SpechtDirectory`, and finally in the SPECHT libraries). If `H.Pq` exists, `DecompositionMatrix` next looks for **crystallized decomposition matrices** (see `CrystalizedDecompositionMatrix` 72.4). If the decomposition matrix `d` is not stored in the library `DecompositionMatrix` will calculate `d` when `H` is a Hecke algebra with a base field `R` of characteristic zero, and will return `false` otherwise (in which case the function `CalculateDecompositionMatrix` 72.15 can be used to force SPECHT to try and calculate this matrix).

For Hecke algebras defined over fields of characteristic zero, SPECHT uses the algorithm of [LLT] to calculate decomposition matrices (this feature can be disabled by unbinding `H.Pq`). The decomposition matrices for the q–Schur algebras for $n \leq 10$ are contained in the SPECHT library, as are those for the symmetric group over fields of positive characteristic when $n < 15$.

Once a decomposition matrix is known, SPECHT keeps an internal copy of it which is used by the functions `H.S`, `H.P`, and `H.D`; these functions also read decomposition matrix files as needed.

If you set the variable `SpechtDirectory`, then SPECHT will also search for decomposition matrix files in this directory. The files in the current directory override those in `SpechtDirectory` and those in the SPECHT libraries.

In the second form of the function, when a *filename* is supplied, `DecompositionMatrix` will read the decomposition matrix in the file *filename*, and this matrix will become SPECHT's internal copy of this matrix.

By default, the rows and columns of the decomposition matrices are ordered lexicographically. This can be changed by supplying `DecompositionMatrix` with an ordering function such as `LengthLexicographic` or `ReverseDominance`. You do not need to specify the ordering you want every time you call `DecompositionMatrix`; SPECHT will keep the same ordering until you change it again. This ordering can also be set "by hand" using the variable `H.Ordering`.

```
gap> DecompositionMatrix(Specht(3),6,LengthLexicographic);
6       | 1
5,1     | 1 1
4,2     | . . 1
3^2     | . 1 . 1
4,1^2   | . 1 . . 1
3,2,1   | 1 1 . 1 1 1
2^3     | 1 . . . . 1
3,1^3   | . . . . 1 1
2^2,1^2| . . . . . . 1
2,1^4   | . . . 1 . 1 .
1^6     | . . . 1 . . .
```

Once you have a decomposition matrix it is often nice to be able to print it. The on screen version is often good enough; there is also a `TeX` command which generates a LaTeX version. There are also functions for converting SPECHT decomposition matrices into GAP3 matrices and visa versa (see `MatrixDecompositionMatrix` 72.16 and `DecompositionMatrixMatrix` 72.17).

Using the function `InducedDecompositionMatrix` (see 72.10), it is possible to induce a decomposition matrix. See also `SaveDecompositionMatrix` 72.14 and `IsNewIndecomposable` 72.11, `Specht` 72.1, `Schur` 72.2, and `CrystalizedDecompositionMatrix` 72.4. This function requires the package "specht" (see 57.1).

72.4 CrystalizedDecompositionMatrix

`CrystalizedDecompositionMatrix(`*H*`, `*n*` [,`*Ordering*`])`
`CrystalizedDecompositionMatrix(`*H*`, `*filename*` [,`*Ordering*`])`

This function is similar to `DecompositionMatrix`, except that it returns a **crystallized decomposition matrix**. The columns of decomposition matrices correspond to projective indecomposables; the columns of crystallized decomposition matrices correspond to the canonical basis elements of the Fock space (see 72.1). Consequently, the entries in these matrices are polynomials (in `v`), and by specializing (i.e. setting `v` equal to 1; see 72.52), the decomposition matrices of H are obtained (see 72.1).

Crystallized decomposition matrices are defined only for Hecke algebras over a base field of characteristic zero. Unlike "normal" decomposition matrices, crystallized decomposition matrices cannot be induced.

```
gap> CrystalizedDecompositionMatrix(Specht(3), 6);
6       |   1
5,1     |   v   1
4,2     |   .   .   1
4,1^2   |   .   v   .   1
3^2     |   .   v   .   .   1
3,2,1   |   v v^2   .   v   v   1
3,1^3   |   .   .   . v^2   .   v
2^3     | v^2   .   .   .   .   v
```

```
2^2,1^2|    .    .    .    .    .    .    1
2,1^4  |    .    .    .    .    v  v^2    .
1^6    |    .    .    .    . v^2    .    .
gap> Specialized(last);    # set v equal to 1.
6      | 1
5,1    | 1 1
4,2    | . . 1
4,1^2  | . 1 . 1
3^2    | . 1 . . 1
3,2,1  | 1 1 . 1 1 1
3,1^3  | . . . 1 . 1
2^3    | 1 . . . . 1
2^2,1^2| . . . . . . 1
2,1^4  | . . . . 1 1 .
1^6    | . . . . 1 . .
```

See also `Specht` 72.1, `Schur` 72.2, `DecompositionMatrix` 72.3, and `Specialized` 72.52. This function requires the package "specht" (see 57.1).

72.5 DecompositionNumber

`DecompositionNumber(`H`, `μ`, `ν`)`
`DecompositionNumber(`d`, `μ`, `ν`)`

This function attempts to calculate the decomposition multiplicity of `D`(ν) in `S`(μ) (equivalently, the multiplicity of `S`(μ) in `P`(ν)). If `P`(ν) is known, we just look up the answer; if not `DecompositionNumber` tries to calculate the answer using "row and column removal" (see [J,Theorem 6.18]).

```
gap> H:=Specht(6);;
gap> DecompositionNumber(H,[6,4,2],[6,6]);
0
```

This function requires the package "specht" (see 57.1).

Partitions in Specht

Many of the functions in Specht take partitions as arguments. Partitions are usually represented by lists in GAP3. In Specht, all the functions which expect a partition will accept their argument either as a list or simply as a sequence of numbers. So, for example:

```
gap> H:=Specht(4);; H.S(H.P(6,4));
S(6,4)+S(6,3,1)+S(5,3,1,1)+S(3,3,2,1,1)+S(2,2,2,2,2)
gap> H.S(H.P([6,4]));
S(6,4)+S(6,3,1)+S(5,3,1,1)+S(3,3,2,1,1)+S(2,2,2,2,2)
```

Some functions require more than one argument, but the convention still applies.

```
gap> ECore(3, [6,4,2]);
[ 6, 4, 2 ]
gap> ECore(3, 6,4,2);
```

```
[ 6, 4, 2 ]
gap> GoodNodes(3, 6,4,2);
[ false, false, 3 ]
gap> GoodNodes(3, [6,4,2], 2);
3
```

Basically, it never hurts to put the extra brackets in, and they can be omitted so long as this is not ambiguous. One function where the brackets are needed is `DecompositionNumber`; this is clear because the function takes two partitions as its arguments.

Inducing and restricting modules

Specht provides four functions `InducedModule`, `RestrictedModule`, `SInducedModule` and `SRestrictedModule` for inducing and restricting modules. All functions can be applied to Specht modules, PIMs, and simple modules. These functions all work by first rewriting all modules as a linear combination of Specht modules (or q–Weyl modules), and then inducing and restricting. Whenever possible the induced or restricted module will be written in the original basis.

All of these functions can also be applied to elements of the Fock space (see 72.1); in which case they correspond to the action of the generators E_i and F_i of $U_q(\widehat{sl_e})$ on $\mathcal{F}$. There is also a function `InducedDecompositionMatrix` 72.10 for inducing decomposition matrices.

72.6 InducedModule

`InducedModule(`x`)`
`InducedModule(`x`, `r_1` [,`r_2`, ...])`

There is an natural embedding of $\mathtt{H}(S_n)$ in $\mathtt{H}(S_{n+1})$ which in the usual way lets us define an **induced** $\mathtt{H}(S_{n+1})$–module for every $\mathtt{H}(S_n)$–module. The function `InducedModule` returns the induced modules of the Specht modules, principal indecomposable modules, and simple modules (more accurately, their image in the Grothendieck ring).

There is also a function `SInducedModule` (see 72.7) which provides a much faster way of r–inducing s times (and inducing s times).

Let μ be a partition. Then the induced module `InducedModule(S(`μ`))` is easy to describe: it has the same composition factors as $\sum \mathtt{S}(\nu)$ where ν runs over all partitions whose diagrams can be obtained by adding a single node to the diagram of μ.

```
gap> H:=Specht(2,2);
Specht(e=2, p=2, S(), P(), D())
gap> InducedModule(H.S(7,4,3,1));
S(8,4,3,1)+S(7,5,3,1)+S(7,4,4,1)+S(7,4,3,2)+S(7,4,3,1,1)
gap> InducedModule(H.P(5,3,1));
P(6,3,1)+2*P(5,4,1)+P(5,3,2)
gap> InducedModule(H.D(11,2,1));
# D(x), unable to rewrite x as a sum of simples
```

```
S(12,2,1)+S(11,3,1)+S(11,2,2)+S(11,2,1,1)
```

When inducing indecomposable modules and simple modules, `InducedModule` first rewrites these modules as a linear combination of Specht modules (using known decomposition matrices), and then induces this linear combination of Specht modules. If possible SPECHT then rewrites the induced module back in the original basis. Note that in the last example above, the decomposition matrix for S_{15} is not known by SPECHT; this is why `InducedModule` was unable to rewrite this module in the `D`–basis.

r–Induction

`InducedModule`(x, r_1 [, r_2, ...])

Two Specht modules `S`(μ) and `S`(ν) belong to the same block if and only if the corresponding partitions μ and ν have the same e–core [JM2] (see 72.39). Because the e–core of a partition is determined by its (multiset of) e–residues, if `S`(μ) and `S`(ν) appear in `InducedModule(S(`τ`))`, for some partition τ, then `S`(μ) and `S`(ν) belong to the same block if and only if μ and ν can be obtained by adding a node of the same e–residue to the diagram of τ. The second form of `InducedModule` allows one to induce "within blocks" by only adding nodes of some fixed e–residue r; this is known as **r-induction**. Note that $0 \leq r < e$.

```
gap> H:=Specht(4); InducedModule(H.S(5,2,1));
S(6,2,1)+S(5,3,1)+S(5,2,2)+S(5,2,1,1)
gap> InducedModule(H.S(5,2,1),0);
0*S()
gap> InducedModule(H.S(5,2,1),1);
S(6,2,1)+S(5,3,1)+S(5,2,1,1)
gap> InducedModule(H.S(5,2,1),2);
0*S()
gap> InducedModule(H.S(5,2,1),3);
S(5,2,2)
```

The function `EResidueDiagram` (72.35), prints the diagram of μ, labeling each node with its e–residue. A quick check of this diagram confirms the answers above.

```
gap> EResidueDiagram(H,5,2,1);
  0  1  2  3  0
  3  0
  2
```

"Quantized" induction

When `InducedModule` is applied to the canonical basis elements `H.Pq`(μ) (or more generally elements of the Fock space; see 72.1), a "quantum analogue" of induction is applied. More precisely, the function `InducedModule(*,i)` corresponds to the action of the generator F_i of the quantum group $U_q(\widehat{sl_e})$ on $\mathcal{F}$ [LLT].

```
gap> H:=Specht(3);; InducedModule(H.Pq(4,2),1,2);
S(6,2)+v*S(4,4)+v^2*S(4,2,2)
gap> H.P(last);
P(6,2)
```

See also `SInducedModule` 72.7, `RestrictedModule` 72.8, and `SRestrictedModule` 72.9. This function requires the package "specht" (see 57.1).

72.7 SInducedModule

`SInducedModule(`x`, `s`)`
`SInducedModule(`x`, `s`, `r`)`

The function `SInducedModule`, standing for "string induction", provides a more efficient way of r–inducing s times (and a way of inducing s times if the residue r is omitted); r–induction is explained in 72.6.

```
gap> H:=Specht(4);; SInducedModule(H.P(5,2,1),3);
P(8,2,1)+3*P(7,3,1)+2*P(7,2,2)+6*P(6,3,2)+6*P(6,3,1,1)+3*P(6,2,1,1,1)
+2*P(5,3,3)+P(5,2,2,1,1)
gap> SInducedModule(H.P(5,2,1),3,1);
P(6,3,1,1)
gap> InducedModule(H.P(5,2,1),1,1,1);
6*P(6,3,1,1)
```

Note that the multiplicity of each summand of `InducedModule(x,r,...,r)` is divisible by $s!$ and that `SInducedModule` divides by this constant.

As with `InducedModule` this function can also be applied to elements of the Fock space (see 72.1), in which case the quantum analogue of induction is used.

See also `InducedModule` 72.6. This function requires the package "specht" (see 57.1).

72.8 RestrictedModule

`RestrictedModule(`x`)`
`RestrictedModule(`x`, `r_1` [, `r_2`, ...])`

Given a module x for $\mathtt{H}(S_n)$ `RestrictedModule` returns the corresponding module for $\mathtt{H}(S_{n-1})$. The restriction of the Specht module $\mathtt{S}(\mu)$ is the linear combination of Specht modules $\sum \mathtt{S}(\nu)$ where ν runs over the partitions whose diagrams are obtained by deleting a node from the diagram of μ. If only nodes of residue r are deleted then this corresponds to first restricting $\mathtt{S}(\mu)$ and then taking one of the block components of the restriction; this process is known as **r-restriction** (cf. r–induction in 72.6).

There is also a function `SRestrictedModule` (see 72.9) which provides a faster way of r–restricting s times (and restricting s times).

When more than one residue if given to `RestrictedModule` it returns

$$\mathtt{RestrictedModule}(x, r_1, r_2, ..., r_k) = \mathtt{RestrictedModule}(\mathtt{RestrictedModule}(x, r_1), r_2, ..., r_k)$$

(cf. `InducedModule` 72.6).

```
gap> H:=Specht(6);; RestrictedModule(H.P(5,3,2,1),4);
2*P(4,3,2,1)
gap> RestrictedModule(H.D(5,3,2),1);
D(5,2,2)
```

"Quantized" restriction

As with `InducedModule`, when `RestrictedModule` is applied to the canonical basis elements `H.Pq`(μ) a quantum analogue of restriction is applied; this time, `RestrictedModule(*,i)` corresponds to the action of the generator E_i of $U_q(\widehat{sl_e})$ on $\mathcal{F}$ [LLT].

See also `InducedModule` 72.6, `SInducedModule` 72.7, and `SRestrictedModule` 72.9. This function requires the package "specht" (see 57.1).

72.9 SRestrictedModule

`SRestrictedModule(`x`, `s`)`
`SRestrictedModule(`x`, `s`, `r`)`

As with `SInducedModule` this function provides a more efficient way of r–restricting s times, or restricting s times if the residue r is omitted (cf. `SInducedModule` 72.7).

```
gap> H:=Specht(6);; SRestrictedModule(H.S(4,3,2),3);
3*S(4,2)+2*S(4,1,1)+3*S(3,3)+6*S(3,2,1)+2*S(2,2,2)
gap> SRestrictedModule(H.P(5,4,1),2,4);
P(4,4)
```

See also `InducedModule` 72.6, `SInducedModule` 72.7, and `RestrictedModule` 72.8. This function requires the package "specht" (see 57.1).

Operations on decomposition matrices

SPECHT is a package for computing decomposition matrices; this section describes the functions available for accessing these matrices directly. In addition to decomposition matrices, SPECHT also calculates the "crystallized decomposition matrices" of [LLT], and the "adjustment matrices" introduced by James [J] (and Geck [G]).

Throughout SPECHT we place an emphasis on calculating the projective indecomposable modules, and hence upon the columns of decomposition matrices. This approach seems more efficient than the traditional approach of calculating decomposition matrices by rows; ideally both approaches should be combined (as is done by `IsNewIndecomposable`).

In principle, all decomposition matrices for all Hecke algebras defined over a field of characteristic zero are available from within SPECHT. In addition, the decomposition matrices for all q–Schur algebras with $n \leq 10$ and all values of e and the p–modular decomposition matrices of the symmetric groups S_n for $n < 15$ are in the SPECHT library files.

If you are using SPECHT regularly to do calculations involving certain values of e it would be advantageous to have SPECHT calculate and save the first 20 odd decomposition matrices that you are interested in. So, for $e = 4$ use the commands:

```
gap> H:=Specht(4);; for n in [8..20] do
>       SaveDecompositionMatrix(DecompositionMatrix(H,n));
>    od;
```

Alternatively, you could save the crystallized decomposition matrices. Note that for $n < 2e$ the decomposition matrices are known (by SPECHT) and easy to compute.

72.10 InducedDecompositionMatrix

`InducedDecompositionMatrix(`d`)`

If d is the decomposition matrix of `H`(S_n), then `InducedDecompositionMatrix(`d`)` attempts to calculate the decomposition matrix of `H`(S_{n+1}). It does this by extracting each projective indecomposable from d and inducing these modules to obtain projective modules for `H`(S_{n+1}). `InducedDecompositionMatrix` then tries to decompose these projectives using the function `IsNewIndecomposable` (see 72.11). In general there will be columns of the decomposition matrix which `InducedDecompositionMatrix` is unable to decompose and these will have to be calculated "by hand". `InducedDecompositionMatrix` prints a list of those columns of the decomposition matrix which it is unable to calculate (this list is also printed by the function `MissingIndecomposables(`d`)`).

```
gap> d:=DecompositionMatrix(Specht(3,3),14);;
gap> InducedDecompositionMatrix(d);;
# Inducing....
The following projectives are missing from <d>:
    [ 15 ]  [ 8, 7 ]
```

Note that the missing indecomposables come in "pairs" which map to each other under the Mullineux map (see `MullineuxMap` 72.25).

Almost all of the decomposition matrices included in SPECHT were calculated directly by `InducedDecompositionMatrix`. When n is "small" `InducedDecompositionMatrix` is usually able to return the full decomposition matrix for `H`(S_{n+1}).

Finally, although the `InducedDecompositionMatrix` can also be applied to the decomposition matrices of the q–Schur algebras (see `Schur` 72.2), `InducedDecompositionMatrix` is much less successful in inducing these decomposition matrices because it contains no special routines for dealing with the indecomposable modules of the q–Schur algebra which are indexed by e–singular partitions. Note also that we use a non–standard labeling of the decomposition matrices of q–Schur algebras; see 72.2.

72.11 IsNewIndecomposable

`IsNewIndecomposable(`d`, `x` [,`μ`])`

`IsNewIndecomposable` is the function which does all of the hard work when the function `InducedDecompositionMatrix` is applied to decomposition matrices (see 72.10). Given a projective module x, `IsNewIndecomposable` returns `true` if it is able to show that x is indecomposable (and this indecomposable is not already listed in d), and `false` otherwise. `IsNewIndecomposable` will also print a brief description of its findings, giving an upper and lower bound on the **first** decomposition number μ for which it is unable to determine the multiplicity of `S`(μ) in x.

`IsNewIndecomposable` works by running through all of the partitions ν such that `P`(ν) could be a summand of x and it uses various results, such as the q-Schaper theorem of [JM2] (see `Schaper` 72.23), the Mullineux map (see `MullineuxMap` 72.25), and inducing simple modules, to determine if `P`(ν) does indeed split off. In addition, if d is the decomposition matrix for `H`(S_n) then `IsNewIndecomposable` will probably use some of the decomposition matrices of `H`(S_m) for $m \leq n$, if they are known. Consequently it is a good idea to save decomposition matrices as they are calculated (see 72.14).

For example, in calculating the 2–modular decomposition matrices of S_r the first projective which `InducedDecompositionMatrix` is unable to calculate is `P`(10).

```
gap> H:=Specht(2,2);;
gap> d:=InducedDecompositionMatrix(DecompositionMatrix(H,9));;
# Inducing.
# The following projectives are missing from d
# [ 10 ]
```

(In fact, given the above commands, Specht will return the full decomposition matrix for S_{10} because this matrix is in the library; these were the commands that I used to calculate the decomposition matrix in the library.)

By inducing `P`(9) we can find a projective H–module which contains `P`(10). We can then use `IsNewIndecomposable` to try and decompose this induced module into a sum of PIMs.

```
gap> SpechtPrettyPrint(); x:=InducedModule(H.P(9),1);
S(10)+S(9,1)+S(8,2)+2S(8,1^2)+S(7,3)+2S(7,1^3)+3S(6,3,1)+3S(6,2^2)
+4S(6,2,1^2)+2S(6,1^4)+4S(5,3,2)+5S(5,3,1^2)+5S(5,2^2,1)+2S(5,1^5)
+2S(4^2,2)+2S(4^2,1^2)+2S(4,3^2)+5S(4,3,1^3)+2S(4,2^3)+5S(4,2^2,1^2)
+4S(4,2,1^4)+2S(4,1^6)+2S(3^3,1)+2S(3^2,2^2)+4S(3^2,2,1^2)
+3S(3^2,1^4)+3S(3,2^2,1^3)+2S(3,1^7)+S(2^3,1^4)+S(2^2,1^6)+S(2,1^8)
+S(1^10)
gap> IsNewIndecomposable(d,x);
# The multiplicity of S(6,3,1) in P(10) is at least 1 and at most 2.
false
gap> x;
S(10)+S(9,1)+S(8,2)+2S(8,1^2)+S(7,3)+2S(7,1^3)+2S(6,3,1)+2S(6,2^2)
+3S(6,2,1^2)+2S(6,1^4)+3S(5,3,2)+4S(5,3,1^2)+4S(5,2^2,1)+2S(5,1^5)
+2S(4^2,2)+2S(4^2,1^2)+2S(4,3^2)+4S(4,3,1^3)+2S(4,2^3)+4S(4,2^2,1^2)
+3S(4,2,1^4)+2S(4,1^6)+2S(3^3,1)+2S(3^2,2^2)+3S(3^2,2,1^2)
+2S(3^2,1^4)+2S(3,2^2,1^3)+2S(3,1^7)+S(2^3,1^4)+S(2^2,1^6)+S(2,1^8)
+S(1^10)
```

Notice that some of the coefficients of the Specht modules in x have changed; this is because `IsNewIndecomposable` was able to determine that the multiplicity of `S`(6,3,1) was at most 2 and so it subtracted one copy of `P`(6,3,1) from x.

In this case, the multiplicity of `S`(6,3,1) in `P`(10) is easy to resolve because general theory says that this multiplicity must be odd. Therefore, $x - \mathtt{P}(6,3,1)$ is projective. After subtracting `P`(6,3,1) from x we again use `IsNewIndecomposable` to see if x is now indecomposable. We can tell `IsNewIndecomposable` that all of the multiplicities up to and including `S`(6,3,1) have already been checked by giving it the addition argument μ=[6,3,1].

```
gap> x:=x-H.P(d,6,3,1);; IsNewIndecomposable(d,x,6,3,1);
true
```

Consequently, $x = \mathtt{P}(10)$ and we add it to the decomposition matrix d (and save it).

```
gap> AddIndecomposable(d,x); SaveDecompositionMatrix(d);
```

A full description of what `IsNewIndecomposable` does can be found by reading the comments in `specht.g`. Any suggestions or improvements on this function would be especially welcome.

See also `DecompositionMatrix` 72.3 and `InducedDecompositionMatrix` 72.10. This function requires the package "specht" (see 57.1).

72.12 InvertDecompositionMatrix

`InvertDecompositionMatrix(d)`

Returns the inverse of the (e–regular part of) d, where d is a decomposition matrix, or crystallized decomposition matrix, of a Hecke algebra or q–Schur algebra. If part of the decomposition matrix d is unknown then `InvertDecompositionMatrix` will invert as much of d as possible.

```
gap> H:=Specht(4);; d:=CrystalizedDecompositionMatrix(H,5);;
gap> InvertDecompositionMatrix(d);
5    |   1
4,1  |   .   1
3,2  |  -v   .   1
3,1^2|   .   .   .   1
2^2,1| v^2   .  -v   .   1
2,1^3|   .   .   .   .   .   1
```

See also `DecompositionMatrix` 72.3, and `CrystalizedDecompositionMatrix` 72.4. This function requires the package "specht" (see 57.1).

72.13 AdjustmentMatrix

`AdjustmentMatrix(`dp`, `d`)`

James [J] noticed, and Geck [G] proved, that the decomposition matrices dp for Hecke algebras defined over fields of positive characteristic admit a factorization

$$dp = d * a$$

where d is a decomposition matrix for a suitable Hecke algebra defined over a field of characteristic zero, and a is the so–called **adjustment matrix**. This function returns the adjustment matrix a.

```
gap> H:=Specht(2);; Hp:=Specht(2,2);;
gap> d:=DecompositionMatrix(H,13);; dp:=DecompositionMatrix(Hp,13);;
gap> a:=AdjustmentMatrix(dp,d);
13      | 1
12,1    | . 1
11,2    | 1 . 1
10,3    | . . . 1
10,2,1  | . . . . 1
9,4     | 1 . 1 . . 1
9,3,1   | 2 . . . . . 1
8,5     | . 1 . . . . . 1
8,4,1   | 1 . . . . . . . 1
8,3,2   | . 2 . . . . . 1 . 1
7,6     | 1 . . . . 1 . . . . 1
7,5,1   | . . . . . . 1 . . . . 1
7,4,2   | 1 . 1 . . 1 . . . . 1 . 1
7,3,2,1 | . . . . . . . . . . . . . 1
```

```
6,5,2  | . 1 . . . . . 1 . 1 . . . . 1
6,4,3  | 2 . . . 1 . . . . . . . . . . 1
6,4,2,1| . 2 . 1 . . . . . . . . . . . . 1
5,4,3,1| 4 . 2 . . . . . . . . . . . . . . 1
gap> MatrixDecompositionMatrix(dp)=
>           MatrixDecompositionMatrix(d)*MatrixDecompositionMatrix(a);
true
```

In the last line we have checked our calculation.

See also `DecompositionMatrix` 72.3, and `CrystalizedDecompositionMatrix` 72.4. This function requires the package "specht" (see 57.1).

72.14 SaveDecompositionMatrix

`SaveDecompositionMatrix(`*d*`)`
`SaveDecompositionMatrix(`*d*`, `*filename*`)`

The function `SaveDecompositionMatrix` saves the decomposition matrix d. After a decomposition matrix has been saved, the functions `H.S`, `H.P`, and `H.D` will automatically access it as needed. So, for example, before saving d in order to retrieve the indecomposable `P(`μ`)` from d it is necessary to type `H.P(`d`, `μ`)`; once d has been saved, the command `H.P(`μ`)` suffices.

Since `InducedDecompositionMatrix(`d`)` consults the decomposition matrices for smaller n, if they are available, it is advantageous to save decomposition matrices as they are calculated. For example, over a field of characteristic 5, the decomposition matrices for the symmetric groups S_n with $n \leq 20$ can be calculated as follows:

```
gap> H:=Specht(5,5);;
gap> d:=DecompositionMatrix(H,9);;
gap> for r in [10..20] do
>      d:=InducedDecompositionMatrix(d);
>      SaveDecompositionMatrix(d);
>    od;
```

If your Hecke algebra record `H` is defined using a non–standard valuation map (see 72.1) then it is also necessary to set the string "`H.HeckeRing`", or to supply the function with a *filename* before it will save your matrix. `SaveDecompositionMatrix` will also save adjustment matrices and the various other matrices that appear in Specht (they can be read back in using `DecompositionMatrix`). Each matrix has a default filename which you can over ride by supplying a *filename*. Using non–standard file names will stop Specht from automatically accessing these matrices in future.

See also 72.3 `DecompositionMatrix` 72.3 and `CrystalizedDecompositionMatrix` 72.4. This function requires the package "specht" (see 57.1).

72.15 CalculateDecompositionMatrix

`CalculateDecompositionMatrix(H,n)`

`CalculateDecompositionMatrix(H,n)` is similar to the function `DecompositionMatrix` 72.3 in that both functions try to return the decomposition matrix `d` of `H(`S_n`)`; the difference is that this function tries to calculate this matrix whereas the later reads the matrix from the library files (in characteristic zero both functions apply the algorithm of [LLT] to compute `d`). In effect this function is only needed when working with Hecke algebras defined over fields of positive characteristic (or when you wish to avoid the libraries).

For example, if you want to do calculations with the decomposition matrix of the symmetric group S_{15} over a field of characteristic two, `DecompositionMatrix` returns false whereas `CalculateDecompositionMatrix`; returns a part of the decomposition matrix.

```
gap> H:=Specht(2,2);
Specht(e=2, p=2, S(), P(), D())
gap> d:=DecompositionMatrix(H,15);
# This decomposition matrix is not known; use CalculateDecompositionMatrix()
# or InducedDecompositionMatrix() to calculate with this matrix.
false
gap> d:=CalculateDecompositionMatrix(H,15);;
# Projective indecomposable P(6,4,3,2) not known.
# Projective indecomposable P(6,5,3,1) not known.
...
gap> MissingIndecomposables(d);
The following projectives are missing from <d>:
    [ 15 ]  [ 14, 1 ]  [ 13, 2 ]  [ 12, 3 ]  [ 12, 2, 1 ]  [ 11, 4 ]
[ 11, 3, 1 ]  [ 10, 5 ]  [ 10, 4, 1 ]  [ 10, 3, 2 ]  [ 9, 6 ]  [ 9, 5, 1 ]
[ 9, 4, 2 ]  [ 9, 3, 2, 1 ]  [ 8, 7 ]  [ 8, 6, 1 ]  [ 8, 5, 2 ]  [ 8, 4, 3]
[ 8, 4, 2, 1 ]  [ 7, 6, 2 ]  [ 7, 5, 3 ]  [ 7, 5, 2, 1 ]  [ 7, 4, 3, 1 ]
[ 6, 5, 4 ]  [ 6, 5, 3, 1 ]  [ 6, 4, 3, 2 ]
```

Actually, you are much better starting with the decomposition matrix of S_{14} and then applying `InducedDecompositionMatrix` to this matrix.

See also 72.3 `DecompositionMatrix`. This function requires the package "specht" (see 57.1).

72.16 MatrixDecompositionMatrix

`MatrixDecompositionMatrix(`d`)`

Returns the GAP3 matrix corresponding to the SPECHT decomposition matrix d. The rows and columns of d are ordered by `H.Ordering`.

```
gap> MatrixDecompositionMatrix(DecompositionMatrix(Specht(3),5));
[ [ 1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 1, 1, 0, 0 ],
  [ 0, 0, 0, 1, 0 ], [ 1, 0, 0, 0, 1 ], [ 0, 0, 0, 0, 1 ],
  [ 0, 0, 1, 0, 0 ] ]
```

See also `DecompositionMatrix` 72.3 and `DecompositionMatrixMatrix` 72.17. This function requires the package "specht" (see 57.1).

72.17 DecompositionMatrixMatrix

`DecompositionMatrixMatrix(`H`, `m`, `n`)`

Given a Hecke algebra H, a GAP3 matrix m, and an integer n this function returns the SPECHT decomposition matrix corresponding to m. If `p` is the number of partitions of n

and r the number of e–regular partitions of n, then m must be either $r \times r$, $p \times r$, or $p \times p$. The rows and columns of m are assumed to be indexed by partitions ordered by H.Ordering (see 72.1).

```
gap> H:=Specht(3);;
gap> m:=[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 1, 0 ],
>         [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ] ];;
gap> DecompositionMatrixMatrix(H,m,4);
4    | 1
3,1  | . 1
2^2  | 1 . 1
2,1^2| . . . 1
1^4  | . . 1 .
```

See also DecompositionMatrix 72.3 and MatrixDecompositionMatrix 72.16. This function requires the package “specht” (see 57.1).

72.18 AddIndecomposable

AddIndecomposable(d, x)

AddIndecomposable(d, x) inserts the indecomposable module x into the decomposition matrix d. If d already contains the indecomposable x then a warning is printed. The function AddIndecomposable also calculates MullineuxMap(x) (see 72.25) and adds this indecomposable to d (or checks to see that it agrees with the corresponding entry of d if this indecomposable is already by d).

See IsNewIndecomposable 72.11 for an example. See also DecompositionMatrix 72.3 and CrystalizedDecompositionMatrix 72.4. This function requires the package “specht” (see 57.1).

72.19 RemoveIndecomposable

RemoveIndecomposable(d, μ)

The function RemoveIndecomposable removes the column from d which corresponds to P(μ). This is sometimes useful when trying to calculate a new decomposition matrix using SPECHT and want to test a possible candidate for a yet to be identified PIM.

See also DecompositionMatrix 72.3 and CrystalizedDecompositionMatrix 72.4. This function requires the package “specht” (see 57.1).

72.20 MissingIndecomposables

MissingIndecomposables(d)

The function MissingIndecomposables prints the list of partitions corresponding to the indecomposable modules which are not listed in d.

See also DecompositionMatrix 72.3 and CrystalizedDecompositionMatrix 72.4. This function requires the package “specht” (see 57.1).

Calculating dimensions

SPECHT has two functions for calculating the dimensions of modules of Hecke algebras; `SimpleDimension` and `SpechtDimension`. As yet, SPECHT does not know how to calculate the dimensions of modules for q–Schur algebras (these depend upon q).

72.21 SimpleDimension

`SimpleDimension(`d`)`
`SimpleDimension(`H`, `n`)`
`SimpleDimension(`H`|`d`, `μ`)`

In the first two forms, `SimpleDimension` prints the dimensions of all of the simple modules specified by d or for the Hecke algebra $\mathtt{H}(S_n)$ respectively. If a partition μ is supplied, as in the last form, then the dimension of the simple module $\mathtt{D}(\mu)$ is returned. At present the function is not implemented for the simple modules of the q–Schur algebras.

```
gap> H:=Specht(6);;
gap> SimpleDimension(H,11,3);
272
gap> d:=DecompositionMatrix(H,5);; SimpleDimension(d,3,2);
5
gap> SimpleDimension(d);
5     : 1
4,1   : 4
3,2   : 5
3,1^2 : 6
2^2,1 : 5
2,1^3 : 4
1^5   : 1
```

This function requires the package "specht" (see 57.1).

72.22 SpechtDimension

`SpechtDimension(`μ`)`

Calculates the dimension of the Specht module $\mathtt{S}(\mu)$, which is equal to the number of standard μ-tableaux; the answer is given by the hook length formula (see [JK]).

```
gap> SpechtDimension(6,3,2,1);
5632
```

See also `SimpleDimension` 72.21. This function requires the package "specht"(see 57.1).

Combinatorics on Young diagrams

These functions range from the representation theoretic q–Schaper theorem and Kleshchev's algorithm for the Mullineux map through to simple combinatorial operations like adding and removing rim hooks from Young diagrams.

72.23 Schaper

`Schaper(`H`, `μ`)`

Given a partition μ, and a Hecke algebra H, `Schaper` returns a linear combination of Specht modules which have the same composition factors as the sum of the modules in the "Jantzen filtration" of `S`(μ); see [JM2]. In particular, if ν strictly dominates μ then `D`(ν) is a composition factor of `S`(μ) if and only if it is a composition factor of `Schaper(`μ`)`.

`Schaper` uses the valuation map `H.valuation` attached to H (see 72.1 and [JM2]).

One way in which the q–Schaper theorem can be applied is as follows. Suppose that we have a projective module x, written as a linear combination of Specht modules, and suppose that we are trying to decide whether the projective indecomposable `P`(μ) is a direct summand of x. Then, providing that we know that `P`(ν) is not a summand of x for all (e–regular) partitions ν which strictly dominate μ (see 47.19), `P`(μ) is a summand of x if and only if `InnerProduct(Schaper(H,`μ`),x)` is non–zero (note, in particular, that we don't need to know the indecomposable `P`(μ) in order to perform this calculation).

The q–Schaper theorem can also be used to check for irreduciblity; in fact, this is the basis for the criterion employed by `IsSimpleModule`.

```
gap> H:=Specht(2);;
gap> Schaper(H,9,5,3,2,1);
S(17,2,1)-S(15,2,1,1,1)+S(13,2,2,2,1)-S(11,3,3,2,1)+S(10,4,3,2,1)-S(9,8,3)
-S(9,8,1,1,1)+S(9,6,3,2)+S(9,6,3,1,1)+S(9,6,2,2,1)
gap> Schaper(H,9,6,5,2);
0*S(0)
```

The last calculation shows that `S`(9,6,5,2) is irreducible when R is a field of characteristic 0 and e`=2` (cf. `IsSimpleModule(H,9,6,5,2)`).

This function requires the package "specht" (see 57.1).

72.24 IsSimpleModule

`IsSimpleModule(`H`, `μ`)`

μ an e–regular partition.

Given an e–regular partition μ, `IsSimpleModule(`H`, `μ`)` returns `true` if `S`(μ) is simple and `false` otherwise. This calculation uses the valuation function `H.valuation`; see 72.1. Note that the criterion used by `IsSimpleModule` is completely combinatorial; it is derived from the q–Schaper theorem [JM2].

```
gap> H:=Specht(3);;
gap> IsSimpleModule(H,45,31,24);
false
```

See also `Schaper` 72.23. This function requires the package "specht" (see 57.1).

72.25 MullineuxMap

`MullineuxMap(`e`|`H`, `μ`)`
`MullineuxMap(`d`, `μ`)`
`MullineuxMap(`x`)`

Given an integer e, or a SPECHT record H, and a partition μ, `MullineuxMap(`e, μ`)` returns the image of μ under the Mullineux map; which we now explain.

The sign representation $\texttt{D}(1^n)$ of the Hecke algebra is the (one dimensional) representation sending T_w to $(-1)^{\ell(w)}$. The Hecke algebra H is not a Hopf algebra so there is no well defined action of H upon the tensor product of two H–modules; however, there is an outer automorphism # of H which corresponds to tensoring with $\texttt{D}(1^n)$. This sends an irreducible module $\texttt{D}(\mu)$ to an irreducible $\texttt{D}(\mu)^\# \cong \texttt{D}(\mu^\#)$ for some e–regular partition $\mu^\#$. In the symmetric group case, Mullineux gave a conjectural algorithm for calculating $\mu^\#$; consequently the map sending μ to $\mu^\#$ is known as the **Mullineux map**.

Deep results of Kleshchev [K] for the symmetric group give another (proven) algorithm for calculating the partition $\mu^\#$ (Ford and Kleshchev have deduced Mullineux's conjecture from this). Using the canonical basis, it was shown by [LLT] that the natural generalization of Kleshchev's algorithm to H gives the Mullineux map for Hecke algebras over fields of characteristic zero. The general case follows from this, so the Mullineux map is now known for all Hecke algebras.

Kleshchev's map is easy to describe; he proved that if *gns* is any good node sequence for μ, then the sequence obtained from *gns* by replacing each residue r by $-r \bmod e$ is a good node sequence for $\mu^\#$ (see `GoodNodeSequence` 72.30).

```
gap> MullineuxMap(Specht(2),12,5,2);
[ 12, 5, 2 ]
gap> MullineuxMap(Specht(4),12,5,2);
[ 4, 4, 4, 2, 2, 1, 1, 1 ]
gap> MullineuxMap(Specht(6),12,5,2);
[ 4, 3, 2, 2, 2, 2, 2, 1, 1 ]
gap> MullineuxMap(Specht(8),12,5,2);
[ 3, 3, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
gap> MullineuxMap(Specht(10),12,5,2);
[ 3, 3, 3, 3, 2, 1, 1, 1, 1, 1 ]
```

`MullineuxMap(`d, μ`)`

The Mullineux map can also be calculated using a decomposition matrix. To see this recall that "tensoring" a Specht module $\texttt{S}(\mu)$ with the sign representation yields a module isomorphic to the dual of $\texttt{S}(\lambda)$, where λ is the partition conjugate to μ. It follows that $d_{\mu\nu} = d_{\lambda\nu^\#}$ for all e–regular partitions ν. Therefore, if μ is the last partition in the lexicographic order such that $d_{\mu\nu} \neq 0$ then we must have $\nu^\# = \lambda$. The second form of `MullineuxMap` uses d to calculate $\mu^\#$ rather than the Kleshchev–[LLT] result.

`MullineuxMap(`x`)`

In the third form, x is a module, and `MullineuxMap` returns $x^\#$, the image of x under #. Note that the above remarks show that $\texttt{P}(\mu)$ is mapped to $\texttt{P}(\mu^\#)$ via the Mullineux map; this observation is useful when calculating decomposition matrices (and is used by the function `InducedDecompositionMatrix`).

See also `GoodNodes` 72.28 and `GoodNodeSequence` 72.30 . This function requires the package "specht" (see 57.1).

72.26 MullineuxSymbol

`MullineuxSymbol(`e`|`H`, `μ`)`

Returns the Mullineux symbol of the e–regular partition μ.

```
gap> MullineuxSymbol(5,[8,6,5,5]);
[ [ 10, 6, 5, 3 ], [ 4, 4, 3, 2 ] ]
```

See also `PartitionMullineuxSymbol` 72.27. This function requires the package "specht" (see 57.1).

72.27 PartitionMullineuxSymbol

`PartitionMullineuxSymbol(`e`|`H`, `ms`)`

Given a Mullineux symbol ms, this function returns the corresponding e–regular partition.

```
gap> PartitionMullineuxSymbol(5, MullineuxSymbol(5,[8,6,5,5]) );
[ 8, 6, 5, 5 ]
```

See also `MullineuxSymbol` 72.26. This function requires the package "specht" (see 57.1).

72.28 GoodNodes

`GoodNodes(`e`|`H`, `μ`)`
`GoodNodes(`e`|`H`, `μ`, `r`)`

Given a partition and an integer e, Kleshchev [K] defined the notion of **good node** for each residue r ($0 \leq r < e$). When e is prime and μ is e–regular, Kleshchev showed that the good nodes describe the restriction of the socle of `D`(μ) in the symmetric group case. Brundan [B] has recently generalized this result to the Hecke algebra.

By definition, there is at most one good node for each residue r, and this node is a removable node (in the diagram of μ). The function `GoodNodes` returns a list of the rows of μ which end in a good node; the good node of residue r (if it exists) is the (r+1)–st element in this list. In the second form, the number of the row which ends with the good node of residue r is returned; or `false` if there is no good node of residue r.

```
gap> GoodNodes(5,[5,4,3,2]);
[ false, false, 2, false, 1 ]
gap> GoodNodes(5,[5,4,3,2],0);
false
gap> GoodNodes(5,[5,4,3,2],4);
1
```

The good nodes also determine the Kleshchev–Mullineux map (see `GoodNodeSequence` 72.30 and `MullineuxMap` 72.25). This function requires the package "specht" (see 57.1).

72.29 NormalNodes

`NormalNodes(`e`|`H`, `μ`)`
`NormalNodes(`e`|`H`, `μ`, `r`)`

Returns the numbers of the rows of μ which end in one of Kleshchev's [K] normal nodes. In the second form, only those rows corresponding to normal nodes of the specified residue are returned.

```
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1]);
[ [ 1, 4 ], [  ], [  ], [ 2, 5 ], [  ] ]
gap> NormalNodes(5,[6,5,4,4,3,2,1,1,1],0);
[ 1, 4 ]
```

See also `GoodNodes` 72.28. This function requires the package "specht" (see 57.1).

72.30 GoodNodeSequence

`GoodNodeSequence(`e`|`H`, `μ`)`
`GoodNodeSequences(`e`|`H`, `μ`)`

μ an e–regular partition.

Given an e–regular partition μ of n, a **good node sequence** for μ is a sequence *gns* of n residues such that μ has a good node of residue r, where r is the last residue in *gns*, and the first $n-1$ residues in *gns* are a good node sequence for the partition obtained from μ by deleting its (unique) good node with residue r (see `GoodNodes` 72.28). In general, μ will have more than one good node sequence; however, any good node sequence uniquely determines μ (see `PartitionGoodNodeSequence` 72.31).

```
gap> H:=Specht(4);; GoodNodeSequence(H,4,3,1);
[ 0, 3, 1, 0, 2, 2, 1, 3 ]
gap> GoodNodeSequence(H,4,3,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3 ]
gap> GoodNodeSequence(H,4,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2 ]
gap> GoodNodeSequence(H,5,4,2);
[ 0, 3, 1, 0, 2, 2, 1, 3, 3, 2, 0 ]
```

The function `GoodNodeSequences` returns the list of all good node sequences for μ.

```
gap> GoodNodeSequences(H,5,2,1);
[ [ 0, 1, 2, 3, 3, 2, 0, 0 ], [ 0, 3, 1, 2, 2, 3, 0, 0 ],
  [ 0, 1, 3, 2, 2, 3, 0, 0 ], [ 0, 1, 2, 3, 3, 0, 2, 0 ],
  [ 0, 1, 2, 3, 0, 3, 2, 0 ], [ 0, 1, 2, 3, 3, 0, 0, 2 ],
  [ 0, 1, 2, 3, 0, 3, 0, 2 ] ]
```

The good node sequences determine the Mullineux map (see `GoodNodes` 72.28 and `MullineuxMap` 72.25). This function requires the package "specht" (see 57.1).

72.31 PartitionGoodNodeSequence

`PartitionGoodNodeSequence(`e`|`H`, `*gns*`)`

Given a good node sequence *gns* (see `GoodNodeSequence` 72.30), this function returns the unique e–regular partition corresponding to *gns* (or `false` if in fact *gns* is not a good node sequence).

```
gap> H:=Specht(4);;
gap> PartitionGoodNodeSequence(H,0, 3, 1, 0, 2, 2, 1, 3, 3, 2);
[ 4, 4, 2 ]
```

See also `GoodNodes` 72.28, `GoodNodeSequence` 72.30 and `MullineuxMap` 72.25. This function requires the package “specht” (see 57.1).

72.32 GoodNodeLatticePath

`GoodNodeLatticePath(`e`|`H`,` μ`)`
`GoodNodeLatticePaths(`e`|`H`,` μ`)`
`LatticePathGoodNodeSequence(`e`|`H`,` *gns*`)`

The function `GoodNodeLatticePath` returns a sequence of partitions which give a path in the e–good partition lattice from the empty partition to μ. The second function returns the list of all paths in the e–good partition lattice which end in μ, and the third function returns the path corresponding to a given good node sequence *gns*.

```
gap> GoodNodeLatticePath(3,3,2,1);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ]
gap> GoodNodeLatticePaths(3,3,2,1);
[ [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 1, 1 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ],
  [ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 2, 2, 1 ], [ 3, 2, 1 ] ] ]
gap> GoodNodeSequence(4,6,3,2);
[ 0, 3, 1, 0, 2, 2, 3, 3, 0, 1, 1 ]
gap> LatticePathGoodNodeSequence(4,last);
[ [ 1 ], [ 1, 1 ], [ 2, 1 ], [ 2, 2 ], [ 3, 2 ], [ 3, 2, 1 ], [ 4, 2, 1 ],
  [ 4, 2, 2 ], [ 5, 2, 2 ], [ 6, 2, 2 ], [ 6, 3, 2 ] ]
```

See also `GoodNodes` 72.28. This function requires the package “specht” (see 57.1).

72.33 LittlewoodRichardsonRule

`LittlewoodRichardsonRule(`μ`,` ν`)`
`LittlewoodRichardsonCoefficient(`μ`,` ν`,` τ`)`

Given partitions μ of n and ν of m the module $\mathtt{S}(\mu)\otimes\mathtt{S}(\nu)$ is naturally an $\mathtt{H}(S_n \times S_m)$-module and, by inducing, we obtain an $\mathtt{H}(S_{n+m})$-module. This module has the same composition factors as

$$\sum_{\nu} a^{\lambda}_{\mu\nu}\mathtt{S}(\lambda),$$

where the sum runs over all partitions λ of $n+m$ and the integers $a^{\lambda}_{\mu\nu}$ are the Littlewood–Richardson coefficients. The integers $a^{\lambda}_{\mu\nu}$ can be calculated using a straightforward combinatorial algorithm known as the Littlewood–Richardson rule (see [JK]).

The function `LittlewoodRichardsonRule` returns an (unordered) list of partitions of $n+m$ in which each partition λ occurs $a^{\lambda}_{\mu\nu}$ times. The Littlewood-Richardson coefficients are independent of e; they can be read more easily from the computation `S(`μ`)*S(`ν`)`.

```
gap> H:=Specht(0);;     # the generic Hecke algebra with R=C[q]
gap> LittlewoodRichardsonRule([3,2,1],[4,2]);
[ [ 4, 3, 2, 2, 1 ],[ 4, 3, 3, 1, 1 ],[ 4, 3, 3, 2 ],[ 4, 4, 2, 1, 1 ],
 [ 4, 4, 2, 2 ],[ 4, 4, 3, 1 ],[ 5, 2, 2, 2, 1 ],[ 5, 3, 2, 1, 1 ],
  [ 5, 3, 2, 2 ],[ 5, 4, 2, 1 ],[ 5, 3, 2, 1, 1 ],[ 5, 3, 3, 1 ],
  [ 5, 4, 1, 1, 1 ],[ 5, 4, 2, 1 ],[ 5, 5, 1, 1 ],[ 5, 3, 2, 2 ],
  [ 5, 3, 3, 1 ],[ 5, 4, 2, 1 ],[ 5, 4, 3 ],[ 5, 5, 2 ],[ 6, 2, 2, 1, 1],
  [ 6, 3, 1, 1, 1 ],[ 6, 3, 2, 1 ],[ 6, 4, 1, 1 ],[ 6, 2, 2, 2 ],
  [ 6, 3, 2, 1 ],[ 6, 4, 2 ],[ 6, 3, 2, 1 ],[ 6, 3, 3 ],[ 6, 4, 1, 1 ],
  [ 6, 4, 2 ], [ 6, 5, 1 ], [ 7, 2, 2, 1 ], [ 7, 3, 1, 1 ], [ 7, 3, 2 ],
  [ 7, 4, 1 ] ]
gap> H.S(3,2,1)*H.S(4,2);
S(7,4,1)+S(7,3,2)+S(7,3,1,1)+S(7,2,2,1)+S(6,5,1)+2*S(6,4,2)+2*S(6,4,1,1)
+S(6,3,3)+3*S(6,3,2,1)+S(6,3,1,1,1)+S(6,2,2,2)+S(6,2,2,1,1)+S(5,5,2)
+S(5,5,1,1)+S(5,4,3)+3*S(5,4,2,1)+S(5,4,1,1,1)+2*S(5,3,3,1)+2*S(5,3,2,2)
+2*S(5,3,2,1,1)+S(5,2,2,2,1)+S(4,4,3,1)+S(4,4,2,2)+S(4,4,2,1,1)+S(4,3,3,2)
+S(4,3,3,1,1)+S(4,3,2,2,1)
gap> LittlewoodRichardsonCoefficient([3,2,1],[4,2],[5,4,2,1]);
3
```

The function `LittlewoodRichardsonCoefficient` returns a single Littlewood–Richardson coefficient (although you are really better off asking for all of them, since they will all be calculated anyway).

See also `InducedModule` 72.6 and `InverseLittlewoodRichardsonRule` 72.34. This function requires the package "specht" (see 57.1).

72.34 InverseLittlewoodRichardsonRule

`InverseLittlewoodRichardsonRule(`τ`)`

Returns a list of all pairs of partitions $[\mu,\nu]$ such that the Littlewood-Richardson coefficient $a^{\tau}_{\mu\nu}$ is non-zero (see 72.33). The list returned is unordered and $[\mu,\nu]$ will appear $a^{\tau}_{\mu\nu}$ times in it.

```
gap> InverseLittlewoodRichardsonRule([3,2,1]);
[ [ [  ],[ 3, 2, 1 ] ],[ [ 1 ],[ 3, 2 ] ],[ [ 1 ],[ 2, 2, 1 ] ],
  [ [ 1 ],[ 3, 1, 1 ] ],[ [ 1, 1 ],[ 2, 2 ] ],[ [ 1, 1 ],[ 3, 1 ] ],
  [ [ 1, 1 ],[ 2, 1, 1 ] ],[ [ 1, 1, 1 ],[ 2, 1 ] ],[ [ 2 ],[ 2, 2 ] ],
  [ [ 2 ],[ 3, 1 ] ],[ [ 2 ],[ 2, 1, 1 ] ],[ [ 2, 1 ],[ 3 ] ],
  [ [ 2, 1 ],[ 2, 1 ] ],[ [ 2, 1 ],[ 2, 1 ] ],[ [ 2, 1 ],[ 1, 1, 1 ] ],
  [ [ 2, 1, 1 ],[ 2 ] ],[ [ 2, 1, 1 ],[ 1, 1 ] ],[ [ 2, 2 ],[ 2 ] ],
  [ [ 2, 2 ],[ 1, 1 ] ],[ [ 2, 2, 1 ],[ 1 ] ],[ [ 3 ],[ 2, 1 ] ],
  [ [ 3, 1 ],[ 2 ] ],[ [ 3, 1 ],[ 1, 1 ] ],[ [ 3, 1, 1 ],[ 1 ] ],
  [ [ 3, 2 ],[ 1 ] ],[ [ 3, 2, 1 ],[ ] ] ]
```

See also `LittlewoodRichardsonRule` 72.33.

This function requires the package "specht" (see 57.1).

72.35 EResidueDiagram

`EResidueDiagram(`H`|`e`, `μ`)`

`EResidueDiagram(`x`)`

The e–residue of the (i,j)–th node in the diagram of a partition μ is $(j-i) \bmod e$. `EResidueDiagram(`e`,` μ`)` prints the diagram of the partition μ replacing each node with its e-residue.

If x is a module then `EResidueDiagram(`x`)` prints the e–residue diagrams of all of the e–regular partitions appearing in x (such diagrams are useful when trying to decide how to restrict and induce modules and also in applying results such as the "Scattering theorem" of [JM1]). It is not necessary to supply the integer e in this case because x "knows" the value of e.

```
gap> H:=Specht(2);; EResidueDiagram(H.S(H.P(7,5)));
[ 7, 5 ]
   0   1   0   1   0   1   0
   1   0   1   0   1
[ 6, 5, 1 ]
   0   1   0   1   0   1
   1   0   1   0   1
   0
[ 5, 4, 2, 1 ]
   0   1   0   1   0
   1   0   1   0
   0   1
   1
# There are 3 2-regular partitions.
```

This function requires the package "specht" (see 57.1).

72.36 HookLengthDiagram

`HookLengthDiagram(`μ`)`

Prints the diagram of μ, replacing each node with its hook length (see [JK]).

```
gap> HookLengthDiagram(11,6,3,2);
 14  13  11   9   8   7   5   4   3   2   1
  8   7   5   3   2   1
  4   3   1
  2   1
```

This function requires the package "specht" (see 57.1).

72.37 RemoveRimHook

`RemoveRimHook(`μ`,` *row*`,` *col*`)`

Returns the partition obtained from μ by removing the (*row*, *col*)–th rim hook from (the diagram of) μ.

```
gap> RemoveRimHook([6,5,4],1,2);
[ 4, 3, 1 ]
gap> RemoveRimHook([6,5,4],2,3);
[ 6, 3, 2 ]
```

```
gap> HookLengthDiagram(6,5,4);
   8   7   6   5   3   1
   6   5   4   3   1
   4   3   2   1
```

See also `AddRimHook` 72.38. This function requires the package "specht" (see 57.1).

72.38 AddRimHook

`AddRimHook(`μ`, `r`, `h`);`

Returns a list [ν, l] where ν is the partition obtained from μ by adding a rim hook of length h with its "foot" in the r–th row of (the diagram of) μ and l is the leg length of the wrapped on rim hook (see, for example, [JK]). If the resulting diagram ν is not the diagram of a partition then `false` is returned.

```
gap> AddRimHook([6,4,3],1,3);
[ [ 9, 4, 3 ], 0 ]
gap> AddRimHook([6,4,3],2,3);
false
gap> AddRimHook([6,4,3],3,3);
[ [ 6, 5, 5 ], 1 ]
gap> AddRimHook([6,4,3],4,3);
[ [ 6, 4, 3, 3 ], 0 ]
gap> AddRimHook([6,4,3],5,3);
false
```

See also `RemoveRimHook` 72.37. This function requires the package "specht" (see 57.1).

Operations on partitions

This section contains functions for manipulating partitions and also several useful orderings on the set of partitions.

72.39 ECore

`ECore(`H`|`e`, `μ`)`

The e-core of a partition μ is what remains after as many rim e-hooks as possible have been removed from the diagram of μ (that this is well defined is not obvious; see [JK]). Thus, `ECore(`μ`)` returns the e–core of the partition μ,

```
gap> H:=Specht(6);; ECore(H,16,8,6,5,3,1);
[ 4, 3, 1, 1 ]
```

The e–core is calculated here using James' notation of an **abacus**; there is also an `EAbacus` function; but it is more "pretty" than useful.

See also `IsECore` 72.40, `EQuotient` 72.41, and `EWeight` 72.43. This function requires the package "specht" (see 57.1).

72.40 IsECore

`IsECore(`H`|`e`, `μ`)`

Returns `true` if μ is an e–core and `false` otherwise; see `ECore` 72.39.

See also `ECore` 72.39. This function requires the package "specht" (see 57.1).

72.41 EQuotient

`EQuotient(`H`|`e`, `μ`)`

Returns the e-quotient of μ; this is a sequence of e partitions whose definition can be found in [JK].

```
gap> H:=Specht(8);; EQuotient(H,22,18,16,12,12,1,1);
[ [ 1, 1 ], [  ], [  ], [  ], [  ], [ 2, 2 ], [  ], [ 1 ] ]
```

See also `ECore` 72.39 and `CombineEQuotientECore` 72.42. This function requires the package "specht" (see 57.1).

72.42 CombineEQuotientECore

`CombineEQuotientECore(`H`|`e`, `Q`, `C`)`

A partition is uniquely determined by its e-quotient and its e-core (see 72.41 and 72.39). `CombineEQuotientECore(`e`, `Q`, `C`)` returns the partition which has e–quotient Q and e–core C. The integer e can be replaced with a record H which was created using the function `Specht`.

```
gap> H:=Specht(11);; mu:=[100,98,57,43,12,1];;
gap> Q:=EQuotient(H,mu);
[ [ 9 ], [  ], [  ], [  ], [  ], [  ], [ 3 ], [ 1 ], [ 9 ], [  ], [ 5 ] ]
gap> C:=ECore(H,mu);
[ 7, 2, 2, 1, 1, 1 ]
gap> CombineEQuotientECore(H,Q,C);
[ 100, 98, 57, 43, 12, 1 ]
```

See also `ECore` 72.39 and `EQuotient` 72.41. This function requires the package "specht" (see 57.1).

72.43 EWeight

`EWeight(`H`|`e`, `μ`)`

The e–weight of a partition is the number of e–hooks which must be removed from the partition to reach the e–core (see `ECore` 72.39).

```
gap> EWeight(6,[16,8,6,5,3,1]);
5
```

This function requires the package "specht" (see 57.1).

72.44 ERegularPartitions

`ERegularPartitions(`H`|`e`, `n`)`

A partition $\mu = (\mu_1, \mu_2, \ldots)$ is e–**regular** if there is no integer i such that $\mu_i = \mu_{i+1} = \cdots = \mu_{i+e-1} > 0$. The function `ERegularPartitions(`$e$`, `$n$`)` returns the list of e–regular partitions of n, ordered reverse lexicographically (see 72.50).

```
gap> H:=Specht(3);
Specht(e=3, S(), P(), D(), Pq());
gap> ERegularPartitions(H,6);
[ [ 2, 2, 1, 1 ], [ 3, 2, 1 ], [ 3, 3 ], [ 4, 1, 1 ], [ 4, 2 ],
  [ 5, 1 ], [ 6 ] ]
```

This function requires the package “specht” (see 57.1).

72.45 IsERegular

`IsERegular(`H`|`e`, `μ`)`

Returns `true` if μ is e–regular and `false` otherwise.

This functions requires the package “specht” (see 57.1).

72.46 ConjugatePartition

`ConjugatePartition(`μ`)`

Given a partition μ, `ConjugatePartition(`μ`)` returns the partition whose diagram is obtained by interchanging the rows and columns in the diagram of μ.

```
gap> ConjugatePartition(6,4,3,2);
[ 4, 4, 3, 2, 1, 1 ]
```

This function requires the package “specht” (see 57.1).

72.47 PartitionBetaSet

`PartitionBetaSet(`bn`)`

Given a **set** of beta numbers bn (see `BetaSet` 47.18), this function returns the corresponding partition. Note in particular that bn must be a set of integers.

```
gap> PartitionBetaSet([ 2, 3, 6, 8 ]);
[ 5, 4, 2, 2 ]
```

This function requires the package “specht” (see 57.1).

72.48 ETopLadder

`ETopLadder(`H`|`e`, `μ`)`

The ladders in the diagram of a partition are the lines connecting nodes of constant e–residue, having slope $e-1$ (see [JK]). A new partition can be obtained from μ by sliding all nodes up to the highest possible rungs on their ladders. `ETopLadder(`e`, `μ`)` returns the partition obtained in this way; it is automatically e–regular (this partition is denoted μ^R in [JK]).

```
gap> H:=Specht(4);;
gap> ETopLadder(H,1,1,1,1,1,1,1,1,1,1);
[ 4, 3, 3 ]
gap> ETopLadder(6,1,1,1,1,1,1,1,1,1,1);
[ 2, 2, 2, 2, 2 ]
```

This function requires the package "specht" (see 57.1).

72.49 LengthLexicographic

`LengthLexicographic(`μ`, `ν`)`

`LengthLexicographic` returns `true` if the length of μ is less than the length of ν or if the length of μ equals the length of ν and `Lexicographic(`μ`, `ν`)`.

```
gap> p:=Partitions(6);;Sort(p,LengthLexicographic); p;
[ [ 6 ],[ 5, 1 ],[ 4, 2 ],[ 3, 3 ],[ 4, 1, 1 ],[ 3, 2, 1 ],[ 2, 2, 2 ],
  [ 3, 1, 1, 1 ],[ 2, 2, 1, 1 ],[ 2, 1, 1, 1, 1 ],[ 1, 1, 1, 1, 1, 1 ] ]
```

This function requires the package "specht" (see 57.1).

72.50 Lexicographic

`Lexicographic(`μ`, `ν`)`

`Lexicographic(`μ`, `ν`)` returns `true` if μ is lexicographically greater than or equal to ν.

```
gap> p:=Partitions(6);;Sort(p,Lexicographic); p;
[ [ 6 ],[ 5, 1 ],[ 4, 2 ],[ 4, 1, 1 ],[ 3, 3 ],[ 3, 2, 1 ],
  [ 3, 1, 1, 1 ],[ 2, 2, 2 ],[ 2, 2, 1, 1 ],[ 2, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1 ] ]
```

This function requires the package "specht" (see 57.1).

72.51 ReverseDominance

`ReverseDominance(`μ`, `ν`)`

This is another total order on partitions which extends the dominance ordering (see 47.19). Here μ is greater than ν if for all $i > 0$

$$\sum_{j\geq i}\mu_j > \sum_{j\geq i}\nu_j.$$

```
gap> p:=Partitions(6);;Sort(p,ReverseDominance); p;
[ [ 6 ], [ 5, 1 ], [ 4, 2 ], [ 3, 3 ], [ 4, 1, 1 ], [ 3, 2, 1 ],
  [ 2, 2, 2 ], [ 3, 1, 1, 1 ], [ 2, 2, 1, 1 ], [ 2, 1, 1, 1, 1 ],
  [ 1, 1, 1, 1, 1, 1 ] ]
```

This is the ordering used by James in the appendix of his Springer lecture notes book.

This function requires the package "specht" (see 57.1).

Miscellaneous functions on modules

This section contains some functions for looking at the partitions in a given module for the Hecke algebras. Most of them are used internally by SPECHT.

72.52 Specialized

`Specialized(x [,q]);`
`Specialized(d [,q]);`

Given an element of the Fock space x (see 72.1), or a crystallized decomposition matrix (see 72.4), `Specialized` returns the corresponding element of the Grothendieck ring or the corresponding decomposition matrix of the Hecke algebra respectively. By default the indeterminate `v` is specialized to 1; however `v` can be specialized to any (integer) q by supplying a second argument.

```
gap> H:=Specht(2);; x:=H.Pq(6,2);
S(6,2)+v*S(6,1,1)+v*S(5,3)+v^2*S(5,1,1,1)+v*S(4,3,1)+v^2*S(4,2,2)
+(v^3 + v)*S(4,2,1,1)+v^2*S(4,1,1,1,1)+v^2*S(3,3,1,1)+v^3*S(3,2,2,1)
+v^3*S(3,1,1,1,1,1)+v^3*S(2,2,2,1,1)+v^4*S(2,2,1,1,1,1)
gap> Specialized(x);
S(6,2)+S(6,1,1)+S(5,3)+S(5,1,1,1)+S(4,3,1)+S(4,2,2)
+2*S(4,2,1,1)+S(4,1,1,1,1)+S(3,3,1,1)+S(3,2,2,1)+S(3,1,1,1,1,1)
+S(2,2,2,1,1)+S(2,2,1,1,1,1)
gap> Specialized(x,2);
S(6,2)+2*S(6,1,1)+2*S(5,3)+4*S(5,1,1,1)+2*S(4,3,1)+4*S(4,2,2)+10*S(4,2,1,1)
+4*S(4,1,1,1,1)+4*S(3,3,1,1)+8*S(3,2,2,1)+8*S(3,1,1,1,1,1)+8*S(2,2,2,1,1)
+16*S(2,2,1,1,1,1)
```

An example of `Specialize` being applied to a crystallized decomposition matrix can be found in 72.4. This function requires the package "specht" (see 57.1).

72.53 ERegulars

`ERegulars(`x`)`
`ERegulars(`d`)`
`ListERegulars(`x`)`

`ERegulars(`x`)` prints a list of the e–regular partitions, together with multiplicities, which occur in the module x. `ListERegulars(`x`)` returns an actual list of these partitions rather than printing them.

```
gap> H:=Specht(8);;
gap> x:=H.S(InducedModule(H.P(8,5,3)) );
S(9,5,3)+S(8,6,3)+S(8,5,4)+S(8,5,3,1)+S(6,5,3,3)+S(5,5,4,3)+S(5,5,3,3,1)
gap> ERegulars(x);
[ 9, 5, 3 ]  [ 8, 6, 3 ]  [ 8, 5, 4 ]  [ 8, 5, 3, 1 ]
[ 6, 5, 3, 3 ]  [ 5, 5, 4, 3 ]  [ 5, 5, 3, 3, 1 ]
```

```
gap> H.P(x);
P(9,5,3)+P(8,6,3)+P(8,5,4)+P(8,5,3,1)
```

This example shows why these functions are useful: given a projective module x, as above, and the list of e–regular partitions in x we know the possible indecomposable direct summands of x.

Note that it is not necessary to specify what e is when calling this function because x "knows" the value of e.

The function `ERegulars` can also be applied to a decomposition matrix d; in this case it returns the unitriangular submatrix of d whose rows and columns are indexed by the e–regular partitions.

These function requires the package "specht" (see 57.1).

72.54 SplitECores

`SplitECores(`x`)`
`SplitECores(`x`, `μ`)`
`SplitECores(`x`, `y`)`

The function `SplitECores(`x`)` returns a list `[`b_1`,...,`b_k`]` where the Specht modules in each b_i all belong to the same block (i.e. they have the same e-core). Similarly, `SplitECores(`x`, `μ`)` returns the component of x which is in the same block as μ, and `SplitECores(`x`, `y`)` returns the component of x which is in the same block as y.

```
gap> H:=Specht(2);;
gap> SplitECores(InducedModule(H.S(5,3,1)));
[ S(6,3,1)+S(5,3,2)+S(5,3,1,1), S(5,4,1) ]
gap> InducedModule(H.S(5,3,1),0);
S(5,4,1)
gap> InducedModule(H.S(5,3,1),1);
S(6,3,1)+S(5,3,2)+S(5,3,1,1)
```

See also `ECore` 72.39, `InducedModule` 72.6, and `RestrictedModule` 72.8.

This function requires the package "specht" (see 57.1).

72.55 Coefficient of Specht module

`Coefficient(`x`, `μ`)`

If x is a sum of Specht (resp. simple, or indecomposable) modules, then `Coefficient(`x`, `μ`)` returns the coefficient of `S(`μ`)` in x (resp. `D(`μ`)`, or `P(`μ`)`).

```
gap> H:=Specht(3);; x:=H.S(H.P(7,3));
S(7,3)+S(7,2,1)+S(6,2,1^2)+S(5^2)+S(5,2^2,1)+S(4^2,1^2)+S(4,3^2)+S(4,3,2,1)
gap> Coefficient(x,5,2,2,1);
1
```

This function requires the package "specht" (see 57.1).

72.56 InnerProduct

`InnerProduct(`x`, `y`)`

Here x and y are some modules of the Hecke algebra (i.e. Specht modules, PIMS, or simple modules). `InnerProduct(`x`, `y`)` computes the standard inner product of these elements. This is sometimes a convenient way to compute decomposition numbers (for example).

```
gap> InnerProduct(H.S(2,2,2,1), H.P(4,3));
1
gap> DecompositionNumber(H,[2,2,2,1],[4,3]);
1
```

This function requires the package "specht" (see 57.1).

72.57 SpechtPrettyPrint

```
SpechtPrettyPrint(true)
SpechtPrettyPrint(false)
SpechtPrettyPrint()
```

This function changes the way in which SPECHT prints modules. The first two forms turn pretty printing on and off respectively (by default it is off), and the third form toggles the printing format.

```
gap> H:=Specht(2);; x:=H.S(H.P(6));;
gap> SpechtPrettyPrint(true); x;
S(6)+S(5,1)+S(4,1^2)+S(3,1^3)+S(2,1^4)+S(1^6)
gap> SpechtPrettyPrint(false); x;
S(6)+S(5,1)+S(4,1,1)+S(3,1,1,1)+S(2,1,1,1,1)+S(1,1,1,1,1,1)
gap> SpechtPrettyPrint(); x;
S(6)+S(5,1)+S(4,1^2)+S(3,1^3)+S(2,1^4)+S(1^6)
```

This function requires the package "specht" (see 57.1).

Semi–standard and standard tableaux

These functions are not really part of SPECHT proper; however they are related and may well be of use to someone. Tableaux are represented as lists, where the first element of the list is the first row of the tableaux and so on.

72.58 SemistandardTableaux

`SemistandardTableaux(`μ`, `ν`)`

μ a partition, ν a composition.

Returns a list of the semistandard μ–tableaux of type ν [JK]. Tableaux are represented as lists of lists, with the first element of the list being the first row of the tableaux and so on.

```
gap> SemistandardTableaux([4,3],[1,1,1,2,2]);
```

```
[ [ [ 1, 2, 3, 4 ], [ 4, 5, 5 ] ], [ [ 1, 2, 3, 5 ], [ 4, 4, 5 ] ],
  [ [ 1, 2, 4, 4 ], [ 3, 5, 5 ] ], [ [ 1, 2, 4, 5 ], [ 3, 4, 5 ] ],
  [ [ 1, 3, 4, 4 ], [ 2, 5, 5 ] ], [ [ 1, 3, 4, 5 ], [ 2, 4, 5 ] ] ]
```

See also `StandardTableaux` 72.59. This function requires the package "specht" (see 57.1).

72.59 StandardTableaux

`StandardTableaux(`μ`)`

μ a partition.

Returns a list of the standard μ–tableaux.

```
gap> StandardTableaux(4,2);
[ [ [ 1, 2, 3, 4 ], [ 5, 6 ] ], [ [ 1, 2, 3, 5 ], [ 4, 6 ] ],
  [ [ 1, 2, 3, 6 ], [ 4, 5 ] ], [ [ 1, 2, 4, 5 ], [ 3, 6 ] ],
  [ [ 1, 2, 4, 6 ], [ 3, 5 ] ], [ [ 1, 2, 5, 6 ], [ 3, 4 ] ],
  [ [ 1, 3, 4, 5 ], [ 2, 6 ] ], [ [ 1, 3, 4, 6 ], [ 2, 5 ] ],
  [ [ 1, 3, 5, 6 ], [ 2, 4 ] ] ]
```

See also `SemistandardTableaux` 72.58. This function requires the package "specht" (see 57.1).

72.60 ConjugateTableau

`ConjugateTableau(`*tab*`)`

Returns the tableau obtained from *tab* by interchangings its rows and columns.

```
gap> ConjugateTableau([ [ 1, 3, 5, 6 ], [ 2, 4 ] ]);
[ [ 1, 2 ], [ 3, 4 ], [ 5 ], [ 6 ] ]
```

This function requires the package "specht" (see 57.1).

72.61 ShapeTableau

`ShapeTableau(`*tab*`)`

Given a tableau *tab* this function returns the partition (or composition).

```
gap> ShapeTableau( [ [ 1, 1, 2, 3 ], [ 4, 5 ] ] );
[ 4, 2 ]
```

This function requires the package "specht" (see 57.1).

72.62 TypeTableau

`TypeTableau(`*tab*`)`

Returns the type of the (semistandard) tableau *tab*; that is, the composition $\sigma = (\sigma_1, \sigma_2, \ldots)$ where σ_i is the number of entries in *tab* which are equal to i.

```
gap> List(SemistandardTableaux([5,4,2],[4,3,0,1,3]),TypeTableau);
[ [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ],
  [ 4, 3, 0, 1, 3 ], [ 4, 3, 0, 1, 3 ] ]
```

This function requires the package "specht" (see 57.1).

Chapter 73

Vector Enumeration

This chapter describes the Vector Enumeration (Version 3) share library package for computing matrix representations of finitely presented algebras. See 57.15 for the installation of the package, and the Vector Enumeration manual [Lin93] for details of the implementation.

The default application of Vector Enumeration, namely the function `Operation` for finitely presented algebras (see chapter 40), is described in 73.1.

The interface between GAP3 and Vector Enumeration is described in 73.2.

In 73.3 the examples given in the Vector Enumeration manual serve as examples for the use of Vector Enumeration with GAP3.

Finally, section 73.4 shows how the MeatAxe share library (see chapter 69) and Vector Enumeration can work hand in hand.

The functions of the package can be used after loading the package with

```
gap> RequirePackage( "ve" );
```

The package is also loaded **automatically** when `Operation` is called for the action of a finitely presented algebra on a quotient module.

73.1 Operation for Finitely Presented Algebras

`Operation(` F`,` Q `)`

This is the default application of Vector Enumeration. F is a finitely presented algebra (see chapter 40), Q is a quotient of a free F-module, and the result is a matrix algebra representing a faithful action on Q.

If Q is the zero module then the matrices have dimension zero, so the result is a null algebra (see 41.9) consisting only of a zero element.

The algebra homomorphism, the isomorphic module for the matrix algebra, and the module homomorphism can be constructed as described in chapters 39 and 42.

```
gap> a:= FreeAlgebra( GF(2), 2 );
UnitalAlgebra( GF(2), [ a.1, a.2 ] )
gap> a:= a / [ a.1^2 - a.one, # group algebra of V4 over GF(2)
```

```
>               a.2^2 - a.one,
>               a.1*a.2 - a.2*a.1 ];
UnitalAlgebra( GF(2), [ a.1, a.2 ] )
gap> op:= Operation( a, a^1 );
UnitalAlgebra( GF(2),
[ [ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2),
          Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ],
  [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ] ] )
gap> Size( op );
16
```

73.2 More about Vector Enumeration

As stated in the introduction to this chapter, Vector Enumeration is a share library package. The computations are done by standalone programs written in C.

The interface between Vector Enumeration and GAP3 consists essentially of two parts, namely the global variable `VE`, and the function `FpAlgebraOps.OperationQuotientModule`.

The VE record

`VE` is a record with components

`Path`
: the full path name of the directory that contains the executables of the standalones `me`, `qme`, `zme`,

`options`
: a string with command line options for Vector Enumeration; it will be appended to the command string of `CallVE` (see below), so the default options chosen there can be overwritten. This may be useful for example in case of the `-v` option to enable the printing of comments (see section 4.3 of [Lin93]), but you should **not** change the output file (using `-o`) when you simply call `Operation` for a finitely presented algebra. `options` is defaulted to the empty string.

FpAlgebraOps.OperationQuotientModule

This function is called automatically by `FpAlgebraOps.Operation` (see 73.1), it can also be called directly as follows.

`FpAlgebraOps.OperationQuotientModule(` A, Q, *opr* `)`
`FpAlgebraOps.OperationQuotientModule(` A, Q, `"mtx")`

It takes a finitely presented algebra A and a list of submodule generators Q, that is, the entries of Q are list of equal length, with entries in A, and returns the matrix representation computed by the Vector Enumeration program.

The third argument must be either one of the operations `OnPoints`, `OnRight`, or the string `"mtx"`. In the latter case the output will be an algebra of MeatAxe matrices, see 73.4 for further explanation.

Accessible Subroutines

The following three functions are used by `FpAlgebraOps.OperationQuotientModule`. They are the real interface that allows to access Vector Enumeration from GAP3.

`PrintVEInput(` *A*`,` *Q*`,` *names* `)`

takes a finitely presented algebra A, a list of submodule generators Q, and a list *names* of names the generators shall have in the presentation that is passed to Vector Enumeration, and prints a string that represents the input presentation for Vector Enumeration. See section 3.1 of the Vector Enumeration manual [Lin93] for a description of the syntax.

```
gap> PrintVEInput( a, [ [ a.zero ] ], [ "A", "B" ] );
2.
A B .
.
.
{1}(0).
A*A, B*B, :
A*B+B*A = 0, .
```

`CallVE(` *commandstr*`,` *infile*`,` *outfile*`,` *options* `)`

calls Vector Enumeration with command string *commandstr*, presentation file *infile*, and command line options *options*, and prescribes the output file *outfile*.

If not overwritten in the string *options*, the default options `"-i -P -v0 -Y VE.out -L# "` are chosen.

Of course it is not necessary that *infile* was produced using `PrintVEInput`, and also the output is independent of GAP3.

```
gap> PrintTo( "infile.pres",
>             PrintVEInput( a, [ [ a.zero ] ], [ "A", "B" ] ) );
gap> CallVE( "me", "infile", "outfile", " -G -vs2" );
```

(The option `-G` sets the output format to GAP3, `-vs2` chooses a more verbose mode.)

`VEOutput(` *A*`,` *Q*`,` *names*`,` *outfile* `)`
`VEOutput(` *A*`,` *Q*`,` *names*`,` *outfile*`, "mtx")`

returns the output record produced by Vector Enumeration that was written to the file *outfile*. A component `operation` is added that contains the information for the construction of the operation homomorphisms.

The arguments A, Q, *names* describe the finitely presented algebra, the quotient module it acts on, and the chosen generators names, i.e., the original structures for that Vector Enumeration was called.

```
gap> out:= VEOutput( a, [ [ a.zero ] ], [ "A", "B" ], "outfile" );;
```

```
gap> out.dim; out.operation.moduleinfo.preimagesBasis;
4
[ [ a.one ], [ a.2 ], [ a.1 ], [ a.1*a.2 ] ]
```

If the optional fifth argument `"mtx"` is present, the output is regarded as an algebra of MeatAxe matrices (see section 73.4). For that, an appropriate command string had to be passed to `CallVE`.

73.3 Examples of Vector Enumeration

We consider those of the examples given in chapter 8 of the Vector Enumeration manual that can be used in GAP3.

8.1 The natural permutation representation of S_3

The symmetric group S_3 is also the dihedral group D_6, and so is presented by two involutions with product of order 3. Taking the permutation action on the cosets of the cyclic group generated by one of the involutions we obtain the following presentation.

```
gap> a:= FreeAlgebra( Rationals, 2 );;
gap> a:= a / [ a.1^2 - a.one, a.2^2 - a.one,
>              (a.1*a.2)^3 - a.one ];
UnitalAlgebra( Rationals, [ a.1, a.2 ] )
gap> a.name:= "a";;
```

We choose as module `q` the quotient of the regular module for `a` by the submodule generated by `a.1 - 1`, and compute the action of `a` on `q`.

```
gap> m:= a^1;;
gap> q:= m / [ [ a.1 - a.one ] ];
Module( a, [ [ a.one ] ] ) / [ [ -1*a.one+a.1 ] ]
gap> op:= Operation( a, q );
UnitalAlgebra( Rationals,
[ [ [ 1, 0, 0 ], [ 0, 0, 1 ], [ 0, 1, 0 ] ],
  [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] ] )
gap> op.name:= "op";;
```

8.2 A Quotient of a Permutation Representation

The permutation representation constructed in example 8.1 fixes the all-ones vector (as do all permutation representations). This is the image of the module element `[ a.one + a.2 + a.2*a.1 ]` in the corresponding module for the algebra `op`.

```
gap> ophom:= OperationHomomorphism( a, op );;
gap> opmod:= OperationModule( op );
Module( op, [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
gap> modhom:= OperationHomomorphism( q, opmod );;
gap> pre:= PreImagesRepresentative( modhom, [ 1, 1, 1 ] );;
gap> pre:= pre.representative;
[ a.one+a.2+a.2*a.1 ]
```

We could have computed such a preimage also by computing a matrix that maps the image of the submodule generator of `q` to the all-ones vector, and applying a preimage to the submodule generator. Of course the we do not necessarily get the same representatives.

```
gap> images:= List( Generators( q ), x -> Image( modhom, x ) );
[ [ 1, 0, 0 ] ]
gap> rep:= RepresentativeOperation( op, images[1], [ 1, 1, 1 ] );
[ [ 1, 1, 1 ], [ 1, 1, 1 ], [ 1, 1, 1 ] ]
gap> PreImagesRepresentative( ophom, rep );
a.one+a.1*a.2+a.2*a.1
```

Now we factor out the fixed submodule by enlarging the denominator of the module `q`. (Note that we could also compute the action of the matrix algebra if we were only interested in the 2-dimensional representation.)

Accordingly we can write down the following presentation for the quotient module.

```
gap> q:= m / [ [ a.1 - a.one ], pre ];;
gap> op:= Operation( a, q );
UnitalAlgebra( Rationals,
[ [ [ 1, 0 ], [ -1, -1 ] ], [ [ 0, 1 ], [ 1, 0 ] ] ] )
```

8.3 A Non-cyclic Module

If we take the direct product of two copies of the permutation representation constructed in example 8.1, we can identify the fixed vectors in the two copies in the following presentation.

```
gap> m:= a^2;;
gap> q:= m / [ [ a.zero, a.1 - a.one ], [ a.1 - a.one, a.zero ],
>              [ a.one+a.2+a.2*a.1, -a.one-a.2-a.2*a.1 ] ];
Module( a, [ [ a.one, a.zero ], [ a.zero, a.one ] ] ) /
[ [ a.zero, -1*a.one+a.1 ], [ -1*a.one+a.1, a.zero ],
  [ a.one+a.2+a.2*a.1, -1*a.one+-1*a.2+-1*a.2*a.1 ] ]
```

We compute the matrix representation.

```
gap> op:= Operation( a, q );
UnitalAlgebra( Rationals,
[ [ [ 1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 0, 1, 0 ],
      [ 0, 0, 1, 0, 0 ], [ 1, -1, 1, 1, -1 ] ],
  [ [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 1 ], [ 1, 0, 0, 0, 0 ],
      [ 0, 0, 0, 1, 0 ], [ 0, 1, 0, 0, 0 ] ] ] )
```

In this case it is interesting to look at the images of the module generators and pre-images of the basis vectors. Note that these preimages are elements of a factor module, corresponding elements of the free module are again found as representatives.

```
gap> ophom:= OperationHomomorphism( a, op );;
gap> opmod:= OperationModule( op );;
gap> opmod.name:= "opmod";;
gap> modhom:= OperationHomomorphism( q, opmod );;
gap> List( Generators( q ), x -> Image( modhom, x ) );
[ [ 1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ] ]
gap> basis:= Basis( opmod );
CanonicalBasis( opmod )
gap> preim:= List( basis.vectors, x ->
>               PreImagesRepresentative( modhom, x ) );;
```

```
gap> preim:= List( preim, Representative );
[ [ a.one, a.zero ], [ a.zero, a.one ], [ a.2, a.zero ],
  [ a.2*a.1, a.zero ], [ a.zero, a.2 ] ]
```

8.4 A Monoid Representation

The Coxeter monoid of type B_2 has a transformation representation on four points. This can be constructed as a matrix representation over GF(3), from the following presentation.

```
gap> a:= FreeAlgebra( GF(3), 2 );;
gap> a:= a / [ a.1^2 - a.1, a.2^2 - a.2,
> (a.1*a.2)^2 - (a.2*a.1)^2 ];;
gap> q:= a^1 / [ [ a.1 - a.one ] ];;
gap> op:= Operation( a, q );
UnitalAlgebra( GF(3),
[ [ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3)^0,
          0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ],
  [ [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ] ] )
```

8.7 A Quotient of a Polynomial Ring

The quotient of a polynomial ring by the ideal generated by some polynomials will be finite-dimensional just when the polynomials have finitely many common roots in the algebraic closure of the ground ring. For example, three polynomials in three variables give us the following presentation for the quotient of their ideal.

Define `a` to be the polynomial algebra on three variables.

```
gap> a:= FreeAlgebra( Rationals, 3 );;
gap> a:= a / [ a.1 * a.2 - a.2 * a.1,
>              a.1 * a.3 - a.3 * a.1,
>              a.2 * a.3 - a.3 * a.2 ];;
```

Define the quotient module by the polynomials `A+B+C`, `AB+BC+CA`, `ABC-1`.

```
gap> q:= a^1 / [ [ a.1+a.2+a.3 ],
>                [ a.1*a.2+a.2*a.3+a.3*a.1 ],
>                [ a.1*a.2*a.3-a.one ]          ];;
```

Compute the representation.

```
gap> op:= Operation( a, q );
UnitalAlgebra( Rationals,
[ [ [ 0, 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1, 0 ],
      [ -1, 0, 0, 0, 0, -1 ], [ 0, 0, 1, 0, 0, 0 ],
      [ 1, 0, 0, 0, 0, 0 ], [ 0, -1, 0, -1, 0, 0 ] ],
  [ [ 0, 0, 0, 1, 0, 0 ], [ 0, 0, 1, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 1 ],
      [ 0, 0, -1, 0, -1, 0 ], [ -1, 0, 0, 0, 0, -1 ],
      [ 0, 1, 0, 0, 0, 0 ] ],
```

```
  [ [ 0, -1, 0, -1, 0, 0 ], [ 0, 0, -1, 0, -1, 0 ],
      [ 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1, 0 ],
      [ 0, 0, 0, 0, 0, 1 ], [ 0, 0, 0, 1, 0, 0 ] ] ] )
```

73.4 Using Vector Enumeration with the MeatAxe

One can deal with the matrix representation constructed by Vector Enumeration also using the MeatAxe share library. This way the matrices are not read into GAP3 but written to files and converted into internal MeatAxe format. See chapter 69 for details.

```
gap> a:= FreeAlgebra( GF(2), 2 );;
gap> a:= a / [ a.1^2 - a.one, a.2^2 - a.one,
>              (a.1*a.2)^3 - a.one ];;
gap> RequirePackage("meataxe");
#I  The MeatAxe share library functions are available now.
#I  All files will be placed in the directory
#I     '/var/tmp/tmp.006103'
#I  Use 'MeatAxe.SetDirectory( <path> )' if you want to change.
gap> op:= Operation( a, a^1, "mtx" );
UnitalAlgebra( GF(2),
[ MeatAxeMat( "/var/tmp/tmp.006103/a/g.1", GF(2), [ 6, 6 ], a.1 ),
  MeatAxeMat( "/var/tmp/tmp.006103/a/g.2", GF(2), [ 6, 6 ], a.2 ) ] )
gap> Display( op.1 );
#I  calling 'maketab' for field of size 2
MeatAxe.Matrix := [
[0,0,1,0,0,0],
[0,0,0,1,0,0],
[1,0,0,0,0,0],
[0,1,0,0,0,0],
[0,0,0,0,0,1],
[0,0,0,0,1,0]
]*Z(2);
gap> MeatAxe.Unbind();
```

Chapter 74

AREP

The share package AREP provides an infrastructure and high level functions to do efficient calculations in constructive representation theory. By the term "constructive" we mean that group representations are constructed and manipulated up to equality – not only up to equivalence as it is done by using characters. Hence you can think of it as working with matrix representations, but in a very efficient way using the special structure of the matrices occuring in representation theory of finite groups. The package is named after its most important class **ARep** (see 74.66) (**A**bstract **Rep**resentations)[1] implementing this idea.

A striking application of constructive representation theory is the decomposition of matrices representing discrete signal transforms into a product of highly structured sparse matrices (realized in 74.147). This decomposition can be viewed as a fast algorithm for the signal transform. Another application is the construction of fast Fourier transforms for solvable groups (realized in 74.123). The package has evolved out of this area of application into a more general tool.

The package AREP consists of the following parts:

- **Monomial Matrices:** A monomial matrix is matrix containing exactly one non-zero entry in every row and column. Hence storing and computing with monomial matrices can be done efficiently. This is realized in the class **Mon**, Sections 74.2 – 74.21.

- **Structured Matrices:** The class **AMat**, Sections 74.22 – 74.65, is created to represent and calculate with structured matrices, like e.g. $2 \cdot (A \oplus B)^C \otimes D \cdot E^2$, where A, B, C, D, E are matrices of compatible size and characteristic.

- **Group Representations:** The class **ARep**, Sections 74.66 – 74.123, is created to represent and manipulate structured representations up to equality, like e.g. $(\phi \uparrow_T G)^M \otimes \psi$. Special care is taken of monomial representations.

[1] A note on the name: We have chosen "abstract" because we manipulate expressions for representations, not constants. However, "concrete" would also be right because the representations are given with respect to a fixed basis of the underlying vector space. The name ARep is thus, for historical reasons, somewhat misleading.

- **Symmetry of Matrices:** In Sections 74.124 – 74.127 functions are provided to compute certain kinds of symmetry of a given matrix. Symmetry allows to describe structure contained in a matrix.

- **Discrete Signal Transforms:** Sections 74.128 – 74.146 describe functions to construct many well-known discrete signal transforms.

- **Matrix Decomposition:** Sections 74.147 – 74.149 describe functions to decompose a discrete signal transform into a product of highly structured sparse matrices.

- **Tools for Complex Numbers, Matrices and Permutations:** Sections 74.151 – 74.169 describe useful tools for the computation with complex numbers, matrices and permutations.

All functions described are written entirely in the GAP3 language. The functions for the computation of the symmetry of a matrix (see 74.124) may use the external C program `desauto` written by J. Leon and contained in the share package GUAVA. However, the use of this program is optional and will only influence the speed and not the executability of the functions.

The package AREP was created in the framework of our theses where the background of constructive representation theory (see [Püs98]) and searching for symmetry of matrices (see [Egn97a]) can be found.

74.1 Loading AREP

After having started GAP3 the AREP package needs to be loaded. This is done by typing:

```
gap> RequirePackage("arep");

                ___   ___   ___  ___
               |   | |   | |    |   |   Version 1.0, 16 Mar 1998
               |___| |___| |___ |___|
               |   | |  \  |    |       by Sebastian Egner
               |   | |   \ |___ |          Markus Pueschel

              Abstract REPresentations
```

If AREP isn't already in memory it is loaded and its banner is displayed. If you are a frequent user of AREP you might consider putting this line into your `.gaprc` file.

74.2 Mons

The class **Mon** is created to represent and calculate efficiently with monomial matrices. A monomial matrix is a matrix which contains exactly one non-zero entry in every row and every column. Hence monomial matrices are always invertible and a generalization of permutation matrices. The elements of the class **Mon** are called "mons". A mon m is a record with at least the following fields.

`isMon`	:=	`true`
`isGroupElement`	:=	`true`
`domain`	:=	`GroupElements`
`operations`	:=	`MonOps`
`char`	:	characteristic of the base field
`perm`	:	a permutation
`diag`	:	a list of non-zero field elements

The MonOps class is derived from the GroupElementOps class, so that groups of mons can be constructed. The monomial matrix represented by a mon m is given by

$$[\delta_{i^p j} \mid i, j \in \{1, \ldots, \texttt{Length}(m.\texttt{diag})\}] \cdot \texttt{ApplyFunc}(\texttt{DiagonalMat}, m.\texttt{diag}),$$

where $p = m$`.perm` and $\delta_{k\ell}$ denotes the Kronecker symbol ($\delta_{k\ell} = 1$ if $k = \ell$ and $= 0$ else). Mons are created using the function `Mon`. The following sections describe functions used for the calculation with mons.

Some remarks on the design of **Mon**: Mons cannot be mixed with GAP3-matrices (which are just lists of lists of field elements); use `MonMat` (74.11) and `MatMon` (74.10) to convert explicitly. Mons are lightweighted, e.g. only the characteristic of the base field is stored. Mons are group elements but there are no efficient functions implemented to compute with mon groups. You should think of mons as being a similar thing as integers or permutations: They are just fundamental objects to work with.

The functions concerning mons are implemented in the file `"arep/lib/mon.g"`.

74.3 Comparison of Mons

m_1 `=` m_2
m_1 `<>` m_2

The equality operator `=` evaluates to `true` if the mons m_1 and m_2 are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the mons m_1 and m_2 are not equal and to `false` otherwise.

Two mons are equal iff they define the same monomial matrix. Note that the monomial matrix being represented has a certain size. The sizes must agree, too.

m_1 `<` m_2
m_1 `<=` m_2
m_1 `>=` m_2
m_1 `>` m_2

The operators `<`, `<=`, `>=`, and `>` evaluate to `true` if the mon m_1 is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the mon m_2.

The ordering of mons m is defined via the ordering of the pairs `[`m`.perm,` m`.diag]`.

74.4 Basic Operations for Mons

The MonOps class is derived from the GroupElementsOps class.

m_1 `*` m_2
m_1 `/` m_2

The operators `*` and `/` evaluate to the product and quotient of the two mons m_1 and m_2. The product is defined via the product of the corresponding (monomial) matrices. Of course the mons must be of equal size and characteristic otherwise an error is signaled.

m_1 `^` m_2

The operator `^` evaluates to the conjugate $m_2^{-1} * m_1 * m_2$ of m_1 under m_2 for two mons m_1 and m_2. The mons must be of equal size and characteristic otherwise an error is signaled.

m `^` i

The powering operator `^` returns the i-th power of the mon m and the integer i.

`Comm(` m_1`,` m_2 `)`

`Comm` returns the commutator $m_1^{-1} * m_2^{-1} * m_1 * m_2$ of two mons m_1 and m_2. The operands must be of equal size and characteristic otherwise an error is signaled.

`LeftQuotient(` m_1`,` m_2 `)`

`LeftQuotient` returns the left quotient $m_1^{-1} * m_2$ of two mons m_1 and m_2. The operands must be of equal size and characteristic otherwise an error is signaled.

74.5 Mon

`Mon(` p`,` D `)`

Let p be a permutation and D a list of field elements $\neq 0$ of the same characteristic. `Mon` returns a mon representing the monomial matrix given by $[\delta_{i^p j} \mid i, j \in \{1, \ldots, \texttt{Length}(D)\}] \cdot$ $\texttt{ApplyFunc}(\texttt{DiagonalMat}, D)$, where $\delta_{k\ell}$ denotes the Kronecker symbol. The function will signal an error if the length of D is less than the largest moved point of p.

```
gap> Mon( (1,2), [1, 2, 3] );
Mon(
  (1,2),
  [ 1, 2, 3 ]
)
gap> Mon( (1,3,4), [Z(3)^0, Z(3)^2, Z(3), Z(9)]);
Mon(
  (1,3,4),
  [ Z(3)^0, Z(3)^0, Z(3), Z(3^2) ]
)
```

`Mon(` D`,` p `)`

`Mon` returns a mon representing the monomial matrix given by $\texttt{ApplyFunc}(\texttt{DiagonalMat}, D) \cdot [\delta_{i^p j} \mid i, j \in \{1, \ldots, \texttt{Length}(D)\}]$, where $\delta_{k\ell}$ denotes the Kronecker symbol. Note that in the output the diagonal is commuted to the right side, but it still represents the same monomial matrix.

```
gap> Mon( [1,2,3], (1,2) );
Mon(
  (1,2),
  [ 2, 1, 3 ]
)
gap> Mon( [Z(3)^0, Z(3)^2, Z(3), Z(9)], (1,3,4) );
Mon(
  (1,3,4),
  [ Z(3^2), Z(3)^0, Z(3)^0, Z(3) ]
)
```

`Mon(` *D* `)`

`Mon` returns a mon representing the (monomial) diagonal matrix given by the list D.

```
gap> Mon( [1, 2, 3, 4] );
Mon( [ 1, 2, 3, 4 ] )
```

`Mon(` *p*`,` *d* `)`
`Mon(` *p*`,` *d*`,` *char* `)`
`Mon(` *p*`,` *d*`,` *field* `)`

Let p be a permutation and d a positive integer. `Mon` returns a mon representing the $(d \times d)$ permutation matrix corresponding to p using the convention $[\delta_{i^p j} \mid i, j \in \{1, \ldots, d\}]$, where $\delta_{k\ell}$ denotes the Kronecker symbol. As optional parameter a characteristic *char* or a *field* can be supplied. The default characteristic is zero. The function will signal an error if the degree d is less than the largest moved point of p.

```
gap> Mon( (1,2), 3 );
Mon( (1,2), 3 )
gap> Mon( (1,2,3), 3, 5 );
Mon( (1,2,3), 3, GF(5) )
```

`Mon(` *m* `)`

Let m a mon. `Mon` returns m.

```
gap> Mon( Mon( (1,2), [1, 2, 3] ) );
Mon(
  (1,2),
  [ 1, 2, 3 ]
)
```

74.6 IsMon

`IsMon(` *obj* `)`

`IsMon` returns `true` if *obj*, which may be an object of arbitrary type, is a mon, and `false` otherwise. The function will signal an error if *obj* is an unbound variable.

```
gap> IsMon( Mon( (1,2), [1, 2, 3] ) );
```

```
true
gap> IsMon( (1,2) );
false
```

74.7 IsPermMon

`IsPermMon(` *m* `)`

`IsPermMon` returns `true` if the mon m represents a permutation matrix and `false` otherwise.

```
gap> IsPermMon( Mon( (1,2), [1, 2, 3] ) );
false
gap> IsPermMon( Mon( (1,2), 2) );
true
```

74.8 IsDiagMon

`IsDiagMon(` *m* `)`

`IsDiagMon` returns `true` if the mon m represents a diagonal matrix and `false` otherwise.

```
gap> IsDiagMon( Mon( (1,2), 2) );
false
gap> IsDiagMon( Mon( [1, 2, 3, 4] ) );
true
```

74.9 PermMon

`PermMon(` *m* `)`

`PermMon` converts the mon m to a permutation if possible and returns `false` otherwise.

```
gap> PermMon( Mon( (1,2), 5) );
(1,2)
gap> PermMon( Mon( [1,2] ) );
false
```

74.10 MatMon

`MatMon(` *m* `)`

`MatMon` converts the mon m to a matrix (i.e. a list of lists of field elements).

```
gap> MatMon( Mon( (1,2), [1, 2, 3] ) );
[ [ 0, 2, 0 ], [ 1, 0, 0 ], [ 0, 0, 3 ] ]
gap> MatMon( Mon( (1,2), 3) );
[ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ]
```

74.11 MonMat

`MonMat(` *M* `)`

`MonMat` converts the matrix M to a mon if possible and returns `false` otherwise.

```
gap> MonMat( [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] );
Mon( (1,2), 3 )
gap> MonMat( [ [ 0, 1, 0 ], [ E(3), 0, 0 ], [ 0, 0, 4 ] ] );
Mon(
  (1,2),
  [ E(3), 1, 4 ]
)
```

74.12 DegreeMon

`DegreeMon(` *m* `)`

`DegreeMon` returns the degree of the mon *m*. The degree is the size of the represented matrix.

```
gap> DegreeMon( Mon( (1,2), [1, 2, 3] ) );
3
```

74.13 CharacteristicMon

`CharacteristicMon(` *m* `)`

`CharacteristicMon` returns the characteristic of the field from which the components of the mon *m* are.

```
gap> CharacteristicMon( Mon( [1,2] ) );
0
gap> CharacteristicMon( Mon( (1,2), 4, 5) );
5
```

74.14 OrderMon

`OrderMon(` *m* `)`

`OrderMon` returns the order of the monomial matrix represented by the mon *m*. The order of *m* is the least positive integer *r* such that m^r is the identity. Note that the order might be infinite.

```
gap> OrderMon( Mon( [1,2] ) );
"infinity"
gap> OrderMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
6
```

74.15 TransposedMon

`TransposedMon(` *m* `)`

`TransposedMon` returns a mon representing the transposed monomial matrix of the mon *m*.

```
gap> TransposedMon( Mon( [1,2] ) );
Mon( [ 1, 2 ] )
gap> TransposedMon( Mon( (1,2,3), 4 ) );
Mon( (1,3,2), 4 )
```

74.16 DeterminantMon

`DeterminantMon(` m `)`

`DeterminantMon` returns the determinant of the monomial matrix represented by the mon m.

```
gap> DeterminantMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
-1
gap> DeterminantMon( Mon( [1,2] ) );
2
```

74.17 TraceMon

`TraceMon(` m `)`

`TraceMon` returns the trace of the monomial matrix represented by the mon m.

```
gap> TraceMon( Mon( (1,2), 4, 5) );
Z(5)
gap> TraceMon( Mon( [1,2] ) );
3
```

74.18 GaloisMon

`GaloisMon(` m, *aut* `)`
`GaloisMon(` m, k `)`

`GaloisMon` returns a mon which is a galois conjugate of the mon m. This means that each component of the represented matrix is mapped with an automorphism of the underlying field. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime.

```
gap> GaloisMon( Mon( (1,2), [1, E(3), E(3)^2] ), -1 );
Mon(
  (1,2),
  [ 1, E(3)^2, E(3) ]
)
gap> aut := FrobeniusAutomorphism( GF(4) );
FrobeniusAutomorphism( GF(2^2) )
gap> GaloisMon( Mon( [ Z(2)^0, Z(2^2), Z(2^2)^2 ] ), aut );
Mon( [ Z(2)^0, Z(2^2)^2, Z(2^2) ] )
```

74.19 DirectSumMon

`DirectSumMon(` m_1, ..., m_k `)`

`DirectSumMon` returns the direct sum of the mons m_1, ..., m_k. The direct sum of mons is defined via the direct sum of the represented matrices. Note that the mons must have the same characteristic.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
```

```
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> DirectSumMon( m1, m2 );
Mon(
  (1,2)(4,5),
  [ 1, E(3), E(3)^2, 1, 1, 1 ]
)
```

`DirectSumMon(` *list* `)`

`DirectSumMon` returns a mon representing the direct sum of the mons in *list*.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> DirectSumMon( [m1, m2] );
Mon(
  (1,2)(4,5),
  [ 1, E(3), E(3)^2, 1, 1, 1 ]
)
```

74.20 TensorProductMon

`TensorProductMon(` m_1`, ...,` m_k `)`

`TensorProductMon` returns the tensor product of the mons m_1, ..., m_k. The tensor product of mons is defined via the tensor product (or Kronecker product) of the represented matrices. Note that the mons must have the same characteristic.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> TensorProductMon( m1, m2 );
Mon(
  (1,5)(2,4)(3,6)(7,8),
  [ 1, 1, 1, E(3), E(3), E(3), E(3)^2, E(3)^2, E(3)^2 ]
)
```

`TensorProductMon(` *list* `)`

`TensorProductMon` returns a mon representing the tensor product of the mons in *list*.

```
gap> m1 := Mon( (1,2), [1, E(3), E(3)^2] );
Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
)
gap> m2 := Mon( (1,2), 3);
Mon( (1,2), 3 )
gap> TensorProductMon( [m1, m2] );
Mon(
  (1,5)(2,4)(3,6)(7,8),
  [ 1, 1, 1, E(3), E(3), E(3), E(3)^2, E(3)^2, E(3)^2 ]
)
```

74.21 CharPolyCyclesMon

`CharPolyCyclesMon(` *m* `)`

`CharPolyCyclesMon` returns the sorted list of the characteristic polynomials of the cycles of the mon m. All polynomials are written in a common polynomial ring. Applying `Product` to the result yields the characteristic polynomial of m.

```
gap> CharPolyCyclesMon( Mon( (1,2), 3 ) );
[ X(Rationals) - 1, X(Rationals)^2 - 1 ]
gap> CharPolyCyclesMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
[ X(CF(3)) + (-E(3)^2), X(CF(3))^2 + (-E(3)) ]
```

74.22 AMats

The class **AMat** (**A**bstract **Mat**rices) is created to represent and calculate efficiently with structured matrices like e.g. $2 \cdot (A \oplus B)^C \otimes D \cdot E^2$, where A, B, C, D, E are matrices of compatible size/characteristic and $\oplus, \otimes$ denote the direct sum and tensor product (Kronecker product) resp. of matrices. The elements of the class **AMat** are called "amats" and implement a recursive datastructure to form expressions like the one above. Basic constructors for amats allow to create permutation matrices (see `AMatPerm`, 74.23), monomial matrices (see `AMatMon`, 74.24) and general matrices (see `AMatMat`, 74.25) in an efficient way (e.g. a permutation matrix is defined by a permutation, the degree and the characteristic). Higher constructors allow to construct direct sums (see `DirectSumAMat`, 74.40), tensor products (see `TensorProductAMat`, 74.41) etc. from given amats. Note that while building up a highly structured amat from other amats no computation is done beside checks for compatibility. To obtain the matrix represented by an amat the appropiate function has to be applied (e.g. `MatAMat`, 74.50).

Some remarks on the design of **AMat**: The class **AMat** is what is called a term algebra for expressions representing highly structured matrices over certain base fields. Amats are not necessarily square but can also be rectangular. Hence, if an amat must be invertible (e.g. when it shall conjugate another amat) this has to be proven by computation. To avoid many of these calculations the result (the inverse) is stored in the object and many functions accept a "hint". E.g. by supplying the hint "invertible" in the example above the explicit check for invertibility is suppressed. Using and passing correct hints is essential for efficient computation. A common problem in the design of non-trivial term algebras is the simplification strategy: Aggressive or conservative simplification? Our approach here is extremely conservative. This means even trivial subexpressions like $1 * A$ are not automatically simplified. This allows the user to write functions that return their result always in a fixed structure, e.g. the result is always a conjugated direct sum of tensor products even though the conjugation might be trivial. Finally, note that amats and normal matrices (i.e. lists of lists of field elements) do not mix – you have to convert explicitly with `AMatMat`, `MatAMat` etc. This greatly simplifies the amat module.

We define an amat recursively in Backus-Naur-Form as the disjoint union of the following cases.

amat ::=			
;	atomic cases		
	perm	;	"perm" (invertible)
|	*mon*	;	"mon" (invertible)
|	*mat*	;	"mat"
;	composed cases		
|	*scalar* · *amat*	;	"scalarMultiple"
|	*amat* · ... · *amat*	;	"product"
|	*amat* ˆ *int*	;	"power"
|	*amat* ˆ *amat*	;	"conjugate"
|	*amat* ⊕ ... ⊕ *amat*	;	"directSum"
|	*amat* ⊗ ... ⊗ *amat*	;	"tensorProduct"
|	GaloisConjugate(*amat*, *aut*)	;	"galoisConjugate".

An amat A is a record with at least the following fields:

`isAMat`	:=	`true`
`operations`	:=	`AMatOps`
`type`	:	a string identifying the type of A
`dimensions`	:	size of the matrix represented (= [rows, columns])
`char`	:	characteristic of the base field

The cases as stated above are distinguished by the field `.type` of an amat. Depending on the type additional fields are mandatory as follows:

`type = "perm"`:
`element` defining permutation

`type = "mon"`:
`element` defining mon-object (see 74.2)

`type = "mat"`:
`element` defining matrix (list of lists of field elements)

`type = "scalarMultiple"`:
`element` the AMat multiplied
`scalar` the scalar

type = "product":

`factors`	list of AMats of compatible dimensions and the same characteristic

type = "power":

`element`	the square AMat to be raised to exponent
`exponent`	the exponent (an integer)

type = "conjugate":

`element`	the square AMat to be conjugated
`conjugation`	the conjugating invertible AMat

type = "directSum":

`summands`	List of AMats of the same characteristic

type = "tensorProduct":

`factors`	List of AMats of the same characteristic

type = "galoisConjugate":

`element`	the AMat to be Galois conjugated
`galoisAut`	the Galois automorphism

Note that there is an important difference between the *type of an amat* and the *type of the matrix being represented by the amat*: An amat can be of type "mat" but the matrix is in fact a permutation matrix. This distinction is refelcted in the naming of the functions: "XAMat" refers to the type of the amat, "XMat" to the type of the matrix being represented,

Here a short overview of the functions concerning amats. sections 74.23 – 74.43 are concerned with the construction of amats, sections 74.44 – 74.53 with the convertability and conversion of amats to permutations, mons and matrices, sections 74.54 – 74.65 contain functions for amats, e.g. computation of the determinant or simplification of amats.

The functions concerning amats are implemented in the file `"arep/lib/amat.g"`.

74.23 AMatPerm

`AMatPerm(` p`,` d `)`
`AMatPerm(` p`,` d`,` *char* `)`
`AMatPerm(` p`,` d`,` *field* `)`

`AMatPerm` returns an amat of type `"perm"` representing the $(d \times d)$ permutation matrix $[\delta_{i^p j} \mid i,j \in \{1,\ldots,d\}]$ corresponding to the permutation p. As optional parameter a characteristic *char* or a *field* can be supplied. The default characteristic is zero. The function will signal an error if the degree d is less than the largest moved point of p.

```
gap> AMatPerm( (1,2), 5 );
AMatPerm((1,2), 5)
gap> AMatPerm( (1,2,3), 5 , 3);
AMatPerm((1,2,3), 5, GF(3))
gap> A := AMatPerm( (1,2,3), 5 , Rationals);
AMatPerm((1,2,3), 5)
gap> A.type;
"perm"
```

74.24 AMatMon

`AMatMon(` *m* `)`

`AMatMon` returns an amat of type `"mon"` representing the monomial matrix given by the mon *m*. For the explanation of mons please refer to 74.2.

```
gap> AMatMon( Mon( (1,2), [1, E(3), E(3)^2] ) );
AMatMon( Mon(
  (1,2),
  [ 1, E(3), E(3)^2 ]
) )
gap> A := AMatMon( Mon( (1,2), 3) );
AMatMon( Mon( (1,2), 3 ) )
gap> A.type;
"mon"
```

74.25 AMatMat

`AMatMat(` *M* `)`
`AMatMat(` *M*`,` *hint* `)`

`AMatMat` returns an amat of type `"mat"` representing the matrix *M*. If the optional *hint* `"invertible"` is supplied then the field `.isInvertible` of the amat is set to `true` (without checking) indicating that the matrix represented is invertible.

```
gap> AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> A := AMatMat( [ [1,2], [3,4] ] , "invertible");
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ],
  "invertible"
)
gap> A.isInvertible;
true
```

74.26 IsAMat

`IsAMat(` *obj* `)`

`IsAMat` returns `true` if *obj*, which may be an object of arbitrary type, is an amat, and `false` otherwise.

```
gap> IsAMat( AMatPerm( (1,2,3), 3 ) );
true
gap> IsAMat( 1/2 );
false
```

74.27 IdentityPermAMat

`IdentityPermAMat(` *n* `)`
`IdentityPermAMat(` *n*`,` *char* `)`
`IdentityPermAMat(` *n*`,` *field* `)`

`IdentityPermAMat` returns an amat of type `"perm"` representing the $(n\times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatPerm`.

```
gap> IdentityPermAMat( 3 );
IdentityPermAMat(3)
gap> AMatPerm( ( ), 3);
IdentityPermAMat(3)
gap> IdentityPermAMat( 3 , GF(3) );
IdentityPermAMat(3, GF(3))
```

74.28 IdentityMonAMat

`IdentityMonAMat(` *n* `)`
`IdentityMonAMat(` *n*`,` *char* `)`
`IdentityMonAMat(` *n*`,` *field* `)`

`IdentityMonAMat` returns an amat of type `"mon"` representing the $(n \times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatMon`.

```
gap> IdentityMonAMat( 3 );
IdentityMonAMat(3)
gap> AMatMon( Mon( ( ), 3 ) );
IdentityMonAMat(3)
gap> IdentityMonAMat( 3, 3 );
IdentityMonAMat(3, GF(3))
```

74.29 IdentityMatAMat

`IdentityMatAMat(` *n* `)`
`IdentityMatAMat(` *n*`,` *char* `)`
`IdentityMatAMat(` *n*`,` *field* `)`

`IdentityMatAMat` returns an amat of type `"mat"` representing the $(n \times n)$ identity matrix. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero. Note that the same result can be obtained by using `AMatMat`.

```
gap> IdentityMatAMat( 3 );
IdentityMatAMat(3)
gap> AMatMat( [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ]);
IdentityMatAMat(3)
gap> IdentityMatAMat( 3, GF(3) );
```

```
IdentityMatAMat(3, GF(3))
```

`IdentityMatAMat(` *dim* `)`
`IdentityMatAMat(` *dim*`,` *char* `)`
`IdentityMatAMat(` *dim*`,` *field* `)`

Let *dim* be a pair of positive integers. `IdentityMatAMat` returns an amat of type `"mat"` representing the rectangular identity matrix with *dim*`[1]` rows and *dim*`[2]` columns. A rectangular identity matrix has the entry 1 at the position (i, j) if $i = j$ and 0 else. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> IdentityMatAMat( [2, 3] );
IdentityMatAMat([ 2, 3 ])
gap> IdentityMatAMat( [2, 3], 3 );
IdentityMatAMat([ 2, 3 ], GF(3))
```

74.30 IdentityAMat

`IdentityAMat(` *dim* `)`
`IdentityAMat(` *dim*`,` *char* `)`
`IdentityAMat(` *dim*`,` *field* `)`

Let *dim* be a pair of positive integers. `IdentityAMat` returns an amat of type `"perm"` if *dim*`[1] =` *dim*`[2]` and an amat of type `"mat"` else, representing the identity matrix with *dim*`[1]` rows and *dim*`[2]` columns. A rectangular identity matrix has the entry 1 at the position (i, j) if $i = j$ and 0 else. Use this function if you do not know whether the matrix is square and you do not care about the type. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the identity matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> IdentityAMat( [2, 2] );
IdentityPermAMat(2)
gap> IdentityAMat( [2, 3] );
IdentityMatAMat([ 2, 3 ])
```

74.31 AllOneAMat

`AllOneAMat(` *n* `)`
`AllOneAMat(` *n*`,` *char* `)`
`AllOneAMat(` *n*`,` *field* `)`

`AllOneAMat` returns an amat of type `"mat"` representing the $(n \times n)$ all-one matrix. An all-one matrix has the entry 1 at each position. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-one matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> AllOneAMat( 3 );
AllOneAMat(3)
gap> AllOneAMat( 3, 3);
AllOneAMat(3, GF(3))
```

`AllOneAMat(` *dim* `)`
`AllOneAMat(` *dim*`,` *char* `)`
`AllOneAMat(` *dim*`,` *field* `)`

Let *dim* a pair of positive integers. `AllOneAMat` returns an amat of type `"mat"` representing the rectangular all-one matrix with *dim*`[1]` rows and *dim*`[2]` columns. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-one matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> AllOneAMat( [3, 2] );
AllOneAMat([ 3, 2 ])
gap> AllOneAMat( [3, 2], GF(5) );
AllOneAMat([ 3, 2 ], GF(5))
```

74.32 NullAMat

`NullAMat(` *n* `)`
`NullAMat(` *n*`,` *char* `)`
`NullAMat(` *n*`,` *field* `)`

`NullAMat` returns an amat of type `"mat"` representing the $(n \times n)$ all-zero matrix. An all-zero matrix has the entry 0 at each position. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-zero matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> NullAMat( 3 );
NullAMat(3)
gap> NullAMat( 3, 3);
NullAMat(3, GF(3))
```

`NullAMat(` *dim* `)`
`NullAMat(` *dim*`,` *char* `)`
`NullAMat(` *dim*`,` *field* `)`

Let *dim* a pair of positive integers. `NullAMat` returns an amat of type `"mat"` representing the rectangular all-zero matrix with *dim*`[1]` rows and *dim*`[2]` columns. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the all-zero matrix of arbitrary characteristic. The default characteristic is zero.

```
gap> NullAMat( [3, 2] );
NullAMat([ 3, 2 ])
gap> NullAMat( [3, 2], GF(5) );
NullAMat([ 3, 2 ], GF(5))
```

74.33 DiagonalAMat

`DiagonalAMat(` *list* `)`

Let *list* contain field elements of the same characteristic. `DiagonalAMat` returns an amat representing the diagonal matrix with diagonal entries in *list*. If all elements in *list* are $\neq 0$ the returned amat is of type `"mon"`, else of type `"directSum"` (see 74.22).

```
gap> DiagonalAMat( [2, 3] );
DiagonalAMat([ 2, 3 ])
```

```
gap> DiagonalAMat( [0, 2, 3] );
DirectSumAMat(
  NullAMat(1),
  AMatMat(
    [ [ 2 ] ]
  ),
  AMatMat(
    [ [ 3 ] ]
  )
)
```

74.34 DFTAMat

`DFTAMat(` *n* `)`
`DFTAMat(` *n*, *char* `)`
`DFTAMat(` *n*, *field* `)`

`DFTAMat` returns a special amat of type `"mat"` representing the matrix

$$\mathrm{DFT}_n = [\omega_n^{i \cdot j} \mid i, j \in \{0, \ldots, n-1\}],$$

with ω_n being a certain primitive n-th root of unity. DFT_n represents the Discrete Fourier Transform on n points (see 74.129). As optional parameter a characteristic *char* or a *field* can be supplied to obtain the DFT of arbitrary characteristic. The default characteristic is zero. Note that for characteristic p prime the DFT_n exists iff $\gcd(p, n) = 1$. For a given finite *field* the DFT_n exists iff $n \mid$ `Size`(F). If these conditions are violated an error is signaled. The choice of ω_n is `E(`*n*`)` if *char* $= 0$ and `Z(`*q*`)^((`*q-1*`)/`*n*`)` for *char* $= p$, q an appropiate p-power.

```
gap> DFTAMat(3);
DFTAMat(3)
gap> DFTAMat(3, 7);
DFTAMat(3, 7)
```

74.35 SORAMat

`SORAMat(` *n* `)`
`SORAMat(` *n*, *char* `)`
`SORAMat(` *n*, *field* `)`

`SORAMat` returns a special amat of type `"mat"` representing the matrix

$$\mathrm{SOR}_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & -1 & 0 & \cdots & 0 \\ 1 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 1 & 0 & 0 & & -1 \end{bmatrix}.$$

The SOR_n is the sparsest matrix that **s**plits off the **o**ne-**r**epresentation in a permutation representation. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the SOR of arbitrary characteristic. The default characteristic is zero.

```
gap> SORAMat( 4 );
SORAMat(4)
gap> SORAMat( 4, 7);
SORAMat(4, 7)
```

74.36 ScalarMultipleAMat

`ScalarMultipleAMat(` s`,` A `)` or s `*` A

Let s be a field element and A an amat. `ScalarMultipleAMat` returns an amat of type `"scalarMultiple"` representing the scalar multiple of s with A, which must have common characteristic otherwise an error is signaled. Note that s and A can be accessed in the fields `.scalar` resp. `.element` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> ScalarMultipleAMat( E(3), A );
E(3) * AMatPerm((1,2,3), 4)
gap> 2 * A;
2 * AMatPerm((1,2,3), 4)
```

74.37 Product and Quotient of AMats

A `*` B

Let A and B be amats. A `*` B returns an amat of type `"product"` representing the product of A and B, which must have compatible sizes and common characteristic otherwise an error is signaled. Note that the factors can be accessed in the field `.factors` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> B := AMatMat( [ [1, 2], [3, 4], [5, 6], [7, 8] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ]
)
gap> A * A;
AMatPerm((1,2,3), 4) *
AMatPerm((1,2,3), 4)
gap> C := A * B;
AMatPerm((1,2,3), 4) *
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ]
)
gap> C.type;
"product"
```

A `/` B

Let A and B be amats. A `/` B returns an amat of type `"product"` representing the quotient of A and B. The sizes and characteristics of A and B must be compatible, B must be square and invertible otherwise an error is signaled.

```
gap> A := AMatPerm( (1,2,3), 4);
```

```
AMatPerm((1,2,3), 4)
gap> B := DiagonalAMat( [1, E(3), 1, 3] );
DiagonalAMat([ 1, E(3), 1, 3 ])
gap> A / B;
AMatPerm((1,2,3), 4) *
DiagonalAMat([ 1, E(3), 1, 3 ]) ^ -1
```

74.38 PowerAMat

`PowerAMat(` A, n `)` or A `^` n
`PowerAMat(` A, n, *hint* `)`

Let A be an amat and n an integer. `PowerAMat` returns an amat of type `"power"` representing the power of A with n. A must be square otherwise an error is signaled. If n is negative then A is checked for invertibility if the hint `"invertible"` is not supplied. Note that A and n can be accessed in the fields `.element` resp. `.exponent` of the result.

```
gap> A := AMatPerm( (1,2,3), 4);
AMatPerm((1,2,3), 4)
gap> B := PowerAMat(A, 3);
AMatPerm((1,2,3), 4) ^ 3
gap> B ^ -2;
( AMatPerm((1,2,3), 4) ^ 3
) ^ -2
```

74.39 ConjugateAMat

`ConjugateAMat(` A, B `)` or A `^` B
`ConjugateAMat(` A, B, *hint* `)`

Let A and B be amats. `ConjugateAMat` returns an amat of type `"conjugate"` representing the conjugate of A with B (i.e. the matrix defined by $B^{-1} \cdot A \cdot B$). A and B must be square otherwise an error is signaled. B is checked for invertibility if the hint `"invertible"` is not supplied. Note that A and B can be accessed in the fields `.element` resp. `conjugation` of the result.

```
gap> A := AMatMon( Mon( (1,2), [1, E(4), -1] ) );
AMatMon( Mon(
  (1,2),
  [ 1, E(4), -1 ]
) )
gap> B := DFTAMat( 3 );
DFTAMat(3)
gap> ConjugateAMat( A, B, "invertible" );
ConjugateAMat(
  AMatMon( Mon(
    (1,2),
    [ 1, E(4), -1 ]
  ) ),
  DFTAMat(3)
```

```
)
gap> B ^ SORAMat( 3 );
ConjugateAMat(
  DFTAMat(3),
  SORAMat(3)
)
```

74.40 DirectSumAMat

`DirectSumAMat(` A_1`, ...,` A_k `)`

`DirectSumAMat` returns an amat of type `"directSum"` representing the direct sum of the amats A_1, ..., A_k, which must have common characteristic otherwise an error is signaled. Note that the direct summands can be accessed in the field `.summands` of the result.

```
gap> A1 := AMatMat( [ [1, 2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> A2 := DFTAMat( 2 );
DFTAMat(2)
gap> A3 := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> DirectSumAMat( E(3) * A1, A2 ^ 2, A3 );
DirectSumAMat(
  E(3) * AMatMat( [ [ 1, 2 ] ] ),
  DFTAMat(2) ^ 2,
  AMatPerm((1,2), 2)
)
```

`DirectSumAMat(` *list* `)`

`DirectSumAMat` returns an amat of type `"directSum"` representing the direct sum of the amats in *list*. The amats must have common characteristic otherwise an error is signaled. The direct summands can be accessed in the field `.summands` of the result.

```
gap> A := DiagonalAMat( [ Z(3), Z(3)^2 ]);
DiagonalAMat([ Z(3), Z(3)^0 ])
gap> B := AMatPerm( (1,2), 3, 3);
AMatPerm((1,2), 3, GF(3))
gap> DirectSumAMat( [A, B] );
DirectSumAMat(
  DiagonalAMat([ Z(3), Z(3)^0 ]),
  AMatPerm((1,2), 3, GF(3))
)
```

74.41 TensorProductAMat

`TensorProductAMat(` A_1`, ...,` A_k `)`

`TensorProductAMat` returns an amat of type `"tensorProduct"` representing the tensor product (or Kronecker product) of the amats A_1, ..., A_k, which must have common characteristic otherwise an error is signaled. Note that the tensor factors can be accessed in the field `.factors` of the result.

```
gap> A := IdentityPermAMat( 2 );
IdentityPermAMat(2)
gap> B := AMatMat( [ [1, 2, 3], [4, 5, 6] ] );
AMatMat(
  [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
)
gap> TensorProductAMat( A, B );
TensorProductAMat(
  IdentityPermAMat(2),
  AMatMat(
    [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
  )
)
```

`TensorProductAMat(` *list* `)`

`TensorPoductAMat` returns an amat of type `"tensorProduct"` representing the tensor product of the amats in *list*. The amats must have common characteristic otherwise an error is signaled. The tensor factors can be accessed in the field `.factors` of the result.

```
gap> A := AMatPerm( (1,2), 3 );
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1], [2] ]);
AMatMat(
  [ [ 1 ], [ 2 ] ]
)
gap> TensorProductAMat( [A ^ 2, 2 * B] );
TensorProductAMat(
  AMatPerm((1,2), 3) ^ 2,
  2 * AMatMat(
    [ [ 1 ], [ 2 ] ]
  )
)
```

74.42 GaloisConjugateAMat

`GaloisConjugateAMat(` A, k `)`
`GaloisConjugateAMat(` A, *aut* `)`

`GaloisConjugateAMat` returns an amat which represents a Galois conjugate of the amat A. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime. Note that A and k/aut can be accessed in the fields `.element` resp. `.galoisAut` of the result.

```
gap> A := DiagonalAMat( [1, E(3)] );
DiagonalAMat([ 1, E(3) ])
gap> GaloisConjugateAMat( A, -1 );
GaloisConjugateAMat(
```

```
  DiagonalAMat([ 1, E(3) ]),
  -1
)
gap> aut := FrobeniusAutomorphism( GF(4) );
FrobeniusAutomorphism( GF(2^2) )
gap> B := AMatMon( Mon( (1,2), [ Z(2)^0, Z(2^2) ] ) );
AMatMon( Mon(
  (1,2),
  [ Z(2)^0, Z(2^2) ]
) )
gap> GaloisConjugateAMat( B, aut );
GaloisConjugateAMat(
  AMatMon( Mon(
    (1,2),
    [ Z(2)^0, Z(2^2) ]
  ) ),
  FrobeniusAutomorphism( GF(2^2) )
)
```

74.43 Comparison of AMats

A = *B*
A <> *B*

The equality operator `=` evaluates to `true` if the amats A and B are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the amats A and B are not equal and to `false` otherwise.

Two amats are equal iff they define the same matrix.

```
gap> A := DiagonalAMat( [E(3), 1] );
DiagonalAMat([ E(3), 1 ])
gap> B := A ^ 3;
DiagonalAMat([ E(3), 1 ]) ^ 3
gap> B = IdentityPermAMat( 2 );
true
```

A < *B*
A <= *B*
A >= *B*
A > *B*

The operators `<`, `<=`, `>=`, and `>` evaluate to `true` if the amat A is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the amat B.

The ordering of amats is defined via the ordering of records.

74.44 Converting AMats

The following sections describe the functions for the convertability and conversion of amats to permutations, mons (see 74.2) and matrices.

The names of the conversion functions are chosen according to the usual GAP3-convention: `ChalkCheese` makes chalk from cheese. The parts in the name (chalk, cheese) are

Perm	– a GAP3-permutation, e.g. (1,2)
Mon	– a mon object, e.g. Mon((1,2), 2) (see 74.2)
Mat	– a GAP3-matrix, e.g. [[1,2],[3,4]]
AMat	– an amat of any type
PermAMat	– an amat of type "perm"
MonAMat	– an amat of type "mon"
MatAMat	– an amat of type "mat"

74.45 IsIdentityMat

`IsIdentityMat(` *A* `)`

`IsIdentityMat` returns `true` if the matrix represented by the amat A is the identity matrix and `false` otherwise. Note that the name of the function is not `IsIdentityAMat` since A can be of any type but represents an identity matrix in the mathematical sense.

```
gap> IsIdentityMat(AMatPerm( (1,2), 3 ));
false
gap> A := DiagonalAMat( [Z(3), Z(3)] ) ^ 2;
DiagonalAMat([ Z(3), Z(3) ]) ^ 2
gap> IsIdentityMat(A);
true
```

74.46 IsPermMat

`IsPermMat(` *A* `)`

`IsPermMat` returns `true` if the matrix represented by the amat A is a permutation matrix and `false` otherwise. The name of the function is not `IsPermAMat` since A can be of any type but represents a permutation matrix in the mathematical sense. Note that `IsPermMat` sets and tests A`.isPermMat`.

```
gap> IsPermMat( AMatMon( Mon( (1,2), [1, -1] )));
false
gap> IsPermMat( DiagonalAMat( [Z(3), Z(9)] ) ^ 8);
true
```

74.47 IsMonMat

`IsMonMat(` *A* `)`

`IsMonMat` returns `true` if the matrix represented by the amat A is a monomial matrix (a matrix containing exactly one entry $\neq 0$ in every row and column) and `false` otherwise. The name of the function is not `IsMonAMat` since A can be of any type but represents a monomial matrix in the mathematical sense. Note that `IsMonMat` sets and tests A`.isMonMat`.

```
gap> IsMonMat( AMatPerm( (1,2), 3 ));
```

```
true
gap> IsMonMat( AMatPerm( (1,2,3), 3 ) ^ DFTAMat(3) );
true
```

74.48 PermAMat

`PermAMat(` *A* `)`

Let *A* be an amat. `PermAMat` returns the permutation represented by *A* if *A* is a permutation matrix (i.e. `IsPermMat(` *A* `) = true`) and `false` otherwise. Note that `PermAMat` sets and tests *A*`.perm`.

```
gap> PermAMat(AMatPerm( (1,2), 5 ));
(1,2)
gap> A := AMatMat( [ [Z(3)^0, Z(3)], [0*Z(3), Z(3)^0] ] );
AMatMat(
  [ [ Z(3)^0, Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
)
gap> PermAMat(A);
false
gap> PermAMat(A ^ 3);
()
```

74.49 MonAMat

`MonAMat(` *A* `)`

Let *A* be an amat. `MonAMat` returns the mon (see 74.2) represented by *A* if *A* is a monomial matrix (i.e. `IsMonMat(` *A* `) = true`) and `false` otherwise. Note that `MonAMat` sets and tests *A*`.mon`.

```
gap> MonAMat(AMatPerm( (1,2,3), 5 ));
Mon( (1,2,3), 5 )
gap> MonAMat(AMatPerm( (1,2,3), 3 ) ^ DFTAMat(3) );
Mon( [ 1, E(3), E(3)^2 ] )
gap> MonAMat( AMatMat( [ [1, 2] ] ));
false
```

74.50 MatAMat

`MatAMat(` *A* `)`

`MatAMat` returns the matrix represented by the amat *A*. Note that `MatAMat` sets and tests *A*`.mat`.

```
gap> MatAMat( AMatPerm( (1,2), 3, 2 ));
[ [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> MatAMat(DFTAMat(3));
[ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ]
gap> A := IdentityPermAMat(2);
IdentityPermAMat(2)
```

```
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> MatAMat(TensorProductAMat(A, B));
[ [ 1, 2, 0, 0 ], [ 3, 4, 0, 0 ], [ 0, 0, 1, 2 ], [ 0, 0, 3, 4 ] ]
```

74.51 PermAMatAMat

`PermAMatAMat(` *A* `)`

Let *A* be an amat. `PermAMatAMat` returns an amat of type `"perm"` equal to *A* if *A* is a permutation matrix (i.e. `IsPermMat(` *A* `) = true`) and `false` otherwise.

```
gap> PermAMatAMat(AMatMon(Mon( (1,2), 3 )));
AMatPerm((1,2), 3)
gap> PermAMatAMat(DiagonalAMat( [E(3), 1] ) ^ 3);
IdentityPermAMat(2)
gap> PermAMatAMat(AMatMat( [ [1,2] ] ));
false
```

74.52 MonAMatAMat

`MonAMatAMat(` *A* `)`

Let *A* be an amat. `MonAMatAMat` returns an amat of type `"mon"` equal to *A* if *A* is a monomial matrix (i.e. `IsMonMat(` *A* `) = true`) and `false` otherwise.

```
gap> MonAMat(AMatPerm( (1,2), 3 ));
Mon( (1,2), 3 )
gap> MonAMat(DFTAMat(3)^2);
Mon(
  (2,3),
  [ 3, 3, 3 ]
)
gap> MonAMat(AMatMat( [ [1, 2] ] ));
false
```

74.53 MatAMatAMat

`MatAMatAMat(` *A* `)`

`MatAMatAMat` returns an amat of type `"mat"` equal to *A*.

```
gap> A := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> B := AMatMat( [ [1,2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> MatAMatAMat(DirectSumAMat(A, B));
AMatMat(
```

```
  [ [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 1, 2 ] ]
)
```

74.54 Functions for AMats

The following sections describe useful functions for the calculation with amats (e.g. calculation of the inverse, determinant of an amat as well as simplifying amats). Most of these functions can take great advantage of the highly structured form of the amats.

74.55 InverseAMat

`InverseAMat(` *A* `)`

`InverseAMat` returns an amat representing the inverse of the amat *A*. If *A* is not invertible, an error is signaled. The function uses mathematical rules to invert the direct sum, tensor product etc. of matrices. Note that `InverseAMat` sets and tests *A*`.inverse`.

```
gap> A := AMatPerm( (1,2), 3);
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1,2], [3,4] ]);
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> C := DiagonalAMat( [ E(3), 1] );
DiagonalAMat([ E(3), 1 ])
gap> D := DirectSumAMat(A, TensorProductAMat(B, C));
DirectSumAMat(
  AMatPerm((1,2), 3),
  TensorProductAMat(
    AMatMat( [ [ 1, 2 ], [ 3, 4 ] ] ),
    DiagonalAMat([ E(3), 1 ])
  )
)
gap> InverseAMat(D);
DirectSumAMat(
  AMatPerm((1,2), 3),
  TensorProductAMat(
    AMatMat(
      [ [ -2, 1 ], [ 3/2, -1/2 ] ],
      "invertible"
    ),
    DiagonalAMat([ E(3)^2, 1 ])
  )
)
```

74.56 TransposedAMat

`TransposedAMat(` *A* `)`

`TransposedAMat` returns an amat representing the transpose of the amat A. The function uses mathematical rules to transpose the direct sum, tensor product etc. of matrices.

```
gap> A := AMatPerm( (1,2,3), 3);
AMatPerm((1,2,3), 3)
gap> B := AMatMat( [ [1, 2] ] );
AMatMat(
  [ [ 1, 2 ] ]
)
gap> TransposedAMat(TensorProductAMat(A, B));
TensorProductAMat(
  AMatPerm((1,3,2), 3),
  AMatMat(
    [ [ 1 ], [ 2 ] ]
  )
)
```

74.57 DeterminantAMat

`DeterminantAMat(` A `)`

`DeterminantAMat` returns the determinant of the amat A. If A is not square an error is signaled. The function uses mathematical rules to calculate the determinant of the direct sum, tensor product etc. of matrices. Note that `DeterminantAMat` sets and tests A`.determinant`.

```
gap> A := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> B := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> DeterminantAMat(TensorProductAMat(A, B));
4
```

74.58 TraceAMat

`TraceAMat(` A `)`

`TraceAMat` returns the trace of the amat A. If A is not square an error is signaled. The function uses mathematical rules to calculate the trace of direct sums, tensor product etc. of matrices. Note that `TraceAMat` sets and tests A`.trace`.

```
gap> A := DFTAMat(2);
DFTAMat(2)
gap> B := DiagonalAMat( [1, 2, 3] );
DiagonalAMat([ 1, 2, 3 ])
gap> TraceAMat(DirectSumAMat( A^2, B ));
10
```

74.59 RankAMat

`RankAMat(` A `)`

RankAMat returns the rank of the amat A. Note that RankAMat sets and tests A.rank.

```
gap> RankAMat(AllOneAMat(100));
1
gap> RankAMat(AMatPerm( (1,2), 10 ));
10
```

74.60 SimplifyAMat

SimplifyAMat(A)

SimplifyAMat returns a simplified amat representing the same matrix as the amat A. The simplification is performed recursively according to certain rules. E.g. the following simplifications are performed:

- If A represents a permutation matrix, monomial matrix then an amat of type "perm", "mon" resp. is returned.
- In a product resp. tensor product, trivial factors are omitted.
- Trivial conjugation is omitted.
- In a direct sum adjacent permutation/monomial matrices are put together.
- In a product adjacent permutation/monomial matrices are multiplied together.
- Successive scalars are multiplied together.
- Successive exponents are multiplied together, negative exponents are evaluated using InverseAMat.

Note that important information about the matrix is shifted to the simplification.

```
gap> A := IdentityPermAMat( 3 );
IdentityPermAMat(3)
gap> B := DiagonalAMat( [E(3), 1, 1] );
DiagonalAMat([ E(3), 1, 1 ])
gap> C := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> D := DirectSumAMat(A ^ -1, 1 * B * A, C);
DirectSumAMat(
  IdentityPermAMat(3) ^ -1,
  ( 1 * DiagonalAMat([ E(3), 1, 1 ])
  ) *
  IdentityPermAMat(3),
  AMatMat(
    [ [ 1, 2 ], [ 3, 4 ] ]
  )
)
gap> SimplifyAMat(D);
```

```
DirectSumAMat(
  IdentityPermAMat(3),
  DiagonalAMat([ E(3), 1, 1 ]),
  AMatMat(
    [ [ 1, 2 ], [ 3, 4 ] ]
  )
)
```

74.61 kbsAMat

`kbsAMat(` A_1`, ...,` A_k `)`

`kbsAMat` returns the joined kbs (conjugated blockstructure) of the amats A_1, ..., A_k. The amats must be square and of common size and characteristic otherwise an error is signaled. The joined kbs of a list of $(n \times n)$-matrices is a partition of $\{1, \ldots, n\}$ representing their common blockstructure. For an exact definition see 74.167.

```
gap> A := IdentityPermAMat(2);
IdentityPermAMat(2)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> kbsAMat(TensorProductAMat(A, B));
[ [ 1, 2 ], [ 3, 4 ] ]
gap> kbsAMat(AMatPerm( (1,3)(2,4), 5 ));
[ [ 1, 3 ], [ 2, 4 ], [ 5 ] ]
```

`kbsAMat(` *list* `)`

`kbsAMat` returns the joined kbs of the amats in *list* (see above).

74.62 kbsDecompositionAMat

`kbsDecompositionAMat(` A `)`

`kbsDecompositionAMat` decomposes the amat A into a conjugated (by an amat of type `"perm"`) direct sum of amats of type `"mat"` as far as possible. If A is not square an error is signaled. The decomposition is performed according to the kbs (see 74.167) of A which is a partition of $\{1, \ldots, n\}$ (n = number of rows of A) describing the blockstructure of A.

```
gap> A := AMatMat( [[1,0,2,0], [0,1,0,2], [3,0,4,0], [0,3,0,4]] );
AMatMat(
  [ [ 1, 0, 2, 0 ], [ 0, 1, 0, 2 ], [ 3, 0, 4, 0 ], [ 0, 3, 0, 4 ] ]
)
gap> kbsDecompositionAMat(A);
ConjugateAMat(
  DirectSumAMat(
    AMatMat(
      [ [ 1, 2 ], [ 3, 4 ] ]
```

```
      ),
      AMatMat(
        [ [ 1, 2 ], [ 3, 4 ] ]
      )
    ),
    AMatPerm((2,3), 4)
  )
```

74.63 AMatSparseMat

`AMatSparseMat(` *M* `)` `AMatSparseMat(` *M*`,` *match-blocks* `)`

Let M be a sparse matrix (i.e. containing entries $\neq 0$). `AMatSparseMat` returns an amat of the form $P_1 \cdot E_1 \cdot D \cdot E_2 \cdot P_2$ where (for $i = 1, 2$) P_i are amats of type `"perm"`, E_i are identity-amats (might be rectangular) and D is an amat of type `"directSum"`. If *match-blocks* is `true` or not provided then, furthermore, the permutations p_1 and p_2 are chosen such that equivalent summands of D are equal and collected together by a tensor product. If *match-blocks* is `false` this is not done. The major part of the work is done by the function `DirectSummandsPermutedMat` (see 74.166). Use the function `SimplifyAMat` (see 74.60) for simplification of the result.

For an explanation of the algorithm see [Egn97a].

```
gap> M := [[0,0,0,0],[0,1,0,2],[0,0,3,0],[0,4,0,5]];;
gap> PrintArray(M);
[ [  0,  0,  0,  0 ],
  [  0,  1,  0,  2 ],
  [  0,  0,  3,  0 ],
  [  0,  4,  0,  5 ] ]
gap> AMatSparseMat(M);
AMatPerm((1,4,3), 4) *
IdentityMatAMat([ 4, 3 ]) *
DirectSumAMat(
  TensorProductAMat(
    IdentityPermAMat(1),
    AMatMat(
      [ [ 3 ] ]
    )
  ),
  TensorProductAMat(
    IdentityPermAMat(1),
    AMatMat(
      [ [ 1, 2 ], [ 4, 5 ] ]
    )
  )
) *
IdentityMatAMat([ 3, 4 ]) *
AMatPerm((1,3,4), 4)
```

74.64 SubmatrixAMat

`SubmatrixAMat(` A, *inds* `)`

Let A be an amat and *inds* a set of positive integers. `SubmatrixAMat` returns an amat of type `"mat"` representing the submatrix of A defined by extracting all entries with row and column index in *inds*.

```
gap> A := AMatPerm( (1,2), 2 );
AMatPerm((1,2), 2)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> SubmatrixAMat(TensorProductAMat(A, B), [2,3] );
AMatMat(
  [ [ 0, 3 ], [ 2, 0 ] ]
)
```

74.65 UpperBoundLinearComplexityAMat

`UpperBoundLinearComplexityAMat(` A `)`

`UpperBoundLinearComplexityAMat` returns an upper bound for the linear complexity of the amat A according to the complexity model L_∞ of Clausen/Baum, [CB93]. The linear complexity is a measure for the complexity of the matrix-vector multiplication of a given matrix with an arbitrary vector.

```
gap> UpperBoundLinearComplexityAMat(DFTAMat(2));
2
gap> UpperBoundLinearComplexityAMat(DiagonalAMat( [2, 3] ));
2
gap> A := AMatPerm( (1,2), 3);
AMatPerm((1,2), 3)
gap> B := AMatMat( [ [1,2], [3,4] ] );
AMatMat(
  [ [ 1, 2 ], [ 3, 4 ] ]
)
gap> UpperBoundLinearComplexityAMat(TensorProductAMat(A, B));
24
```

74.66 AReps

The class **ARep** (**A**bstract **Rep**resentations) is created to represent and calculate efficiently with structured matrix representations of finite groups up to equality, e.g. expressions like $(\phi \uparrow_T G)^M \otimes \psi$ where ϕ, ψ are representations and $\uparrow, \otimes$ denotes the induction resp. inner tensor product of representations. The implementation idea is the same as with the class **AMat** (see 74.22), i.e. a representation is a record containing the necessary information (e.g. degree, characteristic, list of images on the generators) to define a representation up to equality. The elements of **ARep** are called "areps" and are no group homomorphisms in the sense of GAP3 (which is the reason for the term "abstract" representation). Special care is taken of permutation and monomial representations, which can be represented very efficiently by storing a list of permutations or mons (see 74.2) instead of matrices as images on the generators.

Areps can represent representations of any finite group and any characteristic including modular (characteristic divides group size) representations, but most of the higher functions will only work in the non-modular case or even only in the case of characteristic zero. These restrictions are always indicated in the description of the respective function.

Basic constructors allow to create areps, e.g. by supplying the list of images on the generators (see `ARepByImages`, 74.73). Since GAP3 allows the manipulation of the generators given to construct a group, it is important for consistency to have a field with generators one can rely on. This is realized in the function `GroupWithGenerators`, 74.67.

Higher constructors allow to construct inductions (see `InductionARep`, 74.81), direct sums (see `DirectSumARep`, 74.77), inner tensor products (see `InnerTensorProductARep`, 74.78) etc. from given areps.

Some remarks on the design of **ARep**: The class **ARep** is a term algebra for matrix representations of finite groups (see also **AMat**, 74.22). The simplification strategy is extremely conservative, which means that even trivial expressions like `GaloisConjugate(`R`, `id`)` are only simplified upon explicit request. As in **AMat** we use the "hint"-concept extensively to suppress unnecessary expensive computations of little interest. The class **AMat** is used in **ARep** in three ways: 1. for images under areps, 2. for conjugating matrices (change of base of the underlying vector space) and 3. for elements of the intertwining space of two areps. Note that 3. requires non-invertible or even rectangular matrices to be represented. A special point that deserves mentioning is the way in which areps act as homomorphisms anf how they are defined. Areps are *no* GAP3-homomorphisms. We simply did not manage to implement **ARep** as a term algebra *and* as GAP3-homomorphisms in a relyable and efficient way which avoids maximal confusion. In addition, working with **ARep** usually involves many representations of the same group. This is supported in the most obvious way by fixing the list of generators used to create the group (see 74.67) and only varying the list of images. Although this strategy differs from the approach in GAP3 (which deliberately manipulates the generating list used to construct the group) it turned out to be very useful and efficient in the situation at hand.

We define an arep recursively in Backus-Naur-Form as the disjoint union of the following cases.

arep ::=			
; atomic cases			
	perm	;	"perm"
|	*mon*	;	"mon"
|	*mat*	;	"mat"
; composed cases			
|	*arep* ˆ *arep*	;	"conjugate"
|	*arep* $\oplus$ ... $\oplus$ *arep*	;	"directSum"
|	*arep* $\otimes$ ... $\otimes$ *arep*	;	"innerTensorProduct"
|	*arep* # ... # *arep*	;	"outerTensorProduct"
|	*arep* $\downarrow$ *subgrp*	;	"restriction"
|	*arep* $\uparrow$ *supgrp*, *transversal*	;	"induction"
|	Extension(*arep*, *ext-character*)	;	"extension"
|	GaloisConjugate(*arep*, *aut*)	;	"galoisConjugate"

An arep R is a record with the following fields mandatory to all types of areps.

`isARep`	:=	`true`
`operations`	:=	`AMatOps`
`char`	:	characteristic of the base field
`degree`	:	degree of the representation
`source`	:	the group being represented, which must contain the field `.theGenerators`, see 74.67
`type`	:	a string identifying the type of R

The cases as stated above are distinguished by the field `.type` of an arep R. Depending on the type additional fields are mandatory as follows.

`type = "perm":`
`theImages` list of permutations for the images of `source.theGenerators`

`type = "mon":`
`theImages` list of mons (see 74.2) for the images of `source.theGenerators`

`type = "mat":`
`theImages` list of matrices for the images of `source.theGenerators`

`type = "mat":`
`rep` an arep to be conjugated
`conjugation` an amat (see 74.22) conjugating `rep`

`type = "directSum":`
`summands` list of areps of the same source and characteristic

`type = "innerTensorProduct":`
`factors` list of areps of the same characteristic

`type = "outerTensorProduct":`
`factors` list of areps of the same characteristic

type = "restriction":

`rep`	an arep of a supergroup of `source`, the group `source` and `rep.source` have the same parent group

type = "induction":

`rep`	an arep of a subgroup of `source`, the group `source` and `rep.source` have the same parent group
`transversal`	a right transversal of `Cosets(source, rep.source)`

type = "galoisConjugate":

`rep`	an arep to be conjugated
`galoisAut`	the Galois automorphism

Note that most of the function concerning areps require calculation in the source group. Hence it is most useful to choose aggroups or permutation groups as sources if possible. Furthermore there is an important difference between the *type of an arep* and the *type of the representation being represented by the arep*: E.g. an arep can be of type "induction" but the representation is in fact a permutation representation. This distinction is reflected in the naming of the functions: "XARep" refers to the type of the arep, "XRep" to the type of the representation being represented,

Here a short overview of the function concerning areps. sections 74.67 – 74.83 are concerned with the construction of areps, sections 74.84 – 74.92 are concerned with the evaluation of an arep at a point, tests for equivalence and irreducibility, construction of an arep with given character etc., sections 74.94 – 74.99 deal with the conversion of areps to areps of type `"perm", "mon", "mat"`. Sections 74.100 – 74.123 provide function for the computation of the intertwining space of areps and a plenty of functions for monomial areps. The most important function here is `DecompositionMonRep` (see 74.123) performing the decomposition of a monomial arep including the computation of a highly structured decomposition matrix.

The basic functions concerning areps are implemented in the file `"arep/lib/arep.g"`, the higher functions in `"arep/lib/arepfcts.g"`.

For details on constructive representation theory and the theoretical background of the higher functions please refer to [Püs98].

74.67 GroupWithGenerators

`GroupWithGenerators(` G `)`

Let G be a group. `GroupWithGenerators` returns G with the field G`.theGenerators` being set to a fixed non-empty generating set of G. This function is created because GAP3 has the freedom to manipulate the generators given to construct a group. Based on the list G`.theGenerators` areps can be constructed, e.g. by the images on that list (`ARepByImages`, 74.73). If an arep for a group G is constructed with the field G`.theGenerators` unbound a warning is signaled and the field is set.

```
gap> G := Group( (1,2) );
Group( (1,2) )
gap> GroupWithGenerators(G);
Group( (1,2) )
```

```
gap> G.theGenerators;
[ (1,2) ]
gap> G := Group( () );
Group( () )
gap> GroupWithGenerators(G);
Group( () )
gap> G.theGenerators;
[ () ]
gap> G.generators;
[  ]
```

`GroupWithGenerators(` *list* `)`

`GroupWithGenerators` returns the group G generated by the elements in *list*. The field G`.theGenerators` is set to *list*. For the reason of this function see above.

```
gap> G := GroupWithGenerators( [ (), (1,2), (1,2,3) ] );
Group( (1,2), (1,2,3) )
gap> G.theGenerators;
[ (), (1,2), (1,2,3) ]
gap> G.generators;
[ (1,2), (1,2,3) ]
```

74.68 TrivialPermARep

`TrivialPermARep(` G `)`
`TrivialPermARep(` G, d `)`
`TrivialPermARep(` G, d, *char* `)`
`TrivialPermARep(` G, d, *field* `)`

`TrivialPermARep` returns an arep of type `"perm"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> TrivialPermARep(G, 2, 3);
TrivialPermARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2, GF(3) )
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialPermARep(G, 2, 3);
TrivialPermARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2, GF(3) )
gap> R.degree;
2
gap> R.char;
3
```

74.69 TrivialMonARep

`TrivialMonARep(` G `)`
`TrivialMonARep(` G, d `)`

`TrivialMonARep(` *G*, *d*, *char* `)`
`TrivialMonARep(` *G*, *d*, *field* `)`

`TrivialMonARep` returns an arep of type `"mon"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialMonARep(G, 2);
TrivialMonARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 2 )
gap> R.theImages;
[ Mon( (), 2 ), Mon( (), 2 ) ]
```

74.70 TrivialMatARep

`TrivialMatARep(` *G* `)`
`TrivialMatARep(` *G*, *d* `)`
`TrivialMatARep(` *G*, *d*, *char* `)`
`TrivialMatARep(` *G*, *d*, *field* `)`

`TrivialMatARep` returns an arep of type `"mat"` representing the one representation of the group G of degree d. The default degree is 1. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the one representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators( [(1,2), (3,4)] );
Group( (1,2), (3,4) )
gap> R := TrivialMatARep(G);
TrivialMatARep( GroupWithGenerators( [ (1,2), (3,4) ] ) )
gap> R.theImages;
[ [ [ 1 ] ], [ [ 1 ] ] ]
```

74.71 RegularARep

`RegularARep(` *G* `)`
`RegularARep(` *G*, *char* `)`
`RegularARep(` *G*, *field* `)`

`RegularARep` returns an arep of type `"induction"` representing the regular representation of G. The regular representation is defined (up to equality) by the induction $R = (1_{\mathsf{E}} \uparrow_T G)$ of the trivial representation (of degree one) of the trivial subgroup E of G with the transversal T being the ordered list of elements of G. As optional parameter a characteristic *char* or a *field* can be supplied to obtain the regular representation of arbitrary characteristic. The default characteristic is zero.

```
gap> G := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,3), (2,3) ] ) )
gap> RegularARep(G, GF(2));
RegularARep( GroupWithGenerators( [ (1,3), (2,3) ] ), GF(2) )
```

74.72 NaturalARep

`NaturalARep(` *G* `)`
`NaturalARep(` *G*`,` *d* `)`
`NaturalARep(` *G*`,` *d*`,` *char* `)`
`NaturalARep(` *G*`,` *d*`,` *field* `)`

Let G be a mongroup or a matrix group (for mons see 74.2). `NaturalARep` returns an arep of type `"mon"` or `"mat"` resp. representing the representation given by G, which means that G is taken as a representation of itself.

For a permutation group G the desired degree d of the representation has to be supplied. The returned arep is of type `"perm"`. If d is smaller than the largest moved point of G an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied (if G is a permutation group). Note that a mongroup or a matrix group as source of an arep slows down most of the calculations with it.

```
gap> G := GroupWithGenerators( [ (1,2), (1,2,3) ] );
Group( (1,2), (1,2,3) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,2), (1,2,3) ] ), 4 )
gap> R.theImages;
[ (1,2), (1,2,3) ]
gap> R.degree;
4
gap> G := GroupWithGenerators( [ Mon( (1,2), [E(4), 1] ) ] );
Group( Mon(
  (1,2),
  [ E(4), 1 ]
) )
gap> NaturalARep(G);
NaturalARep(
  GroupWithGenerators( [ Mon(
      (1,2),
      [ E(4), 1 ]
    ) ] ) )
```

74.73 ARepByImages

`ARepByImages(` *G*`,` *list* `)`
`ARepByImages(` *G*`,` *list*`,` *hint* `)`

`ARepByImages(` *G*`,` *list*`,` *d* `)`
`ARepByImages(` *G*`,` *list*`,` *d*`,` *hint* `)`
`ARepByImages(` *G*`,` *list*`,` *d*`,` *char* `)`
`ARepByImages(` *G*`,` *list*`,` *d*`,` *field* `)`
`ARepByImages(` *G*`,` *list*`,` *d*`,` *char*`,` *hint* `)`
`ARepByImages(` *G*`,` *list*`,` *d*`,` *field*`,` *hint* `)`

`ARepByImages` allows to construct an arep of the group G by supplying the *list* of images on the list G`.theGenerators`.

Let *list* contain mons (see 74.2). `ARepByImages` returns an arep of type `"mon"` defined by mapping G`.theGenerators` elementwise onto *list*.

Let *list* contain matrices. `ARepByImages` returns an arep of type `"mat"` defined by mapping G`.theGenerators` elementwise onto *list*.

Let *list* contain permutations. `ARepByImages` returns an arep of type `"perm"` and degree d defined by mapping G`.theGenerators` elementwise onto *list*. If d is smaller than the largest moved point of G an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied to obtain an arep of arbitrary characteristic.

In all cases the *hint* `"hom"` or `"faithful"` can be supplied to indicate that the list of images does define a homomorphism or even a faithful homomorphism respectively. If no hint is supplied it is checked whether the list of images defines a homomorphism.

```
gap> G := GroupWithGenerators( [(1,2), (1,2,3)] );
Group( (1,2), (1,2,3) )
gap> ARepByImages(G, [ Mon( [-1] ), Mon( [1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ Mon( [ -1 ] ), Mon( (), 1 ) ],
  "hom"
)
gap> L := [ [ [Z(2), Z(2)], [0*Z(2), Z(2)] ], IdentityMat(2, GF(2)) ];
[ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ]
gap> ARepByImages(G, L);
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ],
    [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
  ],
  GF(2),
  "hom"
)
gap> ARepByImages(G, [ (1,2), () ], 3);
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ (1,2), () ],
  3, # degree
  "hom"
)
gap> ARepByImages(G, [ (1,2), () ], 3, "hom");
ARepByImages(
  GroupWithGenerators( [ (1,2), (1,2,3) ] ),
  [ (1,2), () ],
  3, # degree
  "hom"
)
```

74.74 ARepByHom

`ARepByHom(` *hom* `)`

`ARepByHom(` *hom*`,` *d* `)`
`ARepByHom(` *hom*`,` *d*`,` *char* `)`
`ARepByHom(` *hom*`,` *d*`,` *char* `)`

Let *hom* be a homomorphism of a group into a mongroup. `ARepByHom` returns an arep of type `"mon"` corresponding to *hom*.

Let *hom* be a homomorphism of a group into a matrix group. `ARepByHom` returns an arep of type `"mat"` corresponding to *hom*.

Let *hom* be a homomorphism of a group into a permutation group and *d* a positive integer. `ARepByHom` returns an arep of type `"perm"` and degree *d* corresponding to *hom*. If *d* is smaller than the largest moved point of *hom*`.range` an error is signaled. As optional parameter a characteristic *char* or a *field* can be supplied to obtain an arep of arbitrary characteristic.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> phi := IdentityMapping(G);
IdentityMapping( Group( (1,4), (2,4), (3,4) ) )
gap> ARepByHom(phi, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> H := GroupWithGenerators( [ Mon( [-1] ) ] );
Group( Mon( [ -1 ] ) )
gap> psi :=
> GroupHomomorphismByImages(G, H, G.generators, [H.1, H.1, H.1]);
GroupHomomorphismByImages(
  Group( (1,4), (2,4), (3,4) ),
  Group( Mon( [ -1 ] ) ),
  [ (1,4), (2,4), (3,4) ],
  [ Mon( [ -1 ] ), Mon( [ -1 ] ), Mon( [ -1 ] ) ] )
gap> ARepByHom(psi);
ARepByImages(
  GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ),
  [ Mon( [ -1 ] ),
    Mon( [ -1 ] ),
    Mon( [ -1 ] )
  ],
  "hom"
)
```

74.75 ARepByCharacter

`ARepByCharacter(` *chi* `)`

Let *chi* be a onedimensional character of a group. `ARepByCharacter` returns a onedimensional arep of type `"mon"` given by *chi*.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
```

```
Group( (1,2) )
gap> L := Irr(G);
[ Character( Group( (1,2) ), [ 1, 1 ] ),
  Character( Group( (1,2) ), [ 1, -1 ] ) ]
gap> ARepByCharacter( L[2] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ -1 ] ) ],
  "hom"
)
```

74.76 ConjugateARep

`ConjugateARep(` R, A `)` or R `^` A
`ConjugateARep(` R, A, *hint* `)`

Let R be an arep and A an amat (see 74.22). `ConjugateARep` returns an arep of type `"conjugate"` representing the conjugated representation $R^A : \; x \mapsto A^{-1} \cdot R(x) \cdot A$. The amat is tested for invertibility if the optional *hint* `"invertible"` is not supplied. R and A must be compatible in size and characteristic otherwise an error is signaled. Note that R and A can be accessed in the fields `.rep` and `.conjugation` of the result.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> A := AMatPerm( (1,2,3,4), 4 );
AMatPerm((1,2,3,4), 4)
gap> R ^ A;
ConjugateARep(
  NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 ),
  AMatPerm((1,2,3,4), 4)
)
```

74.77 DirectSumARep

`DirectSumARep(` R_1, ..., R_k `)`

`DirectSumARep` returns an arep of type `"directSum"` representing the direct sum $R_1 \oplus \ldots \oplus R_k$ of the areps R_1, ..., R_k, which must have common source and characteristic otherwise an error is signaled.

The direct sum $R = R_1 \oplus \ldots \oplus R_k$ of representations is defined as $x \mapsto R_1(x) \oplus \ldots \oplus R_k(x)$.

Note that the summands R_1, ..., R_k can be accessed in the field `.summands` of the result.

```
gap> G := GroupWithGenerators( [(1,2,3,4), (1,3)] );
Group( (1,2,3,4), (1,3) )
gap> R1 := RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,2,3,4), (1,3) ] ) )
gap> R2 := ARepByImages(G, [ [[1]], [[-1]] ]);
ARepByImages(
```

```
  GroupWithGenerators( [ (1,2,3,4), (1,3) ] ),
  [ [ [ 1 ] ], [ [ -1 ] ] ],
  "hom"
)
gap> DirectSumARep(R1, R2);
DirectSumARep(
  RegularARep( GroupWithGenerators( [ (1,2,3,4), (1,3) ] ) ),
  ARepByImages(
    GroupWithGenerators( [ (1,2,3,4), (1,3) ] ),
    [ [ [ 1 ] ], [ [ -1 ] ] ],
    "hom"
  )
)
```

`DirectSumARep(` *list* `)`

`DirectSumARep` returns an arep of type `"directSum"` representing the direct sum of the areps in *list* (see above).

74.78 InnerTensorProductARep

`InnerTensorProductARep(` R_1`, ...,` R_k `)`

`InnerTensorProductARep` returns an arep of type `"innerTensorProduct"` representing the inner tensor product $R = R_1 \otimes \ldots \otimes R_k$ of the areps R_1, ..., R_k, which must have common source and characteristic otherwise an error is signaled.

The inner tensor product $R = R_1 \otimes \ldots \otimes R_k$ of representations is defined as $x \mapsto R_1(x) \otimes \ldots \otimes R_k(x)$. Note that the inner tensor product yields a representation of the same source (in contrast to the outer tensor product, see 74.79).

Note that the tensor factors R_1, ..., R_k can be accessed in the field `.factors` of the result.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> R1 := ARepByImages(G, [ Mon( (1,2), 2 ), Mon( [-1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2), (3,4) ] ),
  [ Mon( (1,2), 2 ), Mon( [ -1, -1 ] ) ],
  "hom"
)
gap> R2 := NaturalARep(G, 5);
NaturalARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 5 )
gap> InnerTensorProductARep(R1, R2);
InnerTensorProductARep(
  ARepByImages(
    GroupWithGenerators( [ (1,2), (3,4) ] ),
    [ Mon( (1,2), 2 ), Mon( [ -1, -1 ] ) ],
    "hom"
  ),
  NaturalARep( GroupWithGenerators( [ (1,2), (3,4) ] ), 5 )
)
```

`InnerTensorProductARep(` *list* `)`

`InnerTensorProductARep` returns an arep of type `"innerTensorProduct"` representing the inner tensor product of the areps in *list* (see above).

74.79 OuterTensorProductARep

`OuterTensorProductARep(` R_1`, ...,` R_k `)`
`OuterTensorProductARep(` G`,` R_1`, ...,` R_k `)`

`OuterTensorProductARep` returns an arep of type `"outerTensorProduct"` representing the outer tensor product $R = R_1 \# \ldots \# R_k$ of the areps R_1, ..., R_k, which must have common characteristic otherwise an error is signaled.

The outer tensor product $R = R_1 \# \ldots \# R_k$ of representations is defined as $x \mapsto R_1(x) \otimes \ldots \otimes R_k(x)$. Note that the outer tensor product of representations is a representation of the direct product of the sources (in contrast to the inner tensor product, see 74.78).

Using the first version `OuterTensorProductARep` returns an arep R with R`.source = DirectProduct(`R_1`.source, ...,` R_k`.source)` using the GAP3 function `DirectProduct`. In the second version the returned arep has as source the group G which must be the inner direct product $G = R_1$`.source` $\times \ldots \times$ R_k`.source`. This property is not checked.

Note that the tensor factors R_1, ..., R_k can be accessed in the field `.factors` of the result.

```
gap> G1 := GroupWithGenerators(DihedralGroup(8));
Group( (1,2,3,4), (2,4) )
gap> G2 := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R1 := NaturalARep(G1, 4);
NaturalARep( GroupWithGenerators( [ (1,2,3,4), (2,4) ] ), 4 )
gap> R2 := ARepByImages(G2, [ [[-1]] ]);
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ -1 ] ] ],
  "hom"
)
gap> OuterTensorProductARep(R1, R2);
OuterTensorProductARep(
  NaturalARep( GroupWithGenerators( [ (1,2,3,4), (2,4) ] ), 4 ),
  ARepByImages(
    GroupWithGenerators( [ (1,2) ] ),
    [ [ [ -1 ] ] ],
    "hom"
  )
)
```

74.80 RestrictionARep

`RestrictionARep(` R`,` H `)`

`RestrictionARep` returns an arep of type `"restriction"` representing the restriction of the arep R to the subgroup H of R`.source`. Here, "subgroup" means, that all elements of H are contained in R`.source`.

The restriction $R \downarrow H$ of a representation R to a subgroup H is defined by $x \mapsto R(x),\ x \in H$.

Note that R can be accessed in the field `.rep` of the result.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> H := GroupWithGenerators(AlternatingGroup(4));
Group( (1,2,4), (2,3,4) )
gap> R := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 )
gap> RestrictionARep(R, H);
RestrictionARep(
  NaturalARep( GroupWithGenerators( [ (1,4), (2,4), (3,4) ] ), 4 ),
  GroupWithGenerators( [ (1,2,4), (2,3,4) ] )
)
```

74.81 InductionARep

`InductionARep(` R, G `)`
`InductionARep(` R, G, T `)`

`InductionARep` returns an arep of type `"induction"` representing the induction of the arep R to the supergroup G with the transversal T of the residue classes R`.source`$\backslash G$. Here, “supergroup” means that all elements of R`.source` are contained in G. If no transversal T is supplied one is chosen by the function `RightTransversal`. If a transversal T is given it is not checked to be one.

The induction $R \uparrow_T G$ of a representation R of H to a supergroup G with transversal $T = \{t_1, \ldots, t_k\}$ of $H \backslash G$ is defined by $x \mapsto \left[\dot{R}\left(t_i \cdot x \cdot t_j^{-1}\right) \mid i, j \in \{1, \ldots, k\}\right]$, where $\dot{R}(y) = R(y)$ for $y \in H$ and 0 else.

Note that R and T can be accessed in the fields `.rep` and `.transversal` resp. of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3,4), (1,2) ] );
Group( (1,2,3,4), (1,2) )
gap> H := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(H, [ [[Z(2), Z(2)], [0*Z(2), Z(2)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ]
  ],
  "hom"
)
gap> R.name := "R";
"R"
gap> InductionARep(R, G);
InductionARep(
  R,
  GroupWithGenerators( [ (1,2,3,4), (1,2) ] ),
  [ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,4,3),
```

```
  (1,4), (1,4,2,3), (1,4)(2,3), (1,2,3), (1,2,3,4) ]
)
```

74.82 ExtensionARep

`ExtensionARep(` *R*, *chi* `)`

Let R be an irreducible arep of characteristic zero and *chi* a character of a supergroup of R`.source` which extends the character of R. `ExtensionARep` returns an arep of type `"extension"` representing an extension of R to *chi*`.source`. Here, “supergroup” means that all elements of R`.source` are contained in G. The extension is evaluated using Minkwitz's formula (see [Min96]).

Note that R and *chi* can be accessed in the fields `.rep` and `.character` of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3,4), (1,2) ] );
Group( (1,2,3,4), (1,2) )
gap> H := GroupWithGenerators(AlternatingGroup(4));
Group( (1,2,4), (2,3,4) )
gap> G.name := "S4";
"S4"
gap> H.name := "A4";
"A4"
gap> R := ARepByImages(H, [ Mon( (1,2,3), [ 1, -1, -1 ] ),
> Mon( (1,2,3), 3 ) ] );
ARepByImages(
  A4,
  [ Mon( (1,2,3), [ 1, -1, -1 ] ),
    Mon( (1,2,3), 3 )
  ],
  "hom"
)
gap> L := Irr(G);
[ Character( Group( (1,2,3,4), (1,2) ), [ 1, 1, 1, 1, 1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 1, -1, 1, 1, -1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 2, 0, -1, 2, 0 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 3, -1, 0, -1, 1 ] ),
  Character( Group( (1,2,3,4), (1,2) ), [ 3, 1, 0, -1, -1 ] ) ]
gap> ExtensionARep(R, L[4]);
ExtensionARep(
  ARepByImages(
    A4,
    [ Mon(
        (1,2,3),
        [ 1, -1, -1 ]
      ),
      Mon( (1,2,3), 3 )
    ],
    "hom"
  ),
```

```
  Character( Group( (1,2,3,4), (1,2) ), [ 3, -1, 0, -1, 1 ] )
)
```

74.83 GaloisConjugateARep

`GaloisConjugateARep(` R, *aut* `)`
`GaloisConjugateARep(` R, k `)`

`GaloisConjugateARep` returns an arep of type `"galoisConjugate"` representing the Galois conjugate of the arep A. The conjugating automorphism may either be a field automorphism *aut* or an integer k specifying the automorphism `x -> GaloisCyc(x,` k`)` in the case characteristic $= 0$ or `x -> x^(FrobeniusAut^`k`)` in the case characteristic $= p$ prime.

The Galois conjugate of a representation R with a field automorphism *aut* is defined by $x \mapsto R(x)^{aut}$.

Note that R and *aut* can be accessed in the fields `.rep` and `.galoisAut` resp. of the result.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R := ARepByImages(G, [ [[E(3)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ [ [ E(3) ] ]
  ],
  "hom"
)
gap> GaloisConjugateARep(R, -1);
GaloisConjugateARep(
  ARepByImages(
    GroupWithGenerators( [ (1,2,3) ] ),
    [ [ [ E(3) ] ]
    ],
    "hom"
  ),
  -1
)
```

74.84 Basic Functions for AReps

The following sections describe basic functions for areps like e.g. testing irreducibility and equivalence, evaluating an arep at a group element, computing kernel and character, and constructing an arep with given character.

74.85 Comparison of AReps

R_1 `=` R_2
R_1 `<>` R_2

The equality operator `=` evaluates to `true` if the areps R_1 and R_2 are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the amats R_1 and R_2 are not equal and to `false` otherwise.

Two areps are equal iff they define the same representation. This means that first the sources have to be equal, i.e. R_1`.source =` R_2`.source` and second the images are pointwise equal.

R_1 `<` R_2
R_1 `<=` R_2
R_1 `>=` R_2
R_1 `>` R_2

The operators `<`, `<=`, `>=`, and `>` evaluate to `true` if the arep R_1 is strictly less than, less than or equal to, greater than or equal to, and strictly greater than the arep R_2.

The ordering of areps is defined via the ordering of records.

74.86 ImageARep

`ImageARep(` x`,` R `)` or x `^` R

Let R be an arep and x a group element of R`.source`. `ImageARep` returns the image of x under R as an amat (see 74.22). For conversion of amats see 74.48 – 74.50.

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> x := Random(G);
c
gap> ImageARep(x, R);
TensorProductAMat(
  AMatPerm((1,2)(3,4)(5,6)(7,8), 8),
  IdentityPermAMat(1)
) *
DirectSumAMat(
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1),
  IdentityPermAMat(1)
)
gap> PermAMat(last);
(1,2)(3,4)(5,6)(7,8)
```

`ImageARep(` *list*`,` R `)`

`ImageARep` returns the list of images of the group elements in *list* under the arep R (see above). The images are amats (see 74.22). For conversion of amats see 74.48 – 74.50.

74.87 IsEquivalentARep

`IsEquivalentARep(` R_1`,` R_2 `)`

Let R_1 and R_2 be two areps with Maschke condition, i.e. `Size(` R_i`.source ) mod` R_i`.char` $\neq 0,\ i = 1, 2$. `IsEquivalentARep` returns `true` if the areps R_1 and R_2 define equivalent representations and `false` otherwise. Two representations (with Maschke condition) are equivalent iff they have the same character. R_1 and R_2 must have identical source (i.e. `IsIdentical(`R_1`,` R_2`) = true`) and characteristic otherwise an error is signaled.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := RegularARep(G);
RegularARep( GroupWithGenerators( [ (1,2,3) ] ) )
gap> IsEquivalentARep(R1, R2);
true
```

74.88 CharacterARep

`CharacterARep(` R `)`

`CharacterARep` returns the character of the arep R. Since GAP3 only provides characters of characteristic zero, `CharacterARep` only works in this case and will signal an error otherwise. Note that `CharacterARep` sets and tests R`.character`.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> CharacterARep(RegularARep(G));
Character( Group( (1,2), (3,4) ), [ 4, 0, 0, 0 ] )
```

74.89 IsIrreducibleARep

`IsIrreducibleARep(` R `)`

Let R an arep of characteristic zero. `IsIrreducibleARep` returns `true` if R represents an irreducible arep and `false` otherwise. To determine irreducibility the character is used, which is the reason for the condition characteristic = 0 (see 74.88). Note that `IsIrreducibleARep` sets and tests R`.isIrreducible`.

```
gap> G := GroupWithGenerators(SolvableGroup(12, 5));
A4
gap> L := Irr(G);
[ Character( A4, [ 1, 1, 1, 1 ] ),
  Character( A4, [ 1, 1, E(3), E(3)^2 ] ),
  Character( A4, [ 1, 1, E(3)^2, E(3) ] ),
  Character( A4, [ 3, -1, 0, 0 ] ) ]
gap> R := ARepByCharacter(L[2]);
ARepByImages(
  A4,
  [ Mon( [ E(3) ] ),
    Mon( (), 1 ),
    Mon( (), 1 )
  ],
```

```
  "hom"
)
gap> IsIrreducibleARep(R);
true
gap> IsIrreducibleARep(RegularARep(G));
false
```

74.90 KernelARep

`KernelARep(` *R* `)`

`KernelARep` returns the kernel of the arep *R*. Note that `KernelARep` sets and tests *R*`.kernel`.

```
gap> G := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> R := ARepByImages(G, [ [[-1]], [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,3), (2,3) ] ),
  [ [ [ -1 ] ],
    [ [ -1 ] ]
  ],
  "hom"
)
gap> KernelARep(R);
Subgroup( Group( (1,3), (2,3) ), [ (1,3,2) ] )
```

74.91 IsFaithfulARep

`IsFaithfulARep(` *R* `)`

`IsFaithfulARep` returns `true` if the arep *R* represents a faithful representation and `false` otherwise. Note that `IsFaithfulARep` sets and tests *R*`.isFaithful`.

```
gap> G := GroupWithGenerators(SolvableGroup(16, 7));
Q8x2
gap> IsFaithfulARep(TrivialPermARep(G));
false
gap> IsFaithfulARep(RegularARep(G));
true
```

74.92 ARepWithCharacter

`ARepWithCharacter(` *chi* `)`

`ARepWithCharacter` constructs an arep with character *chi*. The group *chi*`.source` must be solvable otherwise an error is signaled. Note that the function returns a monomial arep if this is possible.

Attention: `ARepWithCharacter` only works in GAP3 3.4.4 after bugfix 9!

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
```

```
gap> L := Irr(G);
[ Character( Q8, [ 1, 1, 1, 1, 1 ] ),
  Character( Q8, [ 1, 1, -1, 1, -1 ] ),
  Character( Q8, [ 1, 1, 1, -1, -1 ] ),
  Character( Q8, [ 1, 1, -1, -1, 1 ] ),
  Character( Q8, [ 2, -2, 0, 0, 0 ] ) ]
gap> MonARepARep(ARepWithCharacter(L[5]));
ARepByImages(
  Q8,
  [ Mon(
      (1,2),
      [ -1, 1 ]
    ),
    Mon( [ E(4), -E(4) ] ),
    Mon( [ -1, -1 ] )
  ],
  "hom"
)
```

74.93 GeneralFourierTransform

`GeneralFourierTransform(` G `)`

`GeneralFourierTransform` returns an amat representing a Fourier transform over the complex numbers for the solvable group G. For an explanation of Fourier transforms see [CB93]. In order to obtain a *fast* Fourier transform for G apply the function `DecompositionMonRep` to any regular representation of G.

Attention: `GeneralFourierTransform` only works in GAP3 3.4.4 after bugfix 9!

```
gap> G := SymmetricGroup(3);
Group( (1,3), (2,3) )
gap> GeneralFourierTransform(G);
AMatMat(
  [ [ 1, 1, 1, 1, 1, 1 ], [ 1, -1, -1, 1, 1, -1 ],
  [ 1, 0, 0, E(3), E(3)^2, 0 ], [ 0, 1, E(3)^2, 0, 0, E(3) ],
  [ 0, 1, E(3), 0, 0, E(3)^2 ], [ 1, 0, 0, E(3)^2, E(3), 0 ] ],
  "invertible"
) ^ -1
```

74.94 Converting AReps

The following sections describe functions for convertibility and conversion of arbitrary areps to areps of type `"perm"`, `"mon"`, and `"mat"`. As in **AMat** (see 74.22) the naming of the functions follows the usual GAP3-convention: `ChalkCheese` makes chalk from cheese. The parts in the name (chalk, cheese) are:

ARep	– an arep of any type
PermARep	– an arep of type “perm”
MonARep	– an arep of type “mon”
MatARep	– an arep of type “mat”

74.95 IsPermRep

`IsPermRep(` *R* `)`

`IsPermRep` returns `true` if R represents a permutation representation and `false` otherwise. Note that the name of this function is not `IsPermARep` since R can be an arep of any type but represents a permutation representation in the mathematical sense (every image is a permutation matrix). Note that `IsPermRep` sets and tests R`.isPermRep`.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ Mon( [1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ 1, -1 ] )
  ],
  "hom"
)
gap> IsPermRep(ConjugateARep(R, DFTAMat(2)));
true
```

74.96 IsMonRep

`IsMonRep(` *R* `)`

`IsMonRep` returns `true` if R represents a monomial representation and `false` otherwise. Note that the name of this function is not `IsMonARep` since R can be an arep of any type but represents a monomial representation in the mathematical sense (every image is a monomial matrix). Note that `IsMonRep` sets and tests R`.isMonRep`.

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> IsMonRep(InnerTensorProductARep(R, R));
true
```

74.97 PermARepARep

`PermARepARep(` *R* `)`

`PermARepARep` returns an arep of type `"perm"` representing the same representation as the arep R if possible. Otherwise `false` is returned. Note that `PermARepARep` sets and tests R`.permARep`.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ Mon( [1, -1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ Mon( [ 1, -1 ] )
  ],
```

```
  "hom"
)
gap> PermARepARep(ConjugateARep(R, DFTAMat(2)));
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
gap> PermARepARep(R);
false
```

74.98 MonARepARep

`MonARepARep(` *R* `)`

`MonARepARep` returns an arep of type `"mon"` representing the same representation as the arep *R* if possible. Otherwise `false` is returned. Note that `MonARepARep` sets and tests *R*`.monARep`.

```
gap> G := GroupWithGenerators( [ (1,2,3), (1,2) ] );
Group( (1,2,3), (1,2) )
gap> R1 := ARepByImages(G, [ [[1]], [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ [ [ 1 ] ],
    [ [ -1 ] ]
  ],
  "hom"
)
gap> R2 := NaturalARep(G, 4);
NaturalARep( GroupWithGenerators( [ (1,2,3), (1,2) ] ), 4 )
gap> MonARepARep(InnerTensorProductARep(R1, R2));
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( (1,2,3), 4 ),
    Mon(
      (1,2),
      [ -1, -1, -1, -1 ]
    )
  ],
  "hom"
)
```

74.99 MatARepARep

`MatARepARep(` *R* `)`

`MatARepARep` returns an arep of type `"mat"` representing the same representation as the arep *R*. Note that `MatARepARep` sets and tests *R*`.matARep`.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> MatARepARep(RegularARep(G, 3));
ARepByImages(
```

```
  GroupWithGenerators( [ (1,2), (3,4) ] ),
  [ [ [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ],
      [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] ],
    [ [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ]
  ],
  "hom"
)
```

74.100 Higher Functions for AReps

The following sections describe functions allowing the structural manipulation of, mainly monomial, areps. The idea is to convert a given arep into a mathematical equal (not only equivalent!) arep having more structure. Examples are: converting a transitive monomial arep into a conjugated induction (see 74.111), converting an induction into a conjugated double induction (see 74.112), changing the transversal of an induction (see 74.115), decomposing a transitive monomial arep into a conjugated outer tensor product (see 74.116) and last but not least decomposing a monomial arep into a conjugated sum of irreducibles (see 74.123). The latter is one of the most interesting functions of the package AREP.

74.101 IsRestrictedCharacter

`IsRestrictedCharacter(` *chi*, *chisub* `)`

`IsRestrictedCharacter` returns `true` if the character *chisub* is a restriction of the character *chi* to *chisub*`.source` and `false` otherwise. All elements of *chisub*`.source` must be contained in *chi*`.source` otherwise an error is signaled.

```
gap> G := SymmetricGroup(3); G.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> H := CyclicGroup(3); H.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> L1 := Irr(G);
[ Character( S3, [ 1, 1, 1 ] ), Character( S3, [ 1, -1, 1 ] ),
  Character( S3, [ 2, 0, -1 ] ) ]
gap> L2 := Irr(H);
[ Character( Z3, [ 1, 1, 1 ] ), Character( Z3, [ 1, E(3), E(3)^2 ] ),
  Character( Z3, [ 1, E(3)^2, E(3) ] ) ]
gap> IsRestrictedCharacter(L1[2], L2[1]);
true
```

74.102 AllExtendingCharacters

`AllExtendingCharacters(` *chi*, *G* `)`

`AllExtendingCharacters` returns the list of all characters of G extending *chi*. All elements of *chi*`.source` must be contained in G otherwise an error is signaled.

```
gap> H := AlternatingGroup(4); H.name := "A4";
Group( (1,2,4), (2,3,4) )
"A4"
gap> G := SymmetricGroup(4); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> L := Irr(H);
[ Character( A4, [ 1, 1, 1, 1 ] ),
  Character( A4, [ 1, 1, E(3)^2, E(3) ] ),
  Character( A4, [ 1, 1, E(3), E(3)^2 ] ),
  Character( A4, [ 3, -1, 0, 0 ] ) ]
gap> AllExtendingCharacters(L[4], G);
[ Character( S4, [ 3, -1, -1, 0, 1 ] ),
  Character( S4, [ 3, 1, -1, 0, -1 ] ) ]
```

74.103 OneExtendingCharacter

`OneExtendingCharacter(` *chi*`,` G `)`

`OneExtendingCharacter` returns one character of G extending *chi* if possible or returns false otherwise. All elements of *chi*`.source` must be contained in G otherwise an error is signaled.

```
gap> H := Group( (1,3)(2,4) ); H.name := "Z2";
Group( (1,3)(2,4) )
"Z2"
gap> G := Group( (1,2,3,4) ); G.name := "Z4";
Group( (1,2,3,4) )
"Z4"
gap> L := Irr(H);
[ Character( Z2, [ 1, 1 ] ), Character( Z2, [ 1, -1 ] ) ]
gap> OneExtendingCharacter(L[2], G);
Character( Z4, [ 1, E(4), -1, -E(4) ] )
```

74.104 IntertwiningSpaceARep

`IntertwiningSpaceARep(` R_1`,` R_2 `)`

`IntertwiningSpaceARep` returns a list of amats (see 74.22) representing a base of the intertwining space $\mathrm{Int}(R_1, R_2)$ of the areps R_1 and R_2, which must have common source and characteristic otherwise an error is signaled.

The intertwining space $\mathrm{Int}(R_1, R_2)$ of two representations R_1 and R_2 of a group G of the same characteristic is the vector space of matrices $\{M \mid R_1(x) \cdot M = M \cdot R_2(x), \text{ for all } x \in G\}$.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
```

```
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := ARepByImages(G, [ Mon( [1, E(3), E(3)^2] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ Mon( [ 1, E(3), E(3)^2 ] )
  ],
  "hom"
)
gap> IntertwiningSpaceARep(R1, R2);
[ AMatMat( [ [ 1, 0, 0 ], [ 1, 0, 0 ], [ 1, 0, 0 ] ] ),
  AMatMat( [ [ 0, 1, 0 ], [ 0, E(3), 0 ], [ 0, E(3)^2, 0 ] ] ),
  AMatMat( [ [ 0, 0, 1 ], [ 0, 0, E(3)^2 ], [ 0, 0, E(3) ] ] ) ]
```

74.105 IntertwiningNumberARep

`IntertwiningNumberARep(` R_1`,` R_2 `)`

`IntertwiningNumberARep` returns the intertwining number of the areps R_1 and R_2. The Maschke condition must hold for both R_1 and R_2, otherwise an error is signaled. R_1 and R_2 must have identical source (i.e. IsIdentical(R_1, R_2) = `true`) and characteristic otherwise an error is signaled.

The intertwining number of two representations R_1 and R_2 (with Maschke condition) is the dimension of the intertwining space or the scalar product of the characters.

```
gap> G := GroupWithGenerators(SolvableGroup(64, 12));
2^3xD8
gap> R := RegularARep(G);
RegularARep( 2^3xD8 )
gap> IntertwiningNumberARep(R, R);
64
```

74.106 UnderlyingPermRep

`UnderlyingPermRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `UnderlyingPermRep` returns an arep of type `"perm"` representing the underlying permutation representation of R.

The underlying permutation representation of a monomial representation R is obtained by replacing all entries $\neq 0$ in the images $R(x),\ x \in G$ by 1.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ [[0, 2], [1/2, 0]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ 0, 2 ], [ 1/2, 0 ] ]
  ],
  "hom"
)
gap> UnderlyingPermARep(R);
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
```

74.107 IsTransitiveMonRep

`IsTransitiveMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `IsTransitiveMonRep` returns `true` if R is transitive and `false` otherwise. Note that `IsTransitiveMonRep` sets and tests R`.isTransitive`.

A monomial representation is transitive iff the underlying permutation representation is.

```
gap> G := GroupWithGenerators( [ (1,2), (3,4) ] );
Group( (1,2), (3,4) )
gap> IsTransitiveMonRep(NaturalARep(G, 4));
false
gap> IsTransitiveMonRep(RegularARep(G));
true
```

74.108 IsPrimitiveMonRep

`IsPrimitiveMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `IsPrimitiveMonRep` returns `true` if R is primitive and `false` otherwise.

A monomial representation is primitive iff the underlying permutation representation is.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(SymmetricGroup(3)); H.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> L := Irr(H);
[ Character( S3, [ 1, 1, 1 ] ), Character( S3, [ 1, -1, 1 ] ),
  Character( S3, [ 2, 0, -1 ] ) ]
gap> R := ARepByCharacter(L[2]);
ARepByImages(
  S3,
  [ Mon( [ -1 ] ),
    Mon( [ -1 ] )
  ],
  "hom"
)
gap> IsPrimitiveMonRep(InductionARep(R, G));
true
```

74.109 TransitivityDegreeMonRep

`TransitivityDegreeMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `TransitivityDegreeMonRep` returns the degree of transitivity of R as an integer. Note that `TransitivityDegreeMonRep` sets and tests R`.transitivity`.

The degree of transitivity of a monomial representation is defined as the degree of transitivity of the underlying permutation representation.

```
gap> G := GroupWithGenerators(AlternatingGroup(5));
Group( (1,2,5), (2,3,5), (3,4,5) )
gap> TransitivityDegreeMonRep(NaturalARep(G, 5));
3
```

74.110 OrbitDecompositionMonRep

`OrbitDecompositionMonRep(` R `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `) = true`). `OrbitDecompositionMonRep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^P$ where $R_i,\ i = 1, \ldots, k$ are transitive areps of type `"mon"` and P is an amat of type `"perm"` (for amats see 74.22).

```
gap> G := GroupWithGenerators( [ (1,2,3,4) ] ); G.name := "Z4";
Group( (1,2,3,4) )
"Z4"
gap> R := ARepByImages(G, [ Mon( (1,2)(3,4), [1,-1,1,1,-1] ) ] );
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4) ] ),
  [ Mon( (1,2)(3,4), [ 1, -1, 1, 1, -1 ] ) ],
  "hom"
)
gap> OrbitDecompositionMonRep(R);
ConjugateARep(
  DirectSumARep(
    ARepByImages(
      Z4,
      [ Mon( (1,2), [ 1, -1 ] ) ],
      "hom"
    ),
    ARepByImages(
      Z4,
      [ Mon( (1,2), 2 ) ],
      "hom"
    ),
    ARepByImages(
      Z4,
      [ Mon( [ -1 ] ) ],
      "hom"
    )
  ),
  IdentityPermAMat(5)
)
```

74.111 TransitiveToInductionMonRep

`TransitiveToInductionMonRep(` R `)`
`TransitiveToInductionMonRep(` R, i `)`

Let R be a transitive monomial arep of a group G. `TransitiveToInductionMonRep` returns an arep equal to R with structure $R = (L \uparrow_T G)^D$. L is an arep of degree one of the stabilizer H of the point i and T a transversal of $H\backslash G$. The default for i is R`.degree`. D is a diagonal amat (see 74.22) of type `"mon"`. Note that `TransitiveToInductionMonRep` sets and tests the field R`.induction` if $i = R$`.degree`.

```
gap> G := GroupWithGenerators(DihedralGroup(8));
Group( (1,2,3,4), (2,4) )
gap> R := ARepByImages(G, [ Mon( [E(4), E(4)^-1] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3,4), (2,4) ] ),
  [ Mon( [ E(4), -E(4) ] ), Mon( (1,2), 2 ) ],
  "hom"
)
gap> TransitiveToInductionMonRep(R);
ConjugateARep(
  InductionARep(
    ARepByImages(
      GroupWithGenerators( [ (1,2,3,4) ] ),
      [ Mon( [ -E(4) ] ) ],
      "hom"
    ),
    GroupWithGenerators( [ (1,2,3,4), (2,4) ] ),
    [ (2,4), () ]
  ),
  IdentityMonAMat(2)
)
```

74.112 InsertedInductionARep

`InsertedInductionARep(` R, H `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_T G$ where L is an arep of $U \leq G$ and $U \leq H \leq G$. `InsertedInductionARep` returns an arep equal to R with structure $((L \uparrow_{T_1} H) \uparrow_{T_2} G)^M$ where M is an amat (see 74.22) with a structure similar to R. If R`.rep` is of degree 1 then M is an amat of type `"mon"`.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(AlternatingGroup(4)); H.name := "A4";
Group( (1,2,4), (2,3,4) )
"A4"
gap> U := GroupWithGenerators(CyclicGroup(3)); U.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> R := ARepByImages(U, [ [[E(3)]] ] );
ARepByImages(
  Z3,
  [ [ [ E(3) ] ]
```

```
  ],
  "hom"
)
gap> InsertedInductionARep(InductionARep(R, G), H);
ConjugateARep(
  InductionARep(
    InductionARep(
      ARepByImages(
        Z3,
        [ [ [ E(3) ] ] ],
        "hom"
      ),
      A4,
      [ (), (2,3,4), (2,4,3), (1,4)(2,3) ]
    ),
    S4,
    [ (), (3,4) ]
  ),
  AMatMon( Mon(
    (2,4,8,7,3,5),
    [ 1, 1, 1, 1, 1, 1, E(3)^2, 1 ]
  ) )
)
```

74.113 ConjugationPermReps

`ConjugationPermReps(` R_1`,` R_2 `)`

Let R_1 and R_2 be permutation representations (i.e. `IsPermRep(` R_i `) = true`, $i = 1, 2$). `ConjugationPermReps` returns an amat A (see 74.22) of type `"perm"` such that ${R_1}^A = R_2$. R_1 and R_2 must have common source and characteristic otherwise an error is signaled.

```
gap> G := GroupWithGenerators( [ (1,2,3) ] );
Group( (1,2,3) )
gap> R1 := NaturalARep(G, 3);
NaturalARep( GroupWithGenerators( [ (1,2,3) ] ), 3 )
gap> R2 := ARepByImages(G, [ (1,3,2) ], 3);
ARepByImages(
  GroupWithGenerators( [ (1,2,3) ] ),
  [ (1,3,2)
  ],
  3, # degree
  "hom"
)
gap> A := ConjugationPermReps(R1, R2);
AMatPerm((2,3), 3)
gap> R1 ^ A = R2;
true
```

74.114 ConjugationTransitiveMonReps

`ConjugationTransitiveMonReps(` R_1`,` R_2 `)`

Let R_1 and R_2 be transitive monomial representations. `ConjugationTransitiveMonReps` returns an amat A (see 74.22) of type `"mon"` such that ${R_1}^A = R_2$ if possible and `false` otherwise. R_1 and R_2 must have common source otherwise an error is signaled.

Note that a conjugating monomial matrix exists iff R_1 and R_2 are induced from inner conjugated representations of degree one (see [Püs98]).

```
gap> G := GroupWithGenerators( [ (1,2,3), (1,2) ] );
Group( (1,2,3), (1,2) )
gap> R1 := ARepByImages(G, [ Mon( [E(3), E(3)^2] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( [ E(3), E(3)^2 ] ),
    Mon( (1,2), 2 )
  ],
  "hom"
)
gap> R2 := ARepByImages(G, [ Mon( [E(3)^2, E(3)] ), Mon( (1,2), 2 ) ]);
ARepByImages(
  GroupWithGenerators( [ (1,2,3), (1,2) ] ),
  [ Mon( [ E(3)^2, E(3) ] ),
    Mon( (1,2), 2 )
  ],
  "hom"
)
gap> ConjugationTransitiveMonReps(R1, R2);
AMatMon( Mon( (1,2), 2 ) )
```

74.115 TransversalChangeInductionARep

`TransversalChangeInductionARep(` R`,` T `)`
`TransversalChangeInductionARep(` R`,` T`,` *hint* `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_S G$ and T another transversal of L`.source`$\backslash G$. `TransversalChangeInductionARep` returns an arep equal to R with structure $(L \uparrow_T G)^M$ where M is an amat (see 74.22). M is of type `"mon"` if L is of degree 1 else M has a structure similar to R. The *hint* `"isTransversal"` suppresses checking T to be a right transversal.

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators(SymmetricGroup(3)); H.name := "S3";
Group( (1,3), (2,3) )
"S3"
gap> R := ARepByImages(H, [ [[-1]], [[-1]] ], "hom" );
ARepByImages(
```

```
    S3,
    [ [ [ -1 ] ], [ [ -1 ] ] ],
    "hom"
  )
gap> RG := InductionARep(R, G);
InductionARep(
  ARepByImages(
    S3,
    [ [ [ -1 ] ], [ [ -1 ] ] ],
    "hom"
  ),
  S4,
  [ (), (3,4), (2,4), (1,4) ]
)
gap> T := [(1,2,3,4), (2,3,4), (3,4), ()];;
gap> TransversalChangeInductionARep(RG, T);
ConjugateARep(
  InductionARep(
    ARepByImages(
      S3,
      [ [ [ -1 ] ], [ [ -1 ] ] ],
      "hom"
    ),
    S4,
    [ (1,2,3,4), (2,3,4), (3,4), () ]
  ),
  AMatMon( Mon( (1,4)(2,3), [ 1, 1, -1, 1 ] ) )
)
gap> last = RG;
true
```

74.116 OuterTensorProductDecompositionMonRep

`OuterTensorProductDecompositionMonRep(` R `)`

Let R be a transitive monomial arep. `OuterTensorProductDecompositionMonRep` returns an arep equal to R with structure $(R_1\# \ \ldots \# R_k)^M$. The R_i are areps of type `"mon"`, M is an amat of type mon.

For a definition of the outer tensor product of representations see 74.79. For an explanation of the algorithm see [Püs98].

```
gap> G := GroupWithGenerators(SolvableGroup(48, 16));
2x4xS3
gap> R := RegularARep(G, 2);
RegularARep( 2x4xS3, GF(2) )
gap> OuterTensorProductDecompositionMonRep(R);
ConjugateARep(
  OuterTensorProductARep(
    2x4xS3,
```

```
    ARepByImages(
      GroupWithGenerators( [ c ] ),
      [ Mon( (1,2), 2, GF(2) ) ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ d, e ] ),
      [ Mon( (1,3,2,4), 4, GF(2) ),
        Mon( (1,2)(3,4), 4, GF(2) )
      ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ a*e, b ] ),
      [ Mon( (1,4)(2,6)(3,5), 6, GF(2) ),
        Mon( (1,2,3)(4,5,6), 6, GF(2) )
      ],
      "hom"
    )
  ),
  AMatMon( Mon( ( 2, 9,18,44,16,28,30,46,31, 6,42,48,47,39,23,35,37, 7)
( 3,17,36,45,24,43, 8,10,25, 5,34,29,38,15,19, 4,26,13)
(11,33,22,27,21,20,12,41,40,32,14), 48, GF(2) ) )
)
gap> last = R;
true
```

74.117 InnerConjugationARep

`InnerConjugationARep(` R, G, t `)`

Let R be an arep with source $H \leq G$ and $t \in G$. `InnerConjugationARep` returns an arep of type `"perm"` or `"mon"` or `"mat"`, the most specific possible, representing the inner conjugate R^t of R with t.

The inner conjugate R^t is a representation of H^t defined by $x \mapsto R(t \cdot x \cdot t^{-1})$.

```
gap> G := GroupWithGenerators(SymmetricGroup(4));
Group( (1,4), (2,4), (3,4) )
gap> H := GroupWithGenerators(SymmetricGroup(3));
Group( (1,3), (2,3) )
gap> R := NaturalARep(H, 3);
NaturalARep( GroupWithGenerators( [ (1,3), (2,3) ] ), 3 )
gap> InnerConjugationARep(R, G, (1,2,3,4));
ARepByImages(
  GroupWithGenerators( [ (2,4), (3,4) ] ),
  [ (1,3), (2,3) ],
  3, # degree
  "hom"
)
```

74.118 RestrictionInductionARep

`RestrictionInductionARep(` R, K `)`

Let R be an arep of type `"induction"`, i.e. $R = L \uparrow_T G$ where L is an arep of $H \leq G$ of degree 1 and $K \leq G$ a subgroup. `RestrictionInductionARep` returns an arep equal to the restriction $R \downarrow K$ with structure $\left(\bigoplus_{i=1}^{k}((L^{s_i} \downarrow (H^{s_i} \cap K)) \uparrow_{T_i} K)\right)^M$. $S = \{s_1, \ldots, s_k\}$ is a transversal of the double cosets $H \backslash G / K$, L^{s_i} denotes the inner conjugate of R with s_i, and M is an amat (see 74.22) of type `"mon"`.

Note that this decomposition is based on a refined version of Mackey's subgroup theorem (see [Püs98]).

```
gap> G := GroupWithGenerators(SymmetricGroup(4)); G.name := "S4";
Group( (1,4), (2,4), (3,4) )
"S4"
gap> H := GroupWithGenerators( [ (1,2) ] ); H.name := "Z2";
Group( (1,2) )
"Z2"
gap> K := GroupWithGenerators( [ (1,2,3) ] ); K.name := "Z3";
Group( (1,2,3) )
"Z3"
gap> L := ARepByImages(H, [ Mon( [-1] ) ] );
ARepByImages(
  Z2,
  [ Mon( [ -1 ] )
  ],
  "hom"
)
gap> RestrictionInductionARep(InductionARep(L, G), K);
ConjugateARep(
  DirectSumARep(
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) ),
    RegularARep( GroupWithGenerators( [ (1,2,3) ] ) )
  ),
  AMatMon( Mon(
    ( 2,12, 4, 6, 9, 5, 8,10),
    [ 1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1 ]
  ) )
)
```

74.119 kbsARep

`kbsARep(` R `)`

`kbsARep` returns the kbs (conjugated blockstructure) of the arep R. The kbs of a representation is a partition of the set $\{1, \ldots, R.\texttt{degree}\}$ representing the blockstructure of R. For an exact definition see 74.167.

Note that for a monomial representation the kbs is exactly the list of orbits.

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ (2,3) ], 4);
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ (2,3) ],
  4, # degree
  "hom"
)
gap> kbsARep(R);
[ [ 1 ], [ 2, 3 ], [ 4 ] ]
```

74.120 RestrictionToSubmoduleARep

`RestrictionToSubmoduleARep(` *R*, *list* `)`
`RestrictionToSubmoduleARep(` *R*, *list*, *hint* `)`

Let R be an arep and *list* a subset of `[1..`R`.degree]`. `RestrictionToSubmoduleARep` returns an arep of type `"perm"` or `"mon"` or `"mat"`, the most specific possible, representing the restriction of R to the submodule generated by the base vectors given through *list*. The optional *hint* `"hom"` avoids the check for homomorphism.

Note that the restriction to the submodule given by *list* defines a homomorphism iff *list* is a union of lists in the kbs of R (see 74.119).

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G, [ (2,4) ], 4);
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ (2,4) ],
  4, # degree
  "hom"
)
gap> RestrictionToSubmoduleARep(R, [2,4]);
NaturalARep( GroupWithGenerators( [ (1,2) ] ), 2 )
```

74.121 kbsDecompositionARep

`kbsDecompositionARep(` *R* `)`

`kbsDecompositionARep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^P$ where P is an amat (see 74.22) of type `"perm"`and all R_i have trivial kbs (see 74.119).

Note that for a monomial arep `kbsDecompositionARep` performs exactly the same as the function `OrbitDecompositionMonRep` (see 74.110).

```
gap> G := GroupWithGenerators( [ (1,2) ] );
Group( (1,2) )
gap> R := ARepByImages(G,
```

```
> [ [[Z(2), Z(2), 0*Z(2), 0*Z(2)], [0*Z(2), Z(2), 0*Z(2), 0*Z(2)],
> [0*Z(2), 0*Z(2), Z(2), Z(2)], [0*Z(2), 0*Z(2), 0*Z(2), Z(2)]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,2) ] ),
  [ [ [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
  ],
  "hom"
)
gap> kbsDecompositionARep(R);
ConjugateARep(
  DirectSumARep(
    ARepByImages(
      GroupWithGenerators( [ (1,2) ] ),
      [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ],
      "hom"
    ),
    ARepByImages(
      GroupWithGenerators( [ (1,2) ] ),
      [ [ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ] ],
      "hom"
    )
  ),
  IdentityPermAMat(4, GF(2))
)
```

74.122 ExtensionOnedimensionalAbelianRep

`ExtensionOnedimensionalAbelianRep(` R, G `)`

Let R be an arep of the subgroup $H \leq G$ and let $G/\mathrm{kernel}(R)$ be an abelian factor group. `ExtensionOnedimensionalAbelianRep` returns an arep of type `"mon"` and degree 1 extending R to G. For the extension the smallest possible extension field is chosen.

```
gap> G := GroupWithGenerators(CyclicGroup(8));
Group( (1,2,3,4,5,6,7,8) )
gap> H := GroupWithGenerators( [ G.1^2 ] );
Group( (1,3,5,7)(2,4,6,8) )
gap> R := ARepByImages(H, [ [[-1]] ] );
ARepByImages(
  GroupWithGenerators( [ (1,3,5,7)(2,4,6,8) ] ),
  [ [ [ -1 ] ]
  ],
  "hom"
)
gap> ExtensionOnedimensionalAbelianRep(R, G);
ARepByImages(
```

```
    GroupWithGenerators( [ (1,2,3,4,5,6,7,8) ] ),
    [ Mon( [ E(4) ] )
    ],
    "hom"
  )
```

74.123 DecompositionMonRep

`DecompositionMonRep(` R `)`
`DecompositionMonRep(` R`,` *hint* `)`

Let R be a monomial arep (i.e. `IsMonRep(` R `)` yields `true`). `DecompositionMonRep` returns an arep equal to R with structure $(R_1 \oplus \ldots \oplus R_k)^{A^{-1}}$ where all R_i are irreducible and A^{-1} is a highly structured amat (see 74.22). A is a decomposition matrix for R and can be accessed in the field `.conjugation.element` of the result. The list of the R_i can be accessed in the field `.rep.summands` of the result. Note that any R_i is monomial if this is possible. If the *hint* `"noOuter"` is supplied, the decomposition of R is performed without any decomposition into an outer tensor product which may speed up the function. The function only works for characteristic zero otherwise an error is signaled. At least the following types of monomial areps can be decomposed: monomial representations of solvable groups, double transitive permutation representations, primitive permutation representations with solvable socle. If `DecompositionMonRep` is not able to decompose R then `false` is returned. The performance of `DecompositionMonRep` depends on the size of the group represented as well as on the degree of R. E.g. the decomposition of a regular representation of a group of size 96 takes less than half a minute (CPU-time on a SUN Ultra-Sparc 150 MHz) if the source group is an ag group.

Note that in the case that R is a regular representation of the solvable group G the structured decomposition matrix A computed by `DecompositionMonRep` represents a fast Fourier transform for G. Hence, `DecompositionMonRep` is able to compute a fast Fourier transform for any solvable group.

The algorithm is a major result of [Püs98] where a thorough explanation can be found.

Set `InfoLatticeDec := Print` to obtain information on the recursive decomposition of R.

An important application of this function is the automatic generation of fast algorithms for discrete signal transforms which is realized in 74.147. (see [Min93], [Egn97a], [Püs98]).

```
gap> G := GroupWithGenerators(SolvableGroup(8, 5));
Q8
gap> R := RegularARep(G);
RegularARep( Q8 )
gap> DecompositionMonRep(R);
ConjugateARep(
  DirectSumARep(
    TrivialMonARep( Q8 ),
    ARepByImages(
      Q8,
      [ Mon( [ -1 ] ), Mon( [ -1 ] ), Mon( (), 1 ) ],
      "hom"
    ),
```

```
    ARepByImages(
      Q8,
      [ Mon( [ -1 ] ), Mon( (), 1 ), Mon( (), 1 ) ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( (), 1 ), Mon( [ -1 ] ), Mon( (), 1 ) ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( (1,2), [ -1, 1 ] ),
        Mon( [ E(4), -E(4) ] ),
        Mon( [ -1, -1 ] )
      ],
      "hom"
    ),
    ARepByImages(
      Q8,
      [ Mon( (1,2), [ -1, 1 ] ),
        Mon( [ E(4), -E(4) ] ),
        Mon( [ -1, -1 ] )
      ],
      "hom"
    )
  ),
  ( AMatPerm((7,8), 8) *
    TensorProductAMat(
      IdentityPermAMat(2),
      AMatPerm((2,3), 4) *
      TensorProductAMat(
        DFTAMat(2),
        IdentityPermAMat(2)
      ) *
      DiagonalAMat([ 1, 1, 1, E(4) ]) *
      TensorProductAMat(
        IdentityPermAMat(2),
        DFTAMat(2)
      ) *
      AMatPerm((2,3), 4)
    ) *
    AMatMon( Mon(
      (2,5,3)(4,8,7),
      [ 1, 1, 1, 1, 1, 1, -1, 1 ]
    ) ) *
    DirectSumAMat(
      TensorProductAMat(
```

```
          DFTAMat(2),
          IdentityPermAMat(2)
        ),
        IdentityPermAMat(4)
      ) *
      AMatPerm((2,4), 8)
    ) ^ -1
)
gap>  last = R;
true
```

74.124 Symmetry of Matrices

The following sections describe functions for the computation of symmetry of a given matrix. A symmetry of a matrix is a pair (R_1, R_2) of representations of the same group G with the property $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$. This definition corresponds to the definition of the intertwining space of R_1, R_2 (see 74.104). The origin of this definition is due to Minkwitz (see [Min95], [Min93]) and was generalized to the definition above by the authors of this package.

Restrictions on the representations R_1, R_2 yield special types of symmetry. We consider the following three types:

- Perm-Irred symmetry: R_1 is a permutation representation, R_2 is a conjugated (by a permutation) direct sum of irreducible representations
- Perm-Perm symmetry: both R_1 and R_2 are permutation representations
- Mon-Mon symmetry: both R_1 and R_2 are monomial representations

There are two implementations for the search algorithm for Perm-Perm-Symmetry. One is entirely in GAP3 by S. Egner, the other uses the external C-program `desauto` bei J. Leon which is distributed with the GUAVA package. By default the GAP3 code is run. In order to use the much faster method of J. Leon based on partitions (see [Leo91]) you should set `UseLeon := true` and make sure that an executable version of `desauto` is placed in `$GAP/pkg/arep/bin`. The implementation of Leon requires the matrix to have ≤ 256 different entries. If this condition is violated the GAP3 implementation is run.

A matrix with symmetry of one of the types above contains structure in a sense and can be decomposed into a product of highly structured sparse matrices (see 74.147).

For details on the concept and computation of symmetry see [Egn97a] and [Püs98].

The following functions are implemented in the file `"arep/lib/symmetry.g"` based on functions from `"arep/lib/permperm.g"`, `"arep/lib/monmon.g"`, `"arep/lib/permblk.g"` and `"arep/lib/permmat.g"`.

74.125 PermPermSymmetry

`PermPermSymmetry(` M `)`

Let M be a matrix or an amat (see 74.22). `PermPermSymmetry` returns a pair (R_1, R_2) of areps of type `"perm"` (see 74.66) of the same group G representing the perm-perm symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$. The returned symmetry is maximal in the sense that for every pair (p_1, p_2) of permutations satisfying $p_1 \cdot M = M \cdot p_2$ there is an x with $p_1 = R_1(x)$ and $p_2 = R_2(x)$.

To use the much faster implementation of J. Leon set `UseLeon := true` as explained in 74.124.

Set `InfoPermSym1 := true` to obtain information about the search.

For the algorithm see [Leo91] resp. [Egn97a].

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> L := PermPermSymmetry(M);
[ ARepByImages(
      GroupWithGenerators( [ g1, g2 ] ),
      [ (2,3,5,4),
        (2,5)(3,4)
      ],
      5, # degree
      "hom"
    ), ARepByImages(
      GroupWithGenerators( [ g1, g2 ] ),
      [ (2,4,5,3),
        (2,5)(3,4)
      ],
      5, # degree
      "hom"
    ) ]
gap> L[1]^AMatMat(M) = L[2];
true
```

74.126 MonMonSymmetry

`MonMonSymmetry(` M `)`

Let M be a matrix or an amat (see 74.22) of characteristic zero. `MonMonSymmetry` returns a pair (R_1, R_2) of areps of type `"mon"` (see 74.66) of the same group G representing a mon-mon symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$.

The non-zero entries in the matrices $R_1(x), R_2(x)$ are all roots of unity of a certain order d. This order is given by the lcm of all quotients of non-zero entries of M with equal absolute value. The returned symmetry is maximal in the sense that for every pair (m_1, m_2) of monomial matrices containing only dth roots of unity (and 0) and satisfying $m_1 \cdot M = M \cdot m_2$ there is an x with $m_1 = R_1(x)$ and $m_2 = R_2(x)$.

`MonMonSymmetry` uses the function `PermPermSymmetry`. Hence you can accelerate the function using the faster implementation of J. Leon by setting `UseLeon := true` as explained in 74.124.

For an explanation of the algorithm see [Püs98].

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> L := MonMonSymmetry(M);
[ ARepByImages(
      GroupWithGenerators( [ g1, g2, g3, g4, g5 ] ),
      [ Mon(
          (2,3,5,4),
          [ 1, E(5)^3, E(5), E(5)^4, E(5)^2 ]
        ),
        Mon(
          (2,5)(3,4),
          [ 1, E(5)^2, E(5)^4, E(5), E(5)^3 ]
        ),
        Mon(
          (1,2,3,4,5),
          [ E(5), E(5), E(5), E(5), E(5) ]
        ),
        Mon( [ E(5), 1, E(5)^4, E(5)^3, E(5)^2 ] ),
        Mon( [ 1, E(5), E(5)^2, E(5)^3, E(5)^4 ] )
      ],
      "hom"
    ), ARepByImages(
      GroupWithGenerators( [ g1, g2, g3, g4, g5 ] ),
      [ Mon( (1,3,4,2), 5 ),
        Mon( (1,4)(2,3), 5 ),
        Mon( [ E(5), E(5)^2, E(5)^3, E(5)^4, 1 ] ),
        Mon(
          (1,2,3,4,5),
          [ E(5), E(5), E(5), E(5), E(5) ]
        ),
        Mon( (1,5,4,3,2), 5 )
      ],
      "hom"
    ) ]
gap> L[1]^AMatMat(M) = L[2];
true
```

74.127 PermIrredSymmetry

`PermIrredSymmetry(` *M* `)`
`PermIrredSymmetry(` *M* `,` *maxblocksize* `)`

Let M be a matrix or an amat (see 74.22) of characteristic zero. `PermIrredSymmetry` returns a list of pairs (R_1, R_2) of areps (see 74.66) of the same group G representing a perm-irred symmetry of M, i.e. $R_1(x) \cdot M = M \cdot R_2(x)$ for all $x \in G$ and R_1 is a permutation representation and R_2 a conjugated (by a permutation) direct sum of irreducible representations. If *maxblocksize* is supplied exactly those perm-irred symmetries are returned where R_2 contains at least one irreducible of degree $\leq$ *maxblocksize*. The default for *maxblocksize* is 2.

Refer to [Egn97a] to understand how the search is done and how to interpret the result.

Note that the perm-irred symmetry is not symmetric. Hence it is possible that a matrix M admits a perm-irred symmetry but its transpose not.

The perm-irred symmetry is a special case of a perm-block symmetry. The perm-block symmetries admitted by a fixed matrix M can be described by two lattices which are in a certain way related to each other (semi-order preserving). To explore this structure (described in [Egn97a]) you should refer to `PermBlockSym` and `DisplayPermBlockSym` in the file `"arep/lib/permblk.g"`.

```
gap> M := DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
gap> PermIrredSymmetry(M);
[ [ NaturalARep( G2, 4 ), ConjugateARep(
          DirectSumARep(
            TrivialMatARep( G2 ),
            ARepByImages(
              G2,
              [ [ [ -1 ] ],
                [ [ E(4) ] ]
              ],
              "hom"
            ),
            ARepByImages(
              G2,
              [ [ [ 1 ] ],
                [ [ -1 ] ]
              ],
              "hom"
            ),
            ARepByImages(
              G2,
              [ [ [ -1 ] ],
                [ [ -E(4) ] ]
              ],
              "hom"
```

```
        )
      ),
      IdentityPermAMat(4)
    ) ], [ NaturalARep( G3, 4 ), ConjugateARep(
      DirectSumARep(
        TrivialMatARep( G3 ),
        ARepByImages(
          G3,
          [ [ [ 0, -E(4) ], [ E(4), 0 ] ],
            [ [ 0, 1 ], [ 1, 0 ] ],
            [ [ 0, -1 ], [ -1, 0 ] ]
          ],
          "hom"
        ),
        ARepByImages(
          G3,
          [ [ [ -1 ] ],
            [ [ 1 ] ],
            [ [ 1 ] ]
          ],
          "hom"
        )
      ),
      AMatPerm((3,4), 4)
    ) ], [ NaturalARep( G1, 4 ), ConjugateARep(
      DirectSumARep(
        TrivialMatARep( G1 ),
        ARepByImages(
          G1,
          [ [ [ 1/2, -1/2+1/2*E(4), 1/2*E(4) ],
      [ -1/2-1/2*E(4), 0, -1/2+1/2*E(4) ],
      [ -1/2*E(4), -1/2-1/2*E(4), 1/2 ] ],
            [ [ 0, 0, 1 ], [ 0, 1, 0 ], [ 1, 0, 0 ] ],
            [ [ 1/2, 1/2+1/2*E(4), -1/2*E(4) ],
      [ 1/2-1/2*E(4), 0, 1/2+1/2*E(4) ],
      [ 1/2*E(4), 1/2-1/2*E(4), 1/2 ] ]
          ],
          "hom"
        )
      ),
      IdentityPermAMat(4)
    ) ] ]
```

74.128 Discrete Signal Transforms

The following sections describe functions for the construction of many well known signal transforms in matrix form, as e.g. the discrete Fourier transform, several discrete cosine transforms etc. For the definition of the mentioned signal transforms see [ER82], [Mal92], [Mer96].

The functions for discrete signal transforms are implemented in `"arep/lib/transf.g"`.

74.129 DiscreteFourierTransform

`DiscreteFourierTransform(` *r* `)`
`DiscreteFourierTransform(` *n* `)`
`DiscreteFourierTransform(` *n*`,` *char* `)`

shortcut: `DFT`

`DiscreteFourierTransform` or `DFT` returns the discrete Fourier transform from a given root of unity *r* or the size *n* and the characteristic *char* (see [CB93]). The default for *char* is zero. Note that the DFT on *n* points and characteristic *char* exists iff *n* and *char* are coprime. If this condition is violenced an error is signaled.

The DFT_n of size n is defined as $\mathrm{DFT}_n = [\omega_n^{k\ell} \mid k, \ell \in \{0, \ldots, n-1\}]$, ω_n a primitive nth root of unity.

```
gap> DFT(Z(3));
[ [ Z(3)^0, Z(3)^0 ], [ Z(3)^0, Z(3) ] ]
gap> DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
```

74.130 InverseDiscreteFourierTransform

`InverseDiscreteFourierTransform(` *r* `)`
`InverseDiscreteFourierTransform(` *n* `)`
`InverseDiscreteFourierTransform(` *n*`,` *char* `)`

shortcut: `InvDFT`

`InverseDiscreteFourierTransform` or `InvDFT` returns the inverse of the discrete Fourier transform from a given root of unity *r* or the size *n* and the characteristic *char* (see 74.129). The default for *char* is zero.

```
gap> InvDFT(3);
[ [ 1/3, 1/3, 1/3 ], [ 1/3, 1/3*E(3)^2, 1/3*E(3) ],
  [ 1/3, 1/3*E(3), 1/3*E(3)^2 ] ]
```

74.131 DiscreteHartleyTransform

`DiscreteHartleyTransform(` *n* `)`

shortcut: `DHT`

`DiscreteHartleyTransform` or `DHT` returns the discrete Hartley transform on *n* points.

The DHT_n of size n is defined by $\mathrm{DHT}_n = [1/\sqrt{n} \cdot (\cos(2\pi k\ell/n) + \sin(2\pi k\ell/n)) \mid k, \ell \in \{0, \ldots, n-1\}]$.

```
gap> DHT(4);
[ [ 1/2, 1/2, 1/2, 1/2 ], [ 1/2, 1/2, -1/2, -1/2 ],
  [ 1/2, -1/2, 1/2, -1/2 ], [ 1/2, -1/2, -1/2, 1/2 ] ]
```

74.132 InverseDiscreteHartleyTransform

`InverseDiscreteHartleyTransform(` n `)`

shortcut: `InvDHT`

`InverseDiscreteHartleyTransform` or `InvDHT` returns the inverse of the discrete Hartley transform on n points. Since the DHT is self inverse the result is exactly the same as from `DHT` above.

```
gap> InvDHT(4);
[ [ 1/2, 1/2, 1/2, 1/2 ], [ 1/2, 1/2, -1/2, -1/2 ],
  [ 1/2, -1/2, 1/2, -1/2 ], [ 1/2, -1/2, -1/2, 1/2 ] ]
```

74.133 DiscreteCosineTransform

`DiscreteCosineTransform(` n `)`

shortcut: `DCT`

`DiscreteCosineTransform` returns the standard cosine transform (type II) on n points. The DCT_n of size n is defined by $\mathrm{DCT}_n = [\sqrt{2/n} \cdot c_k \cdot (\cos(k(\ell+1/2)\pi/n) \mid k, \ell \in \{0, \dots, n-1\}]$, $c_k = 1/\sqrt{2}$ for $k = 0$ and $c_k = 1$ else.

```
gap> DCT(3);
[ [ 1/3*E(12)^7-1/3*E(12)^11, 1/3*E(12)^7-1/3*E(12)^11,
      1/3*E(12)^7-1/3*E(12)^11 ],
  [ -1/2*E(8)+1/2*E(8)^3, 0, 1/2*E(8)-1/2*E(8)^3 ],
  [ -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

74.134 InverseDiscreteCosineTransform

`InverseDiscreteCosineTransform(` n `)`

shortcut: `InvDCT`

`InverseDiscreteCosineTransform` returns the inverse of the standard cosine transform (type II) on n points. Since the DCT is orthogonal, the result is the transpose of the DCT, which is exactly the discrete cosine transform of type III.

```
[ [ 1/3*E(12)^7-1/3*E(12)^11, -1/2*E(8)+1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 0,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 1/2*E(8)-1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

74.135 DiscreteCosineTransformIV

`DiscreteCosineTransformIV(` n `)`

shortcut: `DCT_IV`

DiscreteCosineTransformIV returns the cosine transform of type IV on n points.

The DCT_IV$_n$ of size n is defined by DCT_IV$_n$ $= [\sqrt{2/n} \cdot (\cos((k+1/2)(\ell+1/2)\pi/n) \mid k, \ell \in \{0, \ldots, n-1\}]$.

```
[ [ 1/2*E(12)^4+1/6*E(12)^7+1/2*E(12)^8-1/6*E(12)^11,
      1/3*E(12)^7-1/3*E(12)^11,
      1/2*E(12)^4-1/6*E(12)^7+1/2*E(12)^8+1/6*E(12)^11 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, -1/3*E(12)^7+1/3*E(12)^11,
      -1/3*E(12)^7+1/3*E(12)^11 ],
  [ 1/2*E(12)^4-1/6*E(12)^7+1/2*E(12)^8+1/6*E(12)^11,
      -1/3*E(12)^7+1/3*E(12)^11,
      1/2*E(12)^4+1/6*E(12)^7+1/2*E(12)^8-1/6*E(12)^11 ] ]
```

74.136 InverseDiscreteCosineTransformIV

InverseDiscreteCosineTransformIV(n)

shortcut: InvDCT_IV

InverseDiscreteCosineTransformIV returns the inverse of the cosine transform of type IV on n points. Since the DCT_IV is orthogonal, the result is the transpose of the DCT_IV.

```
[ [ 1/3*E(12)^7-1/3*E(12)^11, -1/2*E(8)+1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 0,
      1/3*E(24)-1/3*E(24)^11-1/3*E(24)^17+1/3*E(24)^19 ],
  [ 1/3*E(12)^7-1/3*E(12)^11, 1/2*E(8)-1/2*E(8)^3,
      -1/6*E(24)+1/6*E(24)^11+1/6*E(24)^17-1/6*E(24)^19 ] ]
```

74.137 DiscreteCosineTransformI

DiscreteCosineTransformI(n)

shortcut: DCT_I

DiscreteCosineTransformI returns the cosine transform of type I on $n+1$ points.

The DCT_I$_n$ of size $n+1$ is defined by DCT_I$_n$ $= [\sqrt{2/n} \cdot c_k \cdot c_\ell \cdot (\cos(k\ell\pi/n) \mid k, \ell \in \{0, \ldots, n\}]$, $c_k = 1/\sqrt{2}$ for $k = 0$ and $c_k = 1$ else.

```
[ [ 1/2, 1/2*E(8)-1/2*E(8)^3, 1/2 ],
  [ 1/2*E(8)-1/2*E(8)^3, 0, -1/2*E(8)+1/2*E(8)^3 ],
  [ 1/2, -1/2*E(8)+1/2*E(8)^3, 1/2 ] ]
```

74.138 InverseDiscreteCosineTransformI

InverseDiscreteCosineTransformI(n)

shortcut: InvDCT_I

InverseDiscreteCosineTransformI returns the inverse of the cosine transform of type I on n points. Since the DCT_I is orthogonal, the result is the transpose of the DCT_I.

```
[ [ 1/2, 1/2*E(8)-1/2*E(8)^3, 1/2 ],
  [ 1/2*E(8)-1/2*E(8)^3, 0, -1/2*E(8)+1/2*E(8)^3 ],
  [ 1/2, -1/2*E(8)+1/2*E(8)^3, 1/2 ] ]
```

74.139 WalshHadamardTransform

`WalshHadamardTransform(` n `)`

shortcut: `WHT`

`WalshHadamardTransform` returns the Walsh-Hadamard transform on n points.

Let $n = \prod_{i=1}^{k} p_i^{\nu_i}$ be the prime factor decomposition of n. Then the WHT_n is defined by $\mathrm{WHT}_n = \bigotimes_{i=1}^{k} \mathrm{DFT}_{p_i}^{\otimes \nu_i}$.

```
gap> WHT(4);
[ [ 1, 1, 1, 1 ], [ 1, -1, 1, -1 ],
  [ 1, 1, -1, -1 ], [ 1, -1, -1, 1 ] ]
```

74.140 InverseWalshHadamardTransform

`InverseWalshHadamardTransform(` n `)`

shortcut: `InvWHT`

`InverseWalshHadamardTransform` returns the inverse of the Walsh-Hadamard transform on n points.

```
gap> InvWHT(4);
[ [ 1/4, 1/4, 1/4, 1/4 ], [ 1/4, -1/4, 1/4, -1/4 ],
  [ 1/4, 1/4, -1/4, -1/4 ], [ 1/4, -1/4, -1/4, 1/4 ] ]
```

74.141 SlantTransform

`SlantTransform(` n `)`

shortcut: `ST`

`SlantTransform` returns the Slant transform on n points, which must be a power of 2, $n = 2^k$

For a definition of the Slant transform see [ER82], 10.9.

```
gap> ST(4);
[ [ 1/2, 1/2, 1/2, 1/2 ],
  [ 3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4,
    1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4,
      -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4 ],
  [ 1/2, -1/2, -1/2, 1/2 ],
  [ 1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4,
    -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4,
      3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4 ] ]
```

74.142 InverseSlantTransform

`InverseSlantTransform(` n `)`

shortcut: `InvST`

`InverseSlantTransform` returns the inverse of the Slant transform on n points, which must be a power of 2, $n = 2^k$. Since ST is orthogonal, this is exactly the transpose of the ST.

```
gap> InvST(4);
[ [ 1/2, 3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4, 1/2,
      1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4 ],
  [ 1/2, 1/10*E(5)-1/10*E(5)^2-1/10*E(5)^3+1/10*E(5)^4, -1/2,
      -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4 ],
  [ 1/2, -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4, -1/2,
      3/10*E(5)-3/10*E(5)^2-3/10*E(5)^3+3/10*E(5)^4 ],
  [ 1/2, -3/10*E(5)+3/10*E(5)^2+3/10*E(5)^3-3/10*E(5)^4, 1/2,
      -1/10*E(5)+1/10*E(5)^2+1/10*E(5)^3-1/10*E(5)^4 ] ]
```

74.143 HaarTransform

`HaarTransform(` n `)`

shortcut: `HT`

`HaarTransform` returns the Haar transform on n points, which must be a power of 2, $n = 2^k$.

For a definition of the Haar transform see [ER82], 10.10.

```
gap> HT(4);
[ [ 1/4, 1/4, 1/4, 1/4 ], [ 1/4, 1/4, -1/4, -1/4 ],
  [ 1/4*E(8)-1/4*E(8)^3, -1/4*E(8)+1/4*E(8)^3, 0, 0 ],
  [ 0, 0, 1/4*E(8)-1/4*E(8)^3, -1/4*E(8)+1/4*E(8)^3 ] ]
```

74.144 InverseHaarTransform

`InverseHaarTransform(` n `)`

shortcut: `InvHT`

`InverseHaarTransform` returns the inverse of the Haar transform on n points, which must be a power of 2, $n = 2^k$.

The inverse is exactly n times the transpose of HT.

```
gap> InvHT(4);
[ [ 1, 1, E(8)-E(8)^3, 0 ], [ 1, 1, -E(8)+E(8)^3, 0 ],
  [ 1, -1, 0, E(8)-E(8)^3 ], [ 1, -1, 0, -E(8)+E(8)^3 ] ]
```

74.145 RationalizedHaarTransform

`RationalizedHaarTransform(` n `)`

shortcut: `RHT`

`RationalizedHaarTransform` returns the rationalized Haar transform on n points, which must be a power of 2, $n = 2^k$.

For a definition of the rationalized Haar transform see [ER82], 10.11.

```
gap> RHT(4);
[ [ 1, 1, 1, 1 ], [ 1, 1, -1, -1 ],
  [ 1, -1, 0, 0 ], [ 0, 0, 1, -1 ] ]
```

74.146 InverseRationalizedHaarTransform

`InverseRationalizedHaarTransform(` n `)`

shortcut: `InvRHT`

`InverseRationalizedHaarTransform` returns the inverse of the rationalized Haar transform on n points, which must be a power of 2, $n = 2^k$.

```
gap> InvRHT(4);
[ [ 1/4, 1/4, 1/2, 0 ], [ 1/4, 1/4, -1/2, 0 ],
  [ 1/4, -1/4, 0, 1/2 ], [ 1/4, -1/4, 0, -1/2 ] ]
```

74.147 Matrix Decomposition

The decomposition of a matrix M with symmetry is a striking application of constructive representation theory and was the original motivation to create the package AREP. Here, decomposition means that M is decomposed into a product of highly structured sparse matrices. Applied to matrices corresponding to discrete signal transforms such a decomposition may represent a fast algorithm for the signal transform.

For the definition of symmetry see 74.124.

The idea of decomposing a matrix with symmetry is due to Minkwitz [Min95], [Min93] and was further developed by the authors of this package. See [Egn97a], chapter 1 or [Püs98], chapter 3 for a thorough explanation of the method.

The following three functions correspond to the three types of symmetry considered in this package (see 74.124). The functions are implemented in the file `"arep/lib/algogen.g"`.

74.148 MatrixDecompositionByPermPermSymmetry

`MatrixDecompositionByPermPermSymmetry(` M `)`

Let M be a matrix or an amat (see 74.22). `MatrixDecompositionByPermPermSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the perm-perm symmetry (see 74.125) to decompose the matrix (see 74.147) and can hence be accelerated by setting `UseLeon := true` as described in 74.124.

The following examples show that `MatrixDecompositionByPermPermSymmetry` discovers automatically the method of Rader (see [Rad68]) for a discrete Fourier transform of prime degree as well as the well-known decomposition of circulant matrices.

```
gap> M := DFT(5);;
gap> PrintArray(M);
[ [       1,       1,       1,       1,       1 ],
  [       1,    E(5),  E(5)^2,  E(5)^3,  E(5)^4 ],
  [       1,  E(5)^2,  E(5)^4,    E(5),  E(5)^3 ],
  [       1,  E(5)^3,    E(5),  E(5)^4,  E(5)^2 ],
  [       1,  E(5)^4,  E(5)^3,  E(5)^2,    E(5) ] ]
gap> MatrixDecompositionByPermPermSymmetry(M);
```

```
AMatPerm((4,5), 5) *
DirectSumAMat(
  IdentityPermAMat(1),
  TensorProductAMat(
    DFTAMat(2),
    IdentityPermAMat(2)
  ) *
  DiagonalAMat([ 1, 1, 1, E(4) ]) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DFTAMat(2)
  ) *
  AMatPerm((2,3), 4)
) *
AMatPerm((1,4,2,5,3), 5) *
DirectSumAMat(
  DiagonalAMat([ E(20)^4-E(20)^13-E(20)^16+E(20)^17,
  E(5)-E(5)^2-E(5)^3+E(5)^4, E(20)^4+E(20)^13-E(20)^16-E(20)^17 ]),
  AMatMat(
    [ [ 1, 4 ], [ 1, -1 ] ]
  )
) *
AMatPerm((1,3,5,2,4), 5) *
DirectSumAMat(
  IdentityPermAMat(1),
  AMatPerm((2,3), 4) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DiagonalAMat([ 1/2, 1/2 ]) *
    DFTAMat(2)
  ) *
  DiagonalAMat([ 1, 1, 1, -E(4) ]) *
  TensorProductAMat(
    DiagonalAMat([ 1/2, 1/2 ]) *
    DFTAMat(2),
    IdentityPermAMat(2)
  )
) *
AMatPerm((3,4,5), 5)

gap> M := [[1, 2, 3], [3, 1, 2], [2, 3, 1]];;
gap> PrintArray(M);
[ [  1,  2,  3 ],
  [  3,  1,  2 ],
  [  2,  3,  1 ] ]
gap> MatrixDecompositionByPermPermSymmetry(M);
DFTAMat(3) *
AMatMon( Mon(
```

```
  (2,3),
  [ 2, 2/3*E(3)+1/3*E(3)^2, 1/3*E(3)+2/3*E(3)^2 ]
) ) *
DFTAMat(3)
```

74.149 MatrixDecompositionByMonMonSymmetry

`MatrixDecompositionByMonMonSymmetry(` M `)`

Let M be a matrix or an amat (see 74.22). `MatrixDecompositionByMonMonSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the mon-mon symmetry (see 74.126) to decompose the matrix (see 74.147) and can hence be accelerated by setting `UseLeon := true` as described in 74.124.

The following example show that `MatrixDecompositionByMonMonSymmetry` is able to find automatically a decomposition of the discrete cosine transform of type IV (see 74.135).

```
gap> M := DCT_IV(8);;
gap> MatrixDecompositionByMonMonSymmetry(M);
AMatMon( Mon(
  (3,4,7,6,8,5),
  [ E(4), E(16)^5, E(8)^3, -E(16)^7, 1, -E(16), E(8), -E(16)^3 ]
) ) *
TensorProductAMat(
  DFTAMat(2),
  IdentityPermAMat(4)
) *
DiagonalAMat([ 1, 1, 1, 1, 1, E(8), E(4), E(8)^3 ]) *
TensorProductAMat(
  IdentityPermAMat(2),
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, E(4), 1, 1, 1, E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(4),
  DFTAMat(2)
) *
DiagonalAMat([ -E(64), -E(64), E(64)^9, -E(64)^9, E(64)^23, -E(64)^23,
  E(64)^31, E(64)^31 ]) *
TensorProductAMat(
  IdentityPermAMat(4),
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2)
) *
DiagonalAMat([ 1, 1, 1, -E(4), 1, 1, 1, -E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(2),
```

```
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, 1, 1, -E(8)^3, -E(4), -E(8) ]) *
TensorProductAMat(
  DiagonalAMat([ 1/2, 1/2 ]) *
  DFTAMat(2),
  IdentityPermAMat(4)
) *
AMatMon( Mon(
  (2,6,3,4,7,5,8),
  [ E(4), E(16)^5, -E(16)^7, E(8), E(8)^3, -E(16)^3, -E(16), 1 ]
) )
```

74.150 MatrixDecompositionByPermIrredSymmetry

`MatrixDecompositionByPermIrredSymmetry(` *M* `)`
`MatrixDecompositionByPermIrredSymmetry(` *M*, *maxblocksize* `)`

Let M be a matrix or an amat (see 74.22). `MatrixDecompositionByPermIrredSymmetry` returns a highly structured amat of type `"product"` with all factors being sparse which represents the matrix M. The returned amat can be viewed as a fast algorithm for the multiplication with M.

The function uses the perm-irred symmetry (see 74.127) to decompose the matrix (see 74.147).

If *maxblocksize* is supplied only those perm-irred symmetries with all irreducibles having degree less than *maxblocksize* are considered. The default for *maxblocksize* is 2.

Note that the perm-irred symmetry is not symmetric. Hence it is possible that a matrix M decomposes but its transpose not.

The following examples show that `MatrixDecompositionByPermIrredSymmetry` discovers automatically the Cooley-Tukey decomposition (see [CT65]) of a discrete Fourier transform as well as a decomposition of the transposed discrete cosine transform of type II (see 74.133).

```
gap> M := DFT(4);
[ [ 1, 1, 1, 1 ], [ 1, E(4), -1, -E(4) ], [ 1, -1, 1, -1 ],
  [ 1, -E(4), -1, E(4) ] ]
gap> MatrixDecompositionByPermIrredSymmetry(M);
TensorProductAMat(
  DFTAMat(2),
  IdentityPermAMat(2)
) *
DiagonalAMat([ 1, 1, 1, E(4) ]) *
TensorProductAMat(
  IdentityPermAMat(2),
  DFTAMat(2)
) *
AMatPerm((2,3), 4)
```

```
gap> M := TransposedMat(DCT(8));;
gap> MatrixDecompositionByPermIrredSymmetry(M);
AMatPerm((1,2,6,7,5,3,8), 8) *
TensorProductAMat(
  IdentityPermAMat(2),
  AMatPerm((3,4), 4) *
  TensorProductAMat(
    IdentityPermAMat(2),
    DFTAMat(2)
  ) *
  AMatPerm((2,3), 4) *
  DirectSumAMat(
    DFTAMat(2),
    IdentityPermAMat(2)
  )
) *
AMatPerm((2,7,5,4,3)(6,8), 8) *
DirectSumAMat(
  IdentityPermAMat(3),
  DirectSumAMat(
    IdentityPermAMat(1),
    AMatMat(
      [ [ -1/2*E(8)+1/2*E(8)^3, 1/2*E(8)-1/2*E(8)^3 ],
        [  1/2*E(8)-1/2*E(8)^3, 1/2*E(8)-1/2*E(8)^3 ] ],
      "invertible"
    )
  ),
  IdentityPermAMat(2)
) *
DirectSumAMat(
  TensorProductAMat(
    DFTAMat(2),
    IdentityPermAMat(3)
  ),
  IdentityPermAMat(2)
) *
AMatPerm((2,7,3,8,4), 8) *
DirectSumAMat(
  DiagonalAMat([ 1/4*E(8)-1/4*E(8)^3, 1/4*E(8)-1/4*E(8)^3 ]),
  AMatMat(
    [ [ 1/4*E(16)-1/4*E(16)^7, 1/4*E(16)^3-1/4*E(16)^5 ],
      [ 1/4*E(16)^3-1/4*E(16)^5, -1/4*E(16)+1/4*E(16)^7 ] ]
  ),
  AMatMat(
    [ [ -1/4*E(32)+1/4*E(32)^15, -1/4*E(32)^7+1/4*E(32)^9 ],
      [ 1/4*E(32)^7-1/4*E(32)^9, -1/4*E(32)+1/4*E(32)^15 ] ]
  ),
```

```
    AMatMat(
      [ [ -1/4*E(32)^3+1/4*E(32)^13, -1/4*E(32)^5+1/4*E(32)^11 ],
        [ -1/4*E(32)^5+1/4*E(32)^11, 1/4*E(32)^3-1/4*E(32)^13 ] ]
    )
  ) *
  AMatPerm((2,5)(4,7)(6,8), 8)
```

74.151 Complex Numbers

The next sections describe basic functions for the calculation with complex numbers which are represented as cyclotomics, e.g. computation of the complex conjugate or certain sine and cosine expressions.

The following functions are implemented in the file "arep/lib/complex.g".

74.152 ImaginaryUnit

`ImaginaryUnit( )`

`ImaginaryUnit` returns `E(4)`.

```
gap> ImaginaryUnit();
E(4)
```

74.153 Re

`Re(` *z* `)`

`Re` returns the real part of the cyclotomic z.

```
gap> z := E(3) + E(4);
E(12)^4-E(12)^7-E(12)^11
gap> Re(z);
-1/2
```

`Re(` *list* `)`

`Re` returns the list of the real parts of the cyclotomics in *list*.

74.154 Im

`Im(` *z* `)`

`Im` returns the imaginary part of the cyclotomic z.

```
gap> z := E(3) + E(4);
E(12)^4-E(12)^7-E(12)^11
gap> Im(z);
-E(12)^4-1/2*E(12)^7-E(12)^8+1/2*E(12)^11
```

`Im(` *list* `)`

`Im` returns the list of the imaginary parts of the cyclotomics in *list*.

74.155 AbsSqr

`AbsSqr( ` *z* ` )`

`AbsSqr` returns the squared absolute value of the cyclotomic z.

```
gap> AbsSqr(z);
-2*E(12)^4-E(12)^7-2*E(12)^8+E(12)^11
```

`AbsSqr( ` *list* ` )`

`AbsSqr` returns the list of the squared absolute values of the cyclotomics in *list*.

74.156 Sqrt

`Sqrt( ` *r* ` )`

`Sqrt` returns the square root of the rational number r.

```
gap> Sqrt(1/3);
1/3*E(12)^7-1/3*E(12)^11
```

74.157 ExpIPi

`ExpIPi( ` *r* ` )`

Let r be a rational number. `ExpIPi` returns $e^{\pi i r}$.

```
gap> ExpIPi(1/5);
-E(5)^3
```

74.158 CosPi

`CosPi( ` *r* ` )`

Let r be a rational number. `CosPi( ` *r* ` )` returns $\cos(\pi r)$.

```
gap> CosPi(1/5);
-1/2*E(5)^2-1/2*E(5)^3
```

74.159 SinPi

`SinPi( ` *r* ` )`

Let r be a rational number. `SinPi( ` *r* ` )` returns $\sin(\pi r)$.

```
gap> SinPi(1/5);
 -1/2*E(20)^13+1/2*E(20)^17
```

74.160 TanPi

`TanPi( ` *r* ` )`

Let r be a rational number. `TanPi( ` *r* ` )` returns $\tan(\pi r)$.

```
gap> TanPi(1/5);
E(20)-E(20)^9+E(20)^13-E(20)^17
```

74.161 Functions for Matrices and Permutations

The following sections describe basic functions for matrices and permutations, like forming the tensor product (Kronecker product) or direct sum and determination of the blockstructure of a matrix.

The following functions are implemented in the files `"arep/lib/permblk.g"` (`kbs`, see 74.167), `"arep/lib/summands.g"` (`DirectSummandsPermutedMat`, see 74.166) and the file `"arep/lib/tools.g"` (the other functions).

74.162 TensorProductMat

`TensorProductMat(` M_1`, ...,` M_k `)`

`TensorProductMat` returns the tensor product of the matrices $M_1, ..., M_k$.

```
gap> TensorProductMat( [[1]], [[1,2], [3,4]], [[5,6], [7,8]] );
[ [ 5, 6, 10, 12 ], [ 7, 8, 14, 16 ],
  [ 15, 18, 20, 24 ], [ 21, 24, 28, 32 ] ]
```

`TensorProductMat(` *list* `)`

`TensorProductMat` returns the tensor product of the matrices in *list*.

74.163 MatPerm

`MatPerm(` p`,` d `) MatPerm(` p`,` d`,` *char* `)`

`MatPerm` returns the permutation matrix of degree d corresponding to the permutation p in characteristic *char*. The default characteristic is 0. If d is less than the largest moved point of p an error is signaled.

We use the following convention to create a permutation matrix from a permutation p with degree d $[\delta_{i^p j} \mid i, j \in \{1, \ldots, d\}]$.

```
gap> MatPerm( (1,2,3), 4 );
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

74.164 PermMat

`PermMat(` M `)`

`PermMat` returns the permutation represented by the matrix M and returns false otherwise. For the convention see 74.163.

```
gap> PermMat( [[0,0,1], [1,0,0], [0,1,0]] );
(1,3,2)
```

74.165 PermutedMat

`PermutedMat(` p_1`,` M`,` p_2 `)`

Let p_1, p_2 be permutations and M a matrix with r rows and c columns. `PermutedMat` returns `MatPerm(` p_1`,` r `)` $\cdot M \cdot$`MatPerm(` p_2`,` c `)` (see 74.163). The largest moved point of p_1 and p_2 must not exceed r resp. c otherwise an error is signaled.

```
gap> PermutedMat( (1,2), [[1,2,3], [4,5,6], [7,8,9]], (1,2,3) );
[ [ 6, 4, 5 ], [ 3, 1, 2 ], [ 9, 7, 8 ] ]
```

74.166 DirectSummandsPermutedMat

`DirectSummandsPermutedMat(` M `)`
`DirectSummandsPermutedMat(` M `,` *match-blocks* `)`

Let M be a matrix. `DirectSummandsPermutedMat` returns the list `[` p_1`, [` M_1`, ...,` M_k `],` p_2 `]` where p_1, p_2 are permutations and M_i, $i = 1, \ldots, k$, are matrices with the property $M =$ `PermutedMat`(p_1, `DiagonalMat`(M_1, ..., M_k), p_2) (see 74.165, 34.12). If *match-blocks* is `true` or not provided then the permutations p_1 and p_2 are chosen such that equivalent M_i are equal and occur next to each other. If *match-blocks* is `false` this is not done.

For an explanation of the algorithm see [Egn97a].

```
gap> M := [ [ 0, 0, 0, 2, 0, 1], [ 3, 1, 0, 0, 0, 0],
 > [ 0, 0, 1, 0, 2, 0], [ 1, 2, 0, 0, 0, 0],
 > [ 0, 0, 0, 1, 0, 3], [ 0, 0, 3, 0, 1, 0] ];;
gap> PrintArray(M);
[ [  0,  0,  0,  2,  0,  1 ],
  [  3,  1,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  2,  0 ],
  [  1,  2,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  3 ],
  [  0,  0,  3,  0,  1,  0 ] ]
gap> DirectSummandsPermutedMat(M);
[ (2,4,3,5),
  [ [ [ 2, 1 ], [ 1, 3 ] ],
    [ [ 2, 1 ], [ 1, 3 ] ],
    [ [ 2, 1 ], [ 1, 3 ] ] ],
  (1,4)(2,6,3) ]
```

74.167 kbs

`kbs(` M `)`

Let M be a square matrix of degree n. `kbs` (konjugierte Blockstruktur = conjugated block structure) returns the partition $\mathrm{kbs}(M) = \{1, \ldots, n\}/R^*$ where R is the reflexive, symmetric, transitive closure of the relation R defined by $(i, j) \in R \Leftrightarrow M[i][j] \neq \mathtt{0}$.

For an investigation of the kbs of a matrix see [Egn97a].

```
gap> M := [[1,0,1,0], [0,2,0,3], [1,0,3,0], [0,4,0,1]];
[ [ 1, 0, 1, 0 ], [ 0, 2, 0, 3 ], [ 1, 0, 3, 0 ], [ 0, 4, 0, 1 ] ]
gap> PrintArray(M);
[ [  1,  0,  1,  0 ],
  [  0,  2,  0,  3 ],
  [  1,  0,  3,  0 ],
  [  0,  4,  0,  1 ] ]
gap> kbs(M);
[ [ 1, 3 ], [ 2, 4 ] ]
```

`kbs(` *list* `)`

`kbs` returns the joined kbs of the matrices in *list*. The matrices in *list* must have common size otherwise an error is signaled.

74.168 DirectSumPerm

`DirectSumPerm(` *list1* `,` *list2* `)`

Let *list2* contain permutations and *list1* be of the same length and contain degrees equal or larger than the corresponding largest moved points. `DirectSumPerm` returns the direct sum of the permutations defined via the direct sum of the corresponding matrices.

```
gap> DirectSumPerm( [3, 3], [(1,2), (1,2,3)] );
(1,2)(4,5,6)
```

74.169 TensorProductPerm

`TensorProductPerm(` *list1* `,` *list2* `)`

Let *list2* contain permutations and *list1* be of the same length and contain degrees equal or larger than the corresponding largest moved points. `TensorProductPerm` returns the tensor product (Kronecker product) of the permutations defined via the tensor product of the corresponding matrices.

```
gap> TensorProductPerm( [3, 3], [(1,2), (1,2,3)] );
(1,5,3,4,2,6)(7,8,9)
```

Chapter 75

Monoids and Semigroups

Semigroups and, even more, monoids are not far away from being like groups. But, surprisingly, they have not received much attention yet in the form of GAP3 programs. This small collection of files and manual chapters is an attempt to start closing this gap.

The only difference between a semigroup and a monoid is one element: the identity. Although this may lead to subtle differences in the behavior of these structures the underlying assumption of these programs is that you can always, by means of adding an element with the properties of an identity, turn a semigroup into a monoid. So most of the functions will only be available for monoids and not for semigroups. The actual process of adding an identity is also not supported at the moment.

The emphasis of this package is on transformation monoids (see chapter 78). However, it seemed to be a good idea to provide some of the framework for general monoids (this chapter) before concentrating on the special case. Separate chapters introduce transformations (see chapter 77) and binary relations (see chapter 76) as special types of monoid elements. Another chapter treats several ways of how a monoid can act on certain domains (see chapter 79).

For a general treatment of the theory of monoids and transformation monoids see [Lal79] and [How95]. A detailed description of this implementation and the theory behind it is given in [LPRR97] and [LPRR].

A semigroup is constructed by the function `SemiGroup` (see 75.4) and a monoid is constructed by the function `Monoid` (see 75.6).

Note that monoid elements usually exist independently of a monoid, e.g., you can write down two transformations and compute their product without ever defining a monoid that contains them.

The chapter starts with a description of monoid elements, i.e. all those objects that can be element of a semigroup or of a monoid (see 75.1, 75.2, and 75.3). Then the functions which construct monoids and semigroups and the functions which test whether a given object is a monoid or a semigroup are described (see 75.4, 75.5, 75.6 and 75.7).

Monoids and semigroups are domains, so every set theoretic function for domains is applicable to them (see 75.8). There are functions which construct Green Classes of various types as subsets of a monoid (see 75.9, 75.10, 75.13, 75.16 and 75.19), functions which test whether a given object is a Green class of a certain type (see 75.11, 75.14, 75.17 and 75.20), and functions which determine the list of all Green Classes of some given type of a monoid (see 75.12, 75.15, 75.18 and 75.21).

The next sections describe how set functions are applied to Green classes (see 75.22) and how to compute various kinds of Schützenberger groups (see 75.24).

The final sections describe how to determine the idempotents of a monoid (see 75.25), the lack of support for homomorphisms of monoids (see 75.26) and how monoids are represented by records in GAP3 (see 75.27).

The functions described here are in the file `"monoid.g"`.

75.1 Comparisons of Monoid Elements

s `=` *t*
s `<>` *t*

The equality operator `=` evaluates to `true` if the monoid elements *s* and *t* are equal and to `false` otherwise. The inequality operator `<>` evaluates to `true` if the monoid elements *s* and *t* are not equal and to `false` otherwise.

You can compare monoid elements with objects of other types. Of course they are never equal. Standard monoid elements are transformations (see chapter 77) and binary relations (see chapter 76).

s `<` *t*
s `<=` *t*
s `>=` *t*
s `>` *t*

The operators `<`, `<=`, `>=` and `>` evaluate to `true` if the monoid element *s* is strictly less than, less than or equal to, greater than or equal to and strictly greater than the monoid element *t*. There is no general ordering on monoid elements.

Standard monoid elements may be compared with objects of other types while generic monoid elements may disallow such a comparison.

75.2 Operations for Monoid Elements

s `*` *t*

The operator `*` evaluates to the product of the two monoid elements *s* and *t*. The operands must of course lie in a common parent monoid, otherwise an error is signaled.

s `^` *i*

The powering operator `^` returns the *i*-th power of a monoid element *s* and a positive integer *i*. If *i* is zero the identity of a parent monoid of *s* is returned.

list `*` *s*
s `*` *list*

In this form the operator `*` returns a new list where each entry is the product of *s* and the corresponding entry of *list*. Of course multiplication must be defined between *s* and each entry of *list*.

75.3 IsMonoidElement

`IsMonoidElement(` *obj* `)`

`IsMonoidElement` returns `true` if the object *obj*, which may be an object of arbitrary type, is a monoid element, and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsMonoidElement( 10 );
false
gap> IsMonoidElement( Transformation( [ 1, 2, 1 ] ) );
true
```

75.4 SemiGroup

`SemiGroup(` *list* `)`

`SemiGroup` returns the semigroup generated by the list *list* of semigroup elements.

```
gap> SemiGroup( [ Transformation( [ 1, 2, 1 ] ) ] );
SemiGroup( [ Transformation( [ 1, 2, 1 ] ) ] )
```

`SemiGroup(` *gen1* `,` *gen2* `, ... )`

In this form `SemiGroup` returns the semigroup generated by the semigroup elements *gen1*, *gen2*, ...

```
gap> SemiGroup( Transformation( [ 1, 2, 1 ] ) );
SemiGroup( [ Transformation( [ 1, 2, 1 ] ) ] )
```

75.5 IsSemiGroup

`IsSemiGroup(` *obj* `)`

`IsSemiGroup` returns `true` if the object *obj*, which may be an object of an arbitrary type, is a semigroup, and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsSemiGroup( SemiGroup( Transformation( [ 1, 2, 1 ] ) ) );
true
gap> IsSemiGroup( Group( (2,3) ) );
false
```

75.6 Monoid

`Monoid(` *list* `)`
`Monoid(` *list* `,` *id* `)`

`Monoid` returns the monoid generated by the list *list* of monoid elements. If present, *id* must be the identity of this monoid.

```
gap> Monoid( [ Transformation( [ 1, 2, 1 ] ) ],
> IdentityTransformation( 3 ) );
Monoid( [ Transformation( [ 1, 2, 1 ] ) ] )
```

`Monoid(` *gen1* , *gen2* , ... `)`

In this form `Monoid` returns the monoid generated by the monoid elements *gen1*, *gen2*, ...

```
gap> Monoid( Transformation( [ 1, 2, 1 ] ) );
Monoid( [ Transformation( [ 1, 2, 1 ] ) ] )
```

75.7 IsMonoid

`IsMonoid(` *obj* `)`

`IsMonoid` returns `true` if the object *obj*, which may be an object of an arbitrary type, is a monoid, and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsMonoid( Monoid( Transformation( [ 1, 2, 1 ] ) ) );
true
gap> IsMonoid( Group( (2,3) ) );
false
```

75.8 Set Functions for Monoids

Monoids and semigroups are domains. Thus all set theoretic functions described in chapter "Domains" should be applicable to monoids. However, no generic method is installed yet. Of particular interest are the functions `Size` and `Elements` which will have special methods depending on the kind of monoid being dealt with.

75.9 Green Classes

Green classes are special subsets of a monoid. In particular, they are domains so all set theoretic functions for domains (see chapter "Domains") can be applied to Green classes. This is described in section 75.22. The following sections describe how Green classes can be constructed.

75.10 RClass

`RClass(` *M* , *s* `)`

`RClass` returns the R class of the element s in the monoid M.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> RClass( M, Transformation( [ 1, 2, 2 ] ) );
RClass( M, Transformation( [ 1, 2, 2 ] ) )
```

The R class of s in M is the set of all elements of M which generate the same right ideal in M, i.e., the set of all m in M with $sM = mM$.

75.11 IsRClass

`IsRClass(` *obj* `)`

`IsRClass` returns `true` if the object *obj*, which may be an object of arbitrary type, is an R class, and `false` otherwise (see 75.10). It will signal an error if *obj* is an unbound variable.

75.12 RClasses

`RClasses(` *M* `)`
`RClasses(` *dClass* `)`

`RClasses` returns the list of R classes the monoid M. In the second form `RClasses` returns the list of R classes in the D class *dClass*.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> RClasses( M );
[ RClass( M, Transformation( [ 1, 2, 3 ] ) ),
  RClass( M, Transformation( [ 2, 1, 2 ] ) ),
  RClass( M, Transformation( [ 1, 2, 2 ] ) ) ]
```

75.13 LClass

`LClass(` *M*`,` *s* `)`

`LClass` returns the L class of the element s in the monoid M.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> LClass( M, Transformation( [ 1, 2, 2 ] ) );
LClass( M, Transformation( [ 1, 2, 2 ] ) )
```

The L class of s in M is the set of all elements of M which generate the same left ideal in M, i.e., the set of all m in M with $Ms = Mm$.

75.14 IsLClass

`IsLClass(` *obj* `)`

`IsLClass` returns `true` if the object *obj*, which may be an object of arbitrary type, is an L class, and `false` otherwise (see 75.13). It will signal an error if *obj* is an unbound variable.

75.15 LClasses

`LClasses(` *M* `)`
`LClasses(` *dClass* `)`

`LClasses` returns the list of L classes the monoid M. In the second form `LClasses` returns the list of L classes in the D class *dClass*.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
```

```
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> LClasses( M );
[ LClass( M, Transformation( [ 1, 2, 3 ] ) ),
  LClass( M, Transformation( [ 2, 1, 2 ] ) ) ]
```

75.16 DClass

`DClass(` *M* `,` *s* `)`

`DClass` returns the D class of the element s in the monoid M.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> DClass( M, Transformation( [ 1, 2, 2 ] ) );
DClass( M, Transformation( [ 1, 2, 2 ] ) )
```

The D class of s in M is the set of all elements of M which generate the same ideal in M, i.e., the set of all m in M with $MsM = MmM$.

75.17 IsDClass

`IsDClass(` *obj* `)`

`IsDClass` returns `true` if the object *obj*, which may be an object of arbitrary type, is a D class, and `false` otherwise (see 75.16). It will signal an error if *obj* is an unbound variable.

75.18 DClasses

`DClasses(` *M* `)`

`DClasses` returns the list of D classes the monoid M.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> DClasses( M );
[ DClass( M, Transformation( [ 1, 2, 3 ] ) ),
  DClass( M, Transformation( [ 2, 1, 2 ] ) ) ]
```

75.19 HClass

`HClass(` *M* `,` *s* `)`

`HClass` returns the H class of the element s in the monoid M.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> HClass( M, Transformation( [ 1, 2, 2 ] ) );
HClass( M, Transformation( [ 1, 2, 2 ] ) )
```

The H class of s in M is the intersection of the R class of s in M and the L class of s in M (see 75.10 and 75.13).

75.20 IsHClass

`IsHClass(` *obj* `)`

`IsHClass` returns `true` if the object *obj*, which may be an object of arbitrary type, is an H class, and `false` otherwise (see 75.19). It will signal an error if *obj* is an unbound variable.

75.21 HClasses

`HClasses(` *M* `)`
`HClasses(` *class* `)`

`HClasses` returns the list of H classes the monoid M. In the second form `HClasses` returns the list of all H classes in *class* where *class* is an R class, an L class or a D class.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> M.name:= "M";;
gap> HClasses( M );
[ HClass( M, Transformation( [ 1, 2, 3 ] ) ),
  HClass( M, Transformation( [ 2, 1, 2 ] ) ),
  HClass( M, Transformation( [ 2, 1, 1 ] ) ) ]
```

75.22 Set Functions for Green Classes

Green classes are domains so all set theoretic functions for domains can be applied to them. Most of the set functions will work via default methods once the following methods have been implemented.

`Size(` *class* `)`

determines the size of Green class *class*.

`Elements(` *class* `)`

returns the set of all elements of the Green class *class*

obj `in` *class*

returns `true` if *obj* is a member of the Green class *class* and `false` otherwise.

However, no generic methods are provided.

75.23 Green Class Records

A Green class is represented by a domain record with the following tag components.

`isDomain`
: is always `true`.

`isRClass`, `isLClass`, `isDClass`, or `isHClass`
: present and `true` depending on what kind of Green class is being dealt with.

The Green class is determined by the following identity components, which every Green class record must have.

`monoid`
: the monoid.

`representative`
: an element of the class. Which one is unspecified.

In addition to these a Green class record may have the following optional information components.

`elements`
: if present the proper set of elements of the class.

`size`
: if present the size of the class.

75.24 SchutzenbergerGroup

`SchutzenbergerGroup(` *M*, *s* `)`
`SchutzenbergerGroup(` *class* `)`

`SchutzenbergerGroup` returns the Schützenberger group of the element *s* in the monoid *M* as a group.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> SchutzenbergerGroup( M, Transformation( [ 2, 1, 2 ] ) );
Group( (1,2) )
```

In the second form `SchutzenbergerGroup` returns the Schützenberger group of the Green class *class* of a monoid.

75.25 Idempotents

`Idempotents(` *M* `)`
`Idempotents(` *class* `)`

returns the set of idempotents in the monoid *M* or in a Green class *class*.

```
gap> M:= Monoid( Transformation( [ 2, 1, 2 ] ),
> Transformation( [ 1, 2, 2 ] ) );;
gap> Idempotents( M );
[ Transformation( [ 1, 2, 1 ] ), Transformation( [ 1, 2, 2 ] ),
  Transformation( [ 1, 2, 3 ] ) ]
gap> Idempotents( DClass( M, Transformation( [ 2, 1, 2 ] ) ) );
[ Transformation( [ 1, 2, 1 ] ), Transformation( [ 1, 2, 2 ] ) ]
```

75.26 Monoid Homomorphisms

The homomorphisms between monoids are of interest as soon as there are monoids. However, no effort has been made to provide any map between monoids. Here certainly some work needs to be done.

75.27 Monoid Records and Semigroup Records

Like other domains semigroups and monoids are represented by records. While it is possible to construct such a record by hand it is preferable to have the functions `SemiGroup` (see 75.4) or `Monoid` (see 75.6) do this for you.

After such a record is created one can add record components. But you may not alter the values of components which are already present.

A semigroup or monoid record has the following category components.

`isDomain`
: is always `true` since a monoid or a semigroup is a domain.

`isSemiGroup`
: is always `true` for semigroups.

`isMonoid`
: is always `true` for monoids.

The following components are the identification components of a semigroup or monoid record.

`generators`
: is a list of generators of the monoid or the semigroup. Duplicates are allowed in this list, but in the case of a monoid none of the generators may be the identity.

`identity`
: is the indentity in the case of a monoid.

Other components which contain information about the semigroup or monoid may be present.

`size`
: is the size of the monoid or the semigroup (see "Size").

`elements`
: is the set of elements of the monoid or the semigroup (see "Elements").

Chapter 76

Binary Relations

A binary **relation** on n points is a subset $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$. It can also be seen as a multivalued map from $\{1,\ldots,n\}$ to itself, or as a directed graph with vertices $\{1,\ldots,n\}$. The number n is called the **degree** of the relation. Thus a binary relation R of degree n associates a set i^R of positive integers less than or equal to n to each number between 1 and n. This set i^R is called the set of **successors** of i under the relation R.

The degree of a binary relation may not be larger than $2^{28} - 1$ which is (currently) the highest index that can be accessed in a list.

Special cases of binary relations are transformations (see chapter 77) and permutations (see chapter "Permutations"). However, an object of one of these types must be converted into a binary relation before most of the functions of this chapter are applicable.

The product of binary relations is defined via composition of mappings, or equivalently, via concatenation of edges of directed graphs. Precisely, if R and S are two relations on $\{1,\ldots,n\}$ then their product RS is defined by saying that two points $x, y \in \{1,\ldots,n\}$ are in relation RS if and only if there is a point $z \in \{1,\ldots,n\}$ such that xRz and zSy. As multivalued map, the product RS is defined by

$$i\hat{}(RS) = (i\hat{}R)\hat{}S \quad \text{for all } i = 1,\ldots,n.$$

With respect to this multiplication the set of all binary relations of degree n forms a monoid, the **full relation monoid** of degree n.

Each relation of degree n is considered an element of the full relation monoid of degree n although it is not necessary to construct a full relation monoid before working with relations. But you can only multiply two relations if they have the same degree. You can, however, multiply a relation of degree n by a transformation or a permutation of degree n.

A binary relation is entered and displayed by giving its lists of successors as an argument to the function `Relation`. The relation $<$ on the set $\{1, 2, 3\}$, for instance, is written as follows.

```
gap> Relation( [ [ 2, 3 ], [ 3 ], [ ] ] );
Relation( [ [ 2, 3 ], [ 3 ], [  ] ] )
```

This chapter describes finite binary relations in GAP3 and the functions that deal with them. The first sections describe the representation of a binary relation (see 76.1) and how an object that represents a binary relation is constructed (see 76.2). There is a function which constructs the identity relation of degree n (see 76.4) and a function which constructs the empty relation of degree n (see 76.5). Then there are a function which tests whether an arbitrary object is a relation (see 76.3) and a function which determines the degree of a relation (see 76.6).

The next sections describe how relations are compared (see 76.7) and which operations are available for relations (see 76.8). There are functions which test certain properties of relations (see 76.9, 76.11, 76.13, 76.15, 76.17, and 76.18) and functions that construct different closures of a relation (see 76.10, 76.12, and 76.14). Moreover there are a function which computes the classes of an equivalence relation (see 76.19) and a function which determines the Hasse diagram of a partial order. Finally, two functions are describe which convert a transformation into a binary relation (see 76.21) and, if possible, a binary relation into a transformation (see 76.22).

The last section of the chapter describes monoids generated by binary relations (see 76.23).

The functions described in this chapter are in the file `"relation.g"`.

76.1 More about Relations

A binary relation seen as a directed graph on n points is completely determined by its degree and its list of edges. This information is represented in the form of a **successors list** which, for each single point $i \in \{1, \ldots, n\}$ contains the set i^R of succesors of i. Here each single set of successors is represented as a subset of $\{1, \ldots, n\}$ by boolean list (see chapter "Boolean Lists").

A relation R of degree n is represented by a record with the following category components.

`isRelation`
: is always set to `true`.

`domain`
: is always set to `Relations`.

Moreover a relation record has the identification component

`successors`
: containing a list which has as its ith entry the boolean list representing the successors of i.

A relation record *rel* can aquire the following knowledge components.

`isReflexive`
: set to `true` if *rel* represents a reflexive relation (see 76.9)

`isSymmetric`
: set to `true` if *rel* represents a symmetric relation (see 76.11)

`isTransitive`
: set to `true` if *rel* represents a transitive relation (see 76.13)

`isPreOrder`
: set to `true` if *rel* represents a preorder (see 76.16)

`isPartialOrder`
: set to `true` if *rel* represents a partial order (see 76.17)

`isEquivalence`
: set to `true` if *rel* represents an equivalence relation (see 76.18)

76.2 Relation

`Relation( list )`

`Relation` returns the binary relation defined by the list *list* of subsets of $\{1, \ldots, n\}$ where n is the length of *list*.

```
gap> Relation( [ [ 1, 2 ], [ ], [ 3, 1 ] ] );
Relation( [ [ 1, 2 ], [  ], [ 1, 3 ] ] )
```

Alternatively, *list* can be a list of boolean lists of length n, each of which is interpreted as a subset of $\{1, \ldots, n\}$ (see chapter "Boolean Lists").

```
gap> Relation( [
> [ true, true, false ],
> [ false, false, false ],
> [ true, false, true ] ] );
Relation( [ [ 1, 2 ], [  ], [ 1, 3 ] ] )
```

76.3 IsRelation

`IsRelation( obj )`

`IsRelation` returns `true` if *obj*, which may be an object of arbitrary type, is a relation and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsRelation( 1 );
false
gap> IsRelation( Relation(  [ [ 1 ], [ 2 ], [ 3 ] ] ) );
true
```

76.4 IdentityRelation

`IdentityRelation( n )`

`IdentityRelation` returns the identity relation of degree n. This is the relation `=` on the set $\{1, \ldots, n\}$.

```
gap> IdentityRelation( 5 );
Relation( [ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ] )
```

The identity relation of degree n acts as the identity in the full relation monoid of degree n.

76.5 EmptyRelation

`EmptyRelation( n )`

`EmptyRelation` returns the empty relation of degree. This is the relation $\{\} \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$.

```
gap> EmptyRelation( 5 ) ;
Relation( [ [  ], [  ], [  ], [  ], [  ] ] )
```

The empty relation of degree n acts as zero in the full relation monoid of degree n.

76.6 Degree of a Relation

`Degree(` *rel* `)`

`Degree` returns the degree of the binary relation *rel*.

```
gap> Degree( Relation( [ [ 1 ], [ 2, 3 ], [ 2, 3 ] ] ) );
3
```

The **degree** of a relation $R \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ is defined as n.

76.7 Comparisons of Relations

rel1 `=` *rel2*
rel1 `<>` *rel2*

The equality operator `=` applied to two relations *rel1* and *rel2* evaluates to `true` if the two relations are equal and to `false` otherwise. The inequality operator `<>` applied to two relations *rel1* and *rel2* evaluates to `true` if the two relations are not equal and to `false` otherwise. A relation can also be compared to any other object that is not a relation, of course they are never equal.

Two relations are considered equal if and only if their successors lists are equal as lists. In particular, they must have the same degree.

```
gap> Relation( [ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ] ) =
> IdentityRelation( 4 );
true
gap> Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) =
> Relation( [ [ ], [ 1 ], [ 1, 2 ], [ ] ] );
false
```

rel1 `<` *rel2*
rel1 `<=` *rel2*
rel1 `>` *rel2*
rel1 `>=` *rel2*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the relation *rel1* is less than, less than or equal to, greater than, or greater than or equal to the relation *rel2*, and to `false` otherwise.

Let *rel1* and *rel2* be two relations that are not equal. Then *rel1* is considered smaller than *rel2* if and only if the successors list of *rel1* is smaller than the successors list of *rel2*.

You can also compare relations with objects of other types. Here any object that is not a relation will be considered smaller than any relation.

76.8 Operations for Relations

rel1 `*` *rel2*

The operator `*` evaluates to the product of the two relations *rel1* and *rel2* if both have the same degree.

rel `*` *trans*
trans `*` *rel*

The operator `*` evaluates to the product of the relation *rel* and the transformation *trans* in the given order provided both have the same degree (see chapter 77).

rel `*` *perm*
perm `*` *rel*

The operator `*` evaluates to the product of the relation *rel* and the permutation *perm* in the given order provided both have the same degree (see chapter "Permutations").

list `*` *rel*
rel `*` *list*

The operator `*` evaluates to the list of products of the elements in *list* with the relation *rel*. That means that the value is a new list *new* such that *new*`[`*i*`] =` *list*`[`*i*`] *` *rel* or *new*`[`*i*`] =` *rel* `*` *list*`[`*i*`]`, respectively.

i `^` *rel*

The operator `^` evaluates to the set of successors *i*^*rel* of the positive integer i under the relation *rel* if i is smaller than or equal to the degree of *rel*.

set `^` *rel*

The operator `^` evaluates to the image or the set *set* under the relation *rel* which is defined as the union of the sets of successors of the elements of *set*.

rel `^ 0`

The operator `^` evaluates to the identity relation on n points if *rel* is a relation on n points (see 76.4).

rel `^` *i*

For a positive integer i the operator `^` evaluates to the i-th power of the relation *rel* which is defined in the usual way as the i-fold product of *rel* by itself.

rel `^ -1`

The operator `^` evaluates to the inverse of the relation *rel*. The inverse of a relation $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ is given by $\{(y,x) \mid (x,y) \in R\}$. Note that, in general, the product of a binary relation and its inverse is not equal to the identity relation. Neither is it in general equal to the product of the inverse and the binary relation.

76.9 IsReflexive

`IsReflexive(` *rel* `)`

`IsReflexive` returns `true` if the binary relation *rel* is reflexive and `false` otherwise.

```
gap> IsReflexive( Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) );
false
gap> IsReflexive( Relation( [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
true
```

A relation $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ is **reflexive** if $(i,i) \in R$ for all $i = 1,\ldots,n$. (See also 76.10.)

76.10 ReflexiveClosure

`ReflexiveClosure(` *rel* `)`

`ReflexiveClosure` returns the reflexive closure of the relation *rel*, i.e., the relation $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ that consists of all pairs in *rel* and the pairs $(1,1)$, ..., (n,n), where n is the degree of *rel*.

```
gap> ReflexiveClosure( Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) );
Relation( [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] )
```

By construction, the reflexive closure of a relation is reflexive (see 76.9).

76.11 IsSymmetric

`IsSymmetric(` *rel* `)`

`IsSymmetric` returns `true` if the binary relation *rel* is symnmetric and `false` otherwise.

```
gap> IsSymmetric( Relation( [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
false
gap> IsSymmetric( Relation( [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) );
true
```

A relation $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ is **symmetric** if $(y,x) \in R$ for all $(x,y) \in R$. (See also 76.12.)

76.12 SymmetricClosure

`SymmetricClosure(` *rel* `)`

`SymmetricClosure` returns the symmetric closure of the binary relation *rel*.

```
gap> SymmetricClosure( Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) );
Relation( [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] )
```

By construction, the symmetric closure of a relation is symmetric (see 76.11).

76.13 IsTransitiveRel

`IsTransitiveRel(` *rel* `)`

`IsTransitiveRel` returns `true` if the binary relation *rel* is transitive and `false` otherwise.

```
gap> IsTransitiveRel( Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) );
true
```

```
gap> IsTransitiveRel( Relation( [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) );
false
```

A relation $R \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ is **transitive** if $(x, z) \in R$ whenever $(x, y) \in R$ and $(y, z) \in R$ for some $y \in \{1, \ldots, n\}$. (See also 76.14.)

76.14 TransitiveClosure

`TransitiveClosure(` *rel* `)`

`TransitiveClosure` returns the transitive closure of the binary relation *rel*.

```
gap> TransitiveClosure( Relation( [ [ ], [ 1 ], [ 2 ], [ 3 ] ] ) );
Relation( [ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] )
```

By construction, the transitive closure of a relation is transitive (see 76.13).

76.15 IsAntisymmetric

`IsAntisymmetric(` *rel* `)`

`IsAntisymmetric` returns `true` if the binary relation *rel* is antisymmetric and `false` otherwise.

```
gap> IsAntisymmetric( Relation( [ [  ], [ 1 ], [ 1, 2 ] ] ) );
true
gap> IsAntisymmetric( Relation( [ [ 2, 3 ], [ 1, 3 ], [ 1, 2 ] ] ) ) ;
false
```

A relation $R \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ is **antisymmetric** if $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$.

76.16 IsPreOrder

`IsPreOrder(` *rel* `)`

`IsPreOrder` returns `true` if the binary relation *rel* is a preoder and `false` otherwise.

```
gap> IsPreOrder( Relation( [ [  ], [ 1 ], [ 1, 2 ] ] ) );
false
gap> IsPreOrder( Relation( [ [ 1, 2 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
true
```

A relation *rel* is called a **preorder** if *rel* is reflexive and transitive.

76.17 IsPartialOrder

`IsPartialOrder(` *rel* `)`

`IsPartialOrder` returns `true` if the binary relation *rel* is a partial order and `false` otherwise.

```
gap> IsPartialOrder( Relation( [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
true
gap> IsPartialOrder( Relation( [ [ 1, 2 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
false
```

A relation *rel* is called a **partial order** if *rel* is reflexive, transitive and antisymmetric, i.e., if *rel* is an antisymmetric preorder (see 76.16).

76.18 IsEquivalence

`IsEquivalence(` *rel* `)`

`IsEquivalence` returns `true` if the binary relation *rel* is an equivalence relation and `false` otherwise.

```
gap> IsEquivalence( Relation( [ [ ], [ 1 ], [ 1, 2 ] ] ) );
false
gap> IsEquivalence( Relation( [ [ 1 ], [ 2, 3 ], [ 2, 3 ] ] ) );
true
```

A relation *rel* is an **equivalence relation** if *rel* is reflexive, symmetric, and transitive, i.e., if *rel* is a symmetric preorder (see 76.16). (See also 76.19.)

76.19 EquivalenceClasses

`EquivalenceClasses(` *rel* `)`

returns the list of equivalence classes of the equivalence relation *rel*. It will signal an error if *rel* is not an equivalence relation (see 76.18).

```
gap> EquivalenceClasses( Relation( [ [ 1 ], [ 2, 3 ], [ 2, 3 ] ] ) );
[ [ 1 ], [ 2, 3 ] ]
```

76.20 HasseDiagram

`HasseDiagram(` *rel* `)`

`HasseDiagram` returns the Hasse diagram of the binary relation *rel* if this is a partial order. It will signal an error if *rel* is not a partial order (see 76.17).

```
gap> HasseDiagram( Relation( [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] ) );
Relation( [ [  ], [ 1 ], [ 2 ] ] )
```

The **Hasse diagram** of a partial order $R \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ is the smallest relation $H \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ such that R is the reflexive and transitive closure of H.

76.21 RelTrans

`RelTrans(` *trans* `)`

`RelTrans` returns the binary relation defined by the transformation *trans* (see chapter 77).

```
gap> RelTrans( Transformation( [ 3, 3, 2, 1, 4 ] ) );
Relation( [ [ 3 ], [ 3 ], [ 2 ], [ 1 ], [ 4 ] ] )
```

76.22 TransRel

`TransRel(` *rel* `)`

`TransRel` returns the transformation defined by the binary relation *rel* (see chapter 77). This can only be applied if every set of successors of *rel* has size 1. Otherwise an error is signaled.

```
gap> TransRel( Relation( [ [ 3 ], [ 3 ], [ 2 ], [ 1 ], [ 4 ] ] ) );
Transformation( [ 3, 3, 2, 1, 4 ] )
```

76.23 Monoids of Relations

There are no special functions provided for monoids generated by binary relations. The action of such a monoid on sets, however, provides a way to convert a relation monoid into a transformation monoid (see chapter 79). This monoid can then be used to investigate the structure of the original relation monoid.

```
gap> a:= Relation( [ [  ], [  ], [ 1, 3, 4 ], [  ], [ 2, 5 ] ] );;
gap> b:= Relation( [ [  ], [ 2 ], [ 4 ], [ 1, 2, 3 ], [ 1 ] ] );;
gap> M:= Monoid( a, b );
Monoid( [ Relation( [ [  ], [  ], [ 1, 3, 4 ], [  ], [ 2, 5 ] ] ),
  Relation( [ [  ], [ 2 ], [ 4 ], [ 1, 2, 3 ], [ 1 ] ] ) ] )
gap> # transform points into singleton sets.
gap> one:= List( [ 1 .. 5 ], x-> [ x ] );
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
gap> # determine all reachable sets.
gap> sets:= Union( Orbits( M, one ) );
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 3 ], [ 1, 2, 3, 4 ], [ 1, 3, 4 ],
  [ 2 ], [ 2, 4 ], [ 2, 5 ], [ 3 ], [ 4 ], [ 5 ] ]
gap> # construct isomorphic transformation monoid.
gap> act:= Action( M, sets );
Monoid( [ Transformation( [ 1, 1, 1, 6, 6, 6, 1, 1, 9, 6, 1, 9 ] ),
  Transformation( [ 1, 1, 7, 8, 5, 5, 7, 4, 3, 11, 4, 2 ] ) ] )
gap> Size(act);
11
```

Chapter 77

Transformations

A transformation of degree n is a map from the set $\{1, ..., n\}$ into itself. Thus a transformation α of degree n associates a positive integer i^α less than or equal to n to each number i between 1 and n.

The degree of a transformation may not be larger than $2^{28} - 1$ which is (currently) the highest index that can be accessed in a list.

Special cases of transformations are permutations (see chapter "Permutations"). However, a permutation must be converted to a transformation before most of the functions in this chapter are applicable.

The product of transformations is defined via composition of maps. Here transformations are multiplied in such a way that they act from the right on the set $\{1, ..., n\}$. That is, the product of the transformations α and β of degree n is defined by

$$i\hat{}(\alpha\beta) = (i\hat{}\alpha)\hat{}\beta \quad \text{for all } i = 1, ..., n.$$

With respect to this multiplication the set of all transformations of degree n forms a monoid: the full transformation monoid of degree n (see chapter 78).

Each transformation of degree n is considered an element of the full transformation monoid of degree n although it is not necessary to construct a full transformation monoid before working with transformations. But you can only multiply two transformations if they have the same degree. You can, however, multiply a transformation of degree n by a permutation of degree n.

Transformations are entered and displayed by giving their lists of images as an argument to the function `Transformation`.

```
gap> Transformation( [ 3, 3, 4, 2, 5 ] );
Transformation( [ 3, 3, 4, 2, 5 ] )
gap> Transformation( [ 3, 3, 2 ] ) * Transformation( [ 1, 2, 1 ] );
Transformation( [ 1, 1, 2 ] )
```

This chapter describes functions that deal with transformations. The first sections describe the representation of a transformation in GAP3 (see 77.1) and how a transformation is constructed as a GAP3 object (see 77.2). The next sections describe the comparisons and the operations which are available for transformations (see 77.4 and 77.5). There are a function to test whether an arbitrary object is a transformation (see 77.6) and a function to construct the identity transformation of a given degree (see 77.3). Then there are functions that compute attributes of transformations (see 77.7, 77.8, 77.9, and 77.10). Finally, there are a function that converts a permutation to a transformation (see 77.12) and a function that, if possible converts a transformation to a permutation (see 77.13).

The functions described here are in the file `"transfor.g"`.

77.1 More about Transformations

A transformation α on n points is completely defined by its list of images. It is stored as a record with the following category components.

`isTransformation`
: is always set to `true`.

`domain`
: is always set to `Transformations`.

Moreover it has the identification component

`images`
: containing the list of images in such a way that $i\hat{}\alpha = \alpha$.`images`[`i`] for all $i \leq n$.

The multiplication of these transformations can be efficiently implemented by using the sublist operator { }. The product `r * l` of two transformations l and r can be computed as `Transformation( r.images{ l.images } )`. Note that the order has been chosen to have transformations act from the right on their domain.

77.2 Transformation

`Transformation( lst )`

`Transformation` returns the transformation defined by the list *lst* of images. Each entry in *lst* must be a positive integer not exceeding the length of *lst*.

```
gap> Transformation( [ 1, 4, 4, 2 ] );
Transformation( [ 1, 4, 4, 2 ] )
```

77.3 IdentityTransformation

`IdentityTransformation( n )`

`IdentityTransformation` returns, for any positive n, the identity transformation of degree n.

```
gap> IdentityTransformation( 4 );
Transformation( [ 1 .. 4 ] )
```

The identity transformation of degree n acts as the identity in the full transformation monoid of degree n (see 78.3).

77.4 Comparisons of Transformations

tr1 `=` *tr2*
tr1 `<>` *tr2*

The equality operator `=` applied to two transformations *tr1* and *tr2* evaluates to `true` if the two transformations are equal and to `false` otherwise. The inequality operator `<>` applied to two transformations *tr1* and *tr2* evaluates to `true` if the two transformations are not equal and to `false` otherwise. A transformation can also be compared to any other object that is not a transformation, of course they are never equal.

Two transformations are considered equal if and only if their image lists are equal as lists. In particular, equal transformations must have the same degree.

```
gap> Transformation( [ 1, 2, 3, 4 ] ) = IdentityTransformation( 4 );
true
gap> Transformation( [ 1, 4, 4, 2 ] ) =
> Transformation( [ 1, 4, 4, 2, 5 ] );
false
```

tr1 `<` *tr2*
tr1 `<=` *tr2*
tr1 `>` *tr2*
tr1 `>=` *tr2*

The operators `<`, `<=`, `>`, and `>=` evaluate to `true` if the transformation *tr1* is less than, less than or equal to, greater than, or greater than or equal to the transformation *tr2*, and to `false` otherwise.

Let *tr1* and *tr2* be two transformations that are not equal. Then *tr1* is considered smaller than *tr2* if and only if the list of images of *tr1* is (lexicographically) smaller than the list of images of *tr2*. Note that this way the smallest transformation of degree n is the transformation that maps every point to 1.

You can also compare transformations with objects of other types. Here any object that is not a transformation will be considered smaller than any transformation.

77.5 Operations for Transformations

tr1 `*` *tr2*

The operator `*` evaluates to the product of the two transformations *tr1* and *tr2*.

tr `*` *perm*
perm `*` *tr*

The operator `*` evaluates to the product of the transformation *tr* and the permutation *perm* in the given order if the degree of *perm* is less than or equal to the degree of *tr*.

list `*` *tr*
tr `*` *list*

The operator `*` evaluates to the list of products of the elements in *list* with the transformation *tr*. That means that the value is a new list *new* such that *new*`[`*i*`] = `*list*`[`*i*`] * `*tr* or *new*`[`*i*`] = `*tr*` * `*list*`[`*i*`]`, respectively.

i `^` *tr*

The operator `^` evaluates to the image *i*^*tr* of the positive integer *i* under the transformation *tr* if *i* is less than the degree of *tr*.

tr `^ 0`

The operator `^` evaluates to the identity transformation on n points if *tr* is a transformation on n points (see 77.3).

tr `^` *i*

For a positive integer *i* the operator `^` evaluates to the *i*-th power of the transformation *tr*.

tr `^ -1`

The operator `^` evaluates to the inverse mapping of the transformation *tr* which is represented as a binary relation (see chapter 76).

77.6 IsTransformation

`IsTransformation(` *obj* `)`

`IsTransformation` returns `true` if *obj*, which may be an object of arbitrary type, is a transformation and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsTransformation( Transformation( [ 2, 1 ] ) );
true
gap> IsTransformation( 1 );
false
```

77.7 Degree of a Transformation

`Degree(` *trans* `)`

`Degree` returns the degree of the transformation *trans*.

```
gap> Degree( Transformation( [ 3, 3, 4, 2, 5 ] ) );
5
```

The **degree** of a transformation is the number of points it is defined upon. It can therefore be read off as the length of the list of images of the transformation.

77.8 Rank of a Transformation

`Rank(` *trans* `)`

`Rank` returns the rank of the transformation *trans*.

```
gap> Rank( Transformation( [ 3, 3, 4, 2, 5 ] ) );
5
```

The **rank** of a transformation is the number of points in its image. It can therefore be determined as the size of the set of images of the transformation.

77.9 Image of a Transformation

`Image(` *trans* `)`

`Image` returns the image of the transformation *trans*.

```
gap> Image( Transformation( [ 3, 3, 4, 2, 5 ] ) );
[ 2, 3, 4, 5 ]
```

The **image** of a transformation is the set of its images. For a transformation of degree n this is always a subset of the set $\{1, ..., n\}$.

77.10 Kernel of a Transformation

`Kernel(` *trans* `)`

`Kernel` returns the kernel of the transformation *trans*.

```
gap> Kernel( Transformation( [ 3, 3, 4, 2, 5 ] ) );
[ [ 1, 2 ], [ 3 ], [ 4 ], [ 5 ] ]
```

The **kernel** of a transformation is the set of its nonempty preimages. For a transformation of degree n this is always a partition of the set $\{1, ..., n\}$.

77.11 PermLeftQuoTrans

`PermLeftQuoTrans(` *tr1* `,` *tr2* `)`

Given transformations *tr1* and *tr2* with equal kernel and image, the permutation induced by *tr1*`^-1 *` *tr2* on the set `Image(` *tr1* `)` is computed.

```
gap> a:= Transformation( [ 8, 7, 5, 3, 1, 3, 8, 8 ] );;
gap> Image(a);  Kernel(a);
[ 1, 3, 5, 7, 8 ]
[ [ 1, 7, 8 ], [ 2 ], [ 3 ], [ 4, 6 ], [ 5 ] ]
gap> b:= Transformation( [ 1, 3, 8, 7, 5, 7, 1, 1 ] );;
gap> Image(b) = Image(a);  Kernel(b) = Kernel(a);
true
true
gap> PermLeftQuoTrans(a, b);
(1,5,8)(3,7)
```

77.12 TransPerm

`TransPerm(` n`,` *perm* `)`

`TransPerm` returns the bijective transformation of degree n that acts on the set $\{1, ..., n\}$ in the same way as the permutation *perm* does.

```
gap> TransPerm( 4, (1,2,3) );
Transformation( [ 2, 3, 1, 4 ] )
```

77.13 PermTrans

`PermTrans(` *trans* `)`

`PermTrans` returns the permutation defined by the transformation *trans*. If *trans* is not bijective, an error is signaled by `PermList` (see "PermList").

```
gap> PermTrans( Transformation( [ 2, 3, 1, 4 ] ) );
(1,2,3)
```

Chapter 78

Transformation Monoids

A transformation monoid is a monoid of transformations of n points (see chapter 77). These monoids occur for example in the theory of finite state automata and as the results of enumerations of finitely presented monoids. In the theory of semigroups and monoids they ply to some extend the role that is taken by permutation groups in group theory. In fact, there are close relations between permutation groups and transformation monoids. One of these relations is manifested by the Schützenberger group of an element of a transformation monoid, which is represented as a permutation group rather than a group of transformations. Another relation which is used by most functions that deal with transformation monoids is the fact that a transformation monoid can be efficiently described in terms of several permutation groups (for details see [LPRR97] and [LPRR]).

This chapter describes the functions that deal with transformation monoids.

The chapter starts with the description of the function that tests whether or not a given object is a transformation monoid (see 78.1). Then there is the function that determines the degree of a transformation monoid (see 78.2).

There are a function to construct the full transformation monoid of degree n (see 78.3) and a function to construct the monoid of all partial transformations of degree n (see 78.4).

Then there are a function that determines all images of a transformation monoid (see 78.5) and a function that determines all kernels of a transformation monoid (see 78.6).

Because each transformation monoid is a domain all set theoretic functions can be applied to it (see chapter "Domains" and 78.7). Also because a transformation monoid is after all a monoid all monoid functions can be applied to it (see chapter 75 and 78.8).

Next the functions that determine Green classes in transformation monoids are described (see 78.10, 78.11, 78.12, and 78.13).

Finally, there is a section about how a transformation monoid is displayed (see 78.14). The last section in this chapter describes how transformation monoids are represented as records in GAP3 (see 78.15).

The functions described here are in the file `"monotran.g"`.

78.1 IsTransMonoid

`IsTransMonoid( obj )`

`IsTransMonoid` returns `true` if the object *obj*, which may be an object of an arbitrary type, is a transformation monoid, and `false` otherwise. It will signal an error if *obj* is an unbound variable.

```
gap> IsTransMonoid( Monoid( [ Transformation( [ 1, 2, 1 ] ) ] ) );
true
gap> IsTransMonoid( Group( (1,2), (1,2,3,4) ) );
false
gap> IsTransMonoid( [ 1, 2, 1 ] );
false
```

78.2 Degree of a Transformation Monoid

`Degree( M )`

`Degree` returns the degree of a transformation monoid M, which is the common degree of all the generators of M.

```
gap> Degree( Monoid( Transformation( [ 1, 2, 1 ] ) ) );
3
```

The **degree** of a transformation monoid is the number of points it acts upon.

78.3 FullTransMonoid

`FullTransMonoid( n )`

`FullTransMonoid` returns the full transformation monoid of degree n.

```
gap> M:= FullTransMonoid( 8 );
Monoid( [ Transformation( [ 2, 1, 3, 4, 5, 6, 7, 8 ] ),
  Transformation( [ 8, 1, 2, 3, 4, 5, 6, 7 ] ),
  Transformation( [ 2, 2, 3, 4, 5, 6, 7, 8 ] ) ] )
gap> Size( M );
16777216
```

The **full transformation monoid** of degree n is the monoid of all transformations of degree n.

78.4 PartialTransMonoid

`PartialTransMonoid( n )`

`PartialTransMonoid` returns the monoid of partial transformations of degree n.

```
gap> M:= PartialTransMonoid( 8 );
Monoid( [ Transformation( [ 2, 1, 3, 4, 5, 6, 7, 8, 9 ] ),
  Transformation( [ 8, 1, 2, 3, 4, 5, 6, 7, 9 ] ),
  Transformation( [ 9, 2, 3, 4, 5, 6, 7, 8, 9 ] ),
  Transformation( [ 2, 2, 3, 4, 5, 6, 7, 8, 9 ] ) ] )
```

```
gap> Size( M );
1000000000
```

A **partial transformation** of degree n is a mapping from $\{1,...,n\}$ to itself where every point $i \in \{1,...,n\}$ has at most one image. Here the undefined point is represented by $n+1$.

78.5 ImagesTransMonoid

`ImagesTransMonoid(` *M* `)`

`ImagesTransMonoid` returns the set of images of all elements of the transformation monoid *M* (see 77.9).

```
gap> M:= Monoid( Transformation( [ 1, 4, 4, 2 ] ),
>       Transformation( [ 2, 4, 4, 4 ] ) );;
gap> ImagesTransMonoid(M);
[ [ 1, 2, 3, 4 ], [ 1, 2, 4 ], [ 2 ], [ 2, 4 ], [ 4 ] ]
```

78.6 KernelsTransMonoid

`KernelsTransMonoid(` *M* `)`

`KernelsTransMonoid` returns the set of kernels of all elements of the transformation monoid *M* (see 77.10).

```
gap> M:= Monoid( [ Transformation( [ 1, 4, 4, 2 ] ),
>       Transformation( [ 2, 4, 4, 4 ] ) ] );;
gap> KernelsTransMonoid(M);
[ [ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ], [ [ 1 ], [ 2, 3 ], [ 4 ] ],
  [ [ 1 ], [ 2, 3, 4 ] ], [ [ 1, 2, 3, 4 ] ] ]
```

78.7 Set Functions for Transformation Monoids

All set theoretic functions described in chapter "Domains"are also applicable to transformation monoids. This section describes which functions are implemented specially for transformation monoids. Functions not mentioned here are handled by the default methods described in the respective sections of chapter "Domains".

`Size(` *M* `)`

`Size` calls `RClasses` (see 75.12), if necessary, and returns the sum of the sizes of all R classes of M.

```
gap> Size( Monoid( Transformation( [ 1, 2, 1 ] ) ) );
2
```

`Elements(` *M* `)`

`Elements` calls `RClasses` (see 75.12) if necessary, and returns the union of the elements of all R classes of M.

```
gap> Elements( Monoid( Transformation( [ 1, 2, 1 ] ) ) );
[ Transformation( [ 1, 2, 1 ] ), Transformation( [ 1 .. 3 ] ) ]
```

obj `in` *M*

The membership test of elements of transformation monoids first checks whether *obj* is a transformation in the first place (see 77.6) and if so whether the degree of *obj* (see 77.7) coincides with the degree of *M* (see 78.2). Then the image and the kernel of *obj* is used to locate the possible *R* class of *obj* in *M* and the membership test is delegated to that *R* class (see 75.22).

```
gap> M:= Monoid( Transformation( [ 1, 2, 1 ] ) );;
 gap> (1,2,3) in M;
 false
 gap> Transformation( [1, 2, 1 ] ) in M;
 true
 gap> Transformation( [ 1, 2, 1, 4 ] ) in M;
 false
```

78.8 Monoid Functions for Transformation Monoids

All functions described in chapter 75 can be applied to transformation monoids.

Green classes are special subsets of a transformation monoid. In particular, they are domains so all set theoretic functions for domains (see chapter "Domains") can be applied to Green classes. This is described in 75.22. Single Green classes of a transformation monoid are constructed by the functions `RClass` (see 75.10 and 78.11), `LClass` (see 75.13 and 78.12), `DClass` (see 75.16 and 78.13), and `HClass` (see 75.19 and 78.10). The list of all Green classes of a given type in a transformation monoid is constructed by the functions `RClasses` (see 75.12), `LClasses` (see 75.15), `DClasses` (see 75.18), and `HClasses` (see 75.21).

78.9 SchutzenbergerGroup for Transformation Monoids

`SchutzenbergerGroup(` *M*`,` *s* `)`
`SchutzenbergerGroup(` *class* `)`

`SchutzenbergerGroup` returns the Schützenberger group of the element *s* in the transformation monoid *M* as a permutation group on the image of *s*.

In the second form `SchutzenbergerGroup` returns the Schützenberger group of the Green class *class* of a transformation monoid, where *class* is either an H class, an R class or a D class. The Schützenberger group of an H class *class* is the same as the Schützenberger group of *class*. The Schützenberger group of an R class *class* is the generalised right Schützenberger group of the representative of *class* and the Schützenberger group of an L class *class* is the generalised left Schützenberger group of the representative of *class*. Note that the Schützenberger of an R class is only unique up to conjugation.

78.10 H Classes for Transformation Monoids

In addition to the usual components of an H class record, the record representing the H class *hClass* of *s* in a transformation monoid can have the following components. They are created by the function `SchutzenbergerGroup` (see 75.24) which is called whenever the size, the list of elements of *hClass*, or a membership test in *hClass* is asked for.

`schutzenbergerGroup`
: set to the Schützenberger group of *hClass* as a permutation group on the set of images of *hClass*`.representative` (see 78.9).

`R`
: the R class of *hClass*`.representative.`

`L`
: the L class of *hClass*`.representative.`

The following functions have a special implementation in terms of these components.

`Size(` *hClass* `)`

returns the size of the H class *hClass*. This function calls `SchutzenbergerGroup` and determines the size of *hClass* as the size of the resulting group.

`Elements(` *hClass* `)`

returns the set of elements of the H class *hClass*. This function calls `SchutzenbergerGroup` and determines the set of elements of *hClass* as the set of elements of the resulting group multiplied by the representative of *hClass*.

x `in` *hClass*

returns `true` if *x* is an element of the H class *hClass* and `false` otherwise. This function calls `SchutzenbergerGroup` and tests whether the quotient of the representative of *hClass* and *x* (see 77.11) is in the resulting group.

78.11 R Classes for Transformation Monoids

In addition to the usual components of an R class record, the record representing the R class *rClass* of *s* in a transformation monoid can have the following components. They are created by the function `SchutzenbergerGroup` (see 75.24) which is called whenever the size, the list of elements of *rClass*, or a membership test in *rClass* is asked for.

`schutzenbergerGroup`
: set to the Schützenberger group of *rClass* as a permutation group on the set of images of *rClass*`.representative` (see 78.9).

`images`
: is the list of different image sets occurring in the R class *rClass*. The first entry in this list is the image set of *rClass*`.representative`.

`rMults`
: is a list of permutations such that the product of the representative of *rClass* and the inverse of the *i*th entry in the list yields an element of *rClass* whose image set is the *i*th entry in the list *rClass*`.images`.

The following functions have a special implementation in terms of these components.

`Size(` *rClass* `)`

returns the size of the R class *rClass*. This function calls `SchutzenbergerGroup` and determines the size of *rClass* as the size of the resulting group times the length of the list *rClass*`.images`.

`Elements(` *rClass* `)`

returns the set of elements of the R class *rClass*. This function calls `SchutzenbergerGroup` and determines the set of elements of *rClass* as the set of elements of the resulting group multiplied by the representative of *rClass* and each single permutation in the list *rClass*`.rMults`.

x `in` *rClass*

returns `true` if *x* is an element of the R class *rClass* and `false` otherwise. This function calls `SchutzenbergerGroup` and tests whether the quotient of the representative of *rClass* and *x* `*` *rClass*`.rMults[`*i*`]` (see 77.11) is in the resulting group where *i* is the position of the image set of *x* in *rClass*`.images`.

`HClasses(` *rClass* `)`

returns the list of H classes contained in the R class *rClass*.

78.12 L Classes for Transformation Monoids

In addition to the usual components of an L class record, the record representing the L class *lClass* of *s* in a transformation monoid can have the following components. They are created by the function `SchutzenbergerGroup` (see 75.24) which is called whenever the size, the list of elements of *lClass*, or a membership test in *lClass* is asked for.

`schutzenbergerGroup`
: set to the Schützenberger group of *lClass* as a permutation group on the set of images of *lClass*`.representative` (see 78.9).

`kernels`
: is the list of different kernels occurring in the L class *lClass*. The first entry in this list is the kernel of *rClass*`.representative`.

`lMults`
: is a list of binary relations such that the product of the inverse of the *i*th entry in the list and the representative of *rClass* yields an element of *rClass* whose kernel is the *i*th entry in the list *rClass*`.kernels`.

The following functions have a special implementation in terms of these components.

`Size(` *lClass* `)`

returns the size of the L class *lClass*. This function calls `SchutzenbergerGroup` and determines the size of *lClass* as the size of the resulting group times the length of the list *lClass*`.kernels`.

`Elements(` *lClass* `)`

returns the set of elements of the L class *lClass*. This function calls `SchutzenbergerGroup` and determines the set of elements of *lClass* as the set of elements of the resulting group premultiplied by the representative of *lClass* and each single binary relation in the list *lClass*`.lMults`.

x `in` *lClass*

returns `true` if *x* is an element of the L class *lClass* and `false` otherwise. This function calls `SchutzenbergerGroup` and tests whether the quotient of the representative of *lClass* and *lClass*`.lMults[`*i*`] * `*x* (see 77.11) is in the resulting group where *i* is the position of the kernel of *x* in *lClass*`.kernels`.

`HClasses(` *lClass* `)`

returns the list of H classes contained in the L class *lClass*.

78.13 D Classes for Transformation Monoids

In addition to the usual components of a D class record, the record representing the D class *dClass* of *s* in a transformation monoid can have the following components. They are created by the function `SchutzenbergerGroup` (see 75.24) which is called whenever the size, the list of elements of *dClass*, or a membership test in *dClass* is asked for.

`schutzenbergerGroup`
: set to the Schützenberger group of *dClass* as a permutation group on the set of images of *dClass*`.representative` (see 78.9).

`H`
: set to the H class of *dClass*`.representative`.

`L`
: set to the L class of *dClass*`.representative`.

`R`
: set to the R class of *dClass*`.representative`.

`rCosets`
: contains a list of (right) coset representatives of the Schützenberger group of *dClass* in the Schützenberger group of the R class *dClass*`.R`.

The following functions have a special implementation in terms of these components.

`Size(` *dClass* `)`

returns the size of the D class *dClass*. This function calls `SchutzenbergerGroup` and determines the size of *dClass* in terms of the sizes of the resulting group and the Schützenberger groups of the R class *dClass*`.R` and the L class *dClass*`.L`.

`Elements(` *dClass* `)`

returns the set of elements of the D class *dClass*. This function calls `SchutzenbergerGroup` and determines the set of elements of *dClass* as the union of cosets of the Schützenberger group of the L class *dClass*.`L` determined through the multipliers *dClass*.`rCosets` and *dClass*.`R.rMults`.

x `in` *dClass*

returns `true` if x is an element of the D class *dClass* and `false` otherwise. This function calls `SchutzenbergerGroup` and tests whether the quotient of the representative of *dClass* and a suitable translate of x can be found in one of the cosets of the Schützenberger group of the L class *dClass*.`L` determined by the list *dClass*.`rCosets`.

`HClasses(` *dClass* `)`

returns the list of H classes contained in the D class *dClass*.

`LClasses(` *dClass* `)`

returns the list of L classes contained in the D class *dClass*.

`RClasses(` *dClass* `)`

returns the list of R classes contained in the D class *dClass*.

78.14 Display a Transformation Monoid

`Display(` M `)`

`Display` displays the Green class structure of the transformation monoid M. Each D class is displayed as a single item on a line according to its rank. A D class displayed as

`[`a`.`b`.`d`]`

is a regular D class with a Schützenberger group of size a, consisting of b L classes, or d R classes. A D class displayed as

`{`a`.`b`x`c`.`d`x`e`}`

is a nonregular D class with a Schützenberger group of size a, consisting of $b \times c$ L classes (of which c have the same kernel), or $d \times e$ R classes (of which e have the same image).

```
gap> M:= Monoid( Transformation( [ 7, 7, 1, 1, 5, 6, 5, 5 ] ),
> Transformation( [ 3, 8, 3, 7, 4, 6, 4, 5 ] ) );;
gap> Size( M );
27
gap> Display( M );
Rank 8: [1.1.1]
Rank 6: {1.1x1.1x1}
Rank 5: {1.1x1.1x1}
Rank 4: {1.1x1.1x1} [2.1.1]
Rank 3: {1.1x1.4x1} [1.3.4]
Rank 2: [1.5.1]
```

78.15 Transformation Monoid Records

In addition to the usual components of a monoid record (see 75.27) the record representing a transformation monoid M has a component

`isTransMonoid`
: which is always set to `true`.

Moreover, such a record will (after a while) acquire the following components.

`orbitClasses`
: a list of R classes of M such that every orbit of image sets is represented exactly once.

`images`
: the list of lists where `images[` k `]` is the list of all different image sets of size k of the elements of M.

`imagePos`
: stores the relation between `orbitClasses` and `images`. The image set `images[`k`][`l`]` occurs in the orbit of the R class with index `imagePos[`k`][`l`]` in the list `orbitClasses`.

`rClassReps`
: a list of lists, where `rClassReps[`l`]` contains the complete list of representatives of the R classes with the same image orbit as the R class `orbitClasses[`l`]`.

`lTrans`
: a list of lists, where `lTrans[`l`][`k`]` is a transformation α such that `lTrans[`l`][`k`] * rClassReps[`l`][`k`]` is an element of the R class `orbitClasses[`l`]`.

`kernels`
: a list of lists, where `kernels[`l`][`k`]` is the common kernel of the elements in the R class of `rClassReps[`l`][`k`]`.

Chapter 79

Actions of Monoids

A very natural concept and important tool in the study of monoids is the idea of having monoids acting on certain (finite) sets. This provides a way to turn any monoid into a (finite) transformation monoid.

Let M be a monoid and D a set. An **action** of M on D is a map

$$(d, m) \mapsto d\hat{\ } m : D \times M \to D$$

such that $d\hat{\ }1 = d$ for all $d \in D$ (and the identity 1 of M), and that $(d\hat{\ }m_1)\hat{\ }m_2 = d\hat{\ }(m_1 m_2)$ for all $d \in D$ and all $m_1, m_2 \in M$. In this situation we also say that M **acts on** D, or, that D is an M**-set**.

In contrast to group operations (see chapter "Operations of Groups"), a monoid action often comes with a natural grading that can be used to carry out certain calculations more efficiently. To be precise we work with the following concept. Let M be a monoid acting on the set D. A **grading** is a map $g : D \to \{1, 2, 3, \dots\}$ such that $g(d) \geq g(d\hat{\ }m)$ for all $d \in D$ and all $m \in M$. The trivial grading is the map given by $g(d) = 1$ for all $d \in D$.

In GAP3 a monoid usually acts on a set via the caret operator `^`. This action is refered to as the **canonical action**. It is, however, possible to define other actions (see 79.1).

This chapter describes functions that deal with finite actions of monoids. There are functions for different types of orbit calculations depending on whether a grading is used and if so how (see 79.2, 79.5, 79.4). Then there are functions which construct the transformation monoid corresponding to a particular action of a monoid M on a set D (see 79.6 and 79.7) where, if necessary, an additional point 0 is added to the domain D.

The functions described here are in the file `"action.g"`.

79.1 Other Actions

Most of the operations for groups can be applied as monoid actions (see "Other Operations"). In addition to these there are a couple of actions which are particular to monoids.

The functions described in this chapter generally deal with the action of monoid elements defined by the canonical action that is denoted with the caret (`^`) in GAP3. However, they also allow you to specify other actions. Such actions are specified by functions, which are accepted as optional argument by all the functions described here.

An action function must accept two arguments. The first argument will be the point and the second will be the monoid element. The function must return the image of the point under the monoid element in the action that it specifies.

As an example, the function OnPairs that specifies the action on pairs could be defined as follows

```
OnPairs := function ( pair, m )
    return [ pair[1] ^ m, pair[2] ^ m ];
end;
```

The following monoid actions are predefined.

`OnPoints`
: specifies the canonical default action. Passing this function is equivalent to specifying no action. This function exists because there are places where the action in not an option.

`OnPairs`
: specifies the componentwise action of monoid elements on pairs of points, which are represented by lists of length 2.

`OnTuples`
: specifies the componentwise action of monoid elements on tuples of points, which are represented by lists. `OnPairs` is the special case of `OnTuples` for tuples with two elements.

`OnSets`
: specifies the action of monoid elements on sets of points, which are represented by sorted lists of points without duplicates (see chapter "Sets").

`OnRight`
: specifies that monoid elements act by multiplication from the right.

`OnLeftAntiAction`
: specifies that monoid elements act by multiplication from the left.

`OnLClasses`
: specifies that monoid elements act by multiplication from the right on L classes (see 75.15).

`OnRClassesAntiAction`
: specifies that monoid elements act by multiplication from the left on R classes (see 75.12).

Note that it is your responsibility to make sure that the elements of the domain D on which you are acting are already in normal form. The reason is that all functions will compare points using the `=` operation. For example, if you are acting on sets with `OnSets`, you will get an error message it not all elements of the domain are sets.

```
gap> OnSets(Transformation( [ 1, 2 ] ), [ 2, 1 ] );
Error, OnSets: <tuple> must be a set
```

79.2 Orbit for Monoids

`Orbit(` *M* `,` *d* `)`
`Orbit(` *M* `,` *d* `,` *action* `)`

The **orbit** of a point d under the action of a monoid M is the set $\{d\hat{\ }m \mid m \in M\}$ of all points that are images of d under some element $m \in M$.

In the first form `Orbit` computes the orbit of point d under the monoid M with respect to the canonical action `OnPoints`.

In the second form `Orbit` computes the orbit of point d under the monoid M with respect to the action *action*.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 1 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] ) ;;
gap> Orbit(M, 1);
[ 1, 5, 2, 4 ]
gap> Orbit(M, 3, OnPoints);
[ 3, 4, 5, 2, 1 ]
gap> Orbit(M, [1,2], OnSets);
[ [ 1, 2 ], [ 4, 5 ], [ 2, 5 ], [ 1, 4 ], [ 1, 5 ], [ 2, 4 ] ]
gap> Orbit(M, [1,2], OnPairs);
[ [ 1, 2 ], [ 5, 4 ], [ 2, 5 ], [ 1, 4 ], [ 4, 1 ], [ 5, 1 ], [ 5, 2 ],
  [ 2, 4 ], [ 4, 2 ], [ 1, 5 ], [ 4, 5 ], [ 2, 1 ] ]
```

79.3 StrongOrbit

`StrongOrbit(` *M* `,` *d* `,` *action* `)`
`StrongOrbit(` *M* `,` *d* `,` *action* `,` *grad* `)`

The **strong orbit** of the point d in D under the action of M with respect to the grading *grad* is the set $\{d\hat{\ }m_1 \mid m_1 \in M, d\hat{\ }(m_1 m_2) = d \text{ for some } m_2 \in M\}$.

Note that the orbit of a point in general consists of several strong orbits.

In the first form `StrongOrbit` determines the strong orbit of point d under M with respect to the action *action* and the trivial grading.

In the second form `StrongOrbit` determines the strong orbit of point d under M with respect to the action *action*. Moreover, the grading *grad* is used to facilitate the calculations. Note, however, that the strong orbit of a point does not depend on the chosen grading.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 1 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] ) ;;
gap> Orbit( M, 3 );
[ 3, 4, 5, 2, 1 ]
gap> StrongOrbit( M, 3, OnPoints );
[ 3 ]
```

Note that `StrongOrbit` always requires the argument *action* specifying how the monoid acts (see 79.1).

79.4 GradedOrbit

`GradedOrbit(` *M*, *d*, *action*, *grad* `)`

The **graded orbit** of the point d in D under the action of M with respect to the grading *grad* is the list `[`O_1`,` O_2`, ... ]` of sets $O_i = \{d\hat{}m \mid m \in M, grad(d\hat{}m) = i\}$. Thus the orbit of d is simply the union of the sets O_i.

The function `GradedOrbit` determines the graded orbit of point d under M with respect to the grading *grad* and the action *action*.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 1 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] ) ;;
gap> Orbit( M, [ 1, 2, 3 ], OnSets );
[ [ 1, 2, 3 ], [ 4, 5 ], [ 2, 5 ], [ 1, 2 ], [ 1, 4 ], [ 1, 5 ],
  [ 2, 4 ] ]
gap> GradedOrbit( M, [ 1, 2, 3 ], OnSets, Size );
[ [  ], [ [ 4, 5 ], [ 2, 5 ], [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 2, 4 ] ],
[ [ 1, 2, 3 ] ] ]
```

Note that `GradedOrbit` always requires the argument *action* specifying how the monoid acts (see 79.1).

79.5 ShortOrbit

`ShortOrbit(` *M*, *d*, *action*, *grad* `)`

The **short orbit** of the point d in D under the action of M with respect to the grading *grad* is the set $\{d\hat{}m \mid m \in M, grad(d\hat{}m) = grad(d)\}$.

The function `ShortOrbit` determines the short orbit of the point d under M with respect to the grading *grad* and the action *action*.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 1 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] ) ;;
gap> Orbit(M, [1, 2, 3], OnSets);
[ [ 1, 2, 3 ], [ 4, 5 ], [ 2, 5 ], [ 1, 2 ], [ 1, 4 ], [ 1, 5 ],
  [ 2, 4 ] ]
gap> ShortOrbit(M, [1, 2, 3], OnSets, Size);
[ [ 1, 2, 3 ] ]
```

Note that `ShortOrbit` always requires the argument *action* specifying how the monoid acts (see 79.1).

79.6 Action

`Action(` *M*, *D* `)`
`Action(` *M*, *D*, *action* `)`

`Action` returns a transformation monoid with the same number of generators as M, such that each generator of the transformation monoid acts on the set `[1..Length(`D`)]` in the same way as the corresponding generator of the monoid M acts on the domain D, which may be a list of arbitrary type.

It is not allowed that D is a proper subset of a domain, i.e., D must be invariant under M.

`Action` accepts a function *action* of two arguments d and m as optional third argument, which specifies how the elements of M act on D (see 79.1).

`Action` calls
M`.operations.Action(` *M* `,` *D* `,` *action* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `MonoidOps.Action`, which simply applies each generator of M to all the points of D, finds the position of the image in D, and finally constructs the transformation (see 77.2) defined by the list of those positions.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 2 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] );;
gap> Action(M, LClasses(M), OnLClasses);
Monoid( [
Transformation( [2, 6, 9, 2, 2, 6, 13, 9, 6, 9, 7, 13, 12, 13, 14] ),
Transformation( [5, 3, 4, 2, 5, 7, 8, 6, 10, 11, 9, 12, 14, 15, 13] )
] )
```

79.7 ActionWithZero

`ActionWithZero(` *M* `,` *D* `)`
`ActionWithZero(` *M* `,` *D* `,` *action* `)`

`ActionWithZero` returns a transformation monoid with the same number of generators as M, such that each generator of the transformation monoid acts on the set `[1..Length(`*D*`)+1]` in the same way as the corresponding generator of the monoid M acts on the domain $D \cup \{0\}$, which may be a list of arbitrary type.

Here it is not required that D be invariant under M. Whenever the image of a point d under the monoid element m does not lie in D it is set to 0. The image of 0 under every monoid element is set to 0. Note that this way the resulting monoid is a homomorphic image of M if and only if D is a union of strong orbits. The point 0 is represented by `Length(`*D*`) + 1` in the domain of the transformation monoid returned by `ActionWithZero`.

`ActionWithZero` accepts a function *action* of two arguments d and m as optional third argument, which specifies how the elements of M act on D (see 79.1).

`ActionWithZero` calls
M`.operations.ActionWithZero(` *M* `,` *D* `,` *action* `)`
and returns the value. Note that the third argument is not optional for functions called this way.

The default function called this way is `MonoidOps.ActionWithZero`, which simply applies each generator of M to all the points of D, finds the position of the image in D, and finally constructs the transformation (see 77.2) defined by the list of those positions and `Length(`*D*`)+1` for every image not in D.

```
gap> M:= Monoid( [ Transformation( [ 5, 4, 4, 2, 2 ] ),
> Transformation( [ 2, 5, 5, 4, 1 ] ) ] );;
gap> M.name:= "M";;
```

```
gap> class:= LClass( M, Transformation( [ 1, 4, 4, 5, 5 ] ) );
LClass( M, Transformation( [ 1, 4, 4, 5, 5 ] ) )
gap> orb:= ShortOrbit(M, class, OnLClasses, Rank);
[ LClass( M, Transformation( [ 1, 4, 4, 5, 5 ] ) ),
  LClass( M, Transformation( [ 2, 4, 4, 1, 1 ] ) ),
  LClass( M, Transformation( [ 4, 2, 2, 5, 5 ] ) ) ]
gap> ActionWithZero(M, orb, OnLClasses);
Monoid( [ Transformation( [ 4, 3, 4, 4 ] ),
 Transformation( [ 2, 3, 1, 4 ] ) ] )
```

Chapter 80

XMOD

80.1 About XMOD

This document describes a package for the GAP3 group theory langauge which enables computations with the equivalent notions of finite, permutation *crossed modules* and *cat1-groups*.

The package divides into six parts, each of which has its own introduction:

- for constructing crossed modules and their morphisms in section 80.2: About crossed modules;
- for cat1-groups, their morphisms, and for converting between crossed modules and cat1-groups, in section 80.47: About cat1-groups;
- for derivations and sections and the monoids which they form under the Whitehead multiplication, in section 80.77: About derivations and sections;
- for actor crossed modules, actor cat1-groups and the actors squares which they form, in section 80.113: About actors;
- for the construction of induced crossed modules and induced cat1-groups, in section 80.127: About induced constructions;
- for a collection of utility functions in section 80.131: About utilities.

These seven `About...` sections are collected together in a separate LaTeXfile, `xmabout.tex`, which forms a short introduction to the package.

The package may be obtained as a compressed file by ftp from one of the sites with a GAP3 archive. After decompression, instructions for installing the package may be found in the `README` file.

The following constructions are planned for the next version of the package. Firstly, although sub-crossed module functions have been included, the equivalent set of sub-cat1-groups functions is not complete. Secondly, functions for pre-crossed modules, the Peiffer subgroup of a pre-crossed module and the associated crossed modules, will be added. Group-graphs provide examples of pre-crossed modules and their implementation will require interaction with graph-theoretic functions in GAP3. Crossed squares and the equivalent cat2-groups are the structures which arise as "three-dimensional groups". Examples of these are implicitly included already, namely inclusions of normal sub-crossed modules, and the inner morphism from a crossed module to its actor (section 80.123).

80.2 About crossed modules

The term crossed module was introduced by J. H. C. Whitehead in [Whi48], [Whi49]. In [Lod82] Loday reformulated the notion of a crossed module as a cat1-group. Norrie [Nor90], [Nor87] and Gilbert [Gil90] have studied derivations, automorphisms of crossed modules and the actor of a crossed module, while Ellis [Ell84] has investigated higher dimensional analogues. Properties of induced crossed modules have been determined by Brown, Higgins and Wensley in [BH78], [BW95] and [BW96]. For further references see [AW97] where we discuss some of the data structures and algorithms used in this package, and also tabulate isomorphism classes of cat1-groups up to size 30.

We first recall the descriptions of three equivalent categories: **XMod**, the category of crossed modules and their morphisms; **Cat1**, the category of cat1-groups and their morphisms; and **GpGpd**, the subcategory of group objects in the category **Gpd** of groupoids. We also include functors between these categories which exhibit the equivalences. Most papers on crossed modules use left actions, but we give the alternative right action axioms here, which are more suitable for use in computational group theory programs.

A crossed module $\mathcal{X} = (\partial : S \to R)$ consists of a group homomorphism ∂, called the *boundary* of $\mathcal{X}$, with *source* S and *range* R, together with an action $\alpha : R \to \mathrm{Aut}(S)$ satisfying, for all $s, s_1, s_2 \in S$ and $r \in R$,

$$\begin{array}{lrcl} \textbf{XMod 1:} & \partial(s^r) & = & r^{-1}(\partial s)r \\ \textbf{XMod 2:} & s_1^{\partial s_2} & = & s_2^{-1}s_1s_2. \end{array}$$

The kernel of ∂ is abelian.

The standard constructions for crossed modules are as follows

1. A *conjugation crossed module* is an inclusion of a normal subgroup $S \trianglelefteq R$, where R acts on S by conjugation.

2. A *central extension crossed module* has as boundary a surjection $\partial : S \to R$ with central kernel, where $r \in R$ acts on S by conjugation with $\partial^{-1}r$.

3. An *automorphism crossed module* has as range a subgroup R of the automorphism group $\mathrm{Aut}(S)$ of S which contains the inner automorphism group of S. The boundary maps $s \in S$ to the inner automorphism of S by s.

4. A *trivial action crossed module* $\partial : S \to R$ has $s^r = s$ for all $s \in S$, $r \in R$, the source is abelian and the image lies in the centre of the range.

5. An *R-Module crossed module* has an R-module as source and the zero map as boundary.

6. The direct product $\mathcal{X}_1 \times \mathcal{X}_2$ of two crossed modules has source $S_1 \times S_2$, range $R_1 \times R_2$ and boundary $\partial_1 \times \partial_2$, with $R_1,\ R_2$ acting trivially on $S_2,\ S_1$ respectively.

A morphism between two crossed modules $\mathcal{X}_1 = (\partial_1 : S_1 \to R_1)$ and $\mathcal{X}_2 = (\partial_2 : S_2 \to R_2)$ is a pair (σ, ρ), where $\sigma : S_1 \to S_2$ and $\rho : R_1 \to R_2$ are homomorphisms satisfying

$$\partial_2\sigma = \rho\partial_1, \quad \sigma(s^r) = (\sigma s)^{\rho r}.$$

When $\mathcal{X}_1 = \mathcal{X}_2$ and σ, ρ are automorphisms then (σ, ρ) is an automorphism of $\mathcal{X}_1$. The group of automorphisms is denoted by $\mathrm{Aut}(\mathcal{X}_1)$.

In this implementation a crossed module `X` is a record with fields

`X.source`,	the source S of ∂,
`X.boundary`,	the homomorphsim ∂,
`X.range`,	the range R of ∂,
`X.aut`,	a group of automorphisms of S,
`X.action`,	a homomorphism from R to `X.aut`,
`X.isXMod`,	a boolean flag, normally `true`,
`X.isDomain`,	always true,
`X.operations`,	special set of operations `XModOps` (see 80.15),
`X.name`,	a concatenation of the names of the source and range.

Here is a simple example of an automorphism crossed module, the holomorph of the cyclic group of size five.

```
gap> c5 := CyclicGroup( 5 );;   c5.name := "c5";;
gap> X1 := AutomorphismXMod( c5 );
Crossed module [c5->PermAut(c5)]
gap> XModPrint( X1 );
Crossed module [c5->PermAut(c5)] :-
: Source group c5 has generators:
  [ (1,2,3,4,5) ]
: Range group = PermAut(c5) has generators:
  [ (1,2,4,3) ]
: Boundary homomorphism maps source generators to:
  [ () ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,4,3) --> { source gens --> [ (1,3,5,2,4) ] }
  This automorphism generates the group of automorphisms.
```

Implementation of the standard constructions is described in sections `ConjugationXMod`, `CentralExtensionXMod`, `AutomorphismXMod`, `TrivialActionXMod` and `RModuleXMod`. With these building blocks, sub-crossed modules `SubXMod`, quotients of normal sub-crossed modules `FactorXMod` and direct products `XModOps.DirectProduct` may be constructed. An extra function `XModSelect` is used to call these constructions using groups of order up to 47 and data from file in `Cat1List`.

A morphism from a crossed module $\mathcal{X}_1$ to a crossed module $\mathcal{X}_2$ is a pair of homomorphisms (σ, ρ), where σ, ρ are respectively homomorphisms between the sources and ranges of $\mathcal{X}_1$ and $\mathcal{X}_2$, which commute with the two boundary maps and which are morphisms for the two actions. In the following code we construct a simple automorphism of `X1`.

```
gap> sigma1 := GroupHomomorphismByImages( c5, c5, [ (1,2,3,4,5) ],
>        [ (1,5,4,3,2) ] );;
gap> rho1 := InclusionMorphism( X1.range, X1.range );;
gap> mor1 := XModMorphism( X1, X1, [ sigma1, rho1 ] );
Morphism of crossed modules <[c5->PermAut(c5)] >-> [c5->PermAut(c5)]>
gap> IsXModMorphism( mor1 );
true
gap> XModMorphismPrint( mor1 );
Morphism of crossed modules :-
: Source = Crossed module [c5->PermAut(c5)] with generating sets:
  [ (1,2,3,4,5) ]
  [ (1,2,4,3) ]
: Range = Source
: Source Homomorphism maps source generators to:
  [ (1,5,4,3,2) ]
: Range Homomorphism maps range generators to:
  [ (1,2,4,3) ]
: isXModMorphism? true
gap> IsAutomorphism( mor1 );
true
```

The functors between **XMod** and **Cat1**, are implemented as functions `XModCat1` and `Cat1XMod`.

An integer variable `XModPrintLevel` is set initially equal to `1`. If it is increased, additional information is printed out during the execution of many of the functions.

80.3 The XMod Function

`XMod(` *f, a* `)`

A crossed module is determined by its boundary and action homomorphisms, f and a. All the standard constructions described below call this function after constructing the two homomorphisms. In the following example we construct a central extension crossed module $s3 \times c4 \to s3$ directly by defining the projection on to the first factor to be the boundary map, and constructing the automorphism group by taking two inner automorphisms as generators.

```
gap> s3c4 := Group( (1,2),(2,3),(4,5,6,7));;
gap> s3c4.name := "s3c4";;
gap> s3 := Subgroup( s3c4, [ (1,2), (2,3) ] );;
gap> s3.name := "s3";;
gap> #  construct the boundary
gap> gen := s3c4.generators;;
gap> imb := [ (1,2), (2,3), () ];;
gap> bX := GroupHomomorphismByImages( s3c4, s3, gen, imb );;
gap> #  construct the inner automorphisms by (1,2) and (2,3)
gap> im1 := List( gen, g -> g^(1,2) );;
gap> a1 := GroupHomomorphismByImages( s3c4, s3c4, gen, im1 );;
```

```
gap> im2 := List( gen, g -> g^(2,3) );;
gap> a2 := GroupHomomorphismByImages( s3c4, s3c4, gen, im2 );;
gap> A := Group( a1, a2 );;
gap> #  construct the action map from s3 to A
gap> aX := GroupHomomorphismByImages( s3, A, [(1,2),(2,3)], [a1,a2] );;
gap> X := XMod( bX, aX );
Crossed module [s3c4->s3]
```

80.4 IsXMod

`IsXMod(` *X* `)`

This Boolean function checks that the five main fields of X exist and that the crossed module axioms are satisfied.

```
gap> IsXMod( X );
true
```

80.5 XModPrint

`XModPrint(` *X* `)`

This function is used to display the main fields of a crossed module.

```
gap> XModPrint( X );
Crossed module [s3c4->s3] :-
: Source group s3c4 has generators:
  [ (1,2), (2,3), (4,5,6,7) ]
: Range group has parent ( s3c4 ) and has generators:
  [ (1,2), (2,3) ]
: Boundary homomorphism maps source generators to:
  [ (1,2), (2,3), () ]
: Action homomorphism maps range generators to automorphisms:
  (1,2) --> { source gens --> [ (1,2), (1,3), (4,5,6,7) ] }
  (2,3) --> { source gens --> [ (1,3), (2,3), (4,5,6,7) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.6 ConjugationXMod

`ConjugationXMod(` *R [,S]* `)`

This construction returns the crossed module whose source S is a normal subgroup of the range R, the boundary is the inclusion map, the group of automorphisms is the inner automorphism group of S, and the action maps an element of $r \in R$ to conjugation of S by r. The default value for S is R.

```
gap> s4 := Group( (1,2,3,4), (1,2) );;
gap> a4 := Subgroup( s4, [ (1,2,3), (2,3,4) ] );;
gap> k4 := Subgroup( a4, [ (1,2)(3,4), (1,3)(2,4) ] );;
gap> s4.name := "s4";;  a4.name := "a4";;  k4.name := "k4";;
gap> CX := ConjugationXMod( a4, k4 );
Crossed module [k4->a4]
```

80.7 XModName

`XModName(` X `)`

Whenever the names of the source or range of `X` are changed, this function may be used to produce the new standard form `[X.source.name->X.range.name]` for the name of `X`. This function is called automatically by `XModPrint`.

```
gap> k4.name := "v4";;
gap> XModName( CX );
"[v4->a4]"
```

80.8 CentralExtensionXMod

`CentralExtensionXMod(` f `)`

This construction returns the crossed module whose boundary f is a surjection from S to R having as kernel a subgroup of the centre of S. The action maps an element of $r \in R$ to conjugation of S by $f^{-1}r$.

```
gap> d8 := Subgroup( s4, [ (1,2,3,4), (1,3) ] );; d8.name := "d8";;
gap> gend8 := d8.generators;; genk4 := k4.generators;;
gap> f := GroupHomomorphismByImages( d8, k4, gend8, genk4 );;
gap> EX := CentralExtensionXMod( f );
Crossed module [d8->v4]
gap> XModPrint( EX );
Crossed module [d8->v4] :-
: Source group d8 has parent s4 and generators:
  [ (1,2,3,4), (1,3) ]
: Range group k4 has parent s4 and generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Boundary homomorphism maps source generators to:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2)(3,4) --> { source gens --> [ (1,2,3,4), (2,4) ] }
  (1,3)(2,4) --> { source gens --> [ (1,4,3,2), (1,3) ] }
These 2 automorphisms generate the group of automorphisms.
```

80.9 AutomorphismXMod

`AutomorphismXMod(` S *[,* A*]* `)`

This construction returns the crossed module whose range R is a permutation representation of a group A which is a group of automorphisms of the source S and which contains the inner automorphism group of S as a subgroup. When A is not specified the full automorphism group is used. The boundary morphism maps $s \in S$ to the representation of the inner automorphism of S by s. The action is the isomorphism $R \to A$.

In the following example, recall that the automorphism group of the quaternion group is isomorphic to the symmetric group of degree 4 and that the inner automorphism group is isomorphic to `k4`. The group `A` is a subgroup of `Aut(q8)` isomorphic to `d8`.

```
gap> q8 := Group( (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) );;
gap> q8.name := "q8";; genq8 := q8.generators;;
gap> iaq8 := InnerAutomorphismGroup( q8 );;
gap> a := GroupHomomorphismByImages( q8, q8, genq8,
>              [(1,5,3,7)(2,6,4,8),(1,4,3,2)(5,6,7,8)]);;
gap> genA := Concatenation( iaq8.generators, [a] );
[ InnerAutomorphism( q8, (1,2,3,4)(5,8,7,6) ),
  InnerAutomorphism( q8, (1,5,3,7)(2,6,4,8) ),
  GroupHomomorphismByImages( q8, q8, [ (1,2,3,4)(5,8,7,6),
   (1,5,3,7)(2,6,4,8) ], [ (1,5,3,7)(2,6,4,8), (1,4,3,2)(5,6,7,8) ] ) ]
gap> id := IdentityMapping( q8 );;
gap> A := Group( genA, id );;
gap> AX := AutomorphismXMod( q8, A );
Crossed module [q8->PermSubAut(q8)]
gap> RecFields( AX );
[ "isDomain", "isParent", "source", "range", "boundary", "action",
  "aut", "isXMod", "operations", "name", "isAutomorphismXMod" ]
```

80.10 InnerAutomorphismXMod

`InnerAutomorphismXMod(` S `)`

This function is equivalent to `AutomorphismXMod(S,A)` in the case when `A` is the inner automorphism group of `S`.

```
gap> IX := InnerAutomorphismXMod( q8 );
Crossed module [q8->PermInn(q8)]
```

80.11 TrivialActionXMod

`TrivialActionXMod(` f `)`

For a crossed module to have trivial action, the axioms require the source to be abelian and the image of the boundary to lie in the centre of the range. A homomorphism `f` can act as the boundary map when these conditions are satisfied.

```
gap> imf := [ (1,3)(2,4), (1,3)(2,4) ];;
gap> f := GroupHomomorphismByImages( k4, d8, genk4, imf );;
gap> TX := TrivialActionXMod( f );
Crossed module [v4->d8]
gap> XModPrint( TX );

Crossed module [v4->d8] :-
: Source group has parent ( s4 ) and has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Range group has parent ( s4 ) and has generators:
  [ (1,2,3,4), (1,3) ]
: Boundary homomorphism maps source generators to:
  [ (1,3)(2,4), (1,3)(2,4) ]
  The automorphism group is trivial
```

80.12 IsRModule for groups

`IsRModule(` *Rmod* `)`

`IsRModuleRecord(` *Rmod* `)`

An R-module consists of a permutation group R with an action $\alpha : R \to A$ where A is a group of automorphisms of an abelian group M. When `R` is not specified, the function `AutomorphismPair` is automatically called to construct it.

This structure is implemented here as a record `Rmod` with fields

`Rmod.module`,	the abelian group M,
`Rmod.perm`,	the group R,
`Rmod.auto`,	the action group A,
`Rmod.isRModule`,	set true.

The `IsRModule` distributor calls this function when the parameter is a record but not a crossed module.

```
gap> k4gen := k4.generators;;
gap> k4im := [ (1,3)(2,4), (1,4)(2,3) ];;
gap> a := GroupHomomorphismByImages( k4, k4, k4gen, k4im );;
gap> Ak4 := Group( a );;
gap> R := rec( );;
gap> R.module := k4;;
gap> R.auto := Ak4;;
gap> IsRModule( R );
true
gap> RecFields( R );
[ "module", "auto", "perm", "isRModule" ]
gap> R.perm;
PermSubAut(v4)
```

80.13 RModuleXMod

`RModuleXMod(` *Rmod* `)`

The crossed module `RX` obtained from an R-module has the abelian group M as source, the zero map as boundary, the group R which acts on M as range, the group A of automorphisms of M as `RX.aut` and $\alpha : R \to A$ as `RX.action`. An appropriate name for `RX` is chosen automatically. Continuing the previous example, M is `k4` and R is cyclic of order 3.

```
gap> RX := RModuleXMod( R );
Crossed module [v4->PermSubAut(v4)]
gap> XModPrint( RX );

Crossed module [v4->PermSubAut(v4)]
: Source group has parent s4 and has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Range group = PermSubAut(v4) has generators:
  [ (1,2,3) ]
: Boundary homomorphism maps source generators to:
```

```
  [ () ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3) --> { source gens --> [ (1,3)(2,4), (1,4)(2,3) ] }
  This automorphism generates the group of automorphisms.
```

80.14 XModSelect

`XModSelect(` *size* `[,` *gpnum*`,` *type*`,` *norm*`] )`

Here the parameter *size* may take any value up to 47, *gpnum* refers to the isomorphism class of groups of order *size* as ordered in the GAP3 library. The *norm* parameter is only used in the case `"conj"` and specifies the position of the source group in the list of normal subgroups of the range R. The list `Cat1List` is used to store the data for these groups. The allowable *types* are `"conj"` for normal inclusions with conjugation, `"aut"` for automorphism groups and `"rmod"` for Rmodules. If *type* is not specified the default is `"conj"`. If *norm* is not specified, then the `AutomorphismXMod` of R is returned.

In the following example the fourteenth class of groups of size 24 is a special linear group `sl(2,3)` and a double cover of `a4`. The third normal subgroup of `sl(2,3)` is a quaternion group, and a conjugation crossed module is returned.

```
gap> SX := XModSelect( 24, 14, "conj", 3 );
Crossed module [N3->sl(2,3)]
gap> XModPrint( SX );

Crossed module [N3->sl(2,3)] :-
: Source group has parent ( sl(2,3) )  and has generators:
  [ (1,2,3,4)(5,8,7,6), ( 1, 5, 3, 7)( 2, 6, 4, 8) ]
: Range group = sl(2,3)  has generators:
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8), (2,5,6)(4,7,8)(9,10,11) ]
: Boundary homomorphism maps source generators to:
  [ (1,2,3,4)(5,8,7,6), ( 1, 5, 3, 7)( 2, 6, 4, 8) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3,4)(5,8,7,6) --> { source gens -->
[ (1,2,3,4)(5,8,7,6), ( 1, 7, 3, 5)( 2, 8, 4, 6) ] }
  (1,5,3,7)(2,6,4,8) --> { source gens -->
[ (1,4,3,2)(5,6,7,8), ( 1, 5, 3, 7)( 2, 6, 4, 8) ] }
  ( 2, 5, 6)( 4, 7, 8)( 9,10,11) --> { source gens -->
[ ( 1, 5, 3, 7)( 2, 6, 4, 8), ( 1, 6, 3, 8)( 2, 7, 4, 5) ] }
  These 3 automorphisms generate the group of automorphisms.
```

80.15 Operations for crossed modules

Special operations defined for crossed modules are stored in the record structure `XModOps`. Every crossed module `X` has `X.operations := XModOps;`.

```
gap> RecFields( XModOps );
[ "name", "operations", "Elements", "IsFinite", "Size", "=", "<",
  "in", "IsSubset", "Intersection", "Union", "IsParent", "Parent",
  "Difference", "Representative", "Random", "Print", "Kernel",
```

```
"IsAspherical", "IsSimplyConnected", "IsConjugation",
"IsTrivialAction", "IsCentralExtension", "DirectProduct",
"IsAutomorphismXMod", "IsZeroBoundary", "IsRModule",
"InclusionMorphism", "WhiteheadPermGroup", "Whitehead", "Norrie",
"Lue", "Actor", "InnerMorphism", "Centre", "InnerActor",
 "AutomorphismPermGroup", "IdentityMorphism", "InnerAutomorphism", ]
```

Crossed modules `X,Y` are considered equal if they have the same source, boundary, range, and action. The remaining functions are discussed below and following section 80.113.

80.16 Print for crossed modules

`XModOps.Print(` *X* `)`

This function is the special print command for crossed modules, producing a single line of output, and is called automatically when a crossed module is displayed. For more detail use `XModPrint( X )`.

```
gap> CX;
Crossed module [v4->a4]
```

80.17 Size for crossed modules

`XModOps.Size(` *X* `)`

This function returns a 2-element list containing the sizes of the source and the range of `X`.

```
gap> Size( CX );
[ 4, 12 ]
```

80.18 Elements for crossed modules

`XModOps.Elements(` *X* `)`

This function returns a 2-element list of lists of elements of the source and range of `X`.

```
gap> Elements( CX );
[ [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ],
  [ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
    (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ] ]
```

80.19 IsConjugation for crossed modules

`XModOps.IsConjugation(` *X* `)`

This Boolean function checks that the source is a normal subgroup of the range and that the boundary is an inclusion.

```
gap> IsConjugation( CX );
true
```

80.20 IsAspherical

`XModOps.IsAspherical(` X `)`

This Boolean function checks that the boundary map is monomorphic.

```
gap> IsAspherical( CX );
true
```

80.21 IsSimplyConnected

`XModOps.IsSimplyConnected(` X `)`

This Boolean function checks that the boundary map is surjective. The corresponding groupoid then has a single connected component.

```
gap> IsSimplyConnected( EX );
true
```

80.22 IsCentralExtension

`XModOps.IsCentralExtension(` X `)`

This Boolean function checks that the boundary is surjective with kernel central in the source.

```
gap> IsCentralExtension( EX );
true
```

80.23 IsAutomorphismXMod

`XModOps.IsAutomorphismXMod(` X `)`

This Boolean function checks that the range group is a subgroup of the automorphism group of the source group containing the group of inner automorphisms, and that the boundary and action homomorphisms are of the correct form.

```
gap> IsAutomorphismXMod( AX );
true
```

80.24 IsTrivialAction

`XModOps.IsTrivialAction(` X `)`

This Boolean function checks that the action is the zero map.

```
gap> IsTrivialAction( TX );
true
```

80.25 IsZeroBoundary

`XModOps.IsZeroBoundary(` X `)`

This Boolean function checks that the boundary is the zero map.

```
gap> IsZeroBoundary( EX );
false
```

80.26 IsRModule for crossed modules

`XModOps.IsRModule(` *X* `)`

This Boolean function checks that the boundary is the zero map and that the source is abelain.

```
gap> IsRModule( RX );
true
```

80.27 WhatTypeXMod

`WhatTypeXMod(` *X* `)`

This function checks whether the crossed module `X` is one or more of the six standard type listed above.

```
gap> WhatTypeXMod( EX );
[ " extn, " ]
```

80.28 DirectProduct for crossed modules

`XModOps.DirectProduct(` *X,Y* `)`

The direct product of crossed modules `X,Y` has as source and range the direct products of the sources and ranges of `X` and `Y`. The boundary map is the product of the two boundaries. The range of `X` acts trivially on the source of `Y` and conversely. Because the standard `DirectProduct` function requires the two parameters to be groups, the `XModOps.` prefix must be used (at least for GAP33.4.3).

```
gap> DX := XModOps.DirectProduct( CX, CX );
Crossed module [v4xv4->a4xa4]
gap> XModPrint( DX );

Crossed module [v4xv4->a4xa4] :-
: Source group v4xv4 has generators:
  [ (1,2)(3,4), (1,3)(2,4), (5,6)(7,8), (5,7)(6,8) ]
: Range group = a4xa4 has generators:
  [ (1,2,3), (2,3,4), (5,6,7), (6,7,8) ]
: Boundary homomorphism maps source generators to:
  [ (1,2)(3,4), (1,3)(2,4), (5,6)(7,8), (5,7)(6,8) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3) --> { source gens -->
      [ (1,4)(2,3), (1,2)(3,4), (5,6)(7,8), (5,7)(6,8) ] }
  (2,3,4) --> { source gens -->
      [ (1,3)(2,4), (1,4)(2,3), (5,6)(7,8), (5,7)(6,8) ] }
  (5,6,7) --> { source gens -->
      [ (1,2)(3,4), (1,3)(2,4), (5,8)(6,7), (5,6)(7,8) ] }
  (6,7,8) --> { source gens -->
      [ (1,2)(3,4), (1,3)(2,4), (5,7)(6,8), (5,8)(6,7) ] }
  These 4 automorphisms generate the group of automorphisms.
```

80.29 XModMorphism

`XModMorphism(` *X, Y, homs* `)`

A morphism of crossed modules is a pair of homomorphisms `[ sourceHom, rangeHom ]`, where `sourceHom`, `rangeHom` are respectively homomorphisms between the sources and ranges of X and Y, which commute with the two boundary maps and which are morphisms for the two actions.

In this implementation a morphism of crossed modules `mor` is a record with fields

`mor.source`,	the source crossed module `X`,
`mor.range`,	the range crossed module `Y`,
`mor.sourceHom`,	a homomorphism from `X.source` to `Y.source`,
`mor.rangeHom`,	a homomorphism from `X.range` to `Y.range`,
`mor.isXModMorphism`,	a Boolean flag, normally `true`,
`mor.operations`,	a special set of operations `XModMorphismOps` (see 80.33),
`mor.name`,	a concatenation of the names of `X` and `Y`.

The function `XModMorphism` requires as parameters two crossed modules and a two-element list containing the source and range homomorphisms. It sets up the required fields for `mor`, but does not check the axioms. The `IsXModMorphism` function should be used to perform these checks. Note that the `XModMorphismPrint` function is needed to print out the morphism in detail.

```
gap> smor := GroupHomomorphismByImages( q8, k4, genq8, genk4 );
GroupHomomorphismByImages( q8, v4,
 [(1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8)], [(1,2)(3,4), (1,3)(2,4)] )
gap> IsHomomorphism(smor);
true
gap> sl23 := SX.range;;
gap> gensl23 := sl23.generators;
 [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8), (2,5,6)(4,7,8)(9,10,11) ]
gap> images := [ (1,2)(3,4), (1,3)(2,4), (2,3,4) ];;
gap> rmor := GroupHomomorphismByImages( sl23, a4, gensl23, images );;
gap> IsHomomorphism(rmor);
true
gap> mor := XModMorphism( SX, CX, [ smor, rmor ] );
Morphism of crossed modules <[N3->sl(2,3)] >-> [v4->a4]>
```

80.30 IsXModMorphism

`IsXModMorphism(` *mor* `)`

This Boolean function checks that *mor* includes homomorphisms between the corresponding source and range crossed modules, and that these homomorphisms commute with the two actions. In the example we increase the value of `XModPrintLevel` to show the effect of such an increase in a simple case.

```
gap> XModPrintLevel := 3;;
gap> IsXModMorphism( mor );
Checking that the diagram commutes :-
```

```
          Y.boundary(morsrc(x)) = morrng(X.boundary(x))
Checking:  morsrc(x2^x1) = morsrc(x2)^(morrng(x1))
true
gap> XModPrintLevel := 1;;
```

80.31 XModMorphismPrint

`XModMorphismPrint(` *mor* `)`

This function is used to display the main fields of a crossed module.

```
gap> XModMorphismPrint( mor );
Morphism of crossed modules :-
: Source = Crossed module [N3->sl(2,3)] with generating sets
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8), (2,5,6)(4,7,8)(9,10,11) ]
:  Range = Crossed module [v4->a4] with generating sets
  [ (1,2)(3,4), (1,3)(2,4) ]]
  [ (1,2,3), (2,3,4) ]
: Source Homomorphism maps source generators to:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Range Homomorphism maps range generators to:
  [ (1,2)(3,4), (1,3)(2,4), (2,3,4) ]
: isXModMorphism? true
```

80.32 XModMorphismName

`XModMorphismName(` *mor* `)`

Whenever the names of the source or range crossed module are changed, this function may be used to produce the new standard form `<mor.source.name >-> mor.range.name>` for the *name* of *mor*. This function is automatically called by `XModMorphismPrint`.

```
gap> k4.name := "k4";;  XModName( CX );;
gap> XModMorphismName( mor );
<[N3->sl(2,3)] >-> [k4->a4]>
```

80.33 Operations for morphisms of crossed modules

Special operations defined for morphisms of crossed modules are stored in the record structure `XModMorphismOps` which is based on `MappingOps`. Every crossed module morphism `mor` has field `mor.operations` set equal to `XModMorphismOps;`.

```
gap> IsMonomorphism( mor );
false
gap> IsEpimorphism( mor );
true
gap> IsIsomorphism( mor );
false
gap> IsEndomorphism( mor );
false
gap> IsAutomorphism( mor );
false
```

80.34 IdentitySubXMod

`IdentitySubXMod(` *X* `)`

Every crossed module `X` has an identity sub-crossed module whose source and range are the identity subgroups of the source and range.

```
gap> IdentitySubXMod( CX );
Crossed module [Id[k4->a4]]
```

80.35 SubXMod

`SubXMod(` *X, subS, subR* `)`

A sub-crossed module of a crossed module `X` has as source a subgroup `subS` of `X.source` and as range a subgroup `subR` of `X.range`. The boundary map and the action are the appropriate restrictions. In the following example we construct a sub-crossed module of `SX` with range `q8` and source a cyclic group of order 4.

```
gap> q8 := SX.source;; genq8 := q8.generators;;
gap> q8.name := "q8";; XModName( SX );;
gap> c4 := Subgroup( q8, [ genq8[1] ] );
Subgroup( sl(2,3), [ (1,2,3,4)(5,8,7,6) ] )
gap> c4.name := "c4";;
gap> subSX := SubXMod( SX, c4, q8 );
Crossed module [c4->q8]
gap> XModPrint( subSX );
Crossed module [c4->q8] :-
: Source group has parent ( sl(2,3) ) and has generators:
  [ (1,2,3,4)(5,8,7,6) ]
: Range group has parent ( sl(2,3) ) and has generators:
  [ (1,2,3,4)(5,8,7,6), ( 1, 5, 3, 7)( 2, 6, 4, 8) ]
: Boundary homomorphism maps source generators to:
  [ ( 1, 2, 3, 4)( 5, 8, 7, 6) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3,4)(5,8,7,6) --> {source gens --> [ (1,2,3,4)(5,8,7,6) ]}
  (1,5,3,7)(2,6,4,8) --> {source gens --> [ (1,4,3,2)(5,6,7,8) ]}
  These 2 automorphisms generate the group of automorphisms.
```

80.36 IsSubXMod

`IsSubXMod(` *X,S* `)`

This boolean function checks that `S` is a sub-crossed module of `X`.

```
gap> IsSubXMod( SX, subSX );
true
```

80.37 InclusionMorphism for crossed modules

`InclusionMorphism(` *S,X* `)`

This function constructs the inclusion of a sub-crossed module S of X. When S = X the identity morphism is returned.

```
gap> inc := InclusionMorphism( subSX, SX );
Morphism of crossed modules <[c4->q8] >-> [q8->sl(2,3)]>
gap> IsXModMorphism( inc );
true
gap> XModMorphismPrint( inc );
Morphism of crossed modules :-
: Source = Crossed module [c4->q8] with generating sets:
  [ (1,2,3,4)(5,8,7,6) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
:  Range = Crossed module [q8->sl(2,3)] with generating sets:
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8), (2,5,6)(4,7,80(9,10,11) ]
: Source Homomorphism maps source generators to:
  [ (1,2,3,4)(5,8,7,6) ]
: Range Homomorphism maps range generators to:
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
:  isXModMorphism? true
```

80.38 IsNormalSubXMod

`IsNormalSubXMod(` X, Y `)`

A sub-crossed module `Y=(N->M)` is normal in `X=(S->R)` when

- N, M are normal subgroups of S, R respectively,
- $n^r \in N$ for all $n \in N,\ r \in R$,
- $s^{-1}\ s^m \in N$ for all $m \in M,\ s \in S$.

These axioms are sufficient to ensure that $M \ltimes N$ is a normal subgroup of $R \ltimes S$. They also ensure that the inclusion morphism of a normal sub-crossed module forms a conjugation crossed square, analogous to the construction of a conjugation crossed module.

```
gap> IsNormalSubXMod( SX, subSX );
false
```

80.39 NormalSubXMods

`NormalSubXMods(` X `)`

This function takes pairs of normal subgroups from the source and range of `X` and constructs a normal sub-crossed module whenever the axioms are satisfied. Appropriate names are chosen where possible.

```
gap> NSX := NormalSubXMods( SX );
[ Crossed module [Id[q8->sl(2,3)]], Crossed module [I->?],
  Crossed module [Sub[q8->sl(2,3)]], Crossed module [?->q8],
  Crossed module [?->q8], Crossed module [q8->sl(2,3)] ]
```

80.40 Factor crossed module

`FactorXMod(` *X, subX* `)`

The quotient crossed module of a crossed module by a normal sub-crossed module has quotient groups as source and range, with the obvious action.

```
gap> Size( NSX[3] );
[ 2, 2 ]
gap> FX := FactorXMod( SX, NSX[3] );
Crossed module [?->?]
gap> Size( FX );
[ 4, 12 ]
```

80.41 Kernel of a crossed module morphism

`Kernel(` *mor* `)`

The kernel of a morphism `mor` : $X \to Y$ of crossed modules is the normal sub-crossed module of X whose source is the kernel of `mor.sourceHom` and whose range is the kernel of `mor.rangeHom`. An appropriate name for the kernel is chosen automatically. A field `.kernel` is added to `mor`.

```
gap> XModMorphismName( mor );;
gap> KX := Kernel( mor );
Crossed module Ker<[q8->sl(2,3)] >-> [k4->a4]>
gap> XModPrint( KX );
Crossed module Ker<[q8->SL(2,3)] >-> [k4->a4]> :-
: Source group has parent ( sl(2,3) )  and has generators:
  [ (1,3)(2,4)(5,7)(6,8) ]
: Range group has parent ( sl(2,3) )  and has generators:
  [ ( 1, 3)( 2, 4)( 5, 7)( 6, 8) ]
: Boundary homomorphism maps source generators to:
  [ (1,3)(2,4)(5,7)(6,8) ]
: The automorphism group is trivial.
gap> IsNormalSubXMod( SX, KX );
true
```

80.42 Image for a crossed module morphism

`ImageXModMorphism(` *mor*`,` *S* `)`

The image of a sub-crossed module `S` of `X` under a morphism `mor` : $X \to Y$ of crossed modules is the sub-crossed module of Y whose source is the image of `S.source` under `mor.sourceHom` and whose range is the image of `S.range` under `mor.rangeHom`. An appropriate name for the image is chosen automatically. A field `.image` is added to `mor`. Note that thjis function should be named `XModMorphismOps.Image`, but the command `J := Image(  mor, S );` does not work with version 3 of GAP3.

```
gap> subSX;
Crossed module [c4->q8]
gap> JX := ImageXModMorphism( mor, subSX );
```

```
Crossed module [Im([c4->q8]) by <[q8->sl(2,3)] >-> [k4->a4]>]
gap> RecFields( mor );
[ "sourceHom", "rangeHom", "source", "range", "name", "isXModMorphism",
   "domain", "kernel", "image", "isMonomorphism", "isEpimorphism",
   "isIsomorphism", "isEndomorphism", "isAutomorphism", "operations" ]
gap> XModPrint( JX );

Crossed module [Im([c4->q8]) by <[q8->sl(2,3)] >-> [k4->a4]>] :-
: Source group has parent ( s4 )  and has generators:
  [ (1,2)(3,4) ]
: Range group has parent ( s4 )  and has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Boundary homomorphism maps source generators to:
  [ (1,2)(3,4) ]
: The automorphism group is trivial.
```

80.43 InnerAutomorphism of a crossed module

`InnerAutomorphism(` *X, r* `)`

Each element `r` of `X.range` determines an automorphism of `X` in which the automorphism of `X.source` is given by the image of `X.action` on `r` and the automorphism of `X.range` is conjugation by `r`. The command `InnerAutomorphism(  X, r );` does not work with version 3 of GAP3.

```
gap> g := Elements( q8 )[8];
(1,8,3,6)(2,5,4,7)
gap> psi := XModOps.InnerAutomorphism( subSX, g );
Morphism of crossed modules <[c4->q8] >-> [c4->q8]>
gap> XModMorphismPrint( psi );
Morphism of crossed modules :-
: Source = Crossed module [c4->q8] with generating sets:
  [ (1,2,3,4)(5,8,7,6) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
:  Range = Crossed module [c4->q8] with generating sets:
  [ (1,2,3,4)(5,8,7,6) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
: Source Homomorphism maps source generators to:
  [ ( 1,4,3,2)(5,6,7,8) ]
: Range Homomorphism maps range generators to:
  [ ( 1,4,3,2)(5,6,7,8), (1,7,3,5)(2,8,4,6) ]
  isXModMorphism? true
```

80.44 Order of a crossed module morphism

`XModMorphismOps.Order(` *mor* `)`

This function calculates the order of an automorphism of a crossed module.

```
gap> XModMorphismOps.Order( psi );
2
```

80.45 CompositeMorphism for crossed modules

`CompositeMorphism(` *mor1, mor2* `)`

Morphisms $\mu_1 : X \to Y$ and $\mu_2 : Y \to Z$ have a composite $\mu = \mu_2 \circ \mu_1 : X \to Z$ whose source and range homomorphisms are the composites of those of μ_1 and μ_2.

In the following example we compose `psi` with the `inc` obtained previously.

```
gap> xcomp := XModMorphismOps.CompositeMorphism( psi, inc );
Morphism of crossed modules <[c4->q8] >-> [q8->sl(2,3)]>
gap> XModMorphismPrint( xcomp );
Morphism of crossed modules :-
: Source = Crossed module [c4->q8] with generating sets:
  [ (1,2,3,4)(5,8,7,6) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
: Range = Crossed module [q8->sl(2,3)] with generating sets:
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8) ]
  [ (1,2,3,4)(5,8,7,6), (1,5,3,7)(2,6,4,8), (2,5,6)(4,7,8)(9,10,11) ]
: Source Homomorphism maps source generators to:
  [ (1,4,3,2)(5,6,7,8) ]
: Range Homomorphism maps range generators to:
  [ (1,4,3,2)(5,6,7,8), (1,7,3,5)(2,8,4,6) ]
: isXModMorphism? true
```

80.46 SourceXModXPModMorphism

`SourceXModXPModMorphism(` *mor* `)`

When crossed modules X, Y have a common range P and `mor` is a morphism from X to Y whose range homomorphism is the identity homomorphism, then `mor.sourceHom : X.source -> Y.source)` is a crossed module.

```
gap> c2 := Subgroup( q8, [ genq8[1]^2 ] );
Subgroup( sl(2,3), [ (1,3)(2,4)(5,7)(6,8) ] )
gap> c2.name := "c2";;
gap> sub2 := SubXMod( subSX, c2, q8 );
Crossed module [c2->q8]
gap>inc2 := InclusionMorphism( sub2, subSX );
Morphism of crossed modules <[c2->q8] >-> [c4->q8]>
gap> PX := SourceXModXPModMorphism( inc2 );
Crossed module [c2->c4]
gap> IsConjugation( PX );
true
```

80.47 About cat1-groups

In [Lod82] Loday reformulated the notion of a crossed module as a cat1-group, namely a group G with a pair of homomorphisms $t, h : G \to G$ having a common image R and satisfying certain axioms. We find it convenient to define a cat1-group $\mathcal{C} = (e; t, h : G \to R)$ as having source group G, range group R, and three homomorphisms: two surjections $t, h : G \to R$ and an embedding $e : R \to G$ satisfying:

Cat 1: $te = he = \mathrm{id}_R$,
Cat 2: $[\ker t, \ker h] = \{1_G\}$.

It follows that $teh = h,\ het = t,\ tet = t,\ heh = h$.

The maps t, h are often referred to as the *source* and *target*, but we choose to call them the *tail* and *head* of $\mathcal{C}$, because *source* is the GAP3 term for the domain of a function.

A morphism $\mathcal{C}_1 \to \mathcal{C}_2$ of cat1-groups is a pair (γ, ρ) where $\gamma : G_1 \to G_2$ and $\rho : R_1 \to R_2$ are homomorphisms satisfying

$$h_2\gamma = \rho h_1,\ \ t_2\gamma = \rho t_1,\ \ e_2\rho = \gamma e_1,$$

(see 80.61 and subsequent sections).

In this implementation a cat1-group $\mathcal{C}$ is a record with the following fields:

`C.source`,	the source `G`,
`C.range`,	the range `R`,
`C.tail`,	the tail homomorphism `t`,
`C.head`,	the head homomorphism `h`,
`C.embedRange`,	the embedding of `R` in `G`,
`C.kernel`,	a permutation group isomorphic to the kernel of `t`,
`C.embedKernel`,	the inclusion of the kernel in `G`,
`C.boundary`,	the restriction of `h` to the kernel,
`C.isDomain`,	set `true`,
`C.operations`,	a special set of operations `Cat1Ops` (see 80.53,
`C.name`,	a concatenation of the names of the source and range.
`C.isCat1`	a boolean flag, normally `true`.

The following listing shows a simple example:

```
gap> s3c4gen := s3c4.generators;
[ (1,2), (2,3), (4,5,6,7) ]
gap> t1 := GroupHomomorphismByImages( s3c4, s3, s3c4gen,
>      [ (1,2), (2,3), () ] );;
gap> C1 := Cat1( s3c4, t1, t1 );
cat1-group [s3c4 ==> s3]
gap> Cat1Print( C1 );
cat1-group [s3c4 ==> s3] :-
: source group has generators:
  [ (1,2), (2,3), (4,5,6,7) ]
:  range group has generators:
  [ (1,2), (2,3) ]
: tail homomorphism maps source generators to:
```

```
  [ ( 1, 2), ( 2, 3), () ]
: head homomorphism maps source generators to:
  [ ( 1, 2), ( 2, 3), () ]
: range embedding maps range generators to:
  [ (1,2), (2,3) ]
: kernel has generators:
  [ (4,5,6,7) ]
: boundary homomorphism maps generators of kernel to:
  [ () ]
: kernel embedding maps generators of kernel to:
  [ (4,5,6,7) ]
```

The category of crossed modules is equivalent to the category of cat1-groups, and the functors between these two categories may be described as follows.

Starting with the crossed module $\mathcal{X} = (\partial : S \to R)$ the group G is defined as the semidirect product $G = R \ltimes S$ using the action from $\mathcal{X}$. The structural morphisms are given by

$$t(r, s) = r,\ h(r, s) = r(\partial s),\ er = (r, 1).$$

On the other hand, starting with a cat1-group $\mathcal{C} = (e; t, h : G \to R)$ we define $S = \ker t$, the range R remains unchanged and $\partial = h\mid_S$. The action of R on S is conjugation in S via the embedding of R in G.

```
gap> X1;
Crossed module [c5->PermAut(c5)]
gap> CX1 := Cat1XMod(X1);
cat1-group [Perm(PermAut(c5) |X c5) ==> PermAut(c5)]
gap> CX1.source.generators;
[ (2,3,5,4), (1,2,3,4,5) ]
gap>
gap> XC1 := XModCat1( C1 );
Crossed module [ker([s3c4 ==> s3])->s3]
gap> WhatTypeXMod( XC1 );
[ " triv, ", " zero, ", " RMod, " ]
```

80.48 Cat1

`Cat1(` *G*, *t*, *h* `)`

This function constructs a cat1-group $\mathcal{C}$ from a group G and a pair of endomorphisms, the tail and head of $\mathcal{C}$. The example uses the holomorph of `c5`, a group of size 20, which was the source group in `XC1` in 80.47. Note that when $t = h$ the boundary is the zero map.

```
gap> h20 := Group( (1,2,3,4,5), (2,3,5,4) );;
gap> h20.name := "h20";;
gap> genh20 := h20.generators;;
gap> imh20 := [ (), (2,3,5,4) ];;
gap> h := GroupHomomorphismByImages( h20, h20, genh20, imh20 );;
gap> t := h;;
gap> C := Cat1( h20, t, h );
cat1-group [h20 ==> R]
```

80.49 IsCat1

`IsCat1(` *C* `)`

This function checks that the axioms of a cat1-group are satisfied and that the main fields of a cat1-group record exist.

```
gap> IsCat1(C);
true
```

80.50 Cat1Print

`Cat1Print(` *C* `)`

This function is used to display the main fields of a cat1-group.

```
gap> Cat1Print(C);

cat1-group [h20 ==> R] :-
: source group has generators:
  [ (1,2,3,4,5), (2,3,5,4) ]
: range group has generators:
  [ ( 2, 3, 5, 4) ]
: tail homomorphism maps source generators to:
  [ (), ( 2, 3, 5, 4) ]
: head homomorphism maps source generators to:
  [ (), ( 2, 3, 5, 4) ]
: range embedding maps range generators to:
  [ ( 2, 3, 5, 4) ]
: kernel has generators:
  [ (1,2,3,4,5) ]
: boundary homomorphism maps generators of kernel to:
  [ () ]
: kernel embedding maps generators of kernel to:
  [ ( 1, 2, 3, 4, 5) ]
```

80.51 Cat1Name

`Cat1Name(` *C* `)`

Whenever the names of the source or the range of `C` are changed, this function may be used to produce the new standard form `[<C.source.name>   ==> <C.range.name>]` for the name of `C`. This function is called automatically by `Cat1Print`. Note the use of `=`, rather than `-` in the arrow shaft, to indicate the pair of maps.

```
gap> C.range.name := "c4";; Cat1Name( C );
"[h20 ==> c4]"
```

80.52 ConjugationCat1

`ConjugationCat1(` *R, S* `)`

When S is a normal subgroup of a group R form the semi-direct product $G = R \ltimes S$ to R and take this as the source, with R as the range. The tail and head homomorphisms are defined by $t(r,s) = r(\partial s)$, $h(r,s) = r$. In the example `h20` is the range, rather than the source.

```
gap> c5 := Subgroup( h20, [(1,2,3,4,5)] );;
gap> c5.name := "c5";;
gap> CC := ConjugationCat1( h20, c5 );
cat1-group [Perm(h20 |X c5) ==> h20]
gap> Cat1Print( CC );

cat1-group [Perm(h20 |X c5) ==> h20] :-
: source group has generators:
  [ ( 6, 7, 8, 9,10), ( 2, 3, 5, 4)( 7, 8,10, 9), (1,2,3,4,5) ]
:  range group has generators:
  [ (1,2,3,4,5), (2,3,5,4) ]
: tail homomorphism maps source generators to:
  [ ( 1, 2, 3, 4, 5), ( 2, 3, 5, 4), () ]
: head homomorphism maps source generators to:
  [ ( 1, 2, 3, 4, 5), ( 2, 3, 5, 4), ( 1, 2, 3, 4, 5) ]
: range embedding maps range generators to:
  [ ( 6, 7, 8, 9,10), ( 2, 3, 5, 4)( 7, 8,10, 9) ]
: kernel has generators:
  [ (1,2,3,4,5) ]
: boundary homomorphism maps generators of kernel to:
  [ ( 1, 2, 3, 4, 5) ]
: kernel embedding maps generators of kernel to:
  [ ( 1, 2, 3, 4, 5) ]
: associated crossed module is Crossed module [c5->h20]

gap> ct := CC.tail;;
gap> ch := CC.head;;
gap> CG := CC.source;;
gap> genCG := CG.generators;;
gap> x := genCG[2] * genCG[3];
( 1, 2, 4, 3 )( 7, 8,10, 9 )
gap> tx := Image( ct, x );
( 2, 3, 5, 4)
gap> hx := Image( ch, x );
( 1, 2, 4, 3)

gap> RecFields( CC );
[ "source", "range", "tail", "head", "embedRange", "kernel",
  "boundary", "embedKernel", "isDomain", "operations", "isCat1",
  "name", "xmod" ]
```

80.53 Operations for cat1-groups

Special operations defined for crossed modules are stored in the record structure `Cat1Ops` based on `DomainOps`. Every cat1-group `C` has `C.operations := Cat1Ops;`.

```
gap> RecFields( Cat1Ops );
[ "name", "operations", "Elements", "IsFinite", "Size", "=", "<",
  "in", "IsSubset", "Intersection", "Union", "IsParent", "Parent",
  "Difference", "Representative", "Random", "Print", "Actor",
  "InnerActor", "InclusionMorphism", "WhiteheadPermGroup" ]
```

Cat1-groups are considered equal if they have the same source, range, tail, head and embedding. The remaining functions are described below.

80.54 Size for cat1-groups

`Cat1Ops.Size(` *C* `)`

This function returns a two-element list containing the sizes of the source and range of `C`.

```
gap> Size( C );
 [ 20, 4 ]
```

80.55 Elements for cat1-groups

`Cat1Ops.Elements(` *C* `)`

This function returns the two-element list of lists of elements of the source and range of C.

```
gap> Elements( C );
[ [ (), (2,3,5,4), (2,4,5,3), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5),
    (1,2,4,3), (1,2,5,4), (1,3,4,2), (1,3)(4,5), (1,3,5,2,4),
    (1,3,2,5), (1,4,5,2), (1,4,3,5), (1,4)(2,3), (1,4,2,5,3),
    (1,5,4,3,2), (1,5,3,4), (1,5,2,3), (1,5)(2,4) ],
  [ (), (2,3,5,4), (2,4,5,3), (2,5)(3,4) ] ]
```

80.56 XModCat1

`XModCat1(` *C* `)`

This function acts as the functor from the category of cat1-groups to the category of crossed modules.

```
gap> XC := XModCat1( C );
Crossed module [ker([h20 ==> c4])->c4]
gap> XModPrint( XC );

Crossed module [ker([h20 ==> c4])->c4] :-
: Source group has parent ( h20 ) and has generators:
  [ (1,2,3,4,5) ]
: Range group has parent ( h20 ) and has generators:
  [ ( 2, 3, 5, 4) ]
: Boundary homomorphism maps source generators to:
```

```
  [ () ]
: Action homomorphism maps range generators to automorphisms:
  (2,3,5,4) --> { source gens --> [ (1,3,5,2,4) ] }
  This automorphism generates the group of automorphisms.
: Associated cat1-group = cat1-group [h20 ==> c4]
```

80.57 Cat1XMod

`Cat1XMod(` X `)`

This function acts as the functor from the category of crossed modules to the category of cat1-groups. A permutation representation of the semidirect product $R \ltimes S$ is constructed for G. See section 80.58 for a version where G is a semidirect product group. The example uses the crossed module `CX` constructed in section 80.6.

```
gap> CX;
Crossed module [k4->a4]
gap> CCX := Cat1XMod( CX );
cat1-group [a4.k4 ==> a4]
gap> Cat1Print( CCX );

cat1-group [a4.k4 ==> a4] :-
: source group has generators:
  [ (2,4,3)(5,6,7), (2,3,4)(6,7,8), (1,2)(3,4), (1,3)(2,4) ]
:  range group has generators:
  [ (1,2,3), (2,3,4) ]
: tail homomorphism maps source generators to:
  [ ( 1, 2, 3), ( 2, 3, 4), (), () ]
: head homomorphism maps source generators to:
  [ ( 1, 2, 3), ( 2, 3, 4), ( 1, 2)( 3, 4), ( 1, 3)( 2, 4) ]
: range embedding maps range generators to:
  [ ( 2, 4, 3)( 5, 6, 7), ( 2, 3, 4)( 6, 7, 8) ]
: kernel has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: boundary homomorphism maps generators of kernel to:
  [ (1,2)(3,4), (1,3)(2,4) ]
: kernel embedding maps generators of kernel to:
  [ (1,2)(3,4), (1,3)(2,4) ]
: associated crossed module is Crossed module [k4->a4]
```

80.58 SemidirectCat1XMod

`SemidirectCat1XMod(` X `)`

This function is similar to the previous one, but a permutation representation for $R \ltimes S$ is not constructed.

```
gap> Unbind( CX.cat1 );
gap> SCX := SemidirectCat1XMod( CX );
cat1-group [a4 |X k4 ==> a4]
```

```
gap> Cat1Print( SCX );

cat1-group [a4 |X k4 ==> a4] :-
: source group has generators:
  [ SemidirectProductElement( (1,2,3), GroupHomomorphismByImages( k4,
      k4, [(1,3)(2,4), (1,4)(2,3)], [(1,2)(3,4), (1,3)(2,4)] ), () ),
    SemidirectProductElement( (2,3,4), GroupHomomorphismByImages( k4,
      k4, [(1,4)(2,3), (1,2)(3,4)], [(1,2)(3,4), (1,3)(2,4)] ), () ),
    SemidirectProductElement( (), IdentityMapping( k4 ), (1,2)(3,4) ),
    SemidirectProductElement( (), IdentityMapping( k4 ), (1,3)(2,4) ) ]
:  range group has generators:
  [ (1,2,3), (2,3,4) ]
: tail homomorphism maps source generators to:
  [ (1,2,3), (2,3,4), (), () ]
: head homomorphism maps source generators to:
  [ (1,2,3), (2,3,4), (1,2)(3,4), (1,3)(2,4) ]
: range embedding maps range generators to:
  [ SemidirectProductElement( (1,2,3), GroupHomomorphismByImages( k4,
      k4, [(1,3)(2,4), (1,4)(2,3)], [(1,2)(3,4), (1,3)(2,4)] ), () ),
    SemidirectProductElement( (2,3,4), GroupHomomorphismByImages( k4,
      k4, [(1,4)(2,3), (1,2)(3,4)], [(1,2)(3,4), (1,3)(2,4)] ), () ) ]
: kernel has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: boundary homomorphism maps generators of kernel to:
  [ (1,2)(3,4), (1,3)(2,4) ]
: kernel embedding maps generators of kernel to:
  [ SemidirectProductElement( (), IdentityMapping( k4 ), (1,2)(3,4) ),
    SemidirectProductElement( (), IdentityMapping( k4 ), (1,3)(2,4) ) ]
: associated crossed module is Crossed module [k4->a4]
```

80.59 Cat1List

`Cat1List` is a list containing data on all cat1-structures on groups of size up to 47. The list is used by `Cat1Select` to construct these small examples of cat1-groups.

```
gap> Length( Cat1List );
198
gap> Cat1List[8];
[ 6, 2, [ (1,2), (2,3) ], "s3",
        [ [ [ (2,3), (2,3) ], "c3", "c2", [ (2,3), (2,3) ],
        [ (2,3), (2,3) ] ] ] ]
```

80.60 Cat1Select

`Cat1Select(` *size, [gpnum, num]* `)`

All cat-structures on groups of order up to 47 are stored in a list `Cat1List` and may be obtained from the list using this function. Global variables `Cat1ListMaxSize := 47` and `NumbersOfIsomorphismClasses` are also stored. The example illustrated is the first case in which $t \neq h$ and the associated conjugation crossed module is given by the normal subgroup c3 of s3.

```
gap> Cat1ListMaxSize;
47
gap> NumbersOfIsomorphismClasses[18];
5
gap> Cat1Select( 18 );
Usage:  Cat1Select( size, gpnum, num )
[ "c6c3", "c18", "d18", "s3c3", "c3^2|Xc2" ]

gap> Cat1Select( 18, 5 );
There are 4 cat1-structures for the group c3^2|Xc2.
[ [range generators], [tail.genimages], [head.genimages] ]  :-
[ [ (1,2,3), (4,5,6), (2,3)(5,6) ],  tail = head = identity mapping ]
[ [ (2,3)(5,6) ], "c3^2", "c2", [ (), (), (2,3)(5,6) ],
    [ (), (), (2,3)(5,6) ] ]
[ [ (4,5,6), (2,3)(5,6) ], "c3", "s3", [ (), (4,5,6), (2,3)(5,6) ],
    [ (), (4,5,6), (2,3)(5,6) ] ]
[ [ (4,5,6), (2,3)(5,6) ], "c3", "s3", [ (4,5,6),(4,5,6),(2,3)(5,6) ],
    [ (), (4,5,6), (2,3)(5,6) ] ]
Usage:  Cat1Select( size, gpnum, num )
Group has generators [ (1,2,3), (4,5,6), (2,3)(5,6) ]

gap> SC := Cat1Select( 18, 5, 4 );
cat1-group [c3^2|Xc2 ==> s3]
gap> Cat1Print( SC );

cat1-group [c3^2|Xc2 ==> s3] :-
: source group has generators:
  [ (1,2,3), (4,5,6), (2,3)(5,6) ]
:  range group has generators:
  [ (4,5,6), (2,3)(5,6) ]
: tail homomorphism maps source generators to:
  [ ( 4, 5, 6), ( 4, 5, 6), ( 2, 3)( 5, 6) ]
: head homomorphism maps source generators to:
  [ (), ( 4, 5, 6), ( 2, 3)( 5, 6) ]
: range embedding maps range generators to:
  [ ( 4, 5, 6), ( 2, 3)( 5, 6) ]
: kernel has generators:
  [ ( 1, 2, 3)( 4, 6, 5) ]
: boundary homomorphism maps generators of kernel to:
  [ ( 4, 6, 5) ]
: kernel embedding maps generators of kernel to:
  [ ( 1, 2, 3)( 4, 6, 5) ]

gap> XSC := XModCat1( SC );
Crossed module [c3->s3]
```

For each group G the first cat1-structure is the identity cat1-structure (id;id,id : G -> G) with trivial kernel. The corresponding crossed module has as boundary the inclusion map of the trivial subgroup.

```
gap> AC := Cat1Select( 12, 5, 1 );
cat1-group [a4 ==> a4]
```

80.61 Cat1Morphism

`Cat1Morphism(` *C, D, L* `)`

A morphism of cat1-groups is a pair of homomorphisms `[ sourceHom, rangeHom ]`, where `sourceHom`, `rangeHom` are respectively homomorphisms between the sources and ranges of C and D, which commute with the two tail homomorphisms with the two head homomorphisms and with the two embeddings.

In this implementation a morphism of cat1-groups `mu` is a record with fields:

`mu.source`,	the source cat1-group `C`,
`mu.range`,	the range cat1-group `D`,
`mu.sourceHom`,	a homomorphism from `C.source` to `D.source`,
`mu.rangeHom`,	a homomorphism from `C.range` to `D.range`,
`mu.isCat1Morphism`,	a Boolean flag, normally `true`,
`mu.operations`,	a special set of operations `Cat1MorphismOps`,
`mu.name`,	a concatenation of the names of `C` and `D`.

The function `Cat1Morphism` requires as parameters two cat1-groups and a two-element list containing the source and range homomorphisms. It sets up the required fields for `mu`, but does not check the axioms. The `IsCat1Morphism` function should be used to perform these checks. Note that the `Cat1MorphismPrint` function is needed to print out the morphism in detail.

```
gap> GCCX := CCX.source;
Perm(a4 |X k4)
gap> GAC := AC.source;
a4
gap> genGAC := GAC.generators;
[ (1,2,3), (2,3,4) ]
gap> im := Sublist( GCCX.generators, [1..2] );
[ (2,4,3)(5,6,7), (2,3,4)(6,7,8) ]

gap> musrc := GroupHomomorphismByImages( GAC, GCCX, genGAC, im );;
gap> murng := InclusionMorphism( a4, a4 );;
gap> mu := Cat1Morphism( AC, CCX, [ musrc, murng ] );
Morphism of cat1-groups <[a4 ==> a4]-->[Perm(a4 |X k4) ==> a4]>
```

80.62 IsCat1Morphism

`IsCat1Morphism(` *mu* `)`

This Boolean function checks that μ includes homomorphisms between the corresponding source and range groups, and that these homomorphisms commute with the pairs of tail and head homomorphisms.

```
gap> IsCat1Morphism( mu );
true
```

80.63 Cat1MorphismName

`Cat1MorphismName(` *mu* `)`

This function concatenates the names of the source and range of a morphism of cat1-groups.

```
gap> CCX.source.name := "a4.k4";; Cat1Name( CCX );
"[a4.k4 ==> a4]"
gap> Cat1MorphismName( mu );
"<[a4 ==> a4]-->[a4.k4 ==> a4]>"
```

80.64 Cat1MorphismPrint

`Cat1MorphismPrint(` *mu* `)`

This printing function for cat1-groups is one of the special functions in `Cat1MorphismOps`.

```
gap> Cat1MorphismPrint( mu );
Morphism of cat1-groups :=
: Source = cat1-group [a4 ==> a4]
:  Range = cat1-group [a4.k4 ==> a4]
: Source homomorphism maps source generators to:
  [ (2,4,3)(5,6,7), (2,3,4)(6,7,8) ]
: Range homomorphism maps range generators to:
  [ (1,2,3), (2,3,4) ]
```

80.65 Operations for morphisms of cat1-groups

Special operations defined for morphisms of cat1-groups are stored in the record structure `Cat1MorphismOps` which is based on `MappingOps`. Every morphism of cat1-groups `mor` has field `mor.operations` set equal to `Cat1MorphismOps;`.

```
gap> IsMonomorphism( mu );
true
gap> IsEpimorphism( mu );
false
gap> IsIsomorphism( mu );
false
gap> IsEndomorphism( mu );
false
gap> IsAutomorphism( mu );
false
```

80.66 Cat1MorphismSourceHomomorphism

`Cat1MorphismSourceHomomorphism ( C, D, phi )`

Given a homomorphism from the source of `C` to the source of `D`, this function defines the corresponding cat1-group morphism.

```
gap> GSC := SC.source;;
gap> homsrc := GroupHomomorphismByImages( a4, GSC,
>               [(1,2,3),(2,3,4)],[(4,5,6),(4,6,5)]);;
gap> musrc := Cat1MorphismSourceHomomorphism( AC, SC, homsrc );
Morphism of cat1-groups <[a4 ==> a4]-->[c3^2|Xc2 ==> s3]>
gap> IsCat1Morphism( musrc );
true
gap> Cat1MorphismPrint( musrc );
Morphism of cat1-groups :=
: Source = cat1-group [a4 ==> a4]
:  Range = cat1-group [c3^2|Xc2 ==> s3]
: Source homomorphism maps source generators to:
   [ (4,5,6), (4,6,5) ]
: Range homomorphism maps range generators to:
   [ (4,5,6), (4,6,5) ]
```

80.67 ReverseCat1

`ReverseCat1(` *C* `)`

The reverse of a cat1-group is an isomorphic cat1-group with the same source, range and embedding, but with the tail and head interchanged (see [AW97], section 2).

```
gap> revCC := ReverseCat1( CC );
cat1-group [h20 |X c5 ==> h20]
```

80.68 ReverseIsomorphismCat1

`ReverseIsomorphismCat1(` *C* `)`

```
gap> revmu := ReverseIsomorphismCat1( CC );
Morphism of cat1-groups
   <[Perm(h20 |X c5) ==> h20]-->[h20 |X c5 ==> h20]>
gap> IsCat1Morphism( revmu );
true
```

80.69 Cat1MorphismXModMorphism

`Cat1MorphismXModMorphism(` *mor* `)`

If `C1, C2` are the cat1-groups produced from `X1, X2` by the function `Cat1XMod`, then for any `mor : X1 -> X2` there is an associated `mu :`
`C1 -> C2`. The result is stored as `mor.cat1Morphism`.

```
gap> CX.Cat1 := CCX;;
gap> CSX := Cat1XMod( SX );
cat1-group [Perm(sl(2,3) |X q8) ==> sl(2,3)]
gap> mor;
Morphism of crossed modules <[q8->sl(2,3)] >-> [k4->a4]>
gap> catmor := Cat1MorphismXModMorphism( mor );
Morphism of cat1-groups
```

```
    <[Perm(sl(2,3) |X q8) ==> sl(2,3)]-->[Perm(a4 |X k4) ==> a4]>
gap> IsCat1Morphism( catmor );
true
gap> Cat1MorphismPrint( catmor );
Morphism of cat1-groups :=
: Source = cat1-group [Perm(sl(2,3) |X q8) ==> sl(2,3)]
:  Range = cat1-group [Perm(a4 |X k4) ==> a4]
: Source homomorphism maps source generators to:
  [ (5,6)(7,8), (5,7)(6,8), (2,3,4)(6,7,8), (1,2)(3,4), (1,3)(2,4) ]
: Range homomorphism maps range generators to:
  [ (1,2)(3,4), (1,3)(2,4), (2,3,4) ]
```

80.70 XModMorphismCat1Morphism

`XModMorphismCat1Morphism` (*mu*)

If `X1,X2` are the two crossed modules produced from `C1,C2` by the function `XModCat1`, then for any `mu : C1 -> C2` there is an associated morphism of crossed modules from `X1` to `X2`. The result is stored as `mu.xmodMorphism`.

```
gap> mu;
Morphism of cat1-groups <[a4 ==> a4]-->[a4.k4 ==> a4]>
gap> xmu := XModMorphismCat1Morphism( mu );
Morphism of crossed modules <[a4->a4] >-> [k4->a4]>
```

80.71 CompositeMorphism for cat1-groups

`Cat1MorphismOps.CompositeMorphism(` *mu1,mu2* `)`

Morphisms $\mu_1 : C \to D$ and $\mu_2 : D \to E$ have a composite $\mu = \mu_2 \circ \mu_1 : C \to E$ whose source and range homomorphisms are the composites of those of μ_1 and μ_2. The example corresponds to that in 80.45.

```
gap> psi;
Morphism of crossed modules <[c4->q8] >-> [c4->q8]>
gap> inc;
Morphism of crossed modules <[c4->q8] >-> [q8->sl(2,3)]>
gap> mupsi := Cat1MorphismXModMorphism( psi );
Morphism of cat1-groups
    <[Perm(q8 |X c4) ==> q8]-->[Perm(q8 |X c4) ==> q8]>
gap> muinc := Cat1MorphismXModMorphism( inc );
Morphism of cat1-groups
    <[Perm(q8 |X c4) ==> q8]-->[Perm(sl(2,3) |X q8) ==> sl(2,3)]>
gap> mucomp := Cat1MorphismOps.CompositeMorphism( mupsi, muinc );
Morphism of cat1-groups
    <[Perm(q8 |X c4) ==> q8]-->[Perm(sl(2,3) |X q8) ==> sl(2,3)]>
gap> muxcomp := Cat1MorphismXModMorphism( xcomp );;
gap> mucomp = muxcomp;
true
```

80.72 IdentitySubCat1

`IdentitySubCat1(` *C* `)`

Every cat1-group $\mathcal{C}$ has an identity sub-cat1-group whose source and range are the identity subgroups of the source and range of $\mathcal{C}$.

```
gap> IdentitySubCat1( SC );
cat1-group [Id[c3^2|Xc2 ==> s3]]
```

80.73 SubCat1

`SubCat1(` *C, H* `)`

When `H` is a subgroup of `C.source` and the restrictions of `C.tail` and `C.head` to `H` have a common image, these homomorphisms determine a sub-cat1-group of `C`.

```
gap> d20 := Subgroup( h20, [ (1,2,3,4,5), (2,5)(3,4) ] );;
gap> subC := SubCat1( C, d20 );
cat1-group [Sub[h20 ==> c4]]
gap> Cat1Print( subC );

cat1-group [Sub[h20 ==> c4]] :-
: source group has generators:
  [ (1,2,3,4,5), (2,5)(3,4) ]
:  range group has generators:
  [ ( 2, 5)( 3, 4) ]
: tail homomorphism maps source generators to:
  [ (), ( 2, 5)( 3, 4) ]
: head homomorphism maps source generators to:
  [ (), ( 2, 5)( 3, 4) ]
: range embedding maps range generators to:
  [ ( 2, 5)( 3, 4) ]
: kernel has generators:
  [ (1,2,3,4,5) ]
: boundary homomorphism maps generators of kernel to:
  [ () ]
: kernel embedding maps generators of kernel to:
  [ ( 1, 2, 3, 4, 5) ]
```

80.74 InclusionMorphism for cat1-groups

`InclusionMorphism(` *S, C* `)`

This function constructs the inclusion morphism `S  ->  C` of a sub-cat1-group `S` of a cat1-group `C`.

```
gap> InclusionMorphism( subC, C );
Morphism of cat1-groups <[Sub[h20 ==> c4]]-->[h20 ==> c4]>
```

80.75 NormalSubCat1s

`NormalSubCat1s(` C `)`

This function takes pairs of normal subgroups from the source and range of $\mathcal{C}$ and constructs a normal sub-cat1-group whenever the axioms are satisfied.

```
gap> NormalSubCat1s( SC );
[ cat1-group [Sub[c3^2|Xc2 ==> s3]] ,
  cat1-group [Sub[c3^2|Xc2 ==> s3]] ,
  cat1-group [Sub[c3^2|Xc2 ==> s3]] ,
  cat1-group [Sub[c3^2|Xc2 ==> s3]]  ]
```

80.76 AllCat1s

`AllCat1s(` G `)`

By a *cat1-structure* on G we mean a cat1-group $\mathcal{C}$ where R is a subgroup of G and e is the inclusion map. For such a structure to exist, G must contain a normal subgroup S with $G/S \cong R$. Furthermore, since t, h are respectively the identity and zero maps on S, we require $R \cap S = \{1_G\}$. This function uses `EndomorphismClasses( G, 3 )` (see 80.134, 80.136) to construct idempotent endomorphisms of `G` as potential tails and heads. A backtrack procedure then tests to see which pairs of idempotents give cat1-groups. A non-documented function `AreIsomorphicCat1s` is called in order that the function returns representatives for isomorphism classes of cat1-structures on `G`. See [AW97] for all cat1-structures on groups of order up to 30.

```
gap> AllCat1s( a4 );
There are 1 endomorphism classes.
Calculating idempotent endomorphisms.
# idempotents mapping to lattice class representatives
[ 1, 0, 1, 0, 1 ]
Isomorphism class 1
: kernel of tail = [ "2x2" ]
:    range group = [ "3" ]
Isomorphism class 2
: kernel of tail = [ "1" ]
:    range group = [ "A4" ]
[ cat1-group [a4 ==> a4.H3] , cat1-group [a4 ==> a4] ]
```

The first class has range `c3` and kernel `k4`. The second class contails all cat1-groups $\mathcal{C} = (\alpha^{-1}; \alpha, \alpha : G \to G)$ where α is an automorphism of G.

80.77 About derivations and sections

The Whitehead monoid $\mathrm{Der}(\mathcal{X})$ of $\mathcal{X}$ was defined in [Whi48] to be the monoid of all *derivations* from R to S, that is the set of all maps $R \to S$, with composition $\circ$, satisfying

$$\begin{array}{lrcl} \textbf{Der 1:} & \chi(qr) & = & (\chi q)^r\,(\chi r) \\ \textbf{Der 2:} & (\chi_1 \circ \chi_2)(r) & = & (\chi_1 r)(\chi_2 r)(\chi_1 \partial \chi_2 r). \end{array}$$

The zero map is the identity for this composition. Invertible elements in the monoid are called *regular*. The Whitehead group of $\mathcal{X}$ is the group of regular derivations in $\mathrm{Der}(\mathcal{X})$. In section 80.113 the *actor* of $\mathcal{X}$ is defined as a crossed module whose source and range are permutation representations of the Whitehead group and the automorphism group of $\mathcal{X}$.

The construction for cat1-groups equivalent to the derivation of a crossed module is the *section*. The monoid of sections of $\mathcal{C}$ is the set of group homomorphisms $\xi : R \to G$, with composition $\circ$, satisfying:

$$\begin{array}{ll} \textbf{Sect 1:} & t\xi = \mathrm{id}_R, \\ \textbf{Sect 2:} & (\xi_1 \circ \xi_2)(r) = (\xi_2 r)(eh\xi_2 r)^{-1}(\xi_1 h \xi_2 r). \end{array}$$

The embedding e is the identity for this composition, and $h(\xi_1 \circ \xi_2) = (h\xi_1)(h\xi_2)$. A section is *regular* when $h\xi$ is an automorphism and, of course, the group of regular sections is isomorphic to the Whitehead group.

Derivations are stored like group homomorphisms by specifying the images of a generating set. Images of the remaining elements may then be obtained using axiom **Der 1**. The function `IsDerivation` is automatically called to check that this procedure is well-defined.

```
gap> X1;
Crossed module [c5->PermAut(c5)]
gap> chi1 := XModDerivationByImages( X1, [ () ] );
XModDerivationByImages( PermAut(c5), c5, [ (1,2,4,3) ], [ () ] )
gap> IsDerivation( chi1 );
true
```

A derivation is stored as a record `chi` with fields:

`chi.source`,	the range group R of $\mathcal{X}$,
`chi.range`,	the source group S of $\mathcal{X}$,
`chi.generators`,	a fixed generating set for R,
`chi.genimages`,	the chosen images of the generators,
`chi.xmod`,	the crossed module $\mathcal{X}$,
`chi.operations`,	special set of operations `XModDerivationByImagesOps`,
`chi.isDerivation`,	a boolean flag, normally `true`.

Sections *are* group homomorphisms, and are stored as such, but with a modified set of operations `Cat1SectionByImagesOps` which includes a special `.Print` function to display the section in the manner shown below. Functions `SectionDerivation` and `DerivationSection` convert derivations to sections, and vice-versa, calling `Cat1XMod` and `XModCat1` automatically.

The equation $\xi r = (er)(\chi r)$ determines a section ξ of $\mathcal{C}$, given a derivation χ of $\mathcal{X}$, and conversely.

```
gap> xi1 := SectionDerivation( chi1 );
Cat1SectionByImages( PermAut(c5), Perm(PermAut(c5) |X c5),
    [ (1,2,4,3) ], [ (2,3,5,4) ] )
gap> xi1.cat1;
cat1-group [Perm(PermAut(c5) |X c5) ==> PermAut(c5)]
```

There are two functions to determine all the elements of the Whitehead group and the Whitehead monoid of $\mathcal{X}$, namely `RegularDerivations` and `AllDerivations`. If the whole monoid is needed at some stage, then the latter function should be used. A field `D = X.derivations` is created which stores all the required information:

`D.areDerivations`,	a boolean flag, normally `true`,
`D.isReg`,	`true` when only the regular derivations are known,
`D.isAll`,	`true` when all the derivations have been found,
`D.generators`,	a *copy* of `R.generators`,
`D.genimageList`,	a list of `.genimages` lists for the derivations,
`D.regular`,	the number of regular derivations (if known),
`D.xmod`,	the crossed module $\mathcal{X}$,
`D.operations`,	a special set of operations `XModDerivationsOps`.

Using our standard example `X1` we find that there are just five derivations, all of them regular, so the associated group is cyclic of size 5.

```
gap> RegularDerivations( X1 );
RegularDerivations record for crossed module [c5->PermAut(c5)],
: 5 regular derivations, others not found.
gap> AllDerivations( X1 );
AllDerivations record for crossed module [c5->PermAut(c5)],
: 5 derivations found but unsorted.
gap> DerivationsSorted( X1 );
true
gap> imder1 := X1.derivations.genimageList;
[ [()], [(1,2,3,4,5)], [(1,3,5,2,4)], [(1,4,2,5,3)], [(1,5,4,3,2)] ]
```

The functions `RegularSections` and `AllSections` perform corresponding tasks for a cat1-group. Two strategies for calculating derivations and sections are implemented, see [AW97]. The default method for `AllDerivations` is to search for all possible sets of images using a backtracking procedure, and when all the derivations are found it is not known which are regular. The function `DerivationsSorted` sorts the `.genImageList` field, placing the regular ones at the top of the list and adding the `.regular` field. The default method for `AllSections( C )` computes all endomorphisms on the range group `R` of `C` as possibilities for the composite $h\xi$. A backtrack method then finds possible images for such a section. When either the set of derivations or the set of sections already exists, the other set is computed using `SectionDerivation` or `DerivationSection`.

```
gap> CX1 := Cat1XMod( X1 );
cat1-group [Perm(PermAut(c5) |X c5) ==> PermAut(c5)]
gap> CX1.source.name := "Hol(c5)";; Cat1Name( CX1 );
gap> RegularSections( CX1 );
RegularSections record for cat1-group [Hol(c5) ==> PermAut(c5)],
: 5 regular sections, others not found.
gap> CX1.sections.genimageList;
```

```
[ [(2,3,5,4)], [(1,2,4,3)], [(1,3,2,5)], [(1,4,5,2)], [(1,5,3,4)] ]
```

The derivation images and the composition table may be listed as follows.

```
gap> chi2 := XModDerivationByImages( X1, imder1[2] );
XModDerivationByImages( PermAut(c5), c5, [(1,2,4,3)], [(1,2,3,4,5)] )
gap> DerivationImage( chi2, (1,4)(2,3) );
( 1, 4, 2, 5, 3)
gap> DerivationImages( chi2 );
[ 1, 2, 3, 4 ]
gap> PrintList( DerivationTable( X1 ) );
[ 1, 1, 1, 1 ]
[ 1, 2, 3, 4 ]
[ 1, 3, 5, 2 ]
[ 1, 4, 2, 5 ]
[ 1, 5, 4, 3 ]
gap> PrintList( WhiteheadGroupTable( X1 ) );
[ 1, 2, 3, 4, 5 ]
[ 2, 3, 4, 5, 1 ]
[ 3, 4, 5, 1, 2 ]
[ 4, 5, 1, 2, 3 ]
[ 5, 1, 2, 3, 4 ]
```

Each χ or ξ determines endomorphisms of $R, S, G, \mathcal{X}$ and $\mathcal{C}$, namely:

$$\begin{array}{rcll}
\rho & : & R \to R, & r \mapsto r(\partial\chi r) = h\xi r, \\
\sigma & : & S \to S, & s \mapsto s(\chi\partial s), \\
\gamma & : & G \to G, & g \mapsto (eh\xi tg)(\xi tg^{-1})g(ehg^{-1})(\xi hg), \\
(\sigma,\rho) & : & \mathcal{X} \to \mathcal{X}, & \\
(\gamma,\rho) & : & \mathcal{C} \to \mathcal{C}. &
\end{array}$$

When these endomorphisms are automorphisms, the derivation is regular. When the boundary of $\mathcal{X}$ is the zero map, both σ and ρ are identity homomorphisms, and every derivation is regular, which is the case in this example.

```
gap> sigma2 := SourceEndomorphismDerivation( chi2 );
GroupHomomorphismByImages( c5, c5, [ (1,2,3,4,5) ], [ (1,2,3,4,5) ] )
gap> rho2 := RangeEndomorphismDerivation( chi2 );
GroupHomomorphismByImages( PermAut(c5), PermAut(c5), [ (1,2,4,3) ],
   [ (1,2,4,3) ] )
gap> xi2 := SectionDerivation( chi2 );;
gap> gamma2 := SourceEndomorphismSection( xi2 );
GroupHomomorphismByImages( Hol(c5), Hol(c5), [(2,3,5,4),(1,2,3,4,5)],
   [(2,3,5,4),(1,2,3,4,5)] )
gap> mor2 := XModMorphism( X1, X1, [sigma2,rho2] );
Morphism of crossed modules <[c5->PermAut(c5)] >-> [c5->PermAut(c5)]>
gap> mu2 := Cat1Morphism( CX1, CX1, [gamma2,rho2] );
Morphism of cat1-groups <[Hol(c5) ==> PermAut(c5)]-->
   [Hol(c5) ==> PermAut(c5)]>
```

80.78 XModDerivationByImages

`XModDerivationByImages(` *X, im* `)`

This function takes a list of images in `S = X.source` for the generators of `R = X.range` and constructs a map $\chi : R \to S$ which is then tested to see whether the axioms of a derivation are satisfied.

```
gap> XSC;
Crossed module [c3->s3]
gap> imchi := [ (1,2,3)(4,6,5), (1,2,3)(4,6,5) ];;
gap> chi := XModDerivationByImages( XSC, imchi );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ (1,2,3)(4,6,5), (1,2,3)(4,6,5) ] )
```

80.79 IsDerivation

`IsDerivation(` *X, im* `)`

`IsDerivation(` *chi* `)`

This function may be called in two ways, and tests that the derivation given by the images of its generators is well-defined.

```
gap> im0 := [ (1,3,2)(4,5,6), () ];;
gap> IsDerivation( XSC, im0 );
true
```

80.80 DerivationImage

`DerivationImage(` *chi, r* `)`

This function returns $\chi(r) \in S$ when χ is a derivation.

```
gap> DerivationImage( chi, (4,6,5) );
(1,3,2)(4,5,6)
```

80.81 DerivationImages

`DerivationImages(` *chi* `)`

All the images of the elements of R are found using `DerivationImage` and their positions in `S.elements` is returned as a list.

```
gap> XSC.source.elements;
[ (), ( 1, 2, 3)( 4, 6, 5), ( 1, 3, 2)( 4, 5, 6) ]
gap> DerivationImages(chi);
[ 1, 2, 3, 2, 3, 1 ]
```

80.82 InnerDerivation

`InnerDerivation(` *X, s* `)`

When S, R are respectively the source and range of X, each $s \in S$ defines a derivation $\eta_s : R \to S, r \mapsto s^r s^{-1}$. These *inner derivations* are often called *principal derivations* in the literature.

```
gap> InnerDerivation( XSC, (1,2,3)(4,6,5) );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ (), (1,2,3)(4,6,5) ] )
```

80.83 ListInnerDerivations

ListInnerDerivations(*X*)

This functions applies InnerDerivation to every element of X.source and outputs a list of genimages for the resulting derivations. This list is stored as .innerImageList in the derivations record.

```
gap> PrintList( ListInnerDerivations( XSC ) );
[ (), () ]
[ (), ( 1, 2, 3)( 4, 6, 5) ]
[ (), ( 1, 3, 2)( 4, 5, 6) ]
```

80.84 Operations for derivations

The operations record for derivations is XModDerivationByImagesOps.

```
gap> RecFields( chi.operations );
[ "name", "operations", "IsMapping", "IsInjective", "IsSurjective",
  "IsBijection", "IsHomomorphism", "IsMonomorphism", "IsEpimorphism",
  "IsIsomorphism", "IsEndomorphism", "IsAutomorphism", "=", "<", "*",
  "/", "mod", "Comm", "^", "ImageElm", "ImagesElm", "ImagesSet",
  "ImagesSource", "ImagesRepresentative", "PreImageElm",
  "PreImagesSet", "PreImagesRange", "PreImagesRepresentative",
  "PreImagesElm", "CompositionMapping", "PowerMapping",
  "IsGroupHomomorphism", "KernelGroupHomomorphism",
  "IsFieldHomomorphism", "KernelFieldHomomorphism",
  "InverseMapping", "Print", "IsRegular" ]
```

80.85 Cat1SectionByImages

Cat1SectionByImages(*C, im*)

This function takes a list of images in G = C.source for the generators of R = C.range and constructs a homomorphism $\xi : R \to G$ which is then tested to see whether the axioms of a section are satisfied.

```
gap> SC;
cat1-group [c3^2|Xc2 ==> s3]
gap> imxi := [ (1,2,3), (1,2)(4,6) ];;
gap> xi := Cat1SectionByImages( SC, imxi );
Cat1SectionByImages( s3, c3^2|Xc2, [ (4,5,6), (2,3)(5,6) ],
   [ (1,2,3), (1,2)(4,6) ] )
```

80.86 IsSection

`IsSection(` *C, im* `)`

`IsSection(` *xi* `)`

This function may be called in two ways, and tests that the section given by the images of its generators is well-defined.

```
gap> im0 := [ (1,2,3), (2,3)(4,5) ];;
gap> IsSection( SC, im0 );
false
```

80.87 IsRegular for Crossed Modules

`IsRegular(` *chi* `)`

This function tests a derivation or a section to see whether it is invertible in the Whitehead monoid.

```
gap> IsRegular( chi );
false
gap> IsRegular( xi );
false
```

80.88 Operations for sections

The operations record for sections is `Cat1SectionByImagesOps`.

```
gap> RecFields( xi.operations );
[ "name", "operations", "IsMapping", "IsInjective", "IsSurjective",
  "IsBijection", "IsHomomorphism", "IsMonomorphism", "IsEpimorphism",
  "IsIsomorphism", "IsEndomorphism", "IsAutomorphism", "=", "<", "*",
  "/", "mod", "Comm", "^", "ImageElm", "ImagesElm", "ImagesSet",
  "ImagesSource","ImagesRepresentative", "PreImageElm", "PreImagesElm",
  "PreImagesSet", "PreImagesRange", "PreImagesRepresentative",
  "CompositionMapping", "PowerMapping", "InverseMapping",
  "IsGroupHomomorphism", "CoKernel", "KernelGroupHomomorphism",
  "MakeMapping", "Print", "IsRegular" ]
```

80.89 RegularDerivations

`RegularDerivations(` *X [,"back"*or *"cat1"]* `)`

By default, this function uses a backtrack search to find all the regular derivations of $\mathcal{X}$. The result is stored in a derivations record. The alternative strategy, for which "cat1"option should be specified is to calculate the regular sections of the associated cat1-group first, and convert these to derivations.

```
gap> regXSC := RegularDerivations( XSC );
RegularDerivations record for crossed module [c3->s3],
: 6 regular derivations, others not found.
gap> PrintList( regXSC.genimageList );
```

```
[ (), () ]
[ (), ( 1, 2, 3)( 4, 6, 5) ]
[ (), ( 1, 3, 2)( 4, 5, 6) ]
[ ( 1, 3, 2)( 4, 5, 6), () ]
[ ( 1, 3, 2)( 4, 5, 6), ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 3, 2)( 4, 5, 6), ( 1, 3, 2)( 4, 5, 6) ]
gap> RecFields( regXSC );
[ "areDerivations", "isReg", "isAll", "genimageList", "operations",
  "xmod", "generators", "regular" ]
```

80.90 AllDerivations

`AllDerivations(` *X [,"back"*or *"cat1"]* `)`

This function calculates all the derivations of $\mathcal{X}$ and overwrites any existing subfields of `X.derivations`.

```
gap> allXSC := AllDerivations( XSC );
AllDerivations record for crossed module [c3->s3],
: 9 derivations found but unsorted.
```

80.91 DerivationsSorted

`DerivationsSorted(` *D* `)`

This function tests the derivations in the derivation record `D` to see which are regular; sorts the list `D.genimageList`, placing the regular images first; and stores the number of regular derivations in `D.regular`. The function returns `true` on successful completion.

```
gap> DerivationsSorted( allXSC );
true
gap> PrintList( allXSC.genimageList );
[ (), () ]
[ (), ( 1, 2, 3)( 4, 6, 5) ]
[ (), ( 1, 3, 2)( 4, 5, 6) ]
[ ( 1, 3, 2)( 4, 5, 6), () ]
[ ( 1, 3, 2)( 4, 5, 6), ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 3, 2)( 4, 5, 6), ( 1, 3, 2)( 4, 5, 6) ]
[ ( 1, 2, 3)( 4, 6, 5), () ]
[ ( 1, 2, 3)( 4, 6, 5), ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 2, 3)( 4, 6, 5), ( 1, 3, 2)( 4, 5, 6) ]
```

80.92 DerivationTable

`DerivationTable(` *D* `)`

The function `DerivationImages` in 80.81 is applied to each derivation in the current derivations record and a list of positions of images in S is returned.

```
gap> PrintList( DerivationTable( allXSC ) );
[ 1, 1, 1, 1, 1, 1 ]
[ 1, 1, 1, 2, 2, 2 ]
```

```
[ 1, 1, 1, 3, 3, 3 ]
[ 1, 3, 2, 1, 3, 2 ]
[ 1, 3, 2, 2, 1, 3 ]
[ 1, 3, 2, 3, 2, 1 ]
[ 1, 2, 3, 1, 2, 3 ]
[ 1, 2, 3, 2, 3, 1 ]
[ 1, 2, 3, 3, 1, 2 ]
```

80.93 AreDerivations

`AreDerivations(` *D* `)`

This function checks that the record `D` has the correct fields for a derivations record (regular or all).

```
gap> AreDerivations( regXSC );
true
```

80.94 RegularSections

`RegularSections(` *C [,"endo"*or *"xmod"]* `)`

By default, this function computes the set of idempotent automorphisms from $R \to R$ and takes these as possible choices for $h\xi$. A backtrack procedure then calculates possible images for such a section. The result is stored in a sections record `C.sections` with fields similar to those of a serivations record. The alternative strategy, for which "xmod"option should be specified is to calculate the regular derivations of the associated crossed module first, and convert the resulting derivations to sections.

```
gap> Unbind( XSC.derivations );
gap> regSC := RegularSections( SC );
RegularSections record for cat1-group [c3^2|Xc2 ==> s3],
: 6 regular sections, others not found.
```

80.95 AllSections

`AllSections(` *C [,"endo"*or *"xmod"]* `)`

By default, this function computes the set of idempotent endomorphisms from $R \to R$ (see sections 80.134, 80.136) and takes these as possible choices for the composite homomorphism $h\xi$. A backtrack procedure then calculates possible images for such a section. This function calculates all the sections of $\mathcal{C}$ and overwrites any existing subfields of `C.sections`.

```
gap> allSC := AllSections( SC );
AllSections record for cat1-group [c3^2|Xc2 ==> s3],
: 6 regular sections,  3 irregular ones found.
gap> RecFields( allSC );
[ "areSections", "isReg", "isAll", "regular", "genimageList",
  "generators", "cat1", "operations" ]
gap> PrintList( allSC.genimageList );
[ ( 4, 5, 6), ( 2, 3)( 5, 6) ]
[ ( 4, 5, 6), ( 1, 3)( 4, 5) ]
```

```
[ ( 4, 5, 6), ( 1, 2)( 4, 6) ]
[ ( 1, 3, 2)( 4, 6, 5), ( 2, 3)( 5, 6) ]
[ ( 1, 3, 2)( 4, 6, 5), ( 1, 3)( 4, 5) ]
[ ( 1, 3, 2)( 4, 6, 5), ( 1, 2)( 4, 6) ]
[ ( 1, 2, 3), ( 2, 3)( 5, 6) ]
[ ( 1, 2, 3), ( 1, 2)( 4, 6) ]
[ ( 1, 2, 3), ( 1, 3)( 4, 5) ]
gap> allXSC := AllDerivations( XSC, "cat1" );
AllDerivations record for crossed module [c3->s3],
: 6 regular derivations,  3 irregular ones found.
```

80.96 AreSections

`AreSections(` *S* `)`

This function checks that the record `S` has the correct fields for a sections record (regular or all).

```
gap> AreSections( allSC );
true
```

80.97 SectionDerivation

`SectionDerivation(` *D, i* `)`

This function converts a derivation of `X` to a section of the associated cat1-group `C`. This function is inverse to `DerivationSection`. In the following examples we note that `allXSC` has been obtained using `allSC`, so the derivations and sections correspond in the same order.

```
gap> chi8 := XModDerivationByImages( XSC, allXSC.genimageList[8] );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ ( 1,2,3)(4,6,5), (1,2,3)(4,6,5) ] )
gap> xi8 := SectionDerivation( chi8 );
GroupHomomorphismByImages( s3, c3^2|Xc2,
    [ (4,5,6), (2,3)(5,6) ], [ (1,2,3), (1,2)(4,6) ] )
```

80.98 DerivationSection

`DerivationSection(` *C, xi* `)`

This function converts a section of `C` to a derivation of the associated crossed module `X`. This function is inverse to `SectionDerivation`.

```
gap> xi4 := Cat1SectionByImages( SC, allSC.genimageList[4] );
Cat1SectionByImages( s3, c3^2|Xc2, [ (4,5,6), (2,3)(5,6) ],
   [ (1,3,2)(4,6,5), (2,3)(5,6) ] )
gap> chi4 := DerivationSection( xi4 );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ (1,3,2)(4,5,6), () ] )
```

80.99 CompositeDerivation

`CompositeDerivation(` *chi, chj* `)`

This function applies the Whitehead product to two derivations and returns the composite. In the example, derivations `chi4`, `chi8` correspond to sections `xi4` and `xi8`.

```
gap> chi48 := CompositeDerivation( chi4, chi8 );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ (1,2,3)(4,6,5), (1,3,2)(4,5,6) ] )
```

80.100 CompositeSection

`CompositeSection(` *xi, xj* `)`

This function applies the Whitehead composition to two sections and returns the composite.

```
gap> xi48 := CompositeSection( xi4, xi8 );
Cat1SectionByImages( s3, c3^2|Xc2, [ (4,5,6), (2,3)(5,6) ],
   [ ( 1,2,3), (1,3)(4,5) ] )
gap> SectionDerivation( chi48 ) = xi48;
true
```

80.101 WhiteheadGroupTable

`WhiteheadGroupTable(` *X* `)`

This function applies `CompositeDerivation` to all pairs of regular derivations, producing the Whitehead group multiplication table. A field `.groupTable` is added to `D`.

```
gap> WGT := WhiteheadGroupTable( XSC );; PrintList( WGT );
returning existing ALL derivations
[ 1, 2, 3, 4, 5, 6 ]
[ 2, 3, 1, 5, 6, 4 ]
[ 3, 1, 2, 6, 4, 5 ]
[ 4, 6, 5, 1, 3, 2 ]
[ 5, 4, 6, 2, 1, 3 ]
[ 6, 5, 4, 3, 2, 1 ]
```

80.102 WhiteheadMonoidTable

`WhiteheadMonoidTable(` *X* `)`

The derivations of `X` form a monoid with the first derivation as identity. This function applies `CompositeDerivation` to all pairs of derivations and produces the multiplication table as a list of lists. A field `.monoidTable` is added to `D`. In our example there are 9 derivations and the three irregular ones, numbers 7,8,9, are all left zeroes.

```
gap> WMT := WhiteheadMonoidTable( XSC );; PrintList(WMT );
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 2, 3, 1, 5, 6, 4, 9, 7, 8 ]
[ 3, 1, 2, 6, 4, 5, 8, 9, 7 ]
[ 4, 6, 5, 1, 3, 2, 7, 9, 8 ]
```

```
[ 5, 4, 6, 2, 1, 3, 9, 8, 7 ]
[ 6, 5, 4, 3, 2, 1, 8, 7, 9 ]
[ 7, 7, 7, 7, 7, 7, 7, 7, 7 ]
[ 8, 8, 8, 8, 8, 8, 8, 8, 8 ]
[ 9, 9, 9, 9, 9, 9, 9, 9, 9 ]
```

80.103 InverseDerivations

`InverseDerivations(` *X, i* `)`

When `T[i]` is a regular derivation, this function returns the position `j` such that `T[j]` is the inverse of `T[i]` in the Whitehead group. When `T[i]` is not regular, a list of values `j` is returned such that the inverse semigroup condition $xyx = x,\ yxy = y$ is satisfied, where $x = T[i], y = T[j]$. Notice that derivation 8 has order 3 and derivation 15 as inverse.

```
gap> inv4 := InverseDerivations( chi4 );
[ 4 ]
gap> inv8 := InverseDerivations( chi8 );
[ 7, 8, 9 ]
```

80.104 ListInverseDerivations

`ListInverseDerivations(` *X* `)`

This function applies `InverseDerivations` to all the derivations. A field `.inverses` is added to `D`.

```
gap> inv := ListInverseDerivations( XSC );
[ [ 1 ], [ 3 ], [ 2 ], [ 4 ], [ 5 ], [ 6 ],
  [ 7, 8, 9 ], [ 7, 8, 9 ], [ 7, 8, 9 ] ]
```

80.105 SourceEndomorphismDerivation

`SourceEndomorphismDerivation(` *chi* `)`

Each derivation χ determines an endomorphism σ of `S` such that $\sigma s = s(\chi \partial s)$. This construction defines a homomorphism from the Whitehead group to $Aut(S)$ which forms the action homomorphism of the Whitehead crossed module described in section 80.116.

```
gap> sigma8 := SourceEndomorphismDerivation( chi8 );
GroupHomomorphismByImages( c3, c3, [ (1,2,3)(4,6,5) ], [ () ] )
gap> sigma4 := SourceEndomorphismDerivation( chi4 );
GroupHomomorphismByImages( c3, c3, [ (1,2,3)(4,6,5) ],
   [ (1,3,2)(4,5,6) ] )
```

80.106 TableSourceEndomorphismDerivations

`TableSourceEndomorphismDerivations(` *X* `)`

Applying `SourceEndomorphismDerivation` to every derivation produces a list of endomorphisms of `S = X.source`. This function returns a list of `.genimages` for these endomorphisms. Note that, in this example, `S = c3` and the irregular derivations produce zero maps.

```
gap> TSE := TableSourceEndomorphismDerivations( XSC );;
gap> PrintList( TSE );
[ ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 2, 3)( 4, 6, 5) ]
[ ( 1, 3, 2)( 4, 5, 6) ]
[ ( 1, 3, 2)( 4, 5, 6) ]
[ ( 1, 3, 2)( 4, 5, 6) ]
[ () ]
[ () ]
[ () ]
```

80.107 RangeEndomorphismDerivation

`RangeEndomorphismDerivation(` *chi* `)`

Each derivation χ determines an endomorphism ρ of `R` such that $\rho r = r(\partial\chi r)$. This construction defines a homomorphism from the Whitehead group to $Aut(R)$.

```
gap> rho8 := RangeEndomorphismDerivation( chi8 );
GroupHomomorphismByImages( s3, s3, [ (4,5,6), (2,3)(5,6) ],
  [ (), (2,3)(4,6) ] )
gap> rho4 := RangeEndomorphismDerivation( chi4 );
GroupHomomorphismByImages( s3, s3, [ (4,5,6), (2,3)(5,6) ],
  [ (4,6,5), (2,3)(5,6) ] )
```

80.108 TableRangeEndomorphismDerivations

`TableRangeEndomorphismDerivations(` *X* `)`

Applying `RangeEndomorphismDerivation` to every derivation produces a list of endomorphisms of `R = X.range`. This function returns a list of `.genimages` for these endomorphisms. Note that, in this example, the 3 irregular derivations map onto the 3 cyclic subgroups of order 2.

```
gap> TRE := TableRangeEndomorphismDerivations( XSC );;
gap> PrintList( TRE );
[ (4,5,6), (2,3)(5,6) ]
[ (4,5,6), (2,3)(4,5) ]
[ (4,5,6), (2,3)(4 6) ]
[ (4,6,5), (2,3)(5,6) ]
[ (4,6,5), (2,3)(4,5) ]
[ (4,6,5), (2,3)(4,6) ]
[ (), (2,3)(5,6) ]
[ (), (2,3)(4,6) ]
[ (), (2,3)(4,5) ]
```

80.109 XModEndomorphismDerivation

`XModEndomorphismDerivation(` *chi* `)`

The endomorphisms `sigma4`, `rho4` together determine a pair which may be used to construct an endomorphism of $\mathcal{X}$. When the derivation is regular, the resulting morphism is an automorphism, and this construction determines a homomorphism from the Whitehead group to the automorphism group of `X`.

```
gap> phi4 := XModEndomorphismDerivation( chi4 );
Morphism of crossed modules <[c3->s3]->[c3->s3]>
```

80.110 SourceEndomorphismSection

`SourceEndomorphismSection(` *xi* `)`

Each section ξ determines an endomorphism γ of `G` such that

$$\gamma g = (eh\xi tg)(\xi tg^{-1})g(ehg^{-1})(\xi hg).$$

```
gap> gamma4 := SourceEndomorphismSection( xi4 );
GroupHomomorphismByImages( c3^2|Xc2, c3^2|Xc2,
  [ (1,2,3), (4,5,6), (2,3)(5,6) ], [ (1,3,2), (4,6,5), (2,3)(5,6) ] )
```

80.111 RangeEndomorphismSection

`RangeEndomorphismSection(` *xi* `)`

Each derivation ξ determines an endomorphism ρ of `R` such that $\rho r = h\xi r$.

```
gap> rho4 := RangeEndomorphismSection( xi4 );
GroupHomomorphismByImages( s3, s3, [ (4,5,6), (2,3)(5,6) ],
  [ (4,6,5), (2,3)(5,6) ] )
```

80.112 Cat1EndomorphismSection

`Cat1EndomorphismSection(` *xi* `)`

The endomorphisms `gamma4`, `rho4` together determine a pair which may be used to construct an endomorphism of $\mathcal{C}$. When the derivation is regular, the resulting morphism is an automorphism, and this construction determines a homomorphism from the Whitehead group to the automorphism group of `C`.

```
gap> psi4 := Cat1EndomorphismSection( xi4 );
Morphism of cat1-groups <[c3^2|Xc2 ==> s3]-->[c3^2|Xc2 ==> s3]>
```

80.113 About actors

The *actor* of $\mathcal{X}$ is a crossed module $(\Delta : \mathcal{W}(\mathcal{X}) \to \mathrm{Aut}(\mathcal{X}))$ which was shown by Lue and Norrie, in [Nor87] and [Nor90] to give the automorphism object of a crossed module $\mathcal{X}$. The source of the actor is a permutation representation W of the Whitehead group of regular derivations and the range is a permutation representation A of the automorphism group $\mathrm{Aut}(\mathcal{X})$ of $\mathcal{X}$.

An automorphism (σ, ρ) of `X` acts on the Whitehead monoid by $\chi^{(\sigma,\rho)} = \sigma^{-1}\chi\rho$, and this action determines the action for the actor.

In fact the four groups R, S, W, A, the homomorphisms between them and the various actions, form five crossed modules:

$\mathcal{X}$	:	$S \to R$	the initial crossed module,
$\mathcal{W}(\mathcal{X})$	:	$S \to W$	the Whitehead crossed module of $\mathcal{X}$,
$\mathcal{L}(\mathcal{X})$	:	$S \to A$	the Lue crossed module of $\mathcal{X}$,
$\mathcal{N}(\mathcal{X})$	:	$R \to A$	the Norrie crossed module of $\mathcal{X}$, and
$\mathrm{Act}(\mathcal{X})$	:	$W \to A$	the actor crossed module of $\mathcal{X}$.

These 5 crossed modules, together with the evaluation $W \times R \to S,\quad (\chi, r) \mapsto \chi r$, form a crossed square:

```
S ------ WX ------> W
:  \                :
:     \             :
X        LX        ActX
:           \       :
:              \    :
V                 \ V
R ------ NX ------> A
```

in which pairs of boundaries or identity mappings provide six morphisms of crossed modules. In particular, the boundaries of `WX` and `NX` form the *inner morphism* of `X`, mapping source elements to inner derivations and range elements to inner automorphisms. The image of `X` under this morphism is the *inner actor* of `X`, while the kernel is the *centre* of `X`.

In the example which follows, using the usual `(X1 : c5 -> Aut(c5))`, `Act(X1)` is isomorphic to `X1` and to `LX1` while the Whitehead and Norrie boundaries are identity homomorphisms.

```
gap> X1;
Crossed module [c5->PermAut(c5)]
gap> WGX1 := WhiteheadPermGroup( X1 );
WG([c5->PermAut(c5)])
gap> WGX1.generators;
[ (1,2,3,4,5) ]
gap> AX1 := AutomorphismPermGroup( X1 );
PermAut([c5->PermAut(c5)])
gap> AX1.generators;
[ (1,2,4,3) ]
gap> XModMorphismAutoPerm( X1, AX1.generators[1] );
Morphism of crossed modules <[c5->PermAut(c5)] >-> [c5->PermAut(c5)]>
```

```
gap> WX1 := Whitehead( X1 );
Crossed module Whitehead[c5->PermAut(c5)]
gap> NX1 := Norrie( X1 );
Crossed module Norrie[c5->PermAut(c5)]
gap> LX1 := Lue( X1 );
Crossed module Lue[c5->PermAut(c5)]
gap> ActX1 := Actor( X1 );;
gap> XModPrint( ActX1);
Crossed module Actor[c5->PermAut(c5)] :-
: Source group WG([c5->PermAut(c5)]) has generators:
  [ (1,2,3,4,5) ]
: Range group has parent ( PermAut(c5)xPermAut(PermAut(c5)) )
    and has generators:  [ (1,2,4,3) ]
: Boundary homomorphism maps source generators to:
  [ () ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,4,3) --> { source gens --> [ (1,3,5,2,4) ] }
  This automorphism generates the group of automorphisms.

gap> InActX1 := InnerActor( X1 );
Crossed module Actor[c5->PermAut(c5)]
gap> InActX1 = ActX1;
true
gap> InnerMorphism( X1 );
Morphism of crossed modules
    <[c5->PermAut(c5)] >-> Actor[c5->PermAut(c5)]>
gap> Centre( X1 );
Crossed module Centre[c5->PermAut(c5)]
```

All of these constructions are stored in a sub-record `X1.actorSquare`.

80.114 ActorSquareRecord

`ActorSquareRecord(` X `)`

`ActorSquareRecord(` C `)`

This function creates a new field `.actorSquare` for the crossed module or cat1-group, initially containing `.isActorSquare := true` and `.xmod` or `.cat1` as appropriate. Components for the actor of `X` or `C` are stored here when constructed.

```
gap> ActorSquareRecord( X1 );
rec(
  isActorSquare := true,
  xmod := Crossed module [c5->PermAut(c5)],
  WhiteheadPermGroup := WG([c5->PermAut(c5)]),
  automorphismPermGroup := PermAut([c5->PermAut(c5)]),
  Whitehead := Crossed module Whitehead[c5->PermAut(c5)],
  Norrie := Crossed module Norrie[c5->PermAut(c5)],
  Lue := Crossed module Lue[c5->PermAut(c5)],
```

```
actor := Crossed module Actor[c5->PermAut(c5)],
innerMorphism :=  Morphism of crossed modules
    <[c5->PermAut(c5)] >-> Actor[c5->PermAut(c5)]>,
innerActor := Crossed module Actor[c5->PermAut(c5)] )
```

80.115 WhiteheadPermGroup

`WhiteheadPermGroup(` X `)`

This function first calls `WhiteheadGroupTable`, see 80.101. These lists are then converted to permutations, producing a permutation group which is effectively a regular representation of the group. A field `.WhiteheadPermGroup` is added to `X.actorSquare` and a field `.genpos` is added to `D = X.derivations`. The latter is a list of the positions in `D.genimageList` corresponding to the chosen generating elements. The group is given the name `WG(<name of X>)`.

For an example, we return to the crossed module `XSC = [c3->s3]` obtained from the cat1-group `SC` in section 80.60 which has Whitehead group and automorphism group isomorphic to `s3`.

```
gap> WG := WhiteheadPermGroup( XSC );
WG([c3->s3])
gap> XSC.derivations.genpos;
[ 2, 4 ]
gap> Elements( WG );
[ (), (1,2,3)(4,6,5), (1,3,2)(4,5,6), (1,4)(2,5)(3,6),
  (1,5)(2,6)(3,4), (1,6)(2,4)(3,5) ]
```

80.116 Whitehead crossed module

`Whitehead(` X `)`

This crossed module has the source of `X` as source, and the Whitehead group `WX` as range. The boundary maps each element to the inner derivation which it defines. The action uses `SourceEndomorphismDerivation`.

```
gap> WXSC := Whitehead( XSC );
Crossed module Whitehead[c3->s3]
gap> XModPrint( WXSC );
Crossed module Whitehead[c3->s3] :-
: Source group has parent ( c3^2|Xc2 ) and has generators:
  [ (1,2,3)(4,6,5) ]
: Range group = WG([c3->s3]) has generators:
  [ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
: Boundary homomorphism maps source generators to:
  [ (1,3,2)(4,5,6) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3)(4,6,5) --> { source gens --> [ (1,2,3)(4,6,5) ] }
  (1,4)(2,5)(3,6) --> { source gens --> [ (1,3,2)(4,5,6) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.117 AutomorphismPermGroup for crossed modules

`XModOps.AutomorphismPermGroup(` *X* `)`

This function constructs a permutation group `PermAut(X)` isomorphic to the group of automorphisms of the crossed module `X`. First the automorphism groups of the source and range of `X` are obtained and `AutomorphismPair` used to obtain permutation representations of these. The direct product of these permutation groups is constructed, and the required automorphism group is a subgroup of this direct product. The result is stored as `X.automorphismPermGroup` which has fields defining the various embeddings and projections.

```
gap> autXSC := AutomorphismPermGroup( XSC );
PermAut([c3->s3])
gap> autXSC.projsrc;
GroupHomomorphismByImages( PermAut([c3->s3]), PermAut(c3),
[ (5,6,7), (1,2)(3,4)(6,7) ], [ (), (1,2) ] )
gap> autXSC.projrng;
GroupHomomorphismByImages( PermAut([c3->s3]), PermAut(s3),
[ (5,6,7), (1,2)(3,4)(6,7) ], [ (3,4,5), (1,2)(4,5) ] )
gap> autXSC.embedSourceAuto;
GroupHomomorphismByImages( PermAut(c3), PermAut(c3)xPermAut(s3),
    [ (1,2) ], [ (1,2) ] )
gap> autXSC.embedRangeAuto;
GroupHomomorphismByImages( PermAut(s3), PermAut(c3)xPermAut(s3),
[ (3,5,4), (1,2)(4,5) ], [ (5,7,6), (3,4)(6,7) ] )
gap> autXSC.autogens;
[ [ GroupHomomorphismByImages( c3, c3, [ (1,2,3)(4,6,5) ],
        [ (1,2,3)(4,6,5) ] ), GroupHomomorphismByImages( s3, s3,
        [ (4,5,6), (2,3)(5,6) ], [ (4,5,6), (2,3)(4,5) ] ) ],
  [ GroupHomomorphismByImages( c3, c3, [ (1,2,3)(4,6,5) ],
        [ (1,3,2)(4,5,6) ] ), GroupHomomorphismByImages( s3, s3,
        [ (4,5,6), (2,3)(5,6) ], [ (4,6,5), (2,3)(5,6) ] ) ] ]
```

80.118 XModMorphismAutoPerm

`XModMorphismAutoPerm(` *X, perm* `)`

Given the isomorphism between the automorphism group of `X` and its permutation representation `PermAut(X)`, an element of the latter determines an automorphism of `X`.

```
gap> XModMorphismAutoPerm( XSC, (1,2)(3,4)(6,7) );
Morphism of crossed modules <[c3->s3] >-> [c3->s3]>
```

80.119 ImageAutomorphismDerivation

`ImageAutomorphismDerivation(` *mor, chi* `)`

An automorphism (σ,ρ) of `X` acts on the left on the Whitehead monoid by ${}^{(\sigma,\rho)}\chi = \sigma\chi\rho^{-1}$. This is converted to a right action on the `WhiteheadPermGroup`. In the example we see that `phi4` maps `chi8` to `chi9`.

```
gap> chi8im := ImageAutomorphismDerivation( phi4, chi8 );
XModDerivationByImages( s3, c3, [ (4,5,6), (2,3)(5,6) ],
   [ (1,2,3)(4,6,5), (1,3,2)(4,5,6) ] )
gap> Position( allXSC.genimageList, chi8im.genimages );
9
```

80.120 Norrie crossed module

`Norrie(` *X* `)`

This crossed module has the range of `X` as source and the automorphism permutation group of `X` as range.

```
gap> NXSC := Norrie( XSC );
Crossed module Norrie[c3->s3]
gap> XModPrint( NXSC );

Crossed module Norrie[c3->s3] :-
: Source group has parent ( c3^2|Xc2 ) and has generators:
  [ (4,5,6), (2,3)(5,6) ]
: Range group has parent ( PermAut(c3)xPermAut(s3) ) and has
    generators: [ (5,6,7), (1,2)(3,4)(6,7) ]
: Boundary homomorphism maps source generators to:
  [ (5,7,6), (1,2)(3,4)(6,7) ]
: Action homomorphism maps range generators to automorphisms:
  (5,6,7) --> { source gens --> [ (4,5,6), (2,3)(4,5) ] }
  (1,2)(3,4)(6,7) --> { source gens --> [ (4,6,5), (2,3)(5,6) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.121 Lue crossed module

`Lue(` *X* `)`

This crossed module has the source of `X` as source, and the automorphism permutation group of `X` as range.

```
gap> LXSC := Lue( XSC );
Crossed module Lue[c3->s3]
gap> XModPrint( LXSC );

Crossed module Lue[c3->s3] :-
: Source group has parent ( c3^2|Xc2 ) and has generators:
  [ (1,2,3)(4,6,5) ]
: Range group has parent ( PermAut(c3)xPermAut(s3) ) and has
    generators: [ (5,6,7), (1,2)(3,4)(6,7) ]
: Boundary homomorphism maps source generators to:
  [ (5,6,7) ]
: Action homomorphism maps range generators to automorphisms:
  (5,6,7) --> { source gens --> [ (1,2,3)(4,6,5) ] }
  (1,2)(3,4)(6,7) --> { source gens --> [ (1,3,2)(4,5,6) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.122 Actor crossed module

`Actor(` *X* `)`

The actor of a crossed module `X` is a crossed module `Act(X)` which has the Whitehead group (of regular derivations) as source group and the automorphism group `PermAut(X)` of `X` as range group. The boundary of `Act(X)` maps each derivation to the automorphism provided by `XModEndomorphismDerivation`. The action of an automorphism on a derivation is that provided by `ImageAutomorphismDerivation`.

```
gap> ActXSC := Actor( XSC );
Crossed module Actor[c3->s3]
gap> XModPrint( ActXSC );

Crossed module Actor[c3->s3] :-
: Source group WG([c3->s3]) has generators:
  [ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
: Range group has parent ( PermAut(c3)xPermAut(s3) ) and has
    generators: [ (5,6,7), (1,2)(3,4)(6,7) ]
: Boundary homomorphism maps source generators to:
  [ (5,7,6), (1,2)(3,4)(6,7) ]
: Action homomorphism maps range generators to automorphisms:
  (5,6,7) --> { source gens --> [ (1,2,3)(4,6,5), (1,6)(2,4)(3,5) ] }
  (1,2)(3,4)(6,7) -->
              { source gens --> [ (1,3,2)(4,5,6), (1,4)(2,5)(3,6) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.123 InnerMorphism for crossed modules

`InnerMorphism(` *X* `)`

The boundary maps of `WX` and `NX` form a morphism from `X` to its actor.

```
gap> innXSC := InnerMorphism( XSC );
Morphism of crossed modules <[c3->s3] >-> Actor[c3->s3]>
gap> XModMorphismPrint( innXSC );
Morphism of crossed modules :-
: Source = Crossed module [c3->s3] with generating sets:
  [ (1,2,3)(4,6,5) ]
  [ (4,5,6), (2,3)(5,6) ]
:  Range = Crossed module Actor[c3->s3]
      with generating sets:
  [ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
  [ (5,6,7), (1,2)(3,4)(6,7) ]
: Source Homomorphism maps source generators to:
  [ (1,3,2)(4,5,6) ]
: Range Homomorphism maps range generators to:
  [ (5,7,6), (1,2)(3,4)(6,7) ]
: isXModMorphism? true
```

80.124 Centre for crossed modules

`XModOps.Centre(` X `)`

The kernel of the inner morphism `X -> ActX` is called the centre of `X`, generalising the centre of a group G, which is the kernel of $G \to Aut(G),\ g \mapsto (h \mapsto h^g)$. In this example the centre is trivial.

```
gap> ZXSC := Centre( XSC );
Crossed module Centre[c3->s3]
```

80.125 InnerActor for crossed modules

`InnerActor(` X `)`

The inner actor of `X` is the image of the inner morphism.

```
gap> InnActXSC := InnerActor( XSC );
Crossed module InnerActor[c3->s3]
gap> XModPrint( InnActXSC );

Crossed module InnerActor[c3->s3] :-
: Source group has parent ( WG([c3->s3]) ) and has generators:
  [ (1,3,2)(4,5,6) ]
: Range group has parent ( PermAut(c3)xPermAut(s3) ) and has
    generators: [ (5,7,6), (1,2)(3,4)(6,7) ]
: Boundary homomorphism maps source generators to:
  [ (5,6,7) ]
: Action homomorphism maps range generators to automorphisms:
  (5,7,6) --> { source gens --> [ (1,3,2)(4,5,6) ] }
  (1,2)(3,4)(6,7) --> { source gens --> [ (1,2,3)(4,6,5) ] }
  These 2 automorphisms generate the group of automorphisms.
```

80.126 Actor for cat1-groups

`Actor(` C `)`

The actor of a cat1-group $\mathcal{C}$ is the cat1-group associated to the actor crossed module of the crossed module $\mathcal{X}$ associated to $\mathcal{C}$. Its range is the automorphism group A and its source is $A \ltimes W$ where W is the Whitehead group.

```
gap> ActSC := Actor( SC );;
gap> Cat1Print( ActSC );
cat1-group Actor[c3^2|Xc2 ==> s3] :-
: source group has generators:
  [ (4,6,5), (2,3)(5,6), (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
:  range group has generators:
  [ (5,6,7), (1,2)(3,4)(6,7) ]
: tail homomorphism maps source generators to:
  [ (5,6,7), (1,2)(3,4)(6,7), (), () ]
: head homomorphism maps source generators to:
  [ (5,6,7), (1,2)(3,4)(6,7), (5,7,6), (1,2)(3,4)(6,7) ]
```

```
: range embedding maps range generators to:
  [ (4,6,5), (2,3)(5,6) ]
: kernel has generators:
  [ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
: boundary homomorphism maps generators of kernel to:
  [ (5,7,6), (1,2)(3,4)(6,7) ]
: kernel embedding maps generators of kernel to:
  [ (1,2,3)(4,6,5), (1,4)(2,5)(3,6) ]
: associated crossed module is Crossed module Actor[c3->s3]
```

80.127 About induced constructions

A morphism of crossed modules $(\sigma, \rho) : \mathcal{X}_1 \to \mathcal{X}_2$ factors uniquely through an induced crossed module $\rho_* \mathcal{X}_1 = (\delta : \rho_* S_1 \to R_2)$. Similarly, a morphism of cat1-groups factors through an induced cat1-group. Calculation of induced crossed modules of $\mathcal{X}$ also provides an algebraic means of determining the homotopy 2-type of homotopy pushouts of the classifying space of $\mathcal{X}$. For more background from algebraic topology see references in [BH78], [BW95], [BW96]. Induced crossed modules and induced cat1-groups also provide the building blocks for constructing pushouts in the categories **XMod** and **Cat1**.

Data for the cases of algebraic interest is provided by a conjugation crossed module $\mathcal{X} = (\partial : S \to R)$ and a homomorphism ι from R to a third group Q. The output from the calculation is a crossed module $\iota_* \mathcal{X} = (\delta : \iota_* S \to Q)$ together with a morphism of crossed modules $\mathcal{X} \to \iota_* \mathcal{X}$. When ι is a surjection with kernel K then $\iota_* S = [S, K]$ (see [BH78]). When ι is an inclusion the induced crossed module may be calculated using a copower construction [BW95] or, in the case when R is normal in Q, as a coproduct of crossed modules ([BW96], not yet implemented). When ι is neither a surjection nor an inclusion, ι is written as the composite of the surjection onto the image and the inclusion of the image in Q, and then the composite induced crossed module is constructed.

Other functions required by the induced crossed module construction include a function to produce a common transversal for the left and right cosets of a subgroup (see 80.150 and 80.149). Also, modifications to some of the Tietze transformation routines in `fptietze.g` are required. These have yet to be released as part of the GAP3 library and so are made available in this package in file `felsch.g`, but are not documented here.

As a simple example we take for $\mathcal{X}$ the conjugation crossed module $(\partial : c4 \to d8)$ and for ι the inclusion of $d8$ in $d16$. The induced crossed module has $c4 \times c4$ as source.

```
gap> d16 := DihedralGroup( 16 );  d16.name := "d16";;
Group( (1,2,3,4,5,6,7,8), (2,8)(3,7)(4,6) )
gap> d8 := Subgroup( d16, [ (1,3,5,7)(2,4,6,8), (1,3)(4,8)(5,7) ] );;
gap> c4 := Subgroup( d8, [ (1,3,5,7)(2,4,6,8) ] );;
gap> d8.name :=  "d8";;  c4.name := "c4";;
gap> DX := ConjugationXMod( d8, c4 );
Crossed module [c4->d8]
gap> iota := InclusionMorphism( d8, d16 );;
gap> IDXincl := InducedXMod( DX, iota );
Action of RQ on generators of I :-
  (1,2,3,4,5,6,7,8) : (1,4)(2,3)
  (2,8)(3,7)(4,6) : (1,2)(3,4)
#I  Protecting the first 1 generators.
#I  there are 2 generators and 3 relators of total length 12
partitioning the generators: [ [ 2 ], [ 1 ] ]
Simplified presentation for I :-
#I  generators: [ fI.1, fI.3 ]
#I  relators:
#I  1.  4  [ 1, 1, 1, 1 ]
#I  2.  4  [ 2, 2, 2, 2 ]
#I  3.  4  [ 2, -1, -2, 1 ]
```

```
 I has Size: 16
****************

Group is abelian
factor 1 is abelian with invariants: [ 4 ]
factor 2 is abelian with invariants: [ 4 ]
Image of I has index 4 in RQ and is generated by :
[ ( 1, 3, 5, 7)( 2, 4, 6, 8), ( 1, 7, 5, 3)( 2, 8, 6, 4) ]

gap> XModPrint( IDXincl );
Crossed module [i*(c4)->d16] :-
: Source group i*(c4) has generators:
  [ ( 1, 2, 4, 7)( 3, 5, 8,11)( 6, 9,12,14)(10,13,15,16),
    ( 1, 3, 6,10)( 2, 5, 9,13)( 4, 8,12,15)( 7,11,14,16) ]
: Range group = d16 has generators:
  [ (1,2,3,4,5,6,7,8), (2,8)(3,7)(4,6) ]
: Boundary homomorphism maps source generators to:
  [ ( 1, 3, 5, 7)( 2, 4, 6, 8), ( 1, 7, 5, 3)( 2, 8, 6, 4) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2,3,4,5,6,7,8) --> { source gens -->
[ ( 1,10, 6, 3)( 2,13, 9, 5)( 4,15,12, 8)( 7,16,14,11),
  ( 1, 7, 4, 2)( 3,11, 8, 5)( 6,14,12, 9)(10,16,15,13) ] }
  (2,8)(3,7)(4,6) --> { source gens -->
[ ( 1, 7, 4, 2)( 3,11, 8, 5)( 6,14,12, 9)(10,16,15,13),
  ( 1,10, 6, 3)( 2,13, 9, 5)( 4,15,12, 8)( 7,16,14,11) ] }
  These 2 automorphisms generate the group of automorphisms.
: Kernel of the crossed module has generators:
  [ ( 1, 5,12,16)( 2, 8,14,10)( 3, 9,15, 7)( 4,11, 6,13) ]
: Induced XMod from Crossed module [c4->d8]  with source morphism:
  [ (1,3,5,7)(2,4,6,8) ]
   --> [ ( 1, 2, 4, 7)( 3, 5, 8,11)( 6, 9,12,14)(10,13,15,16) ]
```

In some of the sections which follow the output is very lengthy and so has been pruned.

80.128 InducedXMod

`InducedXMod(` *X, iota* `)`

`InducedXMod(` *Q, P, M* `)`

This function requires as data a conjugation crossed module $\mathcal{X} = (\partial : M \to P)$ and a homomorphism $\iota : P \to Q$. This data may be specified using either of the two forms shown, where the latter form required $Q \geq P \geq M$.

In the first example, ι is a surjection from `d8` to `k4`.

```
gap> d8gen := d8.generators;
[ (1,3,5,7)(2,4,6,8), (1,3)(4,8)(5,7) ]
gap> k4gen := k4.generators;
[ (1,2)(3,4), (1,3)(2,4) ]
```

```
gap> DX;
Crossed module [c4->d8]
gap> iota := GroupHomomorphismByImages( d8, k4, d8gen, k4gen );;
gap> IDXsurj := InducedXMod( DX, iota );
Crossed module [c4/ker->k4]
gap> XModPrint( IDXsurj );
Crossed module [c4/ker->k4] :-
: Source group c4/ker has generators:
  [ (1,2,3,4) ]
: Range group has parent ( s4 ) and has generators:
  [ (1,2)(3,4), (1,3)(2,4) ]
: Boundary homomorphism maps source generators to:
  [ ( 1, 2)( 3, 4) ]
: Action homomorphism maps range generators to automorphisms:
  (1,2)(3,4) --> { source gens --> [ (1,2,3,4) ] }
  (1,3)(2,4) --> { source gens --> [ (1,4,3,2) ] }
  These 2 automorphisms generate the group of automorphisms.
: Induced XMod from Crossed module [c4->d8]  with source morphism:
  [ (1,3,5,7)(2,4,6,8) ]
   --> [ (1,2,3,4) ]
```

In a second example we take `(c3 -> s3)` as the initial crossed module and `s3 -> s4` as the inclusion. The induced group turns out to be the special linear group `sl(2,3)`.

```
gap> s3 := Subgroup( s4, [ (2,3), (1,2,3) ] );;
gap> c3 := Subgroup( s3, [ (1,2,3) ] );
gap> s3.name := "s3";;  c3.name := "c3";;
gap> InducedXMod( s4, s3, c3 );

Action of RQ on generators of I :-
  (1,2,3,4) : (1,7,6,3)(2,8,5,4)
  (1,2) : (1,2)(3,4)(5,8)(6,7)
#I  Protecting the first 1 generators.
#I  there are 2 generators and 3 relators of total length 12
Simplified presentation for I :-
#I  generators: [ fI.1, fI.5 ]
#I  relators:
#I  1.  3  [ 2, 2, 2 ]
#I  2.  3  [ 1, 1, 1 ]
#I  3.  6  [ 2, -1, -2, 1, -2, -1 ]

 I has Size: 24
****************
 Searching Solvable Groups Library:
 GroupId =
rec(
  catalogue := [ 24, 14 ],
  names := [ "SL(2,3)" ],
  size := 24 )
```

```
Image of I has index 2 in RQ and is generated by :
[ (1,2,3), (1,2,4), (1,4,3), (2,3,4) ]

Crossed module [i*(c3)->s4]
```

80.129 AllInducedXMods

`AllInducedXMods(` *Q* `)`

This function calculates `InducedXMod( Q, P, M )` where `P` runs over all conjugacy classes of subgroups of `Q` and `M` runs over all normal subgroups of `P`.

```
gap> AllInducedXMods( d8 );
        ⋮
Number of induced crossed modules calculated = 11
```

80.130 InducedCat1

`InducedCat1(` *C, iota* `)`

When $\mathcal{C}$ is the induced cat1-group associated to $\mathcal{X}$ the induced cat1-group may be obtained by construction the induced crossed module and then using the `Cat1XMod` function. An experimental, alternative procedure is to calculate the induced cat1-group $\iota_* G = G *_R Q$ directly. This has been implemented for the case when $\mathcal{C} = (e; t, h : G \to R)$ and $\iota : R \to Q$ is an inclusion.

The output from the calculation is a cat1-group $\mathcal{C}_* = (e_*; t_*, h_* : \iota_* G \to Q)$ together with a morphism of crossed modules $\mathcal{C} \to \mathcal{C}_*$.

In the example an induced cat1-group is constructed whose associated crossed module has source $c4 \times c4$ and range $d16$, so the source of the cat1-group is $d16 \ltimes (c4 \times c4)$.

```
gap> CDX := Cat1XMod( DX );
cat1-group [Perm(d8 |X c4) ==> d8]
gap> inc := InclusionMorphism( d8, d16 );;
gap> ICDX := InducedCat1( CDX, inc );
       ⋮
new perm group size 256
cat1-group <ICG([Perm(d8 |X c4) ==> d8])>
gap> XICDX := XModCat1( ICDX );
Crossed module [ker(<ICG([Perm(d8 |X c4) ==> d8])>)->d16]
gap> AbelianInvariants( XICDX.source );
[ 4, 4 ]
```

80.131 About utilities

By a utility function we mean a GAP3 function which is:

- needed by other functions in this package,
- not (as far as we know) provided by the standard GAP3 library,
- more suitable for inclusion in the main library than in this package.

The first two utilities give particular group homomorphisms, `InclusionMorphism(H,G)` and `ZeroMorphism(G,H)`. We often prefer

```
gap> incs3 := InclusionMorphism( s3, s3 );
IdentityMapping( s3 )
gap> incs3.genimages;
[ (1,2), (2,3) ]
```

to `IdentityMapping(s3)` because the latter does not provide the fields `.generators` and the `.genimages` which many of the functions in this package expect homomorphisms to possess.

The second set of utilities involve endomorphisms and automorphisms of groups. For example:

```
gap> end8 := EndomorphismClasses( d8 );;
gap> RecFields( end8 );
[ "isDomain", "isEndomorphismClasses", "areNonTrivial", "classes",
  "intersectionFree", "group", "latticeLength", "latticeReps" ]
gap> Length( end8.classes );
11
gap> end8.classes[3];
rec(
  quotient := d8.Q3,
  projection := OperationHomomorphism( d8, d8.Q3 ),
  autoGroup := Group( IdentityMapping( d8.Q3 ) ),
  rangeNumber := 2,
  isomorphism := GroupHomomorphismByImages( d8.Q3, d8.H2, [ (1,2) ],
    [ (1,5)(2,6)(3,7)(4,8) ] ),
  conj := [ () ] )
gap> innd8 := InnerAutomorphismGroup( d8 );
Inn(d8)
gap> innd8.generators;
[ InnerAutomorphism( d8, (1,3,5,7)(2,4,6,8) ),
  InnerAutomorphism( d8, (1,3)(4,8)(5,7) ) ]
gap> IsAutomorphismGroup( innd8 );
true
```

The third set of functions construct isomorphic pairs of groups, where a faithful permutation representation of a given type of group is constructed. Types covered include finitely presented groups, groups of automorphisms and semidirect products. A typical pair record includes the following fields:

.type	the given group G,
.perm	the permutation representation P,
.t2p	the isomorphism $G \to P$,
.p2t	the inverse isomorphism $P \to G$,
.isTypePair	a boolean flag, normally `true`.

The inner automorphism group of the dihedral group `d8` is isomorphic to `k4`:

```
gap> Apair := AutomorphismPair( innd8 );
rec(
  auto := Inn(d8),
  perm := PermInn(d8),
  a2p := OperationHomomorphism( Inn(d8), PermInn(d8) ),
  p2a := GroupHomomorphismByImages( PermInn(d8), Inn(d8),
    [ (1,3), (2,4) ],
    [ InnerAutomorphism( d8, (1,3,5,7)(2,4,6,8) ),
      InnerAutomorphism( d8, (1,3)(4,8)(5,7) ) ] ),
  isAutomorphismPair := true )
gap> IsAutomorphismPair( Apair );
true
```

The final set of functions deal with lists of subsets of `[1..n]` and construct systems of distinct and common representatives using simple, non-recursive, combinatorial algorithms. The latter function returns two lists: the set of representatives, and a permutation of the subsets of the second list. It may also be used to provide a common transversal for sets of left and right cosets of a subgroup H of a group G, although a greedy algorithm is usually quicker.

```
gap> L := [ [1,4], [1,2], [2,3], [1,3], [5] ];;
gap> DistinctRepresentatives( L );
[ 4, 2, 3, 1, 5 ]
gap> M := [ [2,5], [3,5], [4,5], [1,2,3], [1,2,3] ];;
gap> CommonRepresentatives( L, M );
[ [ 4, 1, 3, 1, 5 ], [ 3, 5, 2, 4, 1 ] ]
gap> CommonTransversal( s4, c3 );
[ (), (3,4), (2,3), (1,3)(2,4), (1,2)(3,4), (2,4), (1,4), (1,4)(2,3) ]
```

80.132 InclusionMorphism

`InclusionMorphism(` *H, G* `)`

This gives the inclusion map of a subgroup H of a group G. In the case that $H = G$ the `IdentityMapping(G)` is returned, with fields `.generators` and `.genimages` added.

```
gap> s4 := Group( (1,2,3,4), (1,2) );; s4.name:="s4";;
gap> a4 := Subgroup( s4, [ (1,2,3), (2,3,4) ] );; a4.name:="a4";;
gap> InclusionMorphism( a4, s4 );
GroupHomomorphismByImages( a4, s4, [ (1,2,3), (2,3,4) ],
  [ (1,2,3), (2,3,4) ] )
```

80.133 ZeroMorphism

`ZeroMorphism(` *G, H* `)`

This gives the zero map from G to the identity subgroup of H.

```
gap> ZeroMorphism( s4, a4 );
GroupHomomorphismByImages( s4, a4, [ (1,2,3,4), (1,2) ], [ (), () ] )
```

80.134 EndomorphismClasses

`EndomorphismClasses(` *G, case* `)`

The monoid of endomorphisms is required when calculating the monoid of derivations of a crossed module and when determining all the cat1-structures on a group G (see sections 80.90 and 80.95).

An endomorphism ϵ of R with image H' is determined by

- a normal subgroup N of R and a permutation representation $\theta : R/N \to Q$ of the quotient, giving a projection $\theta \circ \nu : R \to Q$, where $\nu : R \to R/N$ is the natural homomorphism;
- an automorphism α of Q;
- a subgroup H' in a conjugacy class $[H]$ of subgroups of R isomorphic to Q having representative H, an isomorphism $\phi : Q \cong H$, and a conjugating element $c \in R$ such that $H^c = H'$,

and takes values

$$\epsilon r \;=\; (\phi\alpha\theta\nu r)^c.$$

Endomorphisms are placed in the same class if they have the same choice of N and $[H]$, so the number of endomorphisms is

$$|\mathrm{End}(R)| = \sum_{classes} |\mathrm{Aut}(Q)| \; |[H]|.$$

The function returns records `E  = R.endomorphismClasses` and subfield `.classes` as shown below. Three cases are catered for as indicated in the example.

```
gap> Ea4 := EndomorphismClasses( a4 , 7);
Usage:  EndomorphismClasses( G [, case] );
 choose  case = 1  to include automorphisms and zero,
default  case = 2  to exclude automorphisms and zero,
         case = 3  when  N meet H  is trivial,
false
gap> Ea4 := EndomorphismClasses( a4 );
rec(
  isDomain := true,
  isEndomorphismClasses := true,
  areNonTrivial := true,
  intersectionFree := false,
  classes := [ rec(
```

```
            quotient := a4.Q2,
            projection := OperationHomomorphism( a4, a4.Q2 ),
            autoGroup := Group( GroupHomomorphismByImages( a4.Q2, a4.Q2,
              [ (1,3,2) ], [ (1,2,3) ] ) ),
            rangeNumber := 3,
            isomorphism := GroupHomomorphismByImages( a4.Q2, a4.H3,
              [ (1,3,2) ], [ (2,3,4) ] ),
            conj := [ (), (1,3,2), (1,2)(3,4), (1,4,2) ] ) ],
    group := a4,
    latticeLength := 5,
    latticeReps := [ a4.id, a4.H2, a4.H3, a4.H4, a4 ] )
```

80.135 EndomorphismImages

`EndomorphismImages(` G `)`

This returns the lists of images of the generators under the endomorphisms, using the data in `G.endomorphismClasses`. In this example two trivial normal subgroups have been excluded. The remaining normal subgroup of `a4` is `k4`, with quotient `c3` and `a4` has 8 elements of order 3 with which to generate a `c3`, and hence 8 endomorphisms in this class.

```
gap> EndomorphismImages( a4 );
[ [ (2,3,4), (2,4,3) ], [ (2,4,3), (2,3,4) ], [ (1,2,4), (1,4,2) ],
  [ (1,4,2), (1,2,4) ], [ (1,4,3), (1,3,4) ], [ (1,3,4), (1,4,3) ],
  [ (1,3,2), (1,2,3) ], [ (1,2,3), (1,3,2) ] ]
```

80.136 IdempotentImages

`IdempotentImages(` G `)`

This return the images of idempotent endomorphisms. Various options are allowed.

```
gap> IdempotentImages( a4, 7 );
Usage: IdempotentImages( G [, case] );
where  case = 1  for ALL idempotent images,
       case = 2  for all non-trivial images,
       case = 3  for case 2 and one group per conj class,
       case = 4  for case 3 and sorted into images.
false
gap> IdempotentImages( a4, 2 );
[ [ (2,4,3), (2,3,4) ], [ (1,4,2), (1,2,4) ], [ (1,3,4), (1,4,3) ],
  [ (1,2,3), (1,3,2) ] ]
gap> IdempotentImages( a4, 3 );
[ [ (2,4,3), (2,3,4) ] ]
```

80.137 InnerAutomorphismGroup

`InnerAutomorphismGroup(` G `)`

This creates the inner automorphism group of `G` as the group generated by the inner automorphisms by generators of `G`. If a field `G.automorphismGroup` exists, it is specified as the parent of `Inn(G)`.

```
gap> inna4 := InnerAutomorphismGroup( a4 );
Inn(a4)
gap> inna4.generators;
[ InnerAutomorphism( a4, (1,2,3) ), InnerAutomorphism( a4, (2,3,4) ) ]
```

80.138 IsAutomorphismGroup

`IsAutomorphismGroup(` A `)`

This tests to see whether `A` is a group of automorphisms.

```
gap> IsAutomorphismGroup( inna4 );
true
```

80.139 AutomorphismPair

`AutomorphismPair(` A `)`

This returns a record `pairA` containing a permutation group isomorphic to the group A obtained using the `OperationHomomorphism` function. The record contains A and `pairA.auto`, P as `pairA.perm`. Isomorphisms in each direction are saved as `pairA.p2a` and `pairA.a2p`.

```
gap> ac3 := AutomorphismGroup( c3 );
Group( GroupHomomorphismByImages( c3, c3, [(1,2,3)], [(1,3,2)] ) )
gap> pairc3 := AutomorphismPair( ac3 );
rec(
  auto := Aut(c3),
  perm := PermAut(c3),
  a2p := OperationHomomorphism( Aut(c3), PermAut(c3) ),
  p2a := GroupHomomorphismByImages( PermAut(c3), Aut(c3), [(1,2)],
    [ GroupHomomorphismByImages( c3, c3, [(1,2,3)], [(1,3,2)] ) ] ),
  isAutomorphismPair := true )
gap> pc3 := pairc3.perm;
PermAut(c3)
```

80.140 IsAutomorphismPair

`IsAutomorphismPair(` *pair* `)`

This tests to see whether `pair` is an (automorphism group, perm group) pair.

```
gap> IsAutomorphismPair( pairc3 );
true
```

80.141 AutomorphismPermGroup

`AutomorphismPermGroup(` G `)`

This combines AutomorphismGroup(G) with the function AutomorphismPair and returns G.automorphismGroup.automorphismPair.perm. The name PermAut(<G.name>) is given automatically.

```
gap> P := AutomorphismPermGroup( a4 );
PermAut(a4)
gap> P.generators;
[ (1,8,4)(2,6,7), (3,6,7)(4,5,8), (1,2)(3,8)(4,7)(5,6) ]
```

80.142 FpPair

FpPair(*G*)

When G is a finitely presented group, this function finds a faithful permutation representation P, which may be the regular representation, and sets up a pairing between G and P.

```
gap> f := FreeGroup( 2 );;
gap> rels := [ f.1^3, f.2^3, (f.1*f.2)^2 ];;
gap> g := f/rels;;
gap> pairg := FpPair( g );
rec(
  perm := Group( (2,4,3), (1,3,2) ),
  fp := Group( f.1, f.2 ),
  f2p := GroupHomomorphismByImages( Group( f.1, f.2 ),
    Group( (2,4,3), (1,3,2) ), [ f.1, f.2 ], [ (2,4,3), (1,3,2) ] ),
  p2f := GroupHomomorphismByImages( Group( (2,4,3), (1,3,2) ),
    Group( f.1, f.2 ), [ (2,4,3), (1,3,2) ], [ f.1, f.2 ] ),
  isFpPair := true,
  isMinTransitivePair := true,
  generators := [ (2,4,3), (1,3,2) ],
  degree := 4,
  position := 3 )
```

When G is a permutation group, the function PresentationViaCosetTable is called to find a presentation for G and hence a finitely presented group F isomorphic to G. When G has a name, the name <name of G>Fp is given automatically to F and <name of G>Pair to the pair.

```
gap> h20.generators;
[ (1,2,3,4,5), (2,3,5,4) ]
gap> pairh := FpPair( h20 );
rec(
  perm := h20,
  fp := h20Fp,
  f2p := GroupHomomorphismByImages( h20Fp, h20, [ f.1, f.2 ],
    [ (1,2,3,4,5), (2,3,5,4) ] ),
  p2f := GroupHomomorphismByImages( h20, h20Fp,
    [ (1,2,3,4,5), (2,3,5,4) ], [ f.1, f.2 ] ),
  isFpPair := true,
  degree := 5,
```

```
  presentation := << presentation with 2 gens and 3 rels
    of total length 14 >>,
  name := [ 'h', '2', '0', 'P', 'a', 'i', 'r' ] )
gap> pairh.fp.relators;
[ f.2^4, f.1^5, f.1*f.2*f.1*f.2^-1*f.1 ]
```

80.143 IsFpPair

`IsFpPair(` *pair* `)`

This tests to see whether `pair` is an (Fp-group, perm group) pair.

```
gap> IsFpPair( pairh );
true
```

80.144 SemidirectPair

`SemidirectPair(` S `)`

When `S` is a semidirect product, this function finds a faithful permutation representation P and sets up a pairing between S and P. The example illustrates `c2|Xc3`≅`s3`.

```
gap> agen := ac3.generators;; pgen := pc3.generators;;
gap> a := GroupHomomorphismByImages( pc3, ac3, pgen, agen );
GroupHomomorphismByImages( PermAut(c3), Aut(c3), [ (1,2) ],
[ GroupHomomorphismByImages( c3, c3, [ (1,2,3) ], [ (1,3,2) ] ) ] )
gap> G := SemidirectProduct( pc3, a, c3 );;
gap> G.name := "G";;  PG := SemidirectPair( G );
rec(
  perm := Perm(G),
  sdp := G,
  s2p := OperationHomomorphism( G, Perm(G) ),
  p2s := GroupHomomorphismByImages( Perm(G), G, [(1,2)(4,5), (3,5,4)],
    [ SemidirectProductElement( (1,2), GroupHomomorphismByImages
          ( c3, c3, [ (1,3,2) ], [ (1,2,3) ] ), () ),
      SemidirectProductElement( (), IdentityMapping(c3), (1,2,3) ) ] ))
```

80.145 IsSemidirectPair

`IsSemidirectPair(` *pair* `)`

This tests to see whether `pair` is a (semidirect product, perm group) pair.

```
gap> IsSemidirectPair( PG );
true
```

80.146 PrintList

`PrintList(` L `)`

This functions prints each of the elements of a list `L` on a separate line.

```
gap> J := [ [1,2,3], [3,4], [3,4], [1,2,4] ];; PrintList( J );
```

```
[ 1, 2, 3 ]
[ 3, 4 ]
[ 3, 4 ]
[ 1, 2, 4 ]
```

80.147 DistinctRepresentatives

`DistinctRepresentatives(` L `)`

When L is a set of n subsets of $[1..n]$ and the Hall condition is satisfied (the union of any k subsets has at least k elements), a standard algorithm for systems of distinct representatives is applied. (A backtrack algorithm would be more efficient.) If the elements of L are lists, they are converted to sets.

```
gap> DistinctRepresentatives( J );
[ 1, 3, 4, 2 ]
```

80.148 CommonRepresentatives

`CommonRepresentatives(` J, K `)`

When `J` and `K` are both lists of `n` sets, the list `L` is formed where $L[i] := \{j \ : \ J[i] \cap K[j] \neq \emptyset\}$. A system of distinct represetatives `reps` for `L` provides a permutation of the elements of `K` such that `J[i]` and `K[i]` have non-empty intersection. Taking the first element in each of these intersections determines a system of common representatives `com`. The function returns the pair `[ com, reps ]`. Note that there is no requirement for the representatives to be distinct. See also the next section.

```
gap> K := [ [3,4], [1,2], [2,3], [2,3,4] ];;
gap> CommonRepresentatives( J, K );
[ [ 3, 3, 3, 1 ], [ 1, 3, 4, 2 ] ]
```

This has produced $3 \in J[1] \cap K[1]$, $3 \in J[2] \cap K[3]$, $3 \in J[3] \cap K[4]$ and $1 \in J[4] \cap K[2]$.

80.149 CommonTransversal

`CommonTransversal(` G, H `)`

The existence of a common transversal for the left and right cosets of a subgroup H of G is a special case of systems of common representatives.

```
gap> T := CommonTransversal( a4, c3 );
[ (), (1,3)(2,4), (1,2)(3,4), (1,4)(2,3) ]
```

80.150 IsCommonTransversal

`IsCommonTransversal(` G, H, T `)`

```
gap> IsCommonTransversal( a4, c3, T );
true
```

Chapter 81

The CHEVIE Package Version 4 – a short introduction

CHEVIE is a joint project of Meinolf Geck, Gerhard Hiss, Frank Lübeck, Gunter Malle, Jean Michel, and Götz Pfeiffer. We document here the development version 4 of the GAP3-part of CHEVIE. This is a package in the GAP33 language, which implements

- algorithms for: finite complex reflection groups and their cyclotomic Hecke algebras, arbitrary Coxeter groups, the corresponding braid groups, Kazhdan-Lusztig bases, left cells, root data, unipotent characters, unipotent and semi-simple elements of algebraic groups, Green functions, etc...

- contains library files holding information for finite complex reflection groups giving conjugacy classes, fake degrees, generic degrees, irreducible characters, representations of the associated Hecke algebras, associated unipotent characters and unipotent classes (for Weyl groups, or more generally, "Spetsial"groups).

The package is automatically loaded if you use the GAP3 distribution `gap3-jm`; otherwise, you need to load it using

```
gap> RequirePackage("chevie");
--- Loading package chevie ------ version of 2018 Feb 19 ------
If you use CHEVIE in your work please cite the authors as follows:
[Jean Michel] The development version of the CHEVIE package of GAP3
 Journal of algebra 435 (2015) 308--336
[Meinolf Geck, Gerhard Hiss, Frank Luebeck, Gunter Malle, Goetz Pfeiffer]
 CHEVIE -- a system for computing and processing generic character tables
 Applicable Algebra in Engineering Comm. and Computing 7 (1996) 175--210
```

Compared to version 3, it is more general. For example, one can now work systematically with arbitrary Coxeter groups, not necessarily represented as permutation groups. Quite a few functions also work for arbitrary finite groups generated by complex reflections. Some functions have changed name to reflect the more general functionality. We have kept most former names working for compatibility, but we do not guarantee that they will survive in future releases.

Many objects associated with finite Coxeter groups admit some canonical labeling which carries additional information. These labels are often important for applications to Lie theory and related areas. The groups constructed in the package are permutation or matrix groups, so all the functions defined for such groups work; but often there are improvements, exploiting the particular nature of these groups. For example, the generic GAP3 function `ConjugacyClasses` applied to a Coxeter group does not invoke the general algorithm for computing conjugacy classes of permutation groups in GAP3, but first decomposes the given Coxeter group into irreducible components, and then reads canonical representatives of minimal length in the various classes of these irreducible components from library files. These canonical representatives also come with some additional information, for example the class names in exceptional groups reflect Carter's admissible diagrams and in classical groups are given in terms of partitions. In a similar way, the function `CharTable` does not invoke the Dixon–Schneider algorithm but proceeds in a similar way as described above. Moreover, in the resulting character table the classes have the labels described above and the characters also have canonical labels, e.g. partitions of n in the case of the symmetric group $\mathfrak{S}_n$, which is also the Coxeter group of type A_{n-1} (see 87.1 and 87.4). The normal forms we use, and the associated labeling of classes and characters for the individual types, are explained in detail in the various to chapters. The same considerations extend to some extent to all finite complex reflection groups.

Thus, most of the disk space required by CHEVIE is occupied by the files containing the basic information about the finite irreducible reflection groups. These files are called `weyla.g`, `cmplxg24.g` etc. up to the biggest file `cmplxg34.g` whose size is about 660 KBytes. These data files are structured in a uniform manner so that any piece of information can be extracted separately from them. (For example, it is not necessary to first compute the character table in order to have labels for the characters and classes.)

Several computations in the literature concerning the irreducible characters of finite Coxeter groups and Iwahori–Hecke algebras can now be checked or re-computed by anyone who is willing to use GAP3 and CHEVIE. Re-doing such computations and comparing with existing tables has helped discover bugs in the programs and misprints in the literature; we believe that having the possibility of repeating such computations and experimenting with the results has increased the reliability of the data and the programs. For example, it is now a trivial matter to re-compute the tables of induce/restrict matrices (with the appropriate labeling of the characters) for exceptional finite Weyl groups (see Section 88.1). These matrices have various applications in the representation theory of finite reductive groups, see chapter 4 of Lusztig's book [Lus85].

We ourselves have used these programs to prove results about the existence of elements with special properties in the conjugacy classes of finite Coxeter groups (see [GP93], [GM97]), and to compute character tables of Iwahori–Hecke algebras of exceptional type (see [Gec94], [GM97]). For a survey, see also [GHL$^+$96]. Quite a few computations with finite complex reflection groups have also been made in CHEVIE.

• The user should observe limitations on storage for working with these programs, e.g., the command `Elements` applied to a Weyl group of type E_8 needs a computer with 360GB of main memory!

• There is a function `InfoChevie` which is set equal to the GAP3 function `Ignore` when you load CHEVIE. If you redefine it by `InfoChevie:=Print;` then the CHEVIE functions will print some additional information in the course of their computations.

Of course, our hope is that more applications will be added in the future! For contributions to CHEVIE have created a directory `contr` in which the corresponding files are distributed with CHEVIE. However, they do remain under the authorship and the responsibility of their authors. Files from that directory can be read into GAP3 using the command `ReadChv("contr/filename")`. At present, the directory `contr` contains the following files:

`affa` by F. Digne it contains functions to work with periodic permutations of the integers, with the affine Coxeter group of type $\tilde{A}$ seen as a group of periodic permutations, and with the corresponding dual Garside monoid.

`arikidec` by N. Jacon it contains functions for computing the canonical basis of an arbitrary irreducible integrable highest weight representation of the quantum group of the affine special linear group $U_v(\widehat{sl}_e)$. It also computes the decomposition matrix of Ariki-Koike algebras, where the parameters are power of a e-th root of unity in a field of characteristic zero.

`braidsup` by J. Michel it contains some supplementary programs for working with braids (or more generally Garside monoids).

`brbase` by M. Geck and S. Kim it contains programs for computing bi-grassmannians and the base for the Bruhat–Chevalley order on finite Coxeter groups (see [GK96]).

`chargood` by M. Geck and J. Michel it contains functions (used in [GM97]) implementing algorithms to compute character tables of Iwahori–Hecke algebras, especially that of type E_8.

`chevlie` by M. Geck it contains function to build matrices for the Lie algebra and the Chevalley group attached to a root datum.

`cp` by J. Michel and G. Neaime it contains a function to construct the Corran-Picantin monoid as an interval monoid, following Neaime's work.

`hecbloc` by M. Geck it contains functions for computing blocks and defects of characters of Iwahori–Hecke algebras specialized at roots of unity over the rational numbers.

`minrep` by M. Geck and G. Pfeiffer it contains programs (used in [GP93]) for computing representatives of minimal length in the conjugacy classes of finite Coxeter groups.

`murphy` by A. Mathas it contains programs which allow calculations with the Murphy basis of the Hecke algebra of type A.

`rouquierblockdata` by M. Chlouveraki and J. Michel it contains functions to compute the Rouquier blocks of 1-cylotomic Hecke algebras of arbitrary complex reflection groups.

`specpie` by M. Geck and G. Malle it contains functions for computing the Green-like polynomials (or rather rational functions) associated with special pieces of the unipotent variety (or "special characters in the case of complex reflection groups).

`spherical` by D. Juteau it contains a function to determine the support of the spherical module for a rational Cherednik algebra. Here is an example `gap> DisplaySphericalCriterion(ComplexReflectionGroup(13));`

```
Maximal parabolic subgroups |                                   q-index
----------------------------------------------------------------------
A1 [ 2 ]                    |P2(x_1)P3(y_1)P2(x_1y_1)P2^2P6(x_1y_1^2)
```

```
A1 [ 1 ]                          |      P2P3(y_1)P2(x_1y_1)P2^2P6(x_1y_1^2)
```

`xy` by J. Michel and R. Rouquier it contains a function to display graphically elements of Hecke modules for affine Weyl groups of rank 2.

Finally, it should be mentioned that the tables of Green functions for finite groups of Lie type which are in the MAPLE-part of CHEVIE are now obtainable by using the CHEVIE routines for unipotent classes and the associated intersection cohomology complexes.

Acknowledgments. We wish to thank the Aachen GAP3 team for general support.

We also gratefully acknowledge financial support by the DFG in the framework of the Forschungsschwerpunkt "Algorithmische Zahlentheorie und Algebra" from 1992 to 1998.

We are indebted to Andrew Mathas for contributing the initial version of functions for the various Kazhdan-Lusztig bases in kl.g.

Chapter 82

Reflections, and reflection groups

Central in CHEVIE is the notion of **reflection groups**.

Let V be a vector space over a subfield K of the complex numbers; in GAP3 this usually means the `Rationals`, the `Cyclotomics`, or a subfield. A **complex reflection** is an element $s \in GL(V)$ of finite order whose fixed point set is an hyperplane (we will in the following just call it a **reflection** to abbreviate; in some literature the term reflection is only employed when the order is 2 and the more general case is called a **pseudo-reflection**). Thus a reflection has a unique eigenvalue not equal to 1. If K is a subfield of the real numbers, we get a real reflection which is necessarily of order 2 and the non-trivial eigenvalue is equal to -1.

A reflection group W is a group generated by a finite number of complex reflections.

Since when W contains a reflection s it contains its powers, W is always generated by reflections s with eigenvalue `E(d)` where d is the order of s; we may in addition assume that s is not a power of another reflection with eigenvalue `E(d')` with `d'>d`. Such a reflection is called **distinguished**; we take it as the canonical generator of the cyclic subgroup it generates. The generators of reflection groups in CHEVIE are always distinguished reflections. In a real reflection group all reflections are distinguished.

Reflection groups in CHEVIE are groups W with the following fields (in the group record) defined

`.nbGeneratingReflections`
: the number of reflections which generate W

`.reflections`
: a list of distinguished reflections, given as elements of W, such that a list of reflections which generate W is `W.reflections[1..W.nbGeneratingReflections]`.

`.OrdersGeneratingReflections`
: a list (of length at least `W.nbGeneratingReflections`) such that its i-th element is the order of `W.reflections[i]`. By the above conventions `W.reflections[i]` thus has `E(W.OrdersGeneratingReflections[i])` as its nontrivial eigenvalue.

Note that W does **not** need to be a matrix group. The meaning of the above fields is just that W has a representation (called the **reflection representation** of W) where the elements `W.reflections` operate as reflections. It is much more efficient to compute with permutation groups which have such fields defined, than with matrix groups, when possible. Information sufficient to determine a particular reflection representation is stored for such groups (see `CartanMat`).

Also note that, although `.reflections` is usually just initialized to the generating reflections, it is usually augmented by adding other reflections to it as computations require. For instance, when W is finite, the set of all reflections in W is finite (they are just the elements of the conjugacy classes of the generating reflections and their powers), and all the distinguished reflections in W are added to `.reflections` when required, for instance when calling `Reflections(` *W* `)` which returns the list of all (distinguished) reflections. Note that when W is finite, the distinguished reflections are in bijection with the reflecting hyperplanes.

There are very few functions in CHEVIE which deal with reflections groups in full generality. Usually the groups one wants to deal with is in a more restricted class (Coxeter groups, finite reflection groups) which are described in the following chapters.

82.1 Reflection

`Reflection(` *root* `,` *coroot* `)`

A (complex) reflection s acting on the vector space V (over some subfield of the complex numbers), is a linear map of finite order whose fixed points are an hyperplane H (called the **reflecting hyperplane** of s); an eigenvector r for the non-trivial eigenvalue ζ (a root of unity) is called a **root** of s. We may chose a linear form $r^\vee$ (called a **coroot** of s) defining H such that $r^\vee(r) = 1-\zeta$ and then as a linear map s is given by $x \mapsto x - r^\vee(x)r$.

A first way of specifying a reflection is by giving a root and a coroot, which are uniquely determined by the reflection up to multiplication of the root by a scalar and of the coroot by the inverse scalar. The function `Reflection` gives the matrix of the corresponding reflection in the standard basis of V, where the *root* and the *coroot* are vectors given in the standard bases of V and $V^\vee$ (thus in GAP3 $r^\vee(r)$ is obtained as *root*`*`*coroot*).

```
gap> r:=Reflection([1,0,0],[2,-1,0]);
[ [ -1, 0, 0 ], [ 1, 1, 0 ], [ 0, 0, 1 ] ]
gap> r=CoxeterGroup("A",3).matgens[1];
true
gap> [1,0,0]*r;
[ -1, 0, 0 ]
```

As we see in the last line, in GAP3 the matrices operate **from the right** on the vector space.

`Reflection(` *root* `[,` *eigenvalue* `] )`

We may give slightly less information if we assume that the standard hermitian scalar product (x, y) on V (given in GAP3 by `x*ComplexConjugate(y)`) is s-invariant. Then, identifying V and $V^\vee$ via this scalar product, s is given by the formula

$$x \mapsto x - (1-\zeta)(x, r)/(r, r)r$$

so s is specified by just *root* and *eigenvalue*. When *eigenvalue* is omitted it is assumed to be equal to -1. The function `Reflection` gives again the matrix of the reflection.

```
gap> Reflection([0,0,1],E(3));
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, E(3) ] ]
gap> last=ComplexReflectionGroup(25).matgens[1];
true
```

`Reflection(` W`,` i `)`

This form returns the reflection with respect to the i-th root in the finite reflection group W (this works only for groups represented as permutation groups of the roots, see 84). Note that one would not get the same result with `W.reflections[i]` since this entry might not yet be bound (not yet have been computed), and also it is not guaranteed apart from the generating roots (and the positive roots of Weyl groups) that the i-th reflection corresponds to the i-th root, since two roots corresponding to the same reflection may have been obtained before all the reflections have been obtained.

```
gap> Reflection(CoxeterGroup("A",3),6);
( 1,11)( 3,10)( 4, 9)( 5, 7)( 6,12)
```

82.2 AsReflection

`AsReflection(` s `[,`r`])`

Here s is a square matrix with entries cyclotomic numbers, and if given r is a vector of the same length as s of cyclotomic numbers. The function determines if s is the matrix of a reflection (resp. if r is given if it is the matrix of a reflection of root r; the point of giving r is to specify exactly the desired root and coroot, which otherwise are determined only up to a scalar and its inverse). The returned result is `false` if s is not a reflection (resp. not a reflection with root r), and otherwise is a record with four fields :

`.root`
: the root of the reflection s (equal to r if given)

`.coroot`
: the coroot of s

`.eigenvalue`
: the non-trivial eigenvalue of s

`.isOrthogonal`
: a boolean which is `true` if and only if s is orthogonal with respect to the usual scalar product (then the root and eigenvalue are sufficient to determine s)

```
gap> AsReflection([[-1,0,0],[1,1,0],[0,0,1]]);
rec(
  root := [ 2, 0, 0 ],
  coroot := [ 1, -1/2, 0 ],
  eigenvalue := -1,
  isOrthogonal := false )
gap> AsReflection([[-1,0,0],[1,1,0],[0,0,1]],[1,0,0]);
rec(
  root := [ 1, 0, 0 ],
```

```
    coroot := [ 2, -1, 0 ],
    eigenvalue := -1,
    isOrthogonal := false )
```

82.3 CartanMat

`CartanMat(` W `)`

Let $s_1, \ldots, s_n$ be a list of reflections with associated root vectors r_i and coroots $r_i^\vee$. Then the matrix $C_{i,j}$ of the $r_i^\vee(r_j)$ is called the **Cartan matrix** of the list of reflections. It is uniquely determined by the reflections up to conjugating by diagonal matrices.

If $s_1, \ldots, s_n$ are the generators of a reflection group W, the matrix `C` up to conjugation by diagonal matrices is an invariant of the reflection representation of W. It actually completely determines this representation if the r_i are linearly independent (which is e.g. the case if `C` is invertible), since in the r_i basis the matrix for the s_i differs from the identity only on the i-th line, where the corresponding line of `C` has been subtracted.

```
gap> W:=CoxeterGroup("A",3);;
gap> CartanMat(W);
[ [ 2, -1, 0 ], [ -1, 2, -1 ], [ 0, -1, 2 ] ]
```

`CartanMat(` W, l `)`

Returns the Cartan matrix of the roots of W specified by the list of integers l (for a finite reflection group represented as a group of permutation of root vectors, these integers are indices in the list of roots of the parent reflection group).

`CartanMat(` *type* `)`

This form returns the Cartan matrix of some standard reflection representations for Coxeter groups, taking a symbolic description of the Coxeter group given by the arguments. See 85.1

82.4 Rank

`Rank(` W `)`

Let W be a reflection group in the vector space V. This function returns the dimension of V, if known. If reflections of W are generated by a root and a coroot, it is the length of the root as a list. If W is a matrix group it is the dimension of the matrices.

```
gap> W:=ReflectionSubgroup(CoxeterGroup("A",3),[1,3]);
ReflectionSubgroup(CoxeterGroup("A",3), [ 1, 3 ])
gap> Rank(W);
3
```

82.5 SemisimpleRank

`SemisimpleRank(` W `)`

Let W be a reflection group in the vector space V. This function returns the dimension of the subspace V' of V where W effectively acts, which is the subspace generated by the roots of the reflections of W. The space V' is W-stable and has a W-stable complement on which

W acts trivially. The `SemisimpleRank` is independent of the reflection representation. W is called **essential** if V'=V.

```
gap> W:=ReflectionSubgroup(CoxeterGroup("A",3),[1,3]);
ReflectionSubgroup(CoxeterGroup("A",3), [ 1, 3 ])
gap> SemisimpleRank(W);
2
```

Chapter 83

Coxeter groups

In this chapter we describe functions for dealing with general Coxeter groups.

A suitable reference for the general theory is, for example, the volume [Bou68] of Bourbaki.

A **Coxeter group** is a group which has the presentation $W = \langle S| \ (st)^{m(s,t)} = 1$ for $s, t \in S\rangle$ for some symmetric integer matrix $m(s,t)$ called the **Coxeter matrix**, where $m(s,t) > 1$ for $s \neq t$ and $m(s,s) = 1$. It is true (but a non-trivial theorem) that in a Coxeter group the order of st is exactly $m(s,t)$, thus a Coxeter group is the same as a **Coxeter system**, that is a pair (W,S) of a group W and a set S of involutions, such that the group is presented by relations describing the order of the product of two elements of S. A Coxeter group has a natural representation on a real vector space V of dimension the number of generators, where each generator acts as a reflection, its **reflection representation** (see `CoxeterGroupByCoxeterMatrix`); the faithfulness of this representation in the main argument to prove that the order of st is exactly $m(s,t)$. Thus Coxeter groups are real reflection groups. The converse need not be true if the set of reflecting hyperplanes has bad topological properties, but it turns out that finite Coxeter groups are the same as finite real reflection groups. The possible Coxeter matrices for finite Coxeter groups have been completely classified; the corresponding finite groups play a deep role in several areas of mathematics.

Coxeter groups have a nice solution to the word problem. The **length** $l(w)$ of an element w of W is the minimum number of elements of S of which it is a product (since the elements of S are involutions, we do not need inverses). An expression of w of minimal length is called a **reduced word** for w. The main property of reduced words is the **exchange lemma** which states that if $s_1 \dots s_k$ is a reduced word for w where $k = l(w)$ and $s \in S$ is such that $l(sw) \leq l(w)$ then one of the s_i in the word for w can be deleted to obtain a reduced word for sw. Thus given $s \in S$ and $w \in W$, either $l(sw) = l(w)+1$ or $l(sw) = l(w)-1$ and we say in this last case that s belongs to the **left descent set** of w. The computation of a reduced word for an element, and other word problems, are easily done if we know the left descent sets. For most Coxeter groups that we will be able to build in CHEVIE, this left descent set can be easily determined (see e.g. `CoxeterGroupSymmetricGroup` below), so this suggests how to deal with Coxeter groups in CHEVIE. They are reflection groups, so the following fields are defined in the group record:

`.nbGeneratingReflections`
the size of S

`.reflections`
a list of elements of W, such that `W.reflections[1..W.nbGeneratingReflections]` is the set S.

the above names are used instead of names like `CoxeterGenerators` and `CoxeterRank` since the Coxeter groups **are** reflection groups and we want the functions for reflection groups applicable to them (similarly, if you have read the chapter on reflections and reflection groups, you will realize that there is also a field `.OrdersGeneratingReflections` which contains only 2's). The main additional function which allows to compute within Coxeter groups is:

`.operations.IsLeftDescending(W,w,i)`
returns true if and only if the i-th element of S is in the left descending set of w.

For Coxeter groups constructed in CHEVIE an `IsLeftDescending` operation is provided, but you can construct your own Coxeter groups just by filling the above fields (see the function `CoxeterGroupSymmetricGroup` below for an example). It should be noted than you can make into a Coxeter group **any** kind of group: finitely presented groups, permutation groups or matrix groups, if you fill appropriately the above fields; and the given generating reflection do not have to be `W.generators` — all functions for Coxeter group and Hecke algebras will then work for your Coxeter groups (using your function `IsLeftDescending`).

A common occurrence in CHEVIE code for Coxeter groups is a loop like:

`First([1..W.semisimpleRank],x->IsLeftDescending(W,w,x))`

which for a reflection subgroup becomes

`First(W.rootRestriction{[1..W.semisimpleRank]},x->IsLeftDescending(W,w,x))`

where the overhead is quite large, since dispatching on the group type is done in `IsLeftDescending`. To improve this code, if you provide a function `FirstLeftDescending(W,w)` it will be called instead of the above loop (if you do not provide one the above loop will be used). Such a function provided by CHEVIE for finite Coxeter groups represented as permutation groups of the roots is 3 times more efficient than the above loop.

Because of the easy solution of the word problem in Coxeter groups, a convenient way to represent their elements is as words in the Coxeter generators. They are represented in CHEVIE as lists of labels for the generators. By default these labels are given as the index of a generator in S, so a Coxeter word is just a list of integers which run from 1 to the length of S. This can be changed to reflect a more conventional notation for some groups, by changing the field `.reflectionsLabels` of the Coxeter group which contains the labels used for the Coxeter words (by default it contains `[1..W.nbGeneratingReflections]`). For a Coxeter group with 2 generators, you could for instance set this field to `"st"` to use words such as `"sts"` instead of `[1,2,1]`. For reflection subgroups, this is used in CHEVIE by setting the reflection labels to the indices of the generators in the set S of the parent group (which is given by `.rootInclusion`).

The functions `CoxeterWord` and `EltWord` will do the conversion between Coxeter words and elements of the group.

```
gap> W := CoxeterGroup( "D", 4 );;
gap> p := EltWord( W, [ 1, 3, 2, 1, 3 ] );
```

```
( 1,14,13, 2)( 3,17, 8,18)( 4,12)( 5,20, 6,15)( 7,10,11, 9)(16,24)
(19,22,23,21)
gap> CoxeterWord( W, p );
[ 1, 3, 1, 2, 3 ]
gap> W.reflectionsLabels:="stuv";
"stuv"
gap> CoxeterWord(W,p);
"sustu"
```

We notice that the word we started with and the one that we ended up with, are not the same. But of course, they represent the same element of W. The reason for this difference is that the function `CoxeterWord` always computes a reduced word which is the lexicographically smallest among all possible expressions of an element of W as a word in the fundamental reflections. The function `ReducedCoxeterWord` does the same but with a word as input (rather than an element of the group). Below are some other possible computations with the same Coxeter group as above:

```
gap> LongestCoxeterWord( W );  # the (unique) longest element in W
[ 1, 2, 3, 1, 2, 3, 4, 3, 1, 2, 3, 4 ]
gap> w0 := LongestCoxeterElement( W ); # = EltWord( W, last )
( 1,13)( 2,14)( 3,15)( 4,16)( 5,17)( 6,18)( 7,19)( 8,20)( 9,21)(10,22)
(11,23)(12,24)
gap> CoxeterLength( W, w0 );
12
gap> List( Reflections( W ), i -> CoxeterWord( W, i ) );
[ "s", "t", "u", "v", "sus", "tut", "uvu", "stust", "suvus", "tuvut",
  "stuvust", "ustuvustu" ]
gap> l := List( [0 .. W.N], x -> CoxeterElements( W, x ) );;
gap> List( l, Length );
[ 1, 4, 9, 16, 23, 28, 30, 28, 23, 16, 9, 4, 1 ]
```

The above line tells us that there is 1 element of length 0, there are 4 elements of length 4, etc.

For many basic functions (like `Bruhat`, `CoxeterLength`, etc.) we have chosen the convention that the input is an element of a Coxeter group (rather than a Coxeter word). The reason is that for a Coxeter group which is a permutation group, if in some application one has to do a lot of computations with Coxeter group elements then using the low level GAP3 functions for permutations is usually much faster than manipulating lists of reduced expressions.

Before describing functions applicable to Coxeter groups and Coxeter words we describe functions which build two familiar examples.

83.1 CoxeterGroupSymmetricGroup

`CoxeterGroupSymmetricGroup(` n `)`

returns the symmetric group on n letters as a Coxeter group. We give the code of this function as it is a good example on how to make a Coxeter group:

```
gap> CoxeterGroupSymmetricGroup := function ( n )
> local  W;
```

```
> W := SymmetricGroup( n );
> W.reflections := List( [ 1 .. n - 1 ], i->(i,i + 1) );
> W.operations.IsLeftDescending := function ( W, w, i )
>       return i ^ w > (i + 1) ^ w;
>   end;
> AbsCoxOps.CompleteCoxeterGroupRecord( W );
> return W;
> end;
function ( n ) ... end
```

In the above, we first set the generating reflections of W to be the elementary transpositions `(i,i+1)` (which are reflections in the natural representation of the symmetric group permuting the standard basis of an n-dimensional vector space), then give the `IsLeftDescending` function (which just checks if `(i,i+1)` is an inversion of the permutation). Finally, `AbsCoxOps.CompleteCoxeterGrou` is a service routine which fills other fields from the ones we gave. We can see what it did by doing:

```
gap> PrintRec(CoxeterGroupSymmetricGroup(3));
rec(
  isDomain                        := true,
  isGroup                         := true,
  identity                        := (),
  generators                      := [ (1,3), (2,3) ],
  operations                      := HasTypeOps,
  isPermGroup                     := true,
  isFinite                        := true,
  1                               := (1,3),
  2                               := (2,3),
  degree                          := 3,
  reflections                     := [ (1,2), (2,3) ],
  nbGeneratingReflections         := 2,
  generatingReflections           := [ 1 .. 2 ],
  EigenvaluesGeneratingReflections:= [ 1/2, 1/2 ],
  isCoxeterGroup                  := true,
  rootInclusion                   := [ 1 .. 2 ],
  rootRestriction                 := [ 1 .. 2 ],
  reflectionsLabels               := [ 1 .. 2 ],
  semisimpleRank                  := 2,
  rank                            := 2,
  coxeterMat                      := [ [ 1, 3 ], [ 3, 1 ] ],
  orbitRepresentative             := [ 1, 1 ],
  OrdersGeneratingReflections     := [ 2, 2 ],
  cartan                          := [ [ 2, -1 ], [ -1, 2 ] ],
  type                            := [ rec(rank    := 2,
      series  := "A",
      indices := [ 1, 2 ]) ],
  longestElm                      := (1,3),
  longestCoxeterWord              := [ 1, 2, 1 ],
  N                               := 3 )
```

We do not explain all the fields here. Some are there for technical reasons and may change from version to version of CHEVIE. Among the added fields, we see `nbGeneratingReflections` (taken to be `Length(W.reflections)` if we do not give it), `.OrdersGeneratingReflections`, the Coxeter matrix `.coxeterMat`, a description of conjugacy classes of the generating reflections given in `.orbitRepresentative` (whose i-th entry is the smallest index of a reflection conjugate to `.reflections[i]`), `.reflectionsLabels` (the default labels used for Coxeter word). At the end are 3 fields which are computed only for finite Coxeter groups: the longest element, as an element and as a Coxeter word, and in `W.N` the number of reflections in W (which is also the length of the longest Coxeter word).

83.2 CoxeterGroupHyperoctaedralGroup

`CoxeterGroupHyperoctaedralGroup(` n `)`

returns the hyperoctaedral group of rank n as a Coxeter group. It is given as a permutation group on $2n$ letters, with Coxeter generators the permutations `(2i-1,2i+1)(2i,2i+2)` and `(1,2)`.

```
gap> CoxeterGroupHyperoctaedralGroup(2);
Group( (1,2), (1,3)(2,4) )
```

83.3 CoxeterMatrix

`CoxeterMatrix(` W `)`

return the Coxeter matrix of the Coxeter group W, that is the matrix whose entry `m[i][j]` contains the order of $g_i * g_j$ where g_i is the i-th Coxeter generator of W. An infinite order is represented by the entry 0.

```
gap> W:=CoxeterGroupSymmetricGroup(4);
CoxeterGroupSymmetricGroup(4)
gap> CoxeterMatrix(W);
[ [ 1, 3, 2 ], [ 3, 1, 3 ], [ 2, 3, 1 ] ]
```

83.4 CoxeterGroupByCoxeterMatrix

`CoxeterGroupByCoxeterMatrix(` m `)`

returns the Coxeter group whose Coxeter matrix is m.

The matrix m should be a symmetric integer matrix such that `m[i,i]=1` and `m[i,j]>=2` (or `m[i,j]=0` to represent an infinite entry).

The group is constructed as a matrix group, using the standard reflection representation for Coxeter groups. This is the representation on a real vector space V of dimension `Length(m)` defined as follows : if e_s is a basis of V indexed by the lines of m, we make the s-th reflection act by $s(x) = x - 2\langle x, e_s\rangle e_s$ where $\langle,\rangle$ is the bilinear form on V defined by $\langle e_s, e_t\rangle = -\cos(\pi/m[s,t])$ (where by convention $\pi/m[s,t] = 0$ if $m[s,t] = \infty$, which is represented in CHEVIE by setting `m[s,t]:=0`). In the example below the affine Weyl group $\tilde{A}_2$ is constructed, and then $\tilde{A}_1$.

```
gap> m:=[[1,3,3],[3,1,3],[3,3,1]];;
gap> W:=CoxeterGroupByCoxeterMatrix(m);
```

```
CoxeterGroupByCoxeterMatrix([[1,3,3],[3,1,3],[3,3,1]])
gap> CoxeterWords(W,3);
[ [ 1, 3, 2 ], [ 1, 2, 3 ], [ 1, 2, 1 ], [ 1, 3, 1 ], [ 2, 1, 3 ],
  [ 3, 1, 2 ], [ 2, 3, 2 ], [ 2, 3, 1 ], [ 3, 2, 1 ] ]
gap> CoxeterGroupByCoxeterMatrix([[1,0],[0,1]]);
CoxeterGroupByCoxeterMatrix([[1,0],[0,1]])
```

83.5 CoxeterGroupByCartanMatrix

`CoxeterGroupByCartanMatrix(` m `)`

m should be a square matrix of real cyclotomic numbers. It returns the reflection group whose Cartan matrix is m. This is a matrix group constructed as follows. Let V be a real vector space of dimension `Length(m)`, and let $\langle,\rangle$ be the bilinear form defined by $\langle e_i, e_j\rangle = m[i,j]$ where e_i is the canonical basis of V. Then the result is the matrix group generated by the reflections $s_i(x) = x - 2\langle x, e_i\rangle e_i$.

This function is used in `CoxeterGroupByCoxeterMatrix`, using also the function `CartanMatFromCoxeterMatrix`.

```
gap> CartanMatFromCoxeterMatrix([[1,0],[0,1]]);
[ [ 2, -2 ], [ -2, 2 ] ]
gap> CoxeterGroupByCartanMatrix(last);
CoxeterGroupByCartanMatrix([[2,-2],[-2,2]])
```

Above is another way to construct $\tilde{A}_1$.

83.6 CartanMatFromCoxeterMatrix

`CartanMatFromCoxeterMatrix(` m `)`

The argument is a CoxeterMatrix for a finite Coxeter group W and the result is a Cartan Matrix for the standard reflection representation of W (see 82.3). Its diagonal terms are 2 and the coefficient between two generating reflections s and t is $-2\cos(\pi/m[s,t])$ (where by convention $\pi/m[s,t] = 0$ if $m[s,t] = \infty$, which is represented in CHEVIE by setting `m[s,t]:=0`).

```
gap> m:=[[1,3],[3,1]];
[ [ 1, 3 ], [ 3, 1 ] ]
gap> CartanMatFromCoxeterMatrix(m);
[ [ 2, -1 ], [ -1, 2 ] ]
```

83.7 Functions for general Coxeter groups

Some functions take advantage of the fact a group is a Coxeter group to use a better algorithm. A typical example is:

`Elements(`W`)`

For finite Coxeter groups, uses a recursive algorithm based on the construction of elements of a chain of parabolic subgroups

`ReflectionSubgroup(`W`,` J`)`

When I is a subset of `[1..W.nbGeneratingReflections]` then the reflection subgroup of W generated by `W.reflections{I}` can be generated abstractly (without any specific knowledge about the representation of W) as a Coxeter group. This is what this routine does: implement a special case of `ReflectionSubgroup` which works for arbitrary Coxeter groups (see 88.1). The actual argument J should be reflection labels for W, i.e. be a subset of `W.reflectionsLabels`.

Similarly, the functions `ReducedRightCosetRepresentatives`, `PermCosetsSubgroup`, work for reflection subgroups of the above form. See the chapter on reflection subgroups for a description of these functions.

`CartanMat(` W `)`

Returns `CartanMatFromCoxeterMatrix(CoxeterMatrix(` W `))` (see 83.6).

The functions `ReflectionType`, `ReflectionName` and all functions depending on the classification of finite Coxeter groups work for finite Coxeter groups. See the chapter on reflection groups for a description of these functions.

`BraidRelations(` W `)`

returns the braid relations implied by the Coxeter matrix of W.

83.8 IsLeftDescending

`IsLeftDescending(` W `,` w`,` i `)`

returns `true` if and only if the i-th generating reflection `W.reflections[i]` is in the left descent set of the element w of W.

```
gap> W:=CoxeterGroupSymmetricGroup(3);
CoxeterGroupSymmetricGroup(3)
gap> IsLeftDescending(W,(1,2),1);
true
```

83.9 FirstLeftDescending

`FirstLeftDescending(` W `,` w `)`

returns the index in the list of generating reflections of W of the first element of the left descent set of the element w of W (i.e., the first i such that if `s=W.reflections[i]` then $l(sw) < l(w)$). It is quite important to think of using this function rather than write a loop like `First([1..W.nbGeneratingReflections],IsLeftDescending)`, since for particular classes of groups (e.g. finite Coxeter groups) the function is much optimized compared to such a loop.

```
gap> W:=CoxeterGroupSymmetricGroup(3);
CoxeterGroupSymmetricGroup(3)
gap> FirstLeftDescending(W,(2,3));
2
```

83.10 LeftDescentSet

`LeftDescentSet(` W`,` w `)`

The set of generators s such that $l(sw) < l(w)$, given as a list of labels for the corresponding generating reflections (for a coxeter subgroup they are the indices of the reflections in `Parent(W)`).

```
gap> W:=CoxeterGroupSymmetricGroup(3);
CoxeterGroupSymmetricGroup(3)
gap> LeftDescentSet( W, (1,3));
[ 1, 2 ]
```

See also 83.11.

83.11 RightDescentSet

`RightDescentSet(` W, w `)`

The set of generators s such that $l(ws) < l(w)$, given as a list of labels for the corresponding generating reflections (for a coxeter subgroup they are the indices of the reflections in `Parent(W)`).

```
gap> W := CoxeterGroup( "A", 2 );;
gap> w := EltWord( W, [ 1, 2 ] );;
gap> RightDescentSet( W, w );
[ 2 ]
```

See also 83.10.

83.12 EltWord

`EltWord(` W , w `)`

returns the element of W which corresponds to the Coxeter word w. Thus it returns a permutation if W is a permutation group (the usual case for finite Coxeter groups) and a matrix for matrix groups (such as affine Coxeter groups). As for `CoxeterWord`, for a Coxeter subgroup, w must consist of indices in `Parent(W)`.

```
gap> W:=CoxeterGroupSymmetricGroup(4);
CoxeterGroupSymmetricGroup(4)
gap> EltWord(W,[1,2,3]);
(1,4,3,2)
```

See also 83.13.

83.13 CoxeterWord

`CoxeterWord(` W , w `)`

returns a reduced word in the standard generators of the Coxeter group W for the element w (represented as the GAP3 list of the corresponding reflection labels; thus for a coxeter subgroup they are the indices of the reflections in `Parent(W)`).

```
gap> W := CoxeterGroup( "A", 3 );;
gap> w := ( 1,11)( 3,10)( 4, 9)( 5, 7)( 6,12);;
gap> w in W;
true
gap> CoxeterWord( W, w );
[ 1, 2, 3, 2, 1 ]
```

The result of `CoxeterWord` is the lexicographically smallest reduced word for w (for the ordering of the Coxeter generators given by `W.reflections`).

See also 83.12 and 83.15.

83.14 CoxeterLength

`CoxeterLength(` W `,` w `)`

returns the length of the element w of W as a reduced expression in the standard generators.

```
gap> W := CoxeterGroup( "F", 4 );;
gap> p := EltWord( W, [ 1, 2, 3, 4, 2 ] );
( 1,44,38,25,20,14)( 2, 5,40,47,48,35)( 3, 7,13,21,19,15)
( 4, 6,12,28,30,36)( 8,34,41,32,10,17)( 9,18)(11,26,29,16,23,24)
(27,31,37,45,43,39)(33,42)
gap> CoxeterLength( W, p );
5
gap> CoxeterWord( W, p );
[ 1, 2, 3, 2, 4 ]
```

83.15 ReducedCoxeterWord

`ReducedCoxeterWord(` W `,` w `)`

returns a reduced expression for an element of the Coxeter group W, which is given as a GAP3 list of reflection labels for the standard generators of W.

```
gap> W := CoxeterGroup( "E", 6 );;
gap> ReducedCoxeterWord( W, [ 1, 1, 1, 1, 1, 2, 2, 2, 3 ] );
[ 1, 2, 3 ]
```

83.16 BrieskornNormalForm

`BrieskornNormalForm(` W `,` w `)`

Brieskorn [Bri71] has noticed that if $L(w)$ is the left descent set of w (see 83.10), and if $w_{L(w)}$ is the longest Coxeter element (see 83.17) of the reflection subgroup $W_{L(w)}$ (note that this element is an involution), then $w_{L(w)}$ divides w, in the sense that $l(w_{L(w)}) + l(w_{L(w)}^{-1}w) = l(w)$. We can now divide w by $w_{L(w)}$ and continue this process with the quotient. In this way, we obtain a reduced expression $w = w_{L_1} \cdots w_{L_r}$ where $L_i = L(w_{L_i} \cdots w_{L_r})$ for all i, which we call the **Brieskorn normal form** of w. The function `BrieskornNormalForm` will return a description of this form, by returning the list of sets $L(w)$ which describe the above decomposition.

```
gap> W:=CoxeterGroup("E",8);
CoxeterGroup("E",8)
gap> w:=[ 2, 3, 4, 2, 3, 4, 5, 4, 2, 3, 4, 5, 6, 5, 4, 2, 3, 4,
>   5, 6, 7, 6, 5, 4, 2, 3, 4, 5, 6, 7, 8 ];;
gap> BrieskornNormalForm(W,EltWord(W,w));
[ [ 2, 3, 4, 5, 6, 7 ], [ 8 ] ]
gap> EltWord(W,w)=Product(last,x->LongestCoxeterElement(W,x));
true
```

83.17 LongestCoxeterElement

`LongestCoxeterElement(` W `[,`I`])`

If W is finite, returns the unique element of maximal length of the Coxeter group W. May loop infinitely otherwise.

```
gap> LongestCoxeterElement( CoxeterGroupSymmetricGroup( 4 ) );
(1,4)(2,3)
```

If a second argument I is given, returns the longest element of the parabolic subgroup generated by the reflections in I (where I is given as `.reflectionsLabels`).

```
gap> LongestCoxeterElement(CoxeterGroupSymmetricGroup(4),[2,3]);
(2,4)
```

83.18 LongestCoxeterWord

`LongestCoxeterWord(` W `)`

If W is finite, returns a reduced expression in the standard generators for the unique element of maximal length of the Coxeter group W. May loop infinitely otherwise.

```
gap> LongestCoxeterWord( CoxeterGroupSymmetricGroup( 5 ) );
[ 1, 2, 1, 3, 2, 1, 4, 3, 2, 1 ]
```

83.19 CoxeterElements

`CoxeterElements(` W`[,` l`] )`

With one argument this is equivalent to `Elements(W)` — this works only if W is finite. The returned elements are sorted by increasing Coxeter length. If the second argument is an integer l, the elements of Coxeter length l are returned. The second argument can also be a list of integers, and the result is a list of same length as l of lists where the i-th list contains the elements of Coxeter length `l[i]`.

```
gap> W := CoxeterGroup( "G", 2 );;
gap> e := CoxeterElements( W, 6 );
[ ( 1, 7)( 2, 8)( 3, 9)( 4,10)( 5,11)( 6,12) ]
gap> e[1] = LongestCoxeterElement( W );
true
```

After the call to `CoxeterElements(W,l)`, the list of elements of `W` of Coxeter length `l` is stored in the component `elts[l+1]` of the record of W. There are a number of programs (like 83.23) which use the lists `W.elts`.

83.20 CoxeterWords

`CoxeterWords(` W`[,` l`] )`

With second argument the integer l returns the list of `CoxeterWord`s for all elements of `CoxeterLength` l in the Coxeter group W.

If only one argument is given, returns all elements of W as Coxeter words, in the same order as

`Concatenation(List([0..W.N],i->CoxeterWords(W,i)))`

this only makes sense for finite Coxeter groups.

```
gap> CoxeterWords( CoxeterGroup( "G", 2 ) );
[ [  ], [ 2 ], [ 1 ], [ 2, 1 ], [ 1, 2 ], [ 2, 1, 2 ], [ 1, 2, 1 ],
  [ 2, 1, 2, 1 ], [ 1, 2, 1, 2 ], [ 2, 1, 2, 1, 2 ],
  [ 1, 2, 1, 2, 1 ], [ 1, 2, 1, 2, 1, 2 ] ]
```

83.21 Bruhat

`Bruhat(` W`,` y`,` w `)`

returns `true`, if the element y is less than or equal to the element w of the Coxeter group W for the Bruhat order, and `false` otherwise (y is less than w if a reduced expression for y can be extracted from one for w). See [Hum90, (5.9) and (5.10)] for properties of the Bruhat order.

```
gap> W := CoxeterGroup( "H", 3 );;
gap> w := EltWord( W, [ 1, 2, 1, 3 ] );;
gap> b := Filtered( Elements( W ), x -> Bruhat( W, x, w) );;
gap> List( b, x -> CoxeterWord( W, x ) );
[ [  ], [ 3 ], [ 2 ], [ 1 ], [ 2, 1 ], [ 2, 3 ], [ 1, 3 ], [ 1, 2 ],
  [ 2, 1, 3 ], [ 1, 2, 1 ], [ 1, 2, 3 ], [ 1, 2, 1, 3 ] ]
```

83.22 BruhatSmaller

`BruhatSmaller(` W`,` w`)`

Returns a list whose i-th element is the list of elements of W smaller for the Bruhat order that w and of Length $i-1$. Thus the first element of the returned list contains only `W.identity` and the `CoxeterLength(W,w)`-th element contains only w.

```
gap> W:=CoxeterGroupSymmetricGroup(3);
CoxeterGroupSymmetricGroup(3)
gap> BruhatSmaller(W,(1,3));
[ [ () ], [ (2,3), (1,2) ], [ (1,2,3), (1,3,2) ], [ (1,3) ] ]
```

83.23 BruhatPoset

`BruhatPoset(` W `[,` w`])`

Returns as a poset (see 110.4) the Bruhat poset of W. If an element w is given, only the poset of the elements smaller than w is given.

```
gap> W:=CoxeterGroup("A",2);
CoxeterGroup("A",2)
gap> BruhatPoset(W);
Poset with 6 elements
gap> Display(last);
<1,2<21,12<121
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> BruhatPoset(W,EltWord(W,[1,3]));
Poset with 4 elements
gap> Display(last);
<3,1<13
```

83.24 ReducedInRightCoset

`ReducedInRightCoset(` W`,` w`)`

Let w be an element of a parent group of W whose action by conjugation induces an automorphism of Coxeter groups on W, that is sends the Coxeter generators of W to a conjugate set (but may not send the tuple of generators to a conjugate tuple). `ReducedInRightCoset` returns the unique element in the right coset $W.w$ which is W-reduced, that is which preserves the set of Coxeter generators of W.

```
gap> W:=CoxeterGroupSymmetricGroup(6);
CoxeterGroupSymmetricGroup(6)
gap> H:=ReflectionSubgroup(W,[2..4]);
ReflectionSubgroup(CoxeterGroupSymmetricGroup(6), [ 2, 3, 4 ])
gap> ReducedInRightCoset(H,(1,6)(2,4)(3,5));
(1,6)
```

83.25 ForEachElement

`ForEachElement(` W`,` f`)`

This functions calls `f(x)` for each element x of the finite Coxeter group W. It is quite useful when the `Size` of W would make impossible to call `Elements(W)`. For example,

```
gap> i:=0;;
gap> W:=CoxeterGroup("E",7);;
gap> ForEachElement(W,function(x)i:=i+1;
> if i mod 1000000=0 then Print("*\c");fi;
> end);Print("\n");
**
```

prints a * about every second on a 3Ghz computer, so enumerates 1000000 elements per second.

83.26 ForEachCoxeterWord

`ForEachCoxeterWord(` W`,` f`)`

This functions calls `f(x)` for each coxeter word x of the finite Coxeter group W. It is quite useful when the `Size` of W would make impossible to call `CoxeterWords(W)`. For example,

```
gap> i:=0;;
gap> W:=CoxeterGroup("E",7);;
gap> ForEachCoxeterWord(W,function(x)i:=i+1;
> if i mod 1000000=0 then Print("*\c");fi;
> end);Print("\n");
**
```

prints a * about every second on a 3Ghz computer, so enumerates 1000000 elements per second.

83.27 StandardParabolicClass

`StandardParabolicClass(` W, I`)`

I should be a subset of `W.reflectionsLabels` describing a subset of the generating reflections for W. The function returns the list of subsets of `W.reflectionsLabels` corresponding to sets of reflections conjugate to the given subset.

```
gap> StandardParabolicClass(CoxeterGroup("E",8),[7,8]);
[ [ 1, 3 ], [ 2, 4 ], [ 3, 4 ], [ 4, 5 ], [ 5, 6 ], [ 6, 7 ],
  [ 7, 8 ] ]
```

83.28 ParabolicRepresentatives

`ParabolicRepresentatives(`W `[,` r`])`

Returns a list of subsets of `W.reflectionsLabels` describing representatives of orbits of parabolic subgroups under conjugation by W. If a second argument r is given, returns only representatives of the parabolic subgroups of semisimple rank r.

```
gap> ParabolicRepresentatives(Affine(CoxeterGroup("A",3)));
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 1, 2, 3 ],
  [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ], [ 1, 2, 3, 4 ] ]
gap> ParabolicRepresentatives(Affine(CoxeterGroup("A",3)),2);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ] ]
```

83.29 ReducedExpressions

`ReducedExpressions(`W `,` w`)`

Returns the list of all reduced expressions of the element w of the Coxeter group W.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> ReducedExpressions(W,LongestCoxeterElement(W));
[ [ 1, 2, 1, 3, 2, 1 ], [ 1, 2, 3, 1, 2, 1 ], [ 1, 2, 3, 2, 1, 2 ],
  [ 1, 3, 2, 1, 3, 2 ], [ 1, 3, 2, 3, 1, 2 ], [ 2, 1, 2, 3, 2, 1 ],
  [ 2, 1, 3, 2, 1, 3 ], [ 2, 1, 3, 2, 3, 1 ], [ 2, 3, 1, 2, 1, 3 ],
  [ 2, 3, 1, 2, 3, 1 ], [ 2, 3, 2, 1, 2, 3 ], [ 3, 1, 2, 1, 3, 2 ],
  [ 3, 1, 2, 3, 1, 2 ], [ 3, 2, 1, 2, 3, 2 ], [ 3, 2, 1, 3, 2, 3 ],
  [ 3, 2, 3, 1, 2, 3 ] ]
```

Chapter 84

Finite Reflection Groups

Let W be a finite reflection group on the vector space V over a subfield k of the complex numbers. An efficient representation that we use in CHEVIE for computing with such group is, is a permutation representation on a W-invariant set of root and coroot vectors for reflections of W; that is, a set R of pairs $(r, r^\vee) \in V \times V^*$ invariant by W and such each distinguished reflection in W is defined by some pair in R (see 82.1). There may be several pairs for each reflection, differing by roots of unity. This generalizes the usual construction for Coxeter groups (the case $k = \mathbb{R}$) where to each reflection of W is associated two roots, a positive and a negative one. For complex reflection groups, we need at least as many roots on a given line as the order of the center of W.

The finite irreducible complex reflection groups have been completely classified by Shepard and Todd. They contain one infinite family depending on 3 parameters, and 34 "exceptional"groups which have been given by Shephard and Todd names which range from G_4 to G_{37}. They cover the exceptional Coxeter groups, e.g., `CoxeterGroup("E",8)` is the same as G_{37}.

CHEVIE provides functions to build any finite reflection group, either by giving a list of roots and corrots defining the generating reflections, or in terms of the classification. The output is a permutation group on set of roots (see `ComplexReflectionGroup` and `PermRootGroup`). In the context e.g. of Weyl groups, one wants to describe the particular root system chosen in term of the traditional classification of crystallographic root systems. This is done via calls to the function `CoxeterGroup` (see the chapter on finite Coxeter groups). There is not yet a general theory on how to construct a nice set of roots for a non-real reflection group; the roots chosen in CHEVIE where obtained case-by-case; however, they satisfy several important properties:

- The generating reflections satisfy braid relations which present the braid group associated to W (see 84.17).
- The **field of definition** of W is the field k generated by the traces of the elements of W acting on V. It is a theorem that W may be realized as a reflection group over k. For almost all irreducible complex reflection groups, the generating matrices for W given by CHEVIE have coefficients in k. Further, the set of matrices for all elements of W is globally invariant under the Galois group of $k/\mathbb{Q}$, thus the Galois action induces

automorphisms of W. The exceptions are G_{22}, G_{27} where the matrices are in a degree two extension of k (this is needed to have a globally invariant model, see [MM10b]) and some dihedral groups as well as H_3 and H_4, where the matrices given (the usual Coxeter reflection representation over k) are not globally invariant.

It turns out that all representations of a complex reflection group W are defined over the field of definition of W (cf. [Ben76] and D. Bessis thesis). This has been known for a long time in the case $k = \mathbb{Q}$, the case of Weyl groups:their representations are defined over the rationals.

- The Cartan matrix (see 82.3) for the generating roots (those which correspond to the generating reflections) has entries in the ring $\mathbb{Z}_k$ of integers of k, and the roots (resp. coroots) are linear combination with coefficients in $\mathbb{Z}_k$ of a linearly independent subset of them.

The finite reflection groups are reflection groups as described in the chapter 82, so in addition to the fields for permutation groups they have the fields `.nbGeneratingReflections`, `.OrdersGeneratingReflections` and `.reflections`. They also have the following additional fields:

`roots`
: a set of complex roots in V, given as a list of lists (vectors), on which W has a faithful permutation representation.

`simpleCoroots`
: the coroots for the first `.nbGeneratingReflections` roots.

In this chapter we describe functions available for finite reflection groups W represented as permutation groups on a set of roots. These functions make use of the classification of W whenever it is known, but work even if it is not known.

Let SV be the symmetric algebra of V. The invariants of W in SV are called the **polynomial invariants** of W. They are generated as a polynomial ring by $\dim V$ homogeneous algebraically independent polynomials $f_1, \ldots, f_{\dim V}$. The polynomials f_i are not uniquely determined but their degrees are. The f_i are called the **basic invariants** of W, and their degrees the **reflection degrees** of W. Let I be the ideal generated by the homogeneous invariants of positive degree in SV. Then SV/I is isomorphic to the regular representation of W as a W-module. It is thus a graded (by the degree of elements of SV) version of the regular representation of W. The polynomial which gives the graded multiplicity of a character χ of W in the graded module SV/I is called the **fake degree** of χ.

84.1 Functions for finite reflection groups

They are permutation groups, so all functions for permutation groups apply, although some are replaced by faster methods when available. A typical example is the function `Size`, which is obtained simply by the product of the reflection degrees, when they are known. Appropriate methods for `String` and `Print` are also defined.

`EltWord`
: Works like for Coxeter groups; in addition, as a special convention, if all `.reflectionsLabels`

of W are positive integers, negative integers are accepted and represent the inverse of the corresponding generator.

*

A*B returns the product of the two reflection groups A and B as a reflection group.

84.2 PermRootGroup

PermRootGroupNC(*roots* [,*eigenvalues*]) PermRootGroup(*roots* [,*eigenvalues*])

PermRootGroupNC(*roots*, *coroots*) PermRootGroup(*roots*, *coroots*)

roots is a list of roots, that is of vectors in some vector space. PermRootGroup returns the reflection group generated by the reflections with respect to these roots (if this group is not finite, the function will never return). The precise way the reflections are constructed as matrices is specified by the second argument. In the second form the i-th reflection is computed as Reflection(roots[i], coroots[i]). In the first form *eigenvalues* represents non-trivial eigenvalues of the reflections to construct, represented as a list of fractions n/d, where such a fraction represents the eigenvalue E(d)^n (the reason for using such a representation instead of E(d)^n is that in GAP3 it is trivial to compute E(d)^n given d/n, but the converse is hard). In this form the i-th reflection is computed as Reflection(roots[i], E(d)^n) where eigenvalues[i]=n/d. If in the first form eigenvalues are omitted, they are all assumed to be 1/2 (which represents the number -1, i.e. all reflections are true reflections).

In the (faster) variant with NC, the group is not classified (thus for instance PrintDiagram will not work).

```
gap> W:=PermRootGroupNC(IdentityMat(3),CartanMat("A",3));
PermRootGroup([ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 2, -1, 0 ], [ -1, 2, -1 ], [ 0, -1, 2 ] ])
gap> PrintDiagram(W);
Error, PermRootGroup([ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 2, -1, 0 ], [ -1, 2, -1 ], [ 0, -1, 2 ] ]) has no method
for PrintDiagram in
Dispatcher( "PrintDiagram" )( W ) called from
PrintDiagram( W ) called from
main loop
brk>
gap> ReflectionType(W);
[ rec(rank    := 3,
      series  := "A",
      indices := [ 1, 2, 3 ]) ]
gap> PrintDiagram(W);
A3 1 - 2 - 3
```

In the above, the call to ReflectionType makes CHEVIE identify the classification of W, after which functions like PrintDiagram can work. Below is another way to build a group of type A_3.

```
gap> W:=PermRootGroup([[1,0,-1],[-1,1,0],[1,0,1]]);
PermRootGroup([[1,0,-1],[-1,1,0],[1,0,1]])
gap> ReflectionDegrees(W);
[ 2, 3, 4 ]
```

84.3 ReflectionType

`ReflectionType(` W `)`

This function returns the type of W, which is a list each element of which describes an irreducible component of W; the elements of the list are objects of type `Reflection Type`, on which some functions can be called to obtain data on groups of that type, like `ReflectionDegrees`, etc...

Such an object is a record with a field `series`, the type (`"A"`, `"B"`, `"D"`, etc. . .) of the component, a field `indices`, the indices in the list of generating reflections of W where it sits, a field `rank` (equal to `Length(indices)` for well-generated complex reflection groups such as Coxeter groups, and to `Length(indices)+1` for the others).

For dihedral groups there is in addition a field `bond` giving the order of the braid relation between the two generators.

For complex reflection groups which are not real, the field `series` is equal to `"ST"`, and there is an additional field `ST`, equal either to an integer n (for exceptional reflection groups G_n), or a triple (p, q, r) of integers (for imprimitive reflection groups $G(p, q, r)$).

This function is called automatically upon construction of a finite reflection group via `PermRootGroup`, or upon constructing a finite Coxeter group by `CoxeterGroup`. But since it is sometimes costly in time (it identifies the type of the group based on the order, the degree, the order of the generators and the Cartan matrix; sometimes it needs to search for another set of generators than the given one), a version `PermRootGroupNC` is given which does not call it.

This function is called automatically prior to calling any function depending on the classification, such as `PrintDiagram`, `ReflectionName`, `ChevieClassInfo`, `BraidRelations`, `CharNames`, `CharParams`, `Representations`, `Invariants`.

```
gap> W:=ComplexReflectionGroup(4)*CoxeterGroup("A",2);;
gap> ReflectionType(W);
[ rec(series  := "ST",
      ST      := 4,
      rank    := 2,
      indices := [ 1, 2 ]), rec(rank    := 2,
      series  := "A",
      indices := [ 3, 4 ]) ]
```

`ReflectionType(` C `)`

C should be a Cartan matrix. This function determines the type of each irreducible component of C which is the Cartan matrix of a finite Coxeter group; the result is a list of Reflection types. The corresponding field is set to `false` if the corresponding submatrix of C is not the Cartan matrix of a finite Coxeter group. Going from the above example:

```
gap> C:=CartanMat(W);
[ [ -2*E(3)-E(3)^2, E(3)^2, 0, 0 ], [ -E(3)^2, -2*E(3)-E(3)^2, 0, 0 ],
  [ 0, 0, 2, -1 ], [ 0, 0, -1, 2 ] ]
gap> ReflectionType(C);
[ false, rec(rank    := 2,
      series  := "A",
```

```
        indices := [ 3, 4 ]) ]
```

Note that a Cartan matrix for a finite Coxeter group is conjugate by a diagonal matrix of the matrices for the root systems given in the introduction of the chapter on root systems. This conjugation corresponds to changing the ratio of the length between long and short roots; for example one could construct a root system for type `B` where the quotient of the two root lengths is any cyclotomic number.

```
gap> M:=[ [ 2, -E(7)^3-E(7)^5-E(7)^6 ], [ -E(7)-E(7)^2-E(7)^4, 2 ] ];;
gap> ReflectionType(M);
[ rec(rank       := 2,
      series     := "B",
      cartanType := E(7)^3+E(7)^5+E(7)^6,
      indices    := [ 1, 2 ]) ]
```

In the above example, the `cartanType` field shows that the two root lengths for `B2` have a ratio which is $\frac{1+\sqrt{-7}}{2}$.

84.4 ReflectionName

`ReflectionName(` *type* `)`

takes as argument a type *type* as returned by `ReflectionType`. Returns the name of the group system with that type, which is the concatenation of the names of its irreducible components, with `x` added in between. For reflection subgroups, it gives an indication about embedding in the parent

```
gap> C := [ [ 2, 0, -1 ], [ 0, 2, 0 ], [ -1, 0, 2 ] ];;
gap> ReflectionName( ReflectionType( C ) );
"A2xA1"
gap> ReflectionName( ReflectionType( CartanMat( "I", 2, 7 ) ) );
"I2(7)"
gap> ReflectionName(ReflectionSubgroup(CoxeterGroup("E",8),[2,3,6,7]));
"A1<2>xA1<3>xA2<6,7>.(q-1)^4"
```

`ReflectionName(` *D* `)`

The argument to `ReflectionType` can also be a record with a field `operations.ReflectionType`, and that function is then called with *rec* as argument — this works for reflection groups and reflection cosets.

84.5 IsomorphismType

`IsomorphismType(` *W* `)`

takes as argument a reflection group or a reflection coset. Returns a description of the isomorphism type of the argument.

```
gap> IsomorphismType(ReflectionSubgroup(CoxeterGroup("E",8),[2,3,6,7]));
"A2+2A1"
```

84.6 ComplexReflectionGroup

`ComplexReflectionGroup(` *STnumber* `)`

`ComplexReflectionGroup(` *p*`,` *q*`,` *r* `)`

The first form of `ComplexReflectionGroup` returns the complex reflection group which has Shephard-Todd number *STnumber*, see [ST54]. The second form returns the imprimitive complex reflection group $G(p,q,r)$.

```
gap> G := ComplexReflectionGroup( 4 );
ComplexReflectionGroup(4)
gap> ReflectionDegrees( G );
[ 4, 6 ]
gap> Size( G );
24
gap> q := X( Cyclotomics );; q.name := "q";;
gap> FakeDegrees( G, q );
[ q^0, q^4, q^8, q^7 + q^5, q^5 + q^3, q^3 + q, q^6 + q^4 + q^2 ]
gap> ComplexReflectionGroup(2,1,6);
CoxeterGroup("B",6)
```

84.7 Reflections

`Reflections(` *W* `)`

returns the list of distinguished reflections of W, as elements of W. We recall that a reflection is distinguished (see 82) if it has eigenvalue `E(e)` where `e` is the cardinality of the cyclic subgroup $C_W(H)$, where H is the hyperplane of fixed points of the reflection (all reflections are distinguished if W is generated by reflections of order 2). The generating reflections of W are `Reflections(W){W.generatingReflections}`.

```
gap> W := CoxeterGroup( "B", 2 );;
gap> Reflections( W );
[ (1,5)(2,4)(6,8), (1,3)(2,6)(5,7), (2,8)(3,7)(4,6), (1,7)(3,5)(4,8) ]
```

the code needed to obtain in general all reflections of W (not only the distinguished ones) is:

```
gap> l:=Reflections(W);;
gap> l:=Concatenation(List(l,s->List([1..Order(W,s)-1],i->s^i)));;
```

for finite Coxeter groups, `Reflections(W)` are in the same order as the positive roots. For general complex reflection groups the relationship with roots is only guaranteed for the generating reflections, that is `Reflections(W){W.generatingReflections}` are the reflections with respect to `W.roots{W.generatingReflections}`. The other reflections are not in the same order as the roots.

84.8 MatXPerm

`MatXPerm(` *W*`,` *w* `)`

Let w be a permutation of the roots of the finite reflection group W with reflection representation V. The function `MatXPerm` returns the matrix of w acting on V. This is the linear transformation of V which acts trivially on the orthogonal of the coroots and has same effect as w on the simple roots. The function makes sense more generally for an element of the normalizer of W in the whole permutation group of the roots.

```
gap> W := CoxeterGroup(
> [ [ 2, 0,-1, 0, 0, 0, 1 ], [ 0, 2, 0,-1, 0, 0, 0 ],
>   [-1, 0, 2,-1, 0, 0,-1 ], [ 0,-1,-1, 2,-1, 0, 0 ],
>   [ 0, 0, 0,-1, 2,-1, 0 ], [ 0, 0, 0, 0,-1, 2, 0 ] ],
> [ [ 1, 0, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 0 ],
>   [ 0, 0, 1, 0, 0, 0, 0 ], [ 0, 0, 0, 1, 0, 0, 0 ],
>   [ 0, 0, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 0, 1, 0 ] ] );;
gap> w0 := LongestCoxeterElement( W );;
gap> mx := MatXPerm( W, w0 );
[ [ 0, 0, 0, 0, 0, -1, 1 ], [ 0, -1, 0, 0, 0, 0, 2 ],
  [ 0, 0, 0, 0, -1, 0, 3 ], [ 0, 0, 0, -1, 0, 0, 4 ],
  [ 0, 0, -1, 0, 0, 0, 3 ], [ -1, 0, 0, 0, 0, 0, 1 ],
  [ 0, 0, 0, 0, 0, 0, 1 ] ]
```

84.9 PermMatX

`PermMatX(` W, M `)`

Let M be a linear transformation of reflection representation of W which preserves the set of roots, and thus normalizes W (remember that matrices act on the right in GAP3). `PermMatX` returns the corresponding permutation of the roots; it returns `false` if M does not normalize the set of roots.

We continue the example from `MatXPerm` and obtain:

```
gap> PermMatX( W, mx ) = w0;
true
```

84.10 MatYPerm

`MatYPerm(` W, w `)`

Let w be a permutation of the roots of the finite reflection group W with reflection representation V. The function `MatYPerm` returns the matrix of w acting on the dual vector space $V^\vee$. This is the linear transformation of $V^\vee$ which acts trivially on the orthogonal of the roots and has same effect as w on the simple coroots. The function makes sense more generally for an element of the normalizer of W in the whole permutation group of the roots.

```
gap>  W:=ReflectionSubgroup(CoxeterGroup("E",7),[1..6]);
ReflectionSubgroup(CoxeterGroup("E",7), [ 1, 2, 3, 4, 5, 6 ])
gap>  w0:=LongestCoxeterElement(W);;
gap> my:=MatYPerm(W,w0);
[ [ 0, 0, 0, 0, 0, -1, 2 ], [ 0, -1, 0, 0, 0, 0, 2 ],
  [ 0, 0, 0, 0, -1, 0, 3 ], [ 0, 0, 0, -1, 0, 0, 4 ],
  [ 0, 0, -1, 0, 0, 0, 3 ], [ -1, 0, 0, 0, 0, 0, 2 ],
  [ 0, 0, 0, 0, 0, 0, 1 ] ]
```

84.11 InvariantForm for finite reflection groups

`InvariantForm(` *W* `)`

This function returns the matrix `F` of an Hermitian form invariant under the action of the reflection group W. That is, if `M` is the matrix of an element of W, then `M*F*Transpose(ComplexConjugate(M))=F`.

```
gap> W:=ComplexReflectionGroup(4);
ComplexReflectionGroup(4)
gap> F:=InvariantForm(W);
[ [ 1, 0 ], [ 0, 2 ] ]
gap> List(W.matgens,m->m*F*ComplexConjugate(TransposedMat(m))=F);
[ true, true ]
```

84.12 ReflectionEigenvalues

`ReflectionEigenvalues(` *W* `[,` *c*`])`

Let W be a reflection group on the vector space V. `ReflectionEigenvalues(` W`)` returns the list for each conjugacy classes of the eigenvalues of an element of that class acting on V. This is returned as a list of fractions `i/n`, where such a fraction represents the eigenvalue `E(n)^i` (the reason for returning such a representation instead of `E(n)^i` is that in GAP3 it is trivial to compute `E(n)^i` given i/n, but the converse is more expensive). If a second argument c is given, returns only the list of eigenvalues of an element of the cth conjugacy class.

```
gap> W:=CoxeterGroup("A",2);
CoxeterGroup("A",2)
gap> ReflectionEigenvalues(W,3);
[ 1/3, 2/3 ]
gap> ReflectionEigenvalues(CoxeterGroup("B",2));
[ [ 0, 0 ], [ 1/2, 0 ], [ 1/2, 1/2 ], [ 1/2, 0 ], [ 1/4, 3/4 ] ]
```

84.13 ReflectionLength

`ReflectionLength(` *W*`,` *w* `)`

This function returns the number of eigenvalues of w in the reflection representation which are not equal to 1. For a finite Coxeter group, this is equal to the minimum number of reflections of which w is a product. This also holds in general in a well-generated complex reflection group if w divides a Coxeter element for the reflection length.

```
gap> W:=CoxeterGroup("A",4);
CoxeterGroup("A",4)
gap> ReflectionLength(W,LongestCoxeterElement(W));
2
gap> ReflectionLength(W,EltWord(W,[1,2,3,4]));
4
```

84.14 ReflectionWord

`ReflectionWord(` *W*`,` *w* `[,` *refs*`])`

This function return a list of minimal length of reflections of which w is the product. The reflections are represented as their index in the list of reflections (which is the index of the corresponding positive root in the list of roots). If a third argument is given, it must be a list of reflections and only these reflections are tried, and the index is with respect to this list of reflections. This function works for all elements of a Coxeter group when no third argument is given, or for w a simple of the dual braid monoid if W is a well-generated complex reflection group and *refs* is the list of atoms of this monoid.

```
gap> W:=CoxeterGroup("A",4);
CoxeterGroup("A",4)
gap> ReflectionWord(W,LongestCoxeterElement(W));
[ 6, 10 ]
gap> ReflectionWord(W,EltWord(W,[1,2,3,4]));
[ 1, 2, 3, 4 ]
```

84.15 HyperplaneOrbits

`HyperplaneOrbits(` W `)`

returns a list of records, one for each hyperplane orbit of W, containing the following fields for each orbit:

`.s`
: index of first generator in orbit

`.e_s`
: order of s

`.classno`
: if `w=W.generators[.s]` returns `List([1..e_s-1],i->PositionClass(W,w^i)`

`.N_s`
: Size of orbit

`.det_s`
: for i in `[1..e_s-1]`, position in CharTable of `(det_s)^i`

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> HyperplaneOrbits(W);
[ rec(
      s := 1,
      e_s := 2,
      classno := [ 2 ],
      N_s := 2,
      det_s := [ 5 ] ), rec(
      s := 2,
      e_s := 2,
      classno := [ 4 ],
      N_s := 2,
      det_s := [ 1 ] ) ]
```

84.16 BraidRelations

`BraidRelations(` *W* `)`

this function returns the relations which present the braid group of W. These are homogeneous (both sides of the same length) relations between generators in bijection with the generating reflections of W. A presentation of W is obtained by adding relations specifying the order of the generators.

```
gap> W:=ComplexReflectionGroup(29);
ComplexReflectionGroup(29)
gap> BraidRelations(W);
[ [ [ 1, 2, 1 ], [ 2, 1, 2 ] ], [ [ 2, 4, 2 ], [ 4, 2, 4 ] ],
  [ [ 3, 4, 3 ], [ 4, 3, 4 ] ], [ [ 2, 3, 2, 3 ], [ 3, 2, 3, 2 ] ],
  [ [ 1, 3 ], [ 3, 1 ] ], [ [ 1, 4 ], [ 4, 1 ] ],
  [ [ 4, 3, 2, 4, 3, 2 ], [ 3, 2, 4, 3, 2, 4 ] ] ]
```

each relation is represented as a pair of lists, specifying that the product of the generators according to the indices on the left side is equal to the product according to the indices on the right side. See also 84.17.

84.17 PrintDiagram

`PrintDiagram(` *W* `)` `PrintDiagram(` *type* `)`

This is a purely descriptive routine, which, by printing a diagram as in [BMR98] for W or the given reflection *type* (a Dynkin diagram for Weyl groups) shows how the generators of W are labeled in the CHEVIE presentation.

```
gap> PrintDiagram(ComplexReflectionGroup(31));
G31 4 - 2 - 5
     \ /3\ /
      1 - 3     i.e. A_5 on 14253 plus 123=231=312
```

84.18 ReflectionCharValue

`ReflectionCharValue(` *W*, *w* `)`

Returns the trace of the element w of the reflection group W as an endomorphism of the vector space V on which W acts. This could also be obtained (less efficiently) by `TraceMat(MatXPerm(W,w))`.

```
gap> W := CoxeterGroup( "A", 3 );
CoxeterGroup("A",3)
gap> List( Elements( W ), x -> ReflectionCharValue( W, x ) );
[ 3, 1, 1, 1, 0, 0, 0, -1, 0, -1, -1, 1, 1, -1, -1, -1, 0, 0, 0, 0,
  -1, -1, 1, -1 ]
```

84.19 ReflectionCharacter

`ReflectionCharacter(` *W* `)`

Returns the reflection character of the reflection group W. This could also be obtained (less efficiently) by `List(ConjugacyClasses(W),c->ReflectionCharValue(W,c))`. When W is irreducible, it can also be written `CharTable(W).irreducibles[ChevieCharInfo(W).extRefl[2]]`

```
gap> W := CoxeterGroup( "A", 3 );
CoxeterGroup("A",3)
gap> ReflectionCharacter(W);
[ 3, 1, -1, 0, -1 ]
```

84.20 ReflectionDegrees

`ReflectionDegrees(` W `)`

returns a list holding the degrees of W as a reflection group on the vector space V on which it acts. These are the degrees $d_1, \ldots, d_{\dim V}$ of the basic invariants of W in SV. They reflect various properties of W; in particular, their product is the size of W.

```
gap> W := ComplexReflectionGroup(30);
CoxeterGroup("H",4)
gap> ReflectionDegrees( W );
[ 2, 12, 20, 30 ]
gap> Size( W );
14400
```

84.21 ReflectionCoDegrees

`ReflectionCoDegrees(` W `)`

returns a list holding the codegrees of W as a reflection group on the vector space V on which it acts. These are one less than the degrees $d_1^*, \ldots, d_{\dim V}^*$ of the basic derivations of W on $SV \otimes V^\vee$.

```
gap> W := ComplexReflectionGroup(4);;
gap> ReflectionCoDegrees( W );
[ 0, 2 ]
```

84.22 GenericOrder

`GenericOrder(W,q)`

returns the "compact"generic order of `W` as a polynomial in `q`. This is $q^{N_h}\prod_i(q^{d_i}-1)$ where d_i are the reflection degrees and N_h the number of reflecting hyperplanes. For a Weyl group, it is the order of the associated semisimple finite reductive group over the field with q elements.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> GenericOrder(ComplexReflectionGroup(4),q);
q^14 - q^10 - q^8 + q^4
```

84.23 TorusOrder

`TorusOrder(W,i,q)`

returns as a polynomial in `q` the toric order of the `i`-th conjugacy class of `W`. This is the characteristic polynomial of an element of that class on the reflection representation of `W`. It is the same as the generic order of the reflection subcoset of `W` determined by the trivial subgroup and a representative of the `i`-th conjugacy class.

```
gap> W:=ComplexReflectionGroup(4);;
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> List([1..NrConjugacyClasses(W)],i->TorusOrder(W,i,q));
[ q^2 - 2*q + 1, q^2 + 2*q + 1, q^2 + 1, q^2 + (-E(3))*q + (E(3)^2),
  q^2 + (E(3))*q + (E(3)^2), q^2 + (E(3)^2)*q + (E(3)),
  q^2 + (-E(3)^2)*q + (E(3)) ]
```

84.24 ParabolicRepresentatives for reflection groups

`ParabolicRepresentatives(`W `[,` r`])`

Returns a list of subsets of `W.reflectionsLabels` describing representatives of orbits of parabolic subgroups under conjugation by W. If a second argument r is given, returns only representatives of the parabolic subgroups of semisimple rank r. Contrary to the case of Coxeter groups, it may happen that for some orbits no representative can be chosen all of whose elements are standard generators.

```
gap> ParabolicRepresentatives(ComplexReflectionGroup(3,3,3));
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 20 ], [ 2, 3 ], [ 1, 2, 3 ] ]
gap> ParabolicRepresentatives(ComplexReflectionGroup(3,3,3),2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 20 ], [ 2, 3 ] ]
```

84.25 IsParabolic

`IsParabolic(W)`

whether the reflection group W is a parabolic subgroup of `Parent(W)`.

```
gap> W:=ComplexReflectionGroup(7);
ComplexReflectionGroup(7)
gap> IsParabolic(ReflectionSubgroup(W,[1,2]));
false
gap> IsParabolic(ReflectionSubgroup(W,[1]));
true
```

84.26 ParabolicClosure

`ParabolicClosure(`W`,`I`)`

I should be a list of indices of reflections of W. Returns J such that `ReflectionSubgroup(W,J)` is the smallest parabolic subgroup of W containing `ReflectionSubgroup(W,I)`.

```
gap> W:=ComplexReflectionGroup(7);
ComplexReflectionGroup(7)
gap> ParabolicClosure(W,[1]);
[ 1 ]
gap> ParabolicClosure(W,[1,2]);
[ 1, 2, 3 ]
```

84.27 Invariants

`Invariants(` W `)`

returns the fundamental invariants of W in its reflection representation V. That is, returns a set of algebraically independent elements of the symmetric algebra of the dual of V which generate the W-invariant polynomial functions on V. Each such invariant function is returned as a GAP3 function: if $e_1, \ldots, e_n$ is a basis of V and `f` is the GAP3 function, then the value of the polynomial function on $a_1 e_1 + \ldots + a_n e_n$ is obtained by calling $f(a_1, \ldots, a_n)$. This function depends on the classification, and is dependent on the exact reflection representation of W. So for the moment it is only implemented when the reflection representation for the irreducible components has the same Cartan matrix as the one provided by CHEVIE for the corresponding irreducible group. The polynomials are invariant for the natural action of the group elements as matrices; that is, if m is `MatXPerm(W,w)` for some w in W, then an invariant f satisfies $f(a_1, \ldots, a_n) = f(v_1, \ldots, v_n)$ where $[v_1, \ldots, v_n] = [a_1, \ldots, a_n] \times m$. This action is implemented on `Mvp`s by the function `OnPolynomials` (see 112.7).

```
gap> W:=CoxeterGroup("A",2);
CoxeterGroup("A",2)
gap> i:=Invariants(W);
[ function ( arg ) ... end, function ( arg ) ... end ]
gap> a:=X(Rationals);;a.name:="a";;
gap> b:=X(RationalsPolynomials);;b.name:="b";;
gap> i[1](a,b);
(-2*a^0)*b^2 + (2*a)*b + (-2*a^2)
gap> i[2](a,b);
(-6*a)*b^2 + (6*a^2)*b
```

Another example using `Mvp` from the package VKCURVE.

```
gap> W:=ComplexReflectionGroup(24);;
gap> i:=Invariants(W);;
gap> v:=List([1..3],i->Mvp(SPrint("x",i)));
[ x1, x2, x3 ]
gap> ApplyFunc(i[1],v);
-42x1^2x2x3-12x1^2x2^2+21/2x1^2x3^2-9/2x2^2x3^2-6x2^3x3+14x1^4+18/7x2^\
4-21/8x3^4
gap> OnPolynomials(W.matgens[1],last)-last;
0
```

84.28 Discriminant

`Discriminant(` W `)`

returns the discriminant of the complex reflection group W, as a polynomial in the fundamental invariants. The discriminant is the invariant obtained by taking the product of the linear forms describing the reflecting hyperplanes of W, each raised to the order of the corresponding reflection. The discriminant is returned as a GAP3 function `f` such that the discriminant in the variables $a_1, \ldots, a_n$ is obtained by calling $f(a_1, \ldots, a_n)$. For the moment, this function is implemented only for the exceptional complex reflection groups G_4 to G_{33}.

```
gap> W:=ComplexReflectionGroup(4);
ComplexReflectionGroup(4)
gap> Discriminant(W)(x,y);
-y^2+x^3
```

84.29 Catalan

`Catalan(` n `)`

returns the n-th Catalan number.

```
gap> Catalan(8);
1430
```

`Catalan(` W `)`

returns the Catalan Number of the irreducible complex reflection group W. For well-generated groups, this number is equal to the number of simples in the dual Braid monoid. For other groups it was defined by Gordon and Griffeth ([GG12]). For Weyl groups, it also counts the number of antichains of roots.

```
gap> Catalan(CoxeterGroup("A",7));
1430
```

`Catalan(` W`,` i`)`

returns the i-th Fuss-Catalan Number of the irreducible complex reflection group W. For well-generated groups, this number is equal to the number of chains $s_1, \ldots, s_i$ of simples in the dual monoid where s_j divides s_{j+1}. For these groups, it is also equal to $\prod_j (ih + d_j)/d_j$ where the product runs over the reflection degrees of W, and where h is the Coxeter number of W. For non-well generated groups, the definition is in [GG12].

```
gap> Catalan(ComplexReflectionGroup(7),2);
16
```

`Catalan(` W`,` q`)`, resp. `Catalan(` W`,` i`,` q`)`

where q is a variable (an indeterminate or an `Mvp`) returns the q-Catalan number (resp. the i-th q-Fuss Catalan number) of W. Again the definitions in general are in [GG12].

```
gap> Catalan(ComplexReflectionGroup(7),2,x);
1+2x^12+3x^24+4x^36+3x^48+2x^60+x^72
```

Chapter 85

Root systems and finite Coxeter groups

In this chapter we describe functions for dealing with finite Coxeter groups as permutation groups of root systems. A suitable reference for the general theory is, for example, the volume of Bourbaki [Bou68]. Finite Coxeter groups coincide with finite real reflection groups. If a finite Coxeter group can be defined over the rational numbers (it is a rational reflection group), it is called a **Weyl group**.

Root systems play an important role in mathematics; they classify semi-simple Lie algebras and algebraic groups. A root system is a set of roots defining reflections (see the chapter on finite reflection groups) generating the Weyl group. We treat at the same time other finite Coxeter groups by using a generalization of root systems to the non-crystallographic (non-rational) case.

We give now the definitions. Let V be a real vector space, $V^\vee$ its dual and let $(\ ,\ )$ be the natural pairing between $V^\vee$ and V. A **root system** in V is a finite set of vectors R (the **roots**), together with a map $r \mapsto r^\vee$ from R to a subset $R^\vee$ of $V^\vee$ (the **coroots**) such that:

For any $r \in R$, we have $(r^\vee, r) = 2$ and the reflection $V \to V : x \mapsto x - (r^\vee, x)r$ with root r and coroot $r^\vee$ stabilizes R. If R does not span V we also have to impose the condition that the dual reflection $V^\vee \to V^\vee : y \mapsto y - (y, r)r^\vee$ stabilizes $R^\vee$. Note $(r^\vee, r) = 2$ is equivalent to the condition that we have true reflections (of order 2).

We will only consider **reduced** root systems, i.e., such that the only elements of R colinear with a root r are r and $-r$.

A root system R is called **crystallographic** if $(r^\vee, s)$ is an integer, for any $s \in R, r^\vee \in R^\vee$ — these are the root systems considered by Bourbaki.

The dimension of the subspace V_R of V spanned by R will be called the **semi-simple rank** of R.

The subgroup $W = W(R)$ of $\mathrm{GL}(V)$ generated by the `Reflection(`r`,`$r^\vee$`)` is a finite Coxeter group (see chapter 83 — we describe explicitly below how to obtain the Coxeter generators from the root system). If the root system is crystallographic, the representation V of W is defined over the rational numbers, thus W is a Weyl group, in which case all finite-dimensional (complex) representations of W can be realized over the rational numbers.

Weyl groups are characterized amongst finite Coxeter groups by the fact that all numbers $m(i,j)$ in the Coxeter matrix are in $\{2,3,4,6\}$.

If we identify V with $V^\vee$ by choosing a W-invariant bilinear form $(\,;\,)$, then we have $r^\vee = 2r/(r;r)$. A root system R is **irreducible** if it is not the union of two orthogonal subsets. If R is reducible then the corresponding Coxeter group is the direct product of the Coxeter groups associated with the irreducible components of R. The irreducible crystallographic root systems are classified by the following list of **Dynkin diagrams**. We show the labeling of the nodes given by the function `CartanMat` described below.

```
        1   2   3           n              1   2   3           n
  A_n   o---o---o-- . . . --o        B_n   o=<=o---o-- . . . --o

      1 o
         \    4             n              1   2   3           n
  D_n   3 o---o---  . . . --o        C_n   o=>=o---o-- . . . --o
         /
      2 o

        1   2            1   2   3   4           1   3   4   5   6
  G_2   0->-0       F_4  o---o=>=o---o       E_6 o---o---o---o---o
          6                                              |
                                                         o 2

        1   3   4   5   6   7             1   3   4   5   6   7   8
  E_7   o---o---o---o---o---o        E_8  o---o---o---o---o---o---o
                |                                 |
                o 2                               o 2
```

These diagrams encode the presentation of the Coxeter group W as follows: the vertices represent the set `S` of generating reflections; an edge is drawn between s and t if the order $m(s,t)$ of st is greater than 2; the edge is single if $m(s,t)=3$, double if $m(s,t)=4$, triple if $m(s,t)=6$. The arrows indicate the relative root lengths when W has more than one orbit on R, as explained below; we get the **Coxeter Diagram**, which describes the underlying Weyl group, if we ignore the arrows:we see that the root systems B_n and C_n correspond to the same Coxeter group.

To complete the classification of finite Coxeter groups, we need to add the following Coxeter diagrams:

```
             1   2                 1   2   3           1   2   3   4
  I_2(m)     o---o            H_3  o---o---o       H_4 o---o---o---o
               m                     5                   5
```

where a single edge has the value $m(s,t)$ written above if $m(s,t) \notin \{2,3,4,6\}$. These correspond to non-crystallographic groups, excepted for the special cases $I_2(3)=A_2$, $I_2(4)=B_2$ and $I_2(6)=G_2$.

Let us now describe how the root systems are encoded in these diagrams. Let R be a root system in V. Then we can choose a linear form on V which vanishes on no element of R. According to the sign of the value of this linear form on a root $r \in R$ we call r **positive** or **negative**. Then there exists a unique subset of the set of positive roots, called the set of **simple roots**, such that any positive root is a linear combination with non-negative coefficients of simple roots. It can be shown that any two sets of simple roots, corresponding to different choices of linear forms, can be transformed into each other by a unique element of $W(R)$. Hence, since the pairing between V and $V^\vee$ is W-invariant, if $\{r_1,\ldots,r_n\}$ is a set of simple roots and if we define the **Cartan matrix** as being the n times n matrix $C=\{r_i^\vee(r_j)\}_{ij}$, this matrix is unique up to simultaneous permutation of rows and columns. It is this matrix which is encoded in a Dynkin diagram, as follows.

The indices for the rows of C label the nodes of the diagram. The edges, for $i\neq j$, are given as follows. If C_{ij} and C_{ji} are integers such that $|C_{ij}|\geq|C_{ji}|$ the vertices are connected by $|C_{ij}|$ lines, and if $|C_{ij}|>1$ then we put an additional arrow on the lines pointing towards the node with label i. In all other cases, we simply put a single line equipped with the unique integer $p_{ij}\geq 1$ such that $C_{ij}C_{ji}=\cos^2(\pi/p_{ij})$.

It is important to note that, conversely, the whole root system can be recovered from the simple roots:the set S of reflections in $W(R)$ corresponding to the simple roots are called **simple** reflections. They are precisely the generators corresponding to the vertices of the Coxeter diagram. Each root is in the orbit of a simple root, so that R is obtained as the orbit of the simple roots under the group generated by the simple reflections. The restriction of the simple reflections to V_R is determined by the Cartan matrix, so R is determined by the Cartan matrix and the set of simple roots.

The Cartan matrix corresponding to one of the above irreducible root systems (with the specified labeling) is returned by the command `CartanMat` which takes as input a string giving the type (e.g., `"A"`, `"B"`, …, `"I"`) and a positive integer giving the rank. For type $I_2(m)$, we give as a third argument the integer m. This function returns a matrix (that is in GAP3, a list of lists) with entries in $\mathbb{Z}$ or in a cyclotomic extension of the rationals. Given two Cartan matrices, their matrix direct sum, corresponding to the orthogonal direct sum of the root systems,) can be produced by the function `DiagonalMat`.

The function `CoxeterGroup` takes as input some data which determine the roots and the coroots and produces a GAP3 permutation group record, where the Coxeter group is represented by its faithful permutation action on the root system R, with additional components holding information about R and the additional components which makes it also a Coxeter group record. If we label the positive roots by `[1 .. N]`, and the negative roots by `[N+1 .. 2*N]`, then each simple reflection is represented by the permutation of `[ 1 .. 2*N ]` which it induces on the roots.

The function `CoxeterGroup` has several forms; in one of them, the argument is the Cartan matrix of the root system This constructs a root system where the simple roots are the canonical basis of V, and the matrix of the coroots expressed in the dual basis of $V^\vee$ is then equal to the Cartan matrix.

If one only wants to work with Cartan matrices with a labeling as specified by the above list, the function call can be simplified. Instead of `CoxeterGroup( CartanMat("D", 4 ) )` the following is also possible.

```
gap> W := CoxeterGroup( "D", 4 );          # Coxeter group of type D4
CoxeterGroup("D",4)
gap> PrintArray(CartanMat(W));
[[ 2,  0, -1,  0],
 [ 0,  2, -1,  0],
```

```
 [-1, -1,  2, -1],
 [ 0,  0, -1,  2]]
```

Also, the Coxeter group record associated to a direct sum of irreducible root systems with the above standard labeling can be obtained by listing the types of the irreducible components:

```
gap> W := CoxeterGroup( "A", 2, "B", 2 );;
gap> PrintArray(CartanMat(W));
[[ 2, -1,  0,  0],
 [-1,  2,  0,  0],
 [ 0,  0,  2, -2],
 [ 0,  0, -1,  2]]
```

The same record is constructed by applying `CoxeterGroup` to the matrix `CartanMat("A",2,"B",2)` or to `DiagonalMat(CartanMat("A",2), CartanMat("B",2))`, or even by calling `CoxeterGroup("A",2)*CoxeterGro`

The following sections give more details on how to work with the elements of W and different representations for them (permutations, reduced expressions, matrices).

85.1 CartanMat for Dynkin types

`CartanMat(` *type*`,` *n* `)`

returns the Cartan matrix of Dynkin type *type* and rank n. Admissible types are the strings `"A"`, `"B"`, `"C"`, `"D"`, `"E"`, `"F"`, `"G"`, `"H"`, `"I"`, `"Bsym"`, `"Gsym"`, `"Fsym"`, `"Isym"`, `"B?"`, `"G?"`, `"F?"`, `"I?"`.

```
gap> C := CartanMat( "F", 4 );;
gap> PrintArray( C );
[[ 2, -1,  0,  0],
 [-1,  2, -1,  0],
 [ 0, -2,  2, -1],
 [ 0,  0, -1,  2]]
```

For type $I_2(m)$, which is in fact an infinity of types depending on the number m, a third argument is needed specifying the integer m so the syntax is in fact `CartanMat( "I", 2,` *m* `)`:

```
gap> CartanMat( "I", 2, 5 );
[ [ 2, E(5)^2+E(5)^3 ], [ E(5)^2+E(5)^3, 2 ] ]
```

The types like `"Bsym"` specify (non crystallographic) root systems where all roots have the same length, which is necessary for some automorphisms to exist, like the outer automorphism of B_2 which exchanges the two generating reflections:

```
gap> CartanMat("Bsym",2);
[ [ 2, -E(8)+E(8)^3 ], [ -E(8)+E(8)^3, 2 ] ]
```

Finally, for irreducible root systems which have two root lengths, the forms like `"B?"` allow to specify arbitrary root systems (up to a scalar) by giving explicitly as a third argument the coefficient by which to multiply the second conjugacy class of roots compared to the default Cartan matrix for that type.

```
gap> CartanMat("B?",2,1); #  the same as C2
[ [ 2, -1 ], [ -2, 2 ] ]
```

`CartanMat(` *type1*`,` *n1*`, ... ,` *typek*`,` *nk* `)` returns the direct sum of `CartanMat(` *type1* `,` *n1* `)`, ..., `CartanMat(` *typek* `,` *nk* `)`. One can use as argument a computed list of types by `ApplyFunc( CartanMat, [` *type1* `,` *n1* `, ... ,` *typek* `,` *nk* `] )`.

85.2 CoxeterGroup

`CoxeterGroup(` *C* `)`

`CoxeterGroup(` *type1* `,` *n1* `, ... ,` *typek* `,` *nk* `)`

`CoxeterGroup(` *rec* `)`

This function returns a permutation group record containing the basic information about the Coxeter group and the root system determined by its arguments. In the first form the canonical basis of a real vector space V of dimension `Length(`C`)` is taken as simple roots, and the lines of the matrix C express the set of coroots in the dual basis of $V^\vee$. The matrix C must be a valid Cartan matrix (see 82.3). The length of C is called the **semisimple rank** of the Coxeter datum. This function creates a **semisimple** root system, where the length of C is also equal to the dimension of V, called the **rank**. The function 88.1 can create a Coxeter group record where the rank is not equal to the semisimple rank.

The second form is equivalent to

`CoxeterGroup(CartanMat(`*type1* `,` *n1* `, ...,` *typek* `,` *nk*`))`.

The resulting record, that we will call a **Coxeter datum**, has additional entries describing various information on the root system and Coxeter group that we describe below.

The last form takes as an argument a record which has a field `coxeter` and returns the value of this field. This is used to return the Coxeter group of objects derived from Coxeter groups, such as Coxeter cosets, Hecke algebras and braid elements.

We document the following entries in a Coxeter datum record which are guaranteed to remain present in future versions of the package. Other undocumented entries should not be relied upon, they may change without notice.

`isCoxeterGroup`, `isDomain`, `isGroup`, `isPermGroup`, `isFinite`
: `true`

`cartan`
: the Cartan matrix C

`roots`
: the root vectors, given as linear combinations of simple roots. The first `N` roots are positive, the next `N` are the corresponding negative roots. Moreover, the first `SemisimpleRank(W)` roots are the simple roots. The positive roots are ordered by increasing height.

`coroots`
: the same information for the coroots. The coroot corresponding to a given root is in the same relative position in the list of coroots as the root in the list of roots.

`N`
: the number of positive roots

`rootLengths`
: the vector of length of roots the simple roots. The shortest roots in an irreducible

subsystem are given the length 1. The others then have length 2 (or 3 in type G_2). The matrix of the W-invariant bilinear form is given by `List([1..SemisimpleRank(W)], i->W.rootLengths[i]*W.cartan[i])/2`.

`orbitRepresentative`
: this is a list of same length as `roots`, which for each root, gives the smallest index of a root in the same W-orbit.

`orbitRepresentativeElements`
: a list of same length as `roots`, which for the i-th root, gives an element w of W of minimal length such that `i=orbitRepresentative[i]^w`.

`matgens`
: the matrices (in row convention — that is the matrices operate **from the right**) of the simple reflections of the Coxeter group.

`generators`
: the generators as permutations of the root vectors. They are given in the same order as the first `SemisimpleRank(W)` roots.

```
gap> W := CoxeterGroup( "A", 4 );;
gap> PrintArray( W.cartan );
[[ 2, -1,  0,  0],
 [-1,  2, -1,  0],
 [ 0, -1,  2, -1],
 [ 0,  0, -1,  2]]
gap> W.matgens;
[ [ [ -1, 0, 0, 0 ], [ 1, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 1, 1, 0, 0 ], [ 0, -1, 0, 0 ], [ 0, 1, 1, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 1, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 0, 0, -1, 0 ], [ 0, 0, 1, 1 ] ],
  [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 1 ], [ 0, 0, 0, -1 ] ]
 ]
gap> W.roots;
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ],
  [ 1, 1, 0, 0 ], [ 0, 1, 1, 0 ], [ 0, 0, 1, 1 ], [ 1, 1, 1, 0 ],
  [ 0, 1, 1, 1 ], [ 1, 1, 1, 1 ], [ -1, 0, 0, 0 ], [ 0, -1, 0, 0 ],
  [ 0, 0, -1, 0 ], [ 0, 0, 0, -1 ], [ -1, -1, 0, 0 ],
  [ 0, -1, -1, 0 ], [ 0, 0, -1, -1 ], [ -1, -1, -1, 0 ],
  [ 0, -1, -1, -1 ], [ -1, -1, -1, -1 ] ]
```

85.3 Operations and functions for finite Coxeter groups

All permutation group operations are defined on Coxeter groups, as well as all functions defined for finite reflection groups. However, the following operations and functions have been specially written to take advantage of the particular structure of real reflection groups:

`=`
: Two Coxeter data are equal if they are equal as permutation groups and the fields `simpleRoots` and `simpleCoroots` agree (independently of the value of any other bound fields).

`Print`
: prints a Coxeter group in a form that can be input back in GAP3 as a Coxeter group.

`Size`
: uses the classification of Coxeter groups to work faster (specifically, uses the function `ReflectionDegrees`).

`Elements`
: returns the set of elements. They are computed using 83.19. (Note that in an earlier version of the package the elements were sorted by length. You can get such a list by `Concatenation( List( [1..W.N], i -> CoxeterElements(W, i)))`)

`ConjugacyClasses`
: Uses classification of Coxeter groups to work faster, and the resulting list is given in the same order as the result of `ChevieClassInfo` (see 87.1). Each `Representative` given by CHEVIE has the property that it is of minimal Coxeter length in its conjugacy class and is a "good"element in the sense of [GM97].

`CharTable`
: Uses the classification of Coxeter groups to work faster, and the result has better labeling than the default (see 87).

`PositionClass`, `ClassInvariants`, `FusionConjugacyClasses`
: Use the classification of Coxeter groups to work faster.

`DecompositionMatrix(W,p)`
: Returns the p-modular decomposition matrix for Weyl groups which have no component of type `D`.

Similarly, all functions for abstract Coxeter groups are available for finite Coxeter groups. However a few of them are implemented by more efficient methods. For instance, an efficient way of coding `IsLeftDescending(W,w,s)` is `s^w>W.N` (for reflection subgroups this has to be changed slightly:elements are represented as permutations of the roots of the parent group, so one needs to write `s^w>W.parentN`, or `W.rootRestriction[s^w]>W.N`). The functions `CoxeterWord`, `CoxeterLength`, `ReducedCoxeterWord`, `IsLeftDescending`, `FirstLeftDescending`, `LeftDescentSet` and `RightDescentSet` also have a special implementation. Finally, some functions for finite reflection groups which are implemented by more efficient methods, are `ReflectionType`, `ReflectionName`, `MatXPerm`, `Reflections`, `ReflectionDegrees`, `ReflectionCharValue`.

`PrintDiagram`
: Prints the Dynkin diagram of the root system (a more specific information that the Coxeter diagram, since it includes an indication of the relative root lengths).

```
gap> C := [ [ 2, 0, -1 ], [ 0, 2, 0 ], [ -1, 0, 2 ] ];;
gap> t := ReflectionType( C );
[ rec(rank    := 2,
      series  := "A",
      indices := [ 1, 3 ]), rec(rank    := 1,
      series  := "A",
      indices := [ 2 ]) ]
gap> PrintDiagram( t );
A2 1 - 3
A1 2
gap> PrintDiagram( CoxeterGroup( "C", 3) );
C3 1 >=> 2 - 3
```

85.4 HighestShortRoot

`HighestShortRoot(` W `)`

Let W be an irreducible Coxeter group. `HighestShortRoot` computes the unique short root of maximal height of W. Note that if all roots have the same length then this is the unique root of maximal height, which can also be obtained by `W.roots[W.N]`. An error message is returned for W not irreducible.

```
gap> W := CoxeterGroup( "G", 2 );;  W.roots;
[ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 3 ],
  [ -1, 0 ], [ 0, -1 ], [ -1, -1 ], [ -1, -2 ], [ -1, -3 ],
  [ -2, -3 ] ]
gap> HighestShortRoot( W );
4
gap> W1 := CoxeterGroup( "A", 1, "B", 3 );;
gap> HighestShortRoot( W1 );
Error, CoxeterGroup("A",1,"B",3) should be irreducible
 in
HighestShortRoot( W1 ) called from
main loop
brk>
```

85.5 BadPrimes

`BadPrimes(` W `)`

`BadPrimes(` R `)`

Let W be a Weyl group. A prime is **bad** for W if it divides a coefficient on the simple roots of some root. The function `BadPrimes` returns the list of primes which are bad for W.

Alternately the argument can be a set of integer vectors and then the function returns all prime numbers which divide one of their entries.

```
gap> W:=CoxeterGroup("E",8);
CoxeterGroup("E",8)
gap> BadPrimes(W);
[ 2, 3, 5 ]
gap> BadPrimes(W.roots{[1..50]});
[ 2 ]
```

85.6 PermMatY

`PermMatY(` W, M `)`

Let M be a linear transformation of the vector space $V^\vee$ on which the Coxeter datum W acts which preserves the set of coroots. `PermMatY` returns the corresponding permutation of the coroots; it signals an error if M does not normalize the set of coroots.

```
gap>  W:=ReflectionSubgroup(CoxeterGroup("E",7),[1..6]);
ReflectionSubgroup(CoxeterGroup("E",7), [ 1, 2, 3, 4, 5, 6 ])
gap>  w0:=LongestCoxeterElement(W);;
```

```
gap> my:=MatYPerm(W,w0);;
gap> PermMatY( W, my ) = w0;
true
```

85.7 Inversions

`Inversions(` W, w `)` Returns the inversions of the element w of the finite Coxeter group W, that is, the list of the indices of roots of the parent of W sent by w to negative roots. The element w can also be given as a word $s_1 \ldots s_n$, in which case the function returns inversions in the order of the roots of the reflections $s_1, s_1 s_2 s_1, \ldots, s_1 s_2 \ldots s_n s_{n-1} \ldots s_1$.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> Inversions(W,W.1^W.2);
[ 1, 2, 4 ]
gap> Inversions(W,[1,2,1]);
[ 1, 4, 2 ]
```

85.8 ElementWithInversions

`ElementWithInversions(` W, N `)`

W should be a finite Coxeter group and N a subset of `[1..W.N]`. Returns the element w of W such that N is the list of indices of positive roots which are sent to negative roots by w. Returns false if no such element exists.

```
gap> W:=CoxeterGroup("A",2);
CoxeterGroup("A",2)
gap> List(Combinations([1..W.N]),N->ElementWithInversions(W,N));
[ (), (1,4)(2,3)(5,6), false, (1,5)(2,4)(3,6), (1,6,2)(3,5,4),
  (1,3)(2,5)(4,6), (1,2,6)(3,4,5), false ]
```

85.9 DescribeInvolution

`DescribeInvolution(` W, w `)`

Given an involution w of a Coxeter group W, by a theorem of Richardson ([Ric82]) there is a unique parabolic subgroup P of W such that that w is the longest element of P, and is central in P. `DescribeInvolution` returns I such that `P=ReflectionSubgroup(W,I)`, so that `w=LongestCoxeterElement(ReflectionSubgroup(W,I))`.

```
gap> W:=CoxeterGroup("A",2);
CoxeterGroup("A",2)
gap> w:=LongestCoxeterElement(W);
(1,5)(2,4)(3,6)
gap> DescribeInvolution(W,w);
[ 3 ]
gap> w=LongestCoxeterElement(ReflectionSubgroup(W,[3]));
true
```

85.10 ParabolicSubgroups

`ParabolicSubgroups(` W `)`

returns the list of all parabolic subgroups of W. These are the conjugates of the groups returned by `ParabolicRepresentatives(W)`; they are also in bijection with the flats of the hyperplane arrangement defined by W. To save memory, the list is given as a list of generating reflections for each group. For each element `I` of this list, one has to call `ReflectionSubgroup(W,I)` to actually get the corresponding group.

```
gap> ParabolicSubgroups(CoxeterGroup("A",3));
[ [  ], [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 1, 2 ], [ 1, 5 ],
  [ 2, 3 ], [ 3, 4 ], [ 1, 3 ], [ 2, 6 ], [ 4, 5 ], [ 1, 2, 3 ] ]
```

85.11 ExtendedReflectionGroup

`ExtendedReflectionGroup(` W`,` M `)`

This function creates an extended reflection group, which is represented as an object with two field, one recording a reflection group W on a vector space V, and the other a subgroup M of the linear group of V which normalizes W. Actually M should normalize the set of roots of W. If W is semisimple, that is `Rank(W)=SemisimpleRank(W)`, then one can give M as a group of permutations (of the roots of W), otherwise one must give M as a matrix group.

```
gap> W:=CoxeterGroup("F",4);
CoxeterGroup("F",4)
gap> D4:=ReflectionSubgroup(W,[1,2,9,16]);
ReflectionSubgroup(CoxeterGroup("F",4), [ 1, 2, 9, 16 ])
gap> t:=ReducedRightCosetRepresentatives(W,D4){[3,4]};
[ ( 2, 9)( 3,27)( 4, 7)( 5,11)(10,13)(12,15)(17,19)(20,22)(26,33)
    (28,31)(29,35)(34,37)(36,39)(41,43)(44,46),
  ( 2, 9,16)( 3, 4,31)( 5,11,18)( 6,13,10)( 7,27,28)( 8,15,12)
    (14,22,20)(17,19,21)(26,33,40)(29,35,42)(30,37,34)(32,39,36)
    (38,46,44)(41,43,45) ]
gap> ExtendedReflectionGroup(D4,Group(t,()));
Extended(D4<9,2,1,16>,(2,9),(2,9,16))
```

85.12 ClosedSubsystems

`ClosedSubsystems(` W `)`

Returns the poset (see 110.4) of closed subsystems of the root system of W. Each subsystem is represented by the indices of the roots it contains.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> p:=ClosedSubsystems(W);
Poset with 12 elements
gap> Display(p);
1 2 3 4 5 6 7 8 9 10 11 12<1 4 7 10<4 10<
1 2 3 4 5 6 7 8 9 10 11 12<1 5 6 7 11 12<1 7<
```

```
1 2 3 4 5 6 7 8 9 10 11 12<2 6 8 12<6 12<
1 2 3 4 5 6 7 8 9 10 11 12<3 5 9 11<5 11<
1 2 3 4 5 6 7 8 9 10 11 12<4 10
1 2 3 4 5 6 7 8 9 10 11 12<1 7
1 2 3 4 5 6 7 8 9 10 11 12<6 12
1 2 3 4 5 6 7 8 9 10 11 12<5 11
1 2 3 4 5 6 7 8 9 10 11 12<2 8<
1 2 3 4 5 6 7 8 9 10 11 12<3 9<
1 4 7 10<1 7
1 5 6 7 11 12<6 12
1 5 6 7 11 12<5 11
2 6 8 12<2 8
3 5 9 11<3 9
```

Chapter 86

Algebraic groups and semi-simple elements

Let us fix an algebraically closed field K and let $\mathbf{G}$ be a connected reductive algebraic group over K. Let $\mathbf{T}$ be a maximal torus of $\mathbf{G}$, let $X(\mathbf{T})$ be the character group of $\mathbf{T}$ (resp. $Y(\mathbf{T})$ the dual lattice of one-parameter subgroups of $\mathbf{T}$) and Φ (resp $\Phi^\vee$) the roots (resp. coroots) of $\mathbf{G}$ with respect to $\mathbf{T}$.

Then $\mathbf{G}$ is determined up to isomorphism by the **root datum** $(X(\mathbf{T}), \Phi, Y(\mathbf{T}), \Phi^\vee)$. In algebraic terms, this consists in giving a free $\mathbb{Z}$-lattice $X = X(\mathbf{T})$ of dimension the **rank** of $\mathbf{T}$ (which is also called the **rank** of $\mathbf{G}$), and a root system $\Phi \subset X$, and giving similarly the dual roots $\Phi^\vee \subset Y = Y(\mathbf{T})$.

This is obtained by a slight generalization of our setup for a Coxeter group W. This time we assume the canonical basis of the vector space V on which W acts is a $\mathbb{Z}$-basis of X, and Φ is specified by a matrix `W.simpleRoots` whose lines are the simple roots expressed in this basis of X. Similarly $\Phi^\vee$ is described by a matrix `W.simpleCoroots` whose lines are the simple coroots in the basis of Y dual to the chosen basis of X. The duality pairing between X and Y is the canonical one, that is the pairing between vectors $x \in X$ and $y \in Y$ is given in GAP3 by `x*y`. Thus, we must have the relation `W.simpleCoroots*TransposedMat(W.simpleRoots)=CartanMat(W)`.

We get that in CHEVIE by a new form of the function `CoxeterGroup`, where the arguments are the two matrices `W.simpleRoots` and `W.simpleCoroots` described above. The roots need not generate V, so the matrices need not be square. For instance, the root datum of the linear group of rank 3 can be specified as:

```
gap> W := CoxeterGroup( [ [ -1, 1, 0], [ 0, -1, 1 ] ],
>                       [ [ -1, 1, 0], [ 0, -1, 1 ] ] );
CoxeterGroup([[-1,1,0],[0,-1,1]],[[-1,1,0],[0,-1,1]])
gap> MatXPerm( W, W.1);
[ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ]
```

here the symmetric group on 3 letters acts by permutation of the basis of X. The dimension of X (the length of the vectors in `.simpleRoots`) is the **rank** and the dimension of the subspace generated by the roots (the length of `.simpleroots`) is called the **semi-simple rank**. In the example the rank is 3 and the semisimple rank is 2.

The default form `W:=CoxeterGroup("A",2)` defines the adjoint algebraic group (the group with a trivial center). In that case Φ is a basis of X, so `W.simpleRoots` is the identity matrix and `W.simpleCoroots` is the Cartan matrix `CartanMat(W)` of the root system. The form `CoxeterGroup("A",2,"sc")` constructs the semisimple simply connected algebraic group, where `W.simpleRoots` is the transposed of `CartanMat(W)` and `W.simpleCoroots` is the identity matrix.

There is an extreme form of root data which requires another function to specify:when W is the trivial `CoxeterGroup()` and there are thus no roots (in this case $\mathbf{G}$ is a torus), the root datum cannot be determined by the roots, but is entirely determined by the rank r. The function `Torus(`r`)` constructs such a root datum.

Finally, there is also a function `RootDatum` which understands some familiar names for the algebraic groups and gives the results that could be obtained by giving the appropriate matrices `W.simpleRoots` and `W.simpleCoroots`:

```
gap> RootDatum("gl",3);     # same as the previous example
RootDatum("gl",3)
```

Semisimple elements

It is also possible to compute with semi-simple elements. The first type are finite order elements of $\mathbf{T}$, which over an algebraically closed field K are in bijection with elements of $Y \otimes \mathbb{Q}/\mathbb{Z}$ whose denominator is prime to the characteristic of K. These are represented as elements of a vector space of rank r over $\mathbb{Q}$, taken `Mod1` whenever the need arises, where `Mod1` is the function which replaces the numerator of a fraction with the numerator `mod` the denominator; the fraction $\frac{p}{q}$ represents a primitive q-th root of unity raised to the p-th power. In this representation, multiplication of roots of unity becomes addition `Mod1` of rationals and raising to the power n becomes multiplication by n. We call this the "additive"representation of semisimple elements.

Here is an example of computations with semisimple-elements given as list of r elements of $\mathbb{Q}/\mathbb{Z}$.

```
gap> G:=RootDatum("sl",4);
RootDatum("sl",4)
gap> L:=ReflectionSubgroup(G,[1,3]);
ReflectionSubgroup(RootDatum("sl",4), [ 1, 3 ])
gap> AlgebraicCentre(L);
rec(
  Z0 :=
   SubTorus(ReflectionSubgroup(RootDatum("sl",4), [ 1, 3 ]),[ [ 1, 2, \
1 ] ]),
  AZ := Group( <0,0,1/2> ),
  ZD := Group( <1/2,0,0>, <0,0,1/2> ),
  descAZ := [ [ 1, 2 ] ] )
gap> SemisimpleSubgroup(last.Z0,3);
Group( <1/3,2/3,1/3> )
gap> e:=Elements(last);
[ <0,0,0>, <1/3,2/3,1/3>, <2/3,1/3,2/3> ]
```

First, the group $\mathbf{G} = SL_4$ is constructed, then the Levi subgroup L consisting of block-diagonal matrices of shape 2×2. The function `AlgebraicCentre` returns a record with : the neutral component Z^0 of the centre Z of L, represented by a basis of $Y(Z^0)$, a complement subtorus S of $\mathbf{T}$ to Z^0 represented similarly by a basis of $Y(S)$, and semi-simple elements representing the classes of Z modulo Z^0 , chosen in S. The classes Z/Z^0 also biject to the fundamental group as given by the field `.descAZ`, see 86.13 for an explanation. Finally the semi-simple elements of order 3 in Z^0 are computed.

```
gap> e[2]^G.2;
<1/3,0,1/3>
gap> Orbit(G,e[2]);
[ <1/3,2/3,1/3>, <1/3,0,1/3>, <2/3,0,1/3>, <1/3,0,2/3>, <2/3,0,2/3>,
  <2/3,1/3,2/3> ]
```

Since over an algebraically closed field K the points of $\mathbf{T}$ are in bijection with $Y \otimes K^\times$ it is also possible to represent any point of $\mathbf{T}$ over K as a list of r non-zero elements of K. This is the "multiplicative"representation of semisimple elements. here is the same computation as above performed with semisimple elements whose coefficients are in the finite field `GF(4)`:

```
gap> s:=SemisimpleElement(G,List([1,2,1],i->Z(4)^i));
<Z(2^2),Z(2^2)^2,Z(2^2)>
gap> s^G.2;
<Z(2^2),Z(2)^0,Z(2^2)>
gap> Orbit(G,s);
[ <Z(2^2),Z(2^2)^2,Z(2^2)>, <Z(2^2),Z(2)^0,Z(2^2)>,
  <Z(2^2)^2,Z(2)^0,Z(2^2)>, <Z(2^2),Z(2)^0,Z(2^2)^2>,
  <Z(2^2)^2,Z(2)^0,Z(2^2)^2>, <Z(2^2)^2,Z(2^2),Z(2^2)^2> ]
```

We can compute the centralizer $C_{\mathbf{G}}(s)$ of a semisimple element in $\mathbf{G}$:

```
gap> G:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
gap> Centralizer(G,s);
(A1xA1)<1,3>.(q+1)
```

The result is an extended reflection group; the reflection group part is the Weyl group of $C^0_{\mathbf{G}}(s)$ and the extended part are representatives of $C_{\mathbf{G}}(s)$ modulo $C^0_{\mathbf{G}}(s)$ taken as diagram automorphisms of the reflection part. Here is is printed as a coset $C^0_{\mathbf{G}}(s)\phi$ which generates $C_{\mathbf{G}}(s)$.

86.1 CoxeterGroup (extended form)

`CoxeterGroup(` *simpleRoots*`,` *simpleCoroots* `)`

`CoxeterGroup(` *C*`[, "sc"] )`

`CoxeterGroup(` *type1*`,` *n1*`, ... ,` *typek*`,` *nk*`[, "sc"] )`

The above are extended forms of the function `CoxeterGroup` allowing to specify more general root data. In the first form a set of roots is given explicitly as the lines of the matrix *simpleRoots*, representing vectors in a vector space V, as well as a set of coroots as the lines of the matrix *simpleCoroots* expressed in the dual basis of $V^\vee$. The product

C=*simpleCoroots*`*TransposedMat(`*simpleRoots*`)` must be a valid Cartan matrix. The dimension of V can be greater than `Length(`C`)`. The length of C is called the **semisimple rank** of the Coxeter datum, while the dimension of V is called its **rank**.

In the second form C is a Cartan matrix, and the call `CoxeterGroup(`C`)` is equivalent to `CoxeterGroup(IdentityMat(Length(`C`)),`C`)`. When the optional `"sc"` argument is given the situation is reversed: the simple coroots are given by the identity matrix, and the simple roots by the transposed of C (this corresponds to the embedding of the root system in the lattice of characters of a maximal torus in a **simply connected** algebraic group).

The argument `"sc"` can also be given in the third form with the same effect.

The following fields in a Coxeter group record complete the description of the corresponding **root datum**:

`simpleRoots`
: the matrix of simple roots

`simpleCoroots`
: the matrix of simple coroots

`matgens`
: the matrices (in row convention — that is, the matrices operate **from the right**) of the simple reflections of the Coxeter group.

86.2 RootDatum

`RootDatum(`*type*`,`n`)`

This function returns the root datum for the algebraic group described by *type* and n. The types understood as of now are:`"gl"`, `"sl"`, `"pgl"`, `"slmod"`, `"tgl"` `"sp"`, `"so"`, `"psp"`, `"csp"`, `"pso"`, `"halfspin"`, `"spin"`, `"gpin"`, `"E6"`, `"E6sc "`, `"E7"`, `"E8"`, `"F4"`, `"G2"`. Most of these names should be familar; let us explain those which may not be:

`gpin(2n)`
: is a group of type D_n simply connected and with a connected center (of dimension 2).

`slmod(n,q)`
: is the quotient of Sl_n by the subgroup of the center isomorphic to $\mathbb{Z}/q\mathbb{Z}$ (q must divide n).

`tgl(n,k)`
: is the "twisted version of Gl_n described in [Tay19, Example 5.17]

```
gap> RootDatum("spin",8);# same as CoxeterGroup("D",4,"sc")
RootDatum("spin",8)
```

86.3 Dual for root Data

`Dual(`W`)`

This function returns the dual root datum of the root datum W, describing the Langlands dual algebraic group. The fields `.simpleRoots` and `.simpleCoroots` are swapped in the dual compared to W.

```
gap> W:=CoxeterGroup("B",3);
```

```
CoxeterGroup("B",3)
gap> Dual(W);
CoxeterGroup("C",3,sc)
```

86.4 Torus

`Torus(`*rank*`)`

This function returns the CHEVIE object corresponding to the notion of a torus of dimension *rank*, a Coxeter group of semisimple rank 0 and given *rank*. This corresponds to a split torus; the extension to Coxeter cosets is more useful (see 96.11).

```
gap> Torus(3);
Torus(3)
gap> ReflectionName(last);
"(q-1)^3"
```

86.5 FundamentalGroup for algebraic groups

`FundamentalGroup(`W`)`

This function returns the fundamental group of the algebraic group defined by the Coxeter group record W. This group is returned as a group of diagram automorphisms of the corresponding affine Weyl group, that is as a group of permutations of the set of simple roots enriched by the lowest root of each irreducible component. The definition we take of the fundamental group of a (not necessarily semisimple) reductive group is $(P \cap Y(\mathbf{T}))/Q$ where P is the coweight lattice (the dual lattice in $Y(\mathbf{T}) \otimes \mathbb{Q}$ of the root lattice) and Q is the coroot latice. The bijection between elements of P/Q and diagram automorphisms is explained in the context of non-irreducible groups for example in [Bon05, §3.B].

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> FundamentalGroup(W);
Group( ( 1,12, 3, 2) )
gap> W:=CoxeterGroup("A",3,"sc");
CoxeterGroup("A",3,"sc")
gap> FundamentalGroup(W);
Group( () )
```

86.6 IntermediateGroup

`IntermediateGroup(`W`, `*indices*`)`

This computes a Weyl group record representing a semisimple algebraic group intermediate between the adjoint group — obtained by a call like `CoxeterGroup("A",3)`— and the simply connected semi-simple group — obtained by a call like `CoxeterGroup("A",3,"sc")`. The group is specified by specifying a subset of the **minuscule weights**, which are weights whose scalar product with every coroot is in $-1, 0, 1$ (the weights are the elements of the **weight lattice**, the lattice in $X(\mathbf{T}) \otimes \mathbb{Q}$ dual to the coroot lattice). The non-trivial characters of the (algebraic) center of a semi-simple simply connected algebraic group are in bijection with the minuscule weights; this set is also in bijection with P/Q where P is the weight lattice and Q is the root lattice. If W is irreducible, the minuscule weights are part of the basis of the weight lattice given by the **fundamental weights**, which is the dual basis of the simple coroots. They can thus be specified by an *index* in the Dynkin diagram (see 84.17). The constructed group has lattice $X(\mathbf{T})$ generated by the sum of the root lattice and the weights with the given *indices*. If W is not irreducible, a minuscule weight is a sum of minuscule weights in different components. An element of *indices* is thus itself a list, interpreted as representing the sum of the corresponding weights.

```
gap> W:=CoxeterGroup("A",3);;
gap> IntermediateGroup(W,[]); # adjoint
CoxeterGroup("A",3)
gap> FundamentalGroup(last);
Group( ( 1,12, 3, 2) )
gap> IntermediateGroup(W,[2]);# intermediate
CoxeterGroup([[2,0,-1],[0,1,0],[0,0,1]],[[1,-1,0],[-1,2,-1],[1,-1,2]])
gap> FundamentalGroup(last);
Group( ( 1, 3)( 2,12) )
```

86.7 WeightInfo

`WeightInfo(`*W*`)`

W is a Coxeter group record describing an algebraic group $\mathbf{G}$, or an irreducible type. The function is independent of the isogeny type of $\mathbf{G}$ (so just depends on `ReflectionType(W)`, that is on the root system). It returns a record with the following fields:

`.minusculeWeights` the minuscule weights, described as their position in the list of fundamental weights. For non-irreducible groups, a weight is the sum of at most one weight in each irreducible component. It is represented as the list of its components. For consistency, in the case of an irreducible system, a weight is represented as a one-element list.

`.minusculeCoweights` the minuscule coweights, represented in the same manner as the minuscule weights

`.decompositions` for each minuscule coweight, its decomposition in terms of the generators of the adjoint fundamental group (given by the list of the exponents of the generators). Together with the next field it enables to work out the group structure of the adjoint fundamental group.

`.moduli` the list of orders of the generators of the fundamental group.

`.AdjointFundamentalGroup` the list of generators of the adjoint fundamental group, given as permutations.

`.CenterSimplyConnected` A list of semisimple elements generating the center of the universal covering of $\mathbf{G}$ (the coweights in terms of the coroots).

`.chosenAdaptedBasis` A basis of the weight lattice adapted to the root lattice. In the basis of the fundamental weights, the root lattice is given by the Cartan matrix C. The property is that the Hermite normal form of `C*.chosenAdaptedBasis` is almost in Smith normal form (it is diagonal but the diagonal entries may be permuted compared to the Smith normal form).

```
gap> WeightInfo(CoxeterGroup("A",2,"B",2));
```

```
rec(
  minusculeWeights := [ [ 1, 3 ], [ 1 ], [ 2, 3 ], [ 2 ], [ 3 ] ],
  minusculeCoweights := [ [ 1, 4 ], [ 1 ], [ 2, 4 ], [ 2 ], [ 4 ] ],
  decompositions :=
   [ [ 2, 1 ], [ 2, 0 ], [ 1, 1 ], [ 1, 0 ], [ 0, 1 ] ],
  moduli := [ 3, 2 ],
  chosenAdaptedBasis :=
   [ [ 1, -1, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ],
  CenterSimplyConnected := [ [ 1/3, 2/3, 0, 0 ], [ 0, 0, 1/2, 0 ] ],
  AdjointFundamentalGroup := [ ( 1,12, 2), ( 4,14) ] )
```

86.8 SemisimpleElement

`SemisimpleElement(` W `,` v `[,` *additive* `])`

W should be a root datum, given as a Coxeter group record for a Weyl group, and v a list of length `W.rank`. The result is a semisimple element record, which has the fields:

`.v` the given list, taken `Mod1` if its elements are rationals.

`.group` the parent of the group W.

```
gap> G:=CoxeterGroup("A",3);;
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
```

If all elements of v are rational numbers, they are converted by `Mod1` to fractions between 0 and 1 representing roots of unity, and these roots of unity are multiplied by adding `Mod1` the fractions. In this way any semisimple element of finite order can be represented.

If the entries are not rational numbers, they are assumed to represent elements of a field which are multiplied or added normally. To explicitly control if the entries are to be treated additively or not, a third argument can be given:if `true` the entries are treated additively, or not if `false`. For entries to be treated additively, they must belong to a domain for which the method `Mod1` had been defined.

86.9 Operations for semisimple elements

The arithmetic operations `*`, `/` and `^` work for semisimple elements. They also have `Print` and `String` methods. We first give an element with elements of $\mathbb{Q}/\mathbb{Z}$ representing roots of unity.

```
gap> G:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
gap> t:=SemisimpleElement(G,[1/2,1/3,1/7]);
<1/2,1/3,1/7>
gap> s*t;
<1/2,5/6,1/7>
gap> t^3;
<1/2,0,3/7>
```

```
gap> t^-1;
<1/2,2/3,6/7>
gap> t^0;
<0,0,0>
gap> String(t);
"<1/2,1/3,1/7>"
```

then a similar example with elements of `GF(5)`

```
gap> s:=SemisimpleElement(G,Z(5)*[1,2,1]);
<Z(5),Z(5)^2,Z(5)>
gap> t:=SemisimpleElement(G,Z(5)*[2,3,4]);
<Z(5)^2,Z(5)^0,Z(5)^3>
gap> s*t;
<Z(5)^3,Z(5)^2,Z(5)^0>
gap> t^3;
<Z(5)^2,Z(5)^0,Z(5)>
gap> t^-1;
<Z(5)^2,Z(5)^0,Z(5)>
gap> t^0;
<Z(5)^0,Z(5)^0,Z(5)^0>
gap> String(t);
"<Z(5)^2,Z(5)^0,Z(5)^3>"
```

The operation `^` also works for applying an element of its defining Weyl group to a semisimple element, which allows orbit computations:

```
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
gap> s^G.2;
<1/2,1/2,1/2>
gap> Orbit(G,s);
[ <0,1/2,0>, <1/2,1/2,1/2>, <1/2,0,1/2> ]
```

The operation `^` also works for applying a root to a semisimple element:

```
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
gap> s^G.roots[4];
1/2
gap> s:=SemisimpleElement(G,Z(5)*[1,1,1]);
<Z(5),Z(5),Z(5)>
gap> s^G.roots[4];
Z(5)^2
```

`Frobenius(` *WF* `)`

If `WF` is a Coxeter coset associated to the Coxeter group W, the function `Frobenius` returns the associated automorphism which can be applied to semisimple elements, see 96.7.

```
gap> W:=CoxeterGroup("D",4);;WF:=CoxeterCoset(W,(1,2,4));;
gap> s:=SemisimpleElement(W,[1/2,0,0,0]);
<1/2,0,0,0>
```

```
gap> F:=Frobenius(WF);
function ( arg ) ... end
gap> F(s);
<0,0,0,1/2>
gap> F(s,-1);
<0,1/2,0,0>
```

86.10 Centralizer for semisimple elements

`Centralizer(` W, s`)`

W should be a Weyl group record or and extended reflection group and s a semisimple element for W. This function returns the stabilizer of the semisimple element s in W, which describes also $C_{\mathbf{G}}(s)$, if $\mathbf{G}$ is the algebraic group described by W. The stabilizer is an extended reflection group, with the reflection group part equal to the Weyl group of $C^0_{\mathbf{G}}(s)$, and the diagram automorphism part being those induced by $C_{\mathbf{G}}(s)/C^0_{\mathbf{G}}(s)$ on $C^0_{\mathbf{G}}(s)$.

```
gap> G:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> s:=SemisimpleElement(G,[0,1/2,0]);
<0,1/2,0>
gap> Centralizer(G,s);
(A1xA1)<1,3>.(q+1)
```

86.11 SubTorus

`SubTorus(`W,Y`)`

The function returns the subtorus $\mathbf{S}$ of the maximal torus $\mathbf{T}$ of the reductive group represented by the Weyl group record W such that $Y(\mathbf{S})$ is the (pure) sublattice of $Y(\mathbf{T})$ generated by the (integral) vectors Y. A basis of $Y(\mathbf{S})$ adapted to $Y(\mathbf{T})$ is computed and stored in the field `S.generators` of the returned subtorus object. Here, adapted means that there is a set of integral vectors, stored in `S.complement`, such that `M:=Concatenation(S.generators,S.complement)` is a basis of $Y(\mathbf{T})$ (equivalently $M \in \mathrm{GL}(\mathbb{Z}^{\mathrm{rank}(W)})$). An error is raised if Y does not define a pure sublattice.

```
gap> W:=CoxeterGroup("A",4);;
gap> SubTorus(W,[[1,2,3,4],[2,3,4,1],[3,4,1,2]]);
Error, not a pure sublattice in
SubTorus( W, [ [ 1, 2, 3, 4 ], [ 2, 3, 4, 1 ], [ 3, 4, 1, 2 ] ]
 ) called from
main loop
brk>
gap> SubTorus(W,[[1,2,3,4],[2,3,4,1],[3,4,1,1]]);
SubTorus(CoxeterGroup("A",4),[ [ 1, 0, 3, -13 ], [ 0, 1, 2, 7 ], [ 0,
0, 4, -3 ] ])
```

86.12 Operations for Subtori

The operation `in` can test if a semisimple element belongs to a subtorus:

```
gap> W:=RootDatum("gl",4);;
gap> r:=AlgebraicCentre(W);
rec(
  Z0 := SubTorus(RootDatum("gl",4),[ [ 1, 1, 1, 1 ] ]),
  AZ := Group( <0,0,0,0> ),
  ZD := Group( <1/4,1/4,1/4,1/4> ),
  descAZ := [ [ 1 ] ] )
gap> SemisimpleElement(W,[1/4,1/4,1/4,1/4]) in r.Z0;
true
```

The operation `Rank` gives the rank of the subtorus:

```
gap> Rank(r.Z0);
1
```

86.13 AlgebraicCentre

`AlgebraicCentre(` W `)`

W should be a Weyl group record, or an extended Weyl group record. This function returns a description of the centre $Z\mathbf{G}$ of the algebraic group $\mathbf{G}$ defined by W as a record with the following fields:

`Z0` the neutral component Z^0 of $Z\mathbf{G}$ as a subtorus of $\mathbf{T}$.

`AZ` representatives in $Z\mathbf{G}$ of $A(Z) := Z\mathbf{G}/(Z\mathbf{G})^0$ given as a group of semisimple elements.

`ZD` center of the derived subgroup of $\mathbf{G}$ given as a group of semisimple elements.

`descAZ` if W is not an extended Weyl group, describes $A(Z)$ as a quotient of the center `pi` of the simply connected covering of $\mathbf{G}$. It contains a list of elements given as words in the generators of `pi` which generate the kernel of the quotient map.

```
gap> G:=CoxeterGroup("A",3,"sc");;
gap> L:=ReflectionSubgroup(G,[1,3]);
ReflectionSubgroup(CoxeterGroup("A",3,"sc"), [ 1, 3 ])
gap> AlgebraicCentre(L);
rec(
  Z0 :=
   SubTorus(ReflectionSubgroup(CoxeterGroup("A",3,"sc"), [ 1, 3 ]),[ [\
 1, 2, 1 ] ]),
  AZ := Group( <0,0,1/2> ),
  ZD := Group( <1/2,0,0>, <0,0,1/2> ),
  descAZ := [ [ 1, 2 ] ] )
gap> G:=CoxeterGroup("A",3);;
gap> s:=SemisimpleElement(G,[0,1/2,0]);;
gap> Centralizer(G,s);
(A1xA1)<1,3>.(q+1)
gap> AlgebraicCentre(last);
rec(
  Z0 := SubTorus(ReflectionSubgroup(CoxeterGroup("A",3), [ 1, 3 ]),),
  AZ := Group( <0,1/2,0> ) )
```

Note that in versions of CHEVIE prior to april 2017, the field `Z0` was not a subtorus but what is now `Z0.generators`, and there was a field `complement` which is now `Z0.complement`.

86.14 SemisimpleSubgroup

`SemisimpleSubgroup(` S, n `)`

This function returns the subgroup of semi-simple elements of order dividing n in the subtorus S.

```
gap> G:=CoxeterGroup("A",3,"sc");;
gap> L:=ReflectionSubgroup(G,[1,3]);;
gap> z:=AlgebraicCentre(L);;
gap> z.Z0;
SubTorus(ReflectionSubgroup(CoxeterGroup("A",3,"sc"), [ 1, 3 ]),[ [ 1,\
 2, 1 ] ])
gap> SemisimpleSubgroup(z.Z0,3);
Group( <1/3,2/3,1/3> )
gap> Elements(last);
[ <0,0,0>, <1/3,2/3,1/3>, <2/3,1/3,2/3> ]
```

86.15 IsIsolated

`IsIsolated(`W,s`)`

s should be a semi-simple element for the algebraic group $\mathbf{G}$ specified by the Weyl group record W. A semisimple element s of an algebraic group $\mathbf{G}$ is isolated if the connected component $C^0_{\mathbf{G}}(s)$ does not lie in a proper parabolic subgroup of $\mathbf{G}$. This function tests this condition.

```
gap> W:=CoxeterGroup("E",6);;
gap> QuasiIsolatedRepresentatives(W);
[ <0,0,0,0,0,0>, <0,0,0,1/3,0,0>, <0,1/6,1/6,0,1/6,0>,
  <0,1/2,0,0,0,0>, <1/3,0,0,0,0,1/3> ]
gap> Filtered(last,x->IsIsolated(W,x));
[ <0,0,0,0,0,0>, <0,0,0,1/3,0,0>, <0,1/2,0,0,0,0> ]
```

86.16 IsQuasiIsolated

`IsQuasiIsolated(`W,s`)`

s should be a semi-simple element for the algebraic group $\mathbf{G}$ specified by the Weyl group record W. A semisimple element s of an algebraic group $\mathbf{G}$ is quasi-isolated if $C_{\mathbf{G}}(s)$ does not lie in a proper parabolic subgroup of $\mathbf{G}$. This function tests this condition.

```
gap> W:=CoxeterGroup("E",6);;
gap> QuasiIsolatedRepresentatives(W);
[ <0,0,0,0,0,0>, <0,0,0,1/3,0,0>, <0,1/6,1/6,0,1/6,0>,
  <0,1/2,0,0,0,0>, <1/3,0,0,0,0,1/3> ]
gap> Filtered(last,x->IsQuasiIsolated(ReflectionSubgroup(W,[1,3,5,6]),x));
[ <0,0,0,0,0,0>, <0,0,0,1/3,0,0>, <0,1/2,0,0,0,0> ]
```

86.17 QuasiIsolatedRepresentatives

`QuasiIsolatedRepresentatives(` W `[,` p `])`

W should be a Weyl group record corresponding to an algebraic group **G**. This function returns a list of semisimple elements for **G**, which are representatives of the **G**-orbits of quasi-isolated semisimple elements. It follows the algorithm given by C. Bonnafé in [Bon05]. If a second argument p is given, it gives representatives of those quasi-isolated elements which exist in characteristic p.

```
gap> W:=CoxeterGroup("E",6);;QuasiIsolatedRepresentatives(W);
[ <0,0,0,0,0,0>, <0,0,0,1/3,0,0>, <0,1/6,1/6,0,1/6,0>,
  <0,1/2,0,0,0,0>, <1/3,0,0,0,0,1/3> ]
gap> List(last,x->IsIsolated(W,x));
[ true, true, false, true, false ]
gap> W:=CoxeterGroup("E",6,"sc");;QuasiIsolatedRepresentatives(W);
[ <0,0,0,0,0,0>, <1/3,0,2/3,0,1/3,2/3>, <1/2,0,0,1/2,0,1/2>,
  <2/3,0,1/3,0,1/3,2/3>, <2/3,0,1/3,0,2/3,1/3>, <2/3,0,1/3,0,2/3,5/6>,
  <5/6,0,2/3,0,1/3,2/3> ]
gap> List(last,x->IsIsolated(W,x));
[ true, true, true, true, true, true, true ]
gap> QuasiIsolatedRepresentatives(W,3);
[ <0,0,0,0,0,0>, <1/2,0,0,1/2,0,1/2> ]
```

86.18 SemisimpleCentralizerRepresentatives

`SemisimpleCentralizerRepresentatives(` W `[,` p `])`

W should be a Weyl group record corresponding to an algebraic group **G**. This function returns a list giving representatives **H** of **G**-orbits of reductive subgroups of **G** which can be the identity component of the centralizer of a semisimple element. Each group **H** is specified by a list h of reflection indices in W such that **H** corresponds to `ReflectionSubgroup(W,h)`. If a second argument p is given, only the list of the centralizers which occur in characteristic p is returned.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> l:=SemisimpleCentralizerRepresentatives(W);
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 5 ], [ 2 ], [ 2, 6 ] ]
gap> List(last,h->ReflectionName(ReflectionSubgroup(W,h)));
[ "(q-1)^2", "A1.(q-1)", "G2", "A2<1,5>", "~A1<2>.(q-1)",
  "~A1<2>xA1<6>" ]
gap> SemisimpleCentralizerRepresentatives(W,2);
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 5 ], [ 2 ] ]
```

Chapter 87

Classes and representations for reflection groups

The CharTable of a finite complex reflection group W is computed in CHEVIE using the decomposition of W in irreducible groups (see 84.3). For each irreducible group the character table is either computed using recursive formulas for the infinite series, or read into the system from a library file for the exceptional types. Thus, character tables can be obtained quickly even for very large groups (e.g., E_8). Similar remarks apply for conjugacy classes.

The conjugacy classes and irreducible characters of irreducible finite complex reflection groups have canonical labelings by certain combinatorial objects; these labelings are used in the tables of CHEVIE. For the classes, these are partitions or partition tuples for the infinite series, or, for exceptional Coxeter groups, Carter's admissible diagrams [Car72a] (for other primitive complex reflection groups we just use words in the generators to specify the classes). For the characters, these are again partitions or partition tuples for the infinite series, and for the others they are pairs of two integers (d, e) where d is the degree of the character and e is the smallest symmetric power of the reflection representation containing the given character as a constituent (the b-invariant of the character). This information is obtained by using the functions ChevieClassInfo and ChevieCharInfo (and some of it is also available more directly via the functions CharParams, CharNames, HighestPowerFakeDegrees). When you display the character table in GAP3, the canonical labelings for classes and characters are those displayed.

A typical example is CoxeterGroup("A",n), the symmetric group $\mathfrak{S}_{n+1}$ where classes and characters are parameterized by partitions of $n+1$.

```
gap> W := CoxeterGroup( "A", 3 );;
gap> Display( CharTable( W ));
A3

      2    3    2    3    .  2
      3    1    .    .    1  .

         1111  211   22   31  4
     2P  1111 1111 1111   31 22
```

```
     3P 1111  211   22 1111  4

1111       1   -1    1    1 -1
211        3   -1   -1    .  1
22         2    .    2   -1  .
31         3    1   -1    . -1
4          1    1    1    1  1
```

The `charTable` record (computed the first time the function `CharTable` is called) is a usual character table record as defined in GAP3, but with some additional components. The components `classtext`, `classnames` contain information as described for `ChevieClassInfo` (see 87.1). There is also a field `irredinfo`, which is a list of records for each irreducible character which have components `charname` and `charparam` as described for `ChevieCharInfo` (see 87.4).

```
gap> W := CoxeterGroup( "G", 2);;
gap> ct := CharTable( W );
CharTable( "G2" )
gap> ct.classtext;
[ [  ], [ 2 ], [ 1 ], [ 1, 2 ], [ 1, 2, 1, 2 ], [ 1, 2, 1, 2, 1, 2 ] ]
gap> ct.classnames;
[ "A0", "~A1", "A1", "G2", "A2", "A1+~A1" ]
gap> ct.irredinfo;
[ rec(
      charparam := [ [ 1, 0 ] ],
      charname := "\\phi_{1,0}" ), rec(
      charparam := [ [ 1, 6 ] ],
      charname := "\\phi_{1,6}" ), rec(
      charparam := [ [ 1, 3, 1 ] ],
      charname := "\\phi_{1,3}'" ), rec(
      charparam := [ [ 1, 3, 2 ] ],
      charname := "\\phi_{1,3}''" ), rec(
      charparam := [ [ 2, 1 ] ],
      charname := "\\phi_{2,1}" ), rec(
      charparam := [ [ 2, 2 ] ],
      charname := "\\phi_{2,2}" ) ]
```

Recall that our groups acts a reflection group on the vector space V, so have fake degrees (see 87.6). The valuation and degree of these give two integers b, B for each irreducible character of W (see 87.7 and 87.8). For finite Coxeter groups, the valuation and degree of the generic degrees of the one-parameter generic Hecke algebra give two more integers a, A (see the functions 87.10, 87.11, and [Car85, Ch.11] for more details). These will also be used in the operations of truncated inductions explained in the chapter 88.

Iwahori-Hecke algebras and cyclotomic Hecke algebras also have character tables, see the corresponding chapters.

We now describe for each type our conventions for labeling the classes and characters.

Type A_n ($n \geq 0$). In this case we have $W \cong \mathfrak{S}_{n+1}$. The classes and characters are labeled by partitions of $n+1$. The partition corresponding to a class describes the cycle type for the elements in that class; the representative in `.classtext` is the concatenation of the words corresponding to each part, and to a part i is associated the product of $i-1$ consecutive generators (starting one higher that the last generator used for the previous parts). The partition corresponding to a character describes the type of the Young subgroup such that the trivial character induced from this subgroup contains that character with multiplicity 1 and such that every other character occurring in this induced character has a higher a-value. Thus, the sign character corresponds to the partition (1^{n+1}) and the trivial character to the partition $(n+1)$. The character of the reflection representation of W is labeled by $(n, 1)$.

Type B_n ($n \geq 2$). In this case $W = W(B_n)$ is isomorphic to the wreath product of the cyclic group of order 2 with the symmetric group $\mathfrak{S}_n$. Hence the classes and characters are parameterized by pairs of partitions such that the total sum of their parts equals n. The pair corresponding to a class describes the signed cycle type for the elements in that class, as in [Car72a]. We use the convention that if (λ, μ) is such a pair then λ corresponds to the positive and μ to the negative cycles. Thus, $(1^n, -)$ and $(-, 1^n)$ label the trivial class and the class containing the longest element, respectively. The pair corresponding to an irreducible character is determined via Clifford theory, as follows.

We have a semidirect product decomposition $W(B_n) = N \rtimes \mathfrak{S}_n$ where N is the standard n-dimensional $\mathcal{F}_2^n$-vector space. For $a, b \geq 0$ such that $n = a + b$ let $\eta_{a,b}$ be the irreducible character of N which takes value 1 on the first a standard basis vectors and value -1 on the next b standard basis vectors of N. Then the inertia subgroup of $\eta_{a,b}$ has the form $T_{a,b} := N.(\mathfrak{S}_a \times \mathfrak{S}_b)$ and we can extend $\eta_{a,b}$ trivially to an irreducible character $\tilde{\eta}_{a,b}$ of $T_{a,b}$. Let α and β be partitions of a and b, respectively. We take the tensor product of the corresponding irreducible characters of $\mathfrak{S}_a$ and $\mathfrak{S}_b$ and regard this as an irreducible character of $T_{a,b}$. Multiplying this character with $\tilde{\eta}_{a,b}$ and inducing to $W(B_n)$ yields an irreducible character $\chi = \chi_{(\alpha,\beta)}$ of $W(B_n)$. This defines the correspondence between irreducible characters and pairs of partitions as above.

For example, the pair $((n), -)$ labels the trivial character and $(-, (1^n))$ labels the sign character. The character of the natural reflection representation is labeled by $((n-1), (1))$.

Type D_n ($n \geq 4$). In this case $W = W(D_n)$ can be embedded as a subgroup of index 2 into the Coxeter group $W(B_n)$. The intersection of a class of $W(B_n)$ with $W(D_n)$ is either empty or a single class in $W(D_n)$ or splits up into two classes in $W(D_n)$. This also leads to a parameterization of the classes of $W(D_n)$ by pairs of partitions (λ, μ) as before but where the number of parts of μ is even and where there are two classes of this type if μ is empty and all parts of λ are even. In the latter case we denote the two classes in $W(D_n)$ by $(\lambda, +)$ and $(\lambda, -)$, where we use the convention that the class labeled by $(\lambda, +)$ contains a representative which can be written as a word in $\{s_1, s_3, \ldots, s_n\}$ and $(\lambda, -)$ contains a representative which can be written as a word in $\{s_2, s_3, \ldots, s_n\}$.

By Clifford theory the restriction of an irreducible character of $W(B_n)$ to $W(D_n)$ is either irreducible or splits up into two irreducible components. Let (α, β) be a pair of partitions with total sum of parts equal to n. If $\alpha \neq \beta$ then the restrictions of the irreducible characters of $W(B_n)$ labeled by (α, β) and (β, α) are irreducible and equal. If $\alpha = \beta$ then the restriction of the character labeled by (α, α) splits into two irreducible components which we denote by $(\alpha, +)$ and $(\alpha, -)$. Note that this can only happen if n is even. In order to fix the notation we use a result of [Ste89] which describes the value of the difference of these two characters on a class of the form $(\lambda, +)$ in terms of the character values of the symmetric group $\mathfrak{S}_{n/2}$.

Recall that it is implicit in the notation $(\lambda,+)$ that all parts of λ are even. Let λ' be the partition of $n/2$ obtained by dividing each part by 2. Then the value of $\chi_{(\alpha,-)} - \chi_{(\alpha,+)}$ on an element in the class $(\lambda,+)$ is given by $2^{k(\lambda)}$ times the value of the irreducible character of $\mathfrak{S}_{n/2}$ labeled by α on the class of cycle type λ'. (Here, $k(\lambda)$ denotes the number of non-zero parts of λ.)

The labels for the trivial, the sign and the natural reflection character are the same as for $W(B_n)$, since these characters are restrictions of the corresponding characters of $W(B_n)$.

The groups $G(d,1,n)$. They are isomorphic to the wreath product of the cyclic group of order d with the symmetric group $\mathfrak{S}_n$. Hence the classes and characters are parameterized by d-tuples of partitions such that the total sum of their parts equals n. The words chosen as representatives of the classes are, when $d>2$, computed in a slightly different way than for B_n, in order to agree with the words on which Ram and Halverson compute the characters of the Hecke algebra. First the parts of the d partitions are merged in one big partition and sorted in increasing order. Then, to a part i coming from the j-th partition is associated the word $(l+1\ldots 1\ldots l+1)^{j-1} l+2\ldots l+i$ where l is the highest generator used to express the previous part.

The d-tuple corresponding to an irreducible character is determined via Clifford theory in a similar way than for the B_n case. The identity character has the first partition with one part equal n and the other ones empty. The character of the reflection representations has the first two partitions with one part equal respectively to $n-1$ and to 1, and the other partitions empty.

The groups $G(de,e,n)$. They are normal subgroups of index e in $G(de,1,n)$. The quotient is cyclic, generated by the image g of the first generator of $G(de,1,n)$. The classes are parameterized as the classes of $G(de,e,n)$ with an extra information for a component of a class which splits.

According to [Hug85], a class C of $G(de,1,n)$ parameterized by a de-partition $(S_0,\ldots,S_{de-1})$ is in $G(de,e,n)$ if e divides $\sum_i i \sum_{p\in S_i} p$. It splits in d classes for the largest d dividing e and all parts of all S_i and such that S_i is empty if d does not divide i. If w is in C then `g^i w g^-i` for `i in [0..d-1]` are representatives of the classes of $G(de,e,n)$ which meet C. They are described by appending the integer i to the label for C.

The characters are described by Clifford theory. We make g act on labels for characters of $G(de,1,n)$. The action of g permutes circularly by d the partitions in the de-tuple. A character has same restriction to $G(de,e,n)$ as its transform by g. The number of irreducible components of its restriction is equal to the order k of its stabilizer under powers of g. We encode a character of $G(de,e,n)$ by first, choosing the smallest for lexicographical order label of a character whose restriction contains it; then this label is periodic with a motive repeated k times; we represent the character by one of these motives, to which we append $E(k)^i$ for `i in [0..k-1]` to describe which component of the restriction we choose.

Types G_2 *and* F_4. The matrices of character values and the orderings and labelings of the irreducible characters are exactly the same as in [Car85, p.412/413] and in [Kon65]:in type G_2 the character $\phi'_{1,3}$ takes the value -1 on the reflection associated to the long simple root; in type F_4, the characters $\phi'_{1,12}$, $\phi'_{2,4}$, $\phi'_{4,7}$, $\phi'_{8,9}$ and $\phi'_{9,6}$ occur in the induced of the identity from the A_2 corresponding to the short simple roots; the pairs ($\phi'_{2,16}$, $\phi''_{2,4}$) and ($\phi'_{8,3}$, $\phi''_{8,9}$) are related by tensoring by sign; and finally $\phi''_{6,6}$ is the exterior square of the reflection representation. Note, however, that in CHEVIE we put the long root at the left of the Dynkin diagrams to be in accordance with the conventions in [Lus85, (4.8) and (4.10)]; compared to [Car85], this is compensated by renaming accordingly the conjugacy class representatives.

The classes are labeled by Carter's admissible diagrams [Car72a]. A character is labeled by a pair (d, b) where d denotes the degree and b the corresponding b-invariant. If there are several characters with the same pair (d, b) we attach a prime to them, as in [Car85].

Types E_6, E_7, E_8. The character tables are obtained by specialization of those of the Hecke algebra. The classes are labeled by Carter's admissible diagrams [Car72a]. A character is labeled by the pair (d, b) where d denotes the degree and b is the corresponding b-invariant. For these types, this gives a unique labeling of the characters.

Non-crystallographic types $I_2(m)$, H_3, H_4. In these cases we do not have canonical labelings for the classes. We label them by reduced expressions.

Each character for type H_3 is uniquely determined by the pair (d, b) where d is the degree and b the corresponding b-invariant. For type H_4 there are just two characters (those of degree 30) for which the corresponding pairs are the same. These two characters are nevertheless distinguished by their fake degrees: the character $\phi'_{30,10}$ has fake degree $q^{10} + q^{12}+$ higher terms, while $\phi''_{30,10}$ has fake degree $q^{10} + q^{14}+$ higher terms. The characters in the CHEVIE-table for type H_4 are ordered in the same way as in [AL82].

Finally, the characters of degree 2 for type $I_2(m)$ are ordered as follows. The matrix representations affording the characters of degree 2 are given by:

$$\rho_j : s_1 s_2 \mapsto \begin{pmatrix} E(m)^j & 0 \\ 0 & E(m)^{-j} \end{pmatrix}, \quad s_1 \mapsto \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

where $1 \leq j \leq \lfloor (m-1)/2 \rfloor$. The reflection representation is ρ_1. The characters in the CHEVIE-table are ordered by listing first the characters of degree 1 and then $\rho_1, \rho_2, \ldots$.

Primitive complex reflection groups G_4 to G_{34}. The groups $G_{23} = H_3$, $G_{28} = F_4$, $G_{30} = H_4$ are exceptional Coxeter groups and have been explained above. Similarly for the other groups labels for characters consist primarily of the pair (d, b) where d denotes the degree and b is the corresponding b-invariant. This is sufficient for G_4, G_{12}, G_{22} and G_{24}. For other groups there are pairs or triples of characters which have the same (d, b) value. We disambiguate these according to the conventions of [Mal00] for $G_{27}, G_{29}, G_{31}, G_{33}$ and G_{34}:

- For G_{27}: The fake degree of $\phi'_{3,5}$ (resp. $\phi'_{3,20}$, $\phi''_{8,9}$) has smaller degree that of $\phi''_{3,5}$ (resp. $\phi''_{3,20}$, $\phi'_{8,9}$). The characters $\phi'_{5,15}$ and $\phi'_{5,6}$ occur with multiplicity 1 in the induced from the trivial character of the parabolic subgroup of type A_2 generated by the first and third generator (it is asserted mistakenly in [Mal00] that $\phi''_{5,6}$ does not occur in this induced; it occurs with multiplicity 2).

- For G_{29}: The character $\phi'''_{6,10}$ is the exterior square of $\phi_{4,1}$; its complex conjugate is $\phi''''_{6,10}$. The character $\phi''_{15,4}$ occurs in $\phi_{4,1} \otimes \phi_{4,3}$; the character $\phi''_{15,12}$ is tensored by the sign character from $\phi''_{15,4}$. Finally $\phi'_{6,10}$ occurs in the induced from the trivial character of the standard parabolic subgroup of type A_3 generated by the first, second and fourth generators.

- For G_{31}: The characters $\phi'_{15,8}$, $\phi'_{15,20}$ and $\phi''_{45,8}$ occur in $\phi_{4,1} \otimes \phi_{20,7}$; the character $\phi'_{20,13}$ is complex conjugate of $\phi_{20,7}$; the character $\phi'_{45,12}$ is tensored by sign of $\phi'_{45,8}$.

The two terms of maximal degree of the fakedegree of $\phi'_{30,10}$ are $q^{50}+q^{46}$ while for $\phi''_{30,10}$ they are $q^{50}+2q^{46}$.

- For G_{33}: The terms of maximal degree of the fakedegree of $\phi'_{10,8}$ are $q^{28}+q^{26}$ while for $\phi'_{10,8}$ they are $q^{28}+q^{24}$. The terms of maximal degree of the fakedegree of $\phi'_{40,5}$ are $q^{31}+q^{29}$ while for $\phi''_{40,5}$ they are $q^{31}+2q^{29}$. The character $\phi'_{10,17}$ is tensored by sign of $\phi'_{10,8}$ and $\phi'_{40,14}$ is tensored by sign of $\phi'_{40,5}$.
- For G_{34}: The character $\phi'_{20,33}$ occurs in $\phi_{6,1}\otimes\phi_{15,14}$. The character $\phi'_{70,9}$ is rational. The character $\phi''_{70,9}$ occurs in $\phi_{6,1}\otimes\phi_{15,14}$. The character $\phi'_{70,45}$ is rational.The character $\phi''_{70,45}$ is tensored by the determinant character of $\phi''_{70,9}$. The character $\phi'_{560,18}$ is rational. The character $\phi'''_{560,18}$ occurs in $\phi_{6,1}\otimes\phi_{336,17}$. The character $\phi'_{280,12}$ occurs in $\phi_{6,1}\otimes\phi_{336,17}$. The character $\phi''_{280,30}$ occurs in $\phi_{6,1}\otimes\phi_{336,17}$. The character $\phi'_{540,21}$ occurs in $\phi_{6,1}\otimes\phi_{105,20}$. The character $\phi'_{105,8}$ is complex conjugate of $\phi_{105,4}$, and $\phi'_{840,13}$ is complex conjugate of $\phi_{840,11}$. The character $\phi'_{840,23}$ is complex conjugate of $\phi_{840,19}$. Finally $\phi'_{120,21}$ occurs in induced from the trivial character of the standard parabolic subgroup of type A_5 generated by the generators of G_{34} with the third one omitted.

For the groups G_5 and G_7 we adopt the following conventions. For G_5 they are compatible with those of [MR03] and [BMM14].

- For G_5: We let `W:=ComplexReflectionGroup(5)`, so the generators in CHEVIE are `W.1` and `W.2`.

  The character $\phi'_{1,4}$ (resp. $\phi'_{1,12}$, $\phi'_{2,3}$) takes the value 1 (resp. `E(3)`, `-E(3)`) on `W.1`. The character $\phi''_{1,8}$ is complex conjugate to $\phi_{1,16}$, and the character $\phi'_{1,8}$ is complex conjugate to $\phi'_{1,4}$. The character $\phi''_{2,5}$ is complex conjugate to $\phi_{2,1}$; $\phi'_{2,5}$ take the value -1 on `W.1`. The character $\phi'_{2,7}$ is complex conjugate to $\phi'_{2,5}$.
- For G_7: We let `W:=ComplexReflectionGroup(7)`, so the generators in CHEVIE are `W.1`, `W.2` and `W.3`.

  The characters $\phi'_{1,4}$ and $\phi'_{1,10}$ take the value 1 on `W.2`. The character $\phi''_{1,8}$ is complex conjugate to $\phi_{1,16}$ and $\phi'_{1,8}$ is complex conjugate to $\phi'_{1,4}$. The characters $\phi'_{1,12}$ and $\phi'_{1,18}$ take the value `E(3)` on `W.2`. The character $\phi''_{1,14}$ is complex conjugate to $\phi_{1,22}$ and $\phi'_{1,14}$ is complex conjugate to $\phi'_{1,10}$. The character $\phi'_{2,3}$ takes the value `-E(3)` on `W.2` and $\phi'_{2,13}$ takes the value -1 on `W.2`. The characters $\phi''_{2,11}$, $\phi''_{2,5}$, $\phi'''_{2,7}$ and $\phi_{2,1}$ are Galois conjugate, as well as the characters $\phi'_{2,7}$, $\phi'_{2,13}$, $\phi'_{2,11}$ and $\phi'_{2,5}$. The character $\phi'_{2,9}$ is complex conjugate to $\phi_{2,15}$ and $\phi'''_{2,9}$ is complex conjugate to $\phi'_{2,3}$.

Finally, for the remaining groups G_6, G_8 to G_{11}, G_{13} to G_{21}, G_{25}, G_{26}, G_{32} and G_{33} there are only pairs of characters with same value (d,b). We give labels uniformly to these characters by applying in order the following rules :

- If the two characters have different fake degrees, label $\phi'_{d,b}$ the one whose fake degree is minimal for the lexicographic order of polynomials (starting with the highest term).
- For the not yet labeled pairs, if only one of the two characters has the property that in its Galois orbit at least one character is distinguished by its (d,b)-invariant, label it $\phi'_{d,b}$.

- For the not yet labeled pairs, if the minimum of the (d, b)-value (for the lexicographic order (d, b)) in the Galois orbits of the two character is different, label $\phi'_{d,b}$ the character with the minimal minimum.
- We define now a new invariant for characters: consider all the pairs of irreducible characters χ and ψ uniquely determined by their (d, b)-invariant such that ϕ occurs with non-zero multiplicity m in $\chi \otimes \psi$. We define $t(\phi)$ to be the minimal (for lexicographic order) possible list $(d(\chi), b(\chi), d(\psi), b(\psi), m)$.

  For the not yet labeled pairs, if the t-invariants are different, label $\phi'_{d,b}$ the character with the minimal t-invariant.

After applying the last rule all the pairs will be labelled for the considered groups. The labelling obtained is compatible for G_{25}, G_{26}, G_{32} and G_{33} with that of [Mal00] and for G_8 with that described in [MR03].

We should emphasize that for all groups with a few exceptions, the parameters for characters do not depend on any non-canonical choice. The exceptions are $G(de, e, n)$ with $e > 1$, and G_5, G_7, G_{27}, G_{28}, G_{29} and G_{34}, groups which admit outer automorphisms preserving the set of reflections, so choices of a particular value on a particular generator must be made for characters which are not invariant by these automorphisms.

Labels for the classes. For the exceptional complex reflection groups, the labels for the classes represent the decomposition of a representative of the class as a product of generators, with the additional conventions that `z` represents the generator of the center and for well-generated groups `c` represents a Coxeter element (a product of the generators which is a regular element for the highest reflection degree).

87.1 ChevieClassInfo

`ChevieClassInfo(` *W* `)`

returns information about the conjugacy classes of the finite reflection group W. The result is a record with the following components:

`classtext`
: contains words in the generators describing representatives of each conjugacy class. Each word is a list of integers where the generator s_i is represented by the integer i. For finite Coxeter groups, it is the same as `List(ConjugacyClasses(W),x->CoxeterWord(W,Representat` and each such representative is of minimal length in its conjugacy class and is a "very good"element in the sense of [GM97].

`classparams`
: The elements of this list are tuples which have one component for each irreducible component of W. These components for the infinite series, contain partitions or partition tuples describing the class (see the introduction). For the exceptional Coxeter groups they contain Carter's admissible diagrams, see [Car72a]. For exceptional complex reflection groups they contain in general the same information as in classtext.

`classnames`
: Contains strings describing the conjugacy classes, made out of the information in `classparams`.

```
gap> ChevieClassInfo(CoxeterGroup( "D", 4 ));
rec(
  classtext :=
   [ [  ], [ 1, 2 ], [ 1, 2, 3, 1, 2, 3, 4, 3, 1, 2, 3, 4 ], [ 1 ],
      [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 4 ], [ 2, 4 ],
      [ 1, 3, 1, 2, 3, 4 ], [ 1, 3 ], [ 1, 2, 3, 4 ], [ 1, 4, 3 ],
      [ 2, 4, 3 ] ],
  classparams :=
   [ [ [ [ 1, 1, 1, 1 ], [  ] ] ], [ [ [ 1, 1 ], [ 1, 1 ] ] ],
      [ [ [  ], [ 1, 1, 1, 1 ] ] ], [ [ [ 2, 1, 1 ], [  ] ] ],
      [ [ [ 1 ], [ 2, 1 ] ] ], [ [ [ 2 ], [ 1, 1 ] ] ],
      [ [ [ 2, 2 ], '+' ] ], [ [ [ 2, 2 ], '-' ] ],
      [ [ [  ], [ 2, 2 ] ] ], [ [ [ 3, 1 ], [  ] ] ],
      [ [ [  ], [ 3, 1 ] ] ], [ [ [ 4 ], '+' ] ], [ [ [ 4 ], '-' ] ] ],
  classnames := [ "1111.", "11.11", ".1111", "211.", "1.21", "2.11",
      "22.+", "22.-", ".22", "31.", ".31", "4.+", "4.-" ])
gap> ChevieClassInfo(ComplexReflectionGroup(3,1,2));
rec(
  classparams :=
   [ [ [ [ 1, 1 ], [  ], [  ] ] ], [ [ [ 1 ], [ 1 ], [  ] ] ],
      [ [ [ 1 ], [  ], [ 1 ] ] ], [ [ [  ], [ 1, 1 ], [  ] ] ],
      [ [ [  ], [ 1 ], [ 1 ] ] ], [ [ [  ], [  ], [ 1, 1 ] ] ],
      [ [ [ 2 ], [  ], [  ] ] ], [ [ [  ], [ 2 ], [  ] ] ],
      [ [ [  ], [  ], [ 2 ] ] ] ],
  classtext :=
   [ [  ], [ 1 ], [ 1, 1 ], [ 1, 2, 1, 2 ], [ 1, 1, 2, 1, 2 ],
      [ 1, 1, 2, 1, 2, 2, 1, 2 ], [ 2 ], [ 1, 2 ], [ 1, 1, 2 ] ],
  classnames := [ "11..", "1.1.", "1..1", ".11.", ".1.1", "..11",
      "2..", ".2.", "..2" ])
```

See also the introduction of this section.

87.2 CharNames for reflection groups

`CharNames(` *W* `[,`*options*`] )`

returns the list of character names for the reflection group W. The optional *options* is a record which can give alternative names in certain cases, or a different formatting of names in general.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> CharNames(W);
[ "phi{1,0}", "phi{1,6}", "phi{1,3}'", "phi{1,3}''", "phi{2,1}",
  "phi{2,2}" ]
gap> CharNames(W,rec(TeX:=true));
[ "\\phi_{1,0}", "\\phi_{1,6}", "\\phi_{1,3}'", "\\phi_{1,3}''",
  "\\phi_{2,1}", "\\phi_{2,2}" ]
gap> CharNames(W,rec(spaltenstein:=true));
[ "1", "eps", "epsl", "epsc", "theta'", "theta''" ]
```

```
gap> CharNames(W,rec(spaltenstein:=true,TeX:=true));
[ "1", "\\varepsilon", "\\varepsilon_l", "\\varepsilon_c",
  "\\theta'", "\\theta''" ]
```

The last two commands show the character names used by Spaltenstein and Lusztig when describing the Springer correspondence.

87.3 CharParams for reflection groups

`CharParams(` W `)`

this function returns the list of parameters for irreducible characters of W:partitions for type `A`, double partitions for type `B`, etc... as described in the introduction. For exceptional groups they are pairs or triples, beginning with the dimension, the valuation of the fake degree, and an ordinal number if more than one character shares the first two invariants.

```
gap> CharParams(CoxeterGroup("G",2));
[ [ [ 1, 0 ] ], [ [ 1, 6 ] ], [ [ 1, 3, 1 ] ], [ [ 1, 3, 2 ] ],
  [ [ 2, 1 ] ], [ [ 2, 2 ] ] ]
```

87.4 ChevieCharInfo

`ChevieCharInfo(` W `)`

returns information about the irreducible characters of the finite reflection group W. The result is a record with the following components:

`charparams`
: contains parameters for the irreducible characters as described in the introduction or returned by `CharParams(`W`)`. The parameters are tuples with one component for each irreducible irreducible component of W (as given by `ReflectionType`). For an irreducible component which is an imprimitive reflection group the component of the `charparam` is a tuple of partitions, and for a primitive irreducible group it is a pair (d, e) where d is the degree of the character and e is the smallest symmetric power of the character of the reflection representation which contains the given character as a component.

`charnames`
: strings describing the irreducible characters, computed from the `charparams`. This is the same as `CharNames(`W`)`.

`positionId`
: the position of the trivial character in the character table of W (which is also returned by the function `PositionId`).

`positionDet`
: Contains the position of the determinant character in the character table of W (which is also returned by the function `PositionDet`). For Coxeter groups this is the sign character.

`extRefl`
: Only present if W is irreducible, in which case the reflection representation of W and all its exterior powers are irreducible. It then contains the position of the exterior powers of the reflection representation in the character table.

`b`
: contains the result of `LowestPowerFakeDegrees(` W `)`.

`B`
: contains the result of `HighestPowerFakeDegrees(` W `)`.

`a`
: Contains the result of `LowestPowerGenericDegrees(` W `)`.

`A`
: Contains the result of `HighestPowerGenericDegrees(` W `)`.

`hgal`
: Contains the permutation of the characters obtained by composing the Opdam involution with complex conjugation. This permutation has an interpretation as a Galois action on the characters of `H:=Hecke(W,x)` where `x:=Indeterminate(Cyclotomics)`:if `H` splits by taking v an e-th root of x, `.hgal` records the permutation effected by the Galois action `v->E(e)*v`.

```
gap> ChevieCharInfo(ComplexReflectionGroup(22));
rec(
  extRefl := [ 1, 5, 2 ],
  charparams :=
   [ [ [ 1, 0 ] ], [ [ 1, 30 ] ], [ [ 2, 11 ] ], [ [ 2, 13 ] ],
      [ [ 2, 1 ] ], [ [ 2, 7 ] ], [ [ 3, 2 ] ], [ [ 3, 6 ] ],
      [ [ 3, 12 ] ], [ [ 3, 16 ] ], [ [ 4, 3 ] ], [ [ 4, 6 ] ],
      [ [ 4, 9 ] ], [ [ 4, 8 ] ], [ [ 5, 4 ] ], [ [ 5, 10 ] ],
      [ [ 6, 7 ] ], [ [ 6, 5 ] ] ],
  hgal := ( 3, 5)( 4, 6)(11,13)(12,14)(17,18),
  b := [ 0, 30, 11, 13, 1, 7, 2, 6, 12, 16, 3, 6, 9, 8, 4, 10, 7, 5 ],
  charnames := [ "\\phi_{1,0}", "\\phi_{1,30}", "\\phi_{2,11}",
      "\\phi_{2,13}", "\\phi_{2,1}", "\\phi_{2,7}", "\\phi_{3,2}",
      "\\phi_{3,6}", "\\phi_{3,12}", "\\phi_{3,16}", "\\phi_{4,3}",
      "\\phi_{4,6}", "\\phi_{4,9}", "\\phi_{4,8}", "\\phi_{5,4}",
      "\\phi_{5,10}", "\\phi_{6,7}", "\\phi_{6,5}" ],
  positionId := 1,
  positionDet := 2,
  B := [ 0, 30, 19, 17, 29, 23, 18, 14, 28, 24, 27, 22, 21, 24, 20,
      26, 23, 25 ],
  a := [ 0, 30, 3, 3, 3, 3, 1, 1, 11, 11, 3, 3, 3, 3, 2, 8, 3, 3 ],
  A := [ 0, 30, 27, 27, 27, 27, 19, 19, 29, 29, 27, 27, 27, 27, 22,
      28, 27, 27 ] )
gap>  ChevieCharInfo( CoxeterGroup( "G", 2 ) );
rec(
  charparams :=
   [ [ [ 1, 0 ] ], [ [ 1, 6 ] ], [ [ 1, 3, 1 ] ], [ [ 1, 3, 2 ] ],
      [ [ 2, 1 ] ], [ [ 2, 2 ] ] ],
  extRefl := [ 1, 5, 2 ],
  a := [ 0, 6, 1, 1, 1, 1 ],
  A := [ 0, 6, 5, 5, 5, 5 ],
  b := [ 0, 6, 3, 3, 1, 2 ],
```

```
    B := [ 0, 6, 3, 3, 5, 4 ],
    spaltenstein :=
     [ "1", "\\varepsilon", "\\varepsilon_l", "\\varepsilon_c",
        "\\theta'", "\\theta''" ],
    charnames := [ "\\phi_{1,0}", "\\phi_{1,6}", "\\phi_{1,3}'",
        "\\phi_{1,3}''", "\\phi_{2,1}", "\\phi_{2,2}" ],
    positionId := 1,
    positionDet := 2 )
```

For irreducible groups, the returned record contains sometimes additional information:

for F_4
: the field `kondo` gives the labeling of the characters given by Kondo, also used in [Lus85, (4.10)].

for E_6, E_7, E_8
: the field `frame` gives the labeling of the characters given by Frame, also used in [Lus85, (4.11), (4.12), and (4.13)].

for G_2
: the field `spaltenstein` gives the labeling of the characters given by Spaltenstein.

for $G(de, e, 2)$ even e and $d > 1$
: the field `malle` gives the parameters for the characters used by Malle in [Mal96].

87.5 FakeDegrees

`FakeDegrees(` W`,` q `)`

returns a list holding the fake degrees of the reflection group W on the vector space V, evaluated at q. These are the graded multiplicities of the irreducible characters of W in the quotient SV/I where SV is the symmetric algebra of V and I is the ideal generated by the homogeneous invariants of positive degree in SV. The ordering of the result corresponds to the ordering of the characters in `CharTable(W)`.

```
gap> q := X( Rationals );; q.name := "q";;
gap> FakeDegrees( CoxeterGroup( "A", 2 ), q );
[ q^3, q^2 + q, q^0 ]
```

87.6 FakeDegree

`FakeDegree(` W`,` phi`,` q `)`

returns the fake degree of the character of parameter phi (see 103.8) of the reflection group W, evaluated at q (see 87.5 for a definition of the fake degrees).

```
gap> q := X( Rationals );; q.name := "q";;
gap> FakeDegree( CoxeterGroup( "A", 2 ), [ [ 2, 1 ] ], q );
q^2 + q
```

87.7 LowestPowerFakeDegrees

`LowestPowerFakeDegrees(` W `)`

return a list holding the b-function for all irreducible characters of W, that is, for each character χ, the valuation of the fake degree of χ. The ordering of the result corresponds to the ordering of the characters in `CharTable(W)`. The advantage of this function compared to calling `FakeDegrees` is that one does not have to provide an indeterminate, and that it may be much faster to compute than the fake degrees.

```
gap> LowestPowerFakeDegrees( CoxeterGroup( "D", 4 ) );
[ 6, 6, 7, 12, 4, 3, 6, 2, 2, 4, 1, 2, 0 ]
```

87.8 HighestPowerFakeDegrees

`HighestPowerFakeDegrees(` W `)`

returns a list holding the B-function for all irreducible characters of W, that is, for each character χ, the degree of the fake degree of χ. The ordering of the result corresponds to the ordering of the characters in `CharTable(W)`. The advantage of this function compared to calling `FakeDegrees` is that one does not have to provide an indeterminate, and that it may be much faster to compute than the fake degrees.

```
gap> HighestPowerFakeDegrees( CoxeterGroup( "D", 4 ) );
[ 10, 10, 11, 12, 8, 9, 10, 6, 6, 8, 5, 6, 0 ]
```

87.9 Representations

`Representations(` W`[,` l`])`

returns a list holding, for each irreducible character of the complex reflection group W, a list of matrices images of the generating reflections of W in a model of the corresponding representation. This function is based on the classification, and is not yet fully implemented for G_{34}; 78 representations are missing out of 169, that is, representations of dim. ¿=140, except half of those of dimensions 315, 420 and 840.

If there is a second argument, it can be a list of indices (or a single integer) and only the representations with these indices (or that index) in the list of all representations are returned.

```
gap> Representations(CoxeterGroup("B",2));
[ [ [ [ 1 ] ], [ [ -1 ] ] ],
  [ [ [ 1, 0 ], [ -1, -1 ] ], [ [ 1, 2 ], [ 0, -1 ] ] ],
  [ [ [ -1 ] ], [ [ -1 ] ] ], [ [ [ 1 ] ], [ [ 1 ] ] ],
  [ [ [ -1 ] ], [ [ 1 ] ] ] ]
gap> Representations(ComplexReflectionGroup(4),7);
[ [ [ E(3)^2, 0, 0 ], [ 2*E(3)^2, E(3), 0 ], [ E(3), 1, 1 ] ],
  [ [ 1, -1, E(3) ], [ 0, E(3), -2*E(3)^2 ], [ 0, 0, E(3)^2 ] ] ]
```

87.10 LowestPowerGenericDegrees

`LowestPowerGenericDegrees(` W `)`

returns a list holding the a-function for all irreducible characters of the Coxeter group or Spetsial reflection group W, that is, for each character χ, the valuation of the generic degree of χ (in the one-parameter Hecke algebra `Hecke(W,X(Cyclotomics))` corresponding to W). The ordering of the result corresponds to the ordering of the characters in `CharTable(W)`.

```
gap> LowestPowerGenericDegrees( CoxeterGroup( "D", 4 ) );
[ 6, 6, 7, 12, 3, 3, 6, 2, 2, 3, 1, 2, 0 ]
```

87.11 HighestPowerGenericDegrees

`HighestPowerGenericDegrees(` W `)`

returns a list holding the A-function for all irreducible characters of the Coxeter group or Spetsial reflection group W, that is, for each character χ, the degree of the generic degree of χ (in the one-parameter Hecke algebra `Hecke(W,X(Cyclotomics))` corresponding to W). The ordering of the result corresponds to the ordering of the characters in `CharTable(W)`.

```
gap> HighestPowerGenericDegrees( CoxeterGroup( "D", 4 ) );
[ 10, 10, 11, 12, 9, 9, 10, 6, 6, 9, 5, 6, 0 ]
```

87.12 PositionDet

`PositionDet(` W `)`

return the position of the determinant character in the character table of the group W (for Coxeter groups this is the sign character).

```
gap> W := CoxeterGroup( "D", 4 );;
gap> PositionDet( W );
4
```

See also `ChevieCharInfo` (87.4).

87.13 DetPerm

`DetPerm(` W `)`

return the permutation of the characters of the reflection group W which is effected when tensoring by the determinant character (for Coxeter groups this is the sign character).

```
gap> W := CoxeterGroup( "D", 4 );;
gap> DetPerm( W );
[ 8, 9, 11, 13, 5, 6, 12, 1, 2, 10, 3, 7, 4 ]
```

Chapter 88

Reflection subgroups

Let W be a finite (possibly complex) reflection group on the vector space V. A **reflection subgroup** H of W is a subgroup generated by the reflections it contains. A **parabolic subgroup** of W is the fixator in W of some subset of V. By a difficult theorem of Steinberg (easy in the real case) a parabolic subgroup is a reflection subgroup.

The function `ReflectionSubgroup` can be used to construct a reflection subgroup of W. It takes as arguments the original record for W and a list of indices for the reflections.

If V is real, so that W is a Coxeter group with generators S, then $\{wsw^{-1} \mid w \in W, s \in S\}$ is the set of all reflections in W. A reflection subgroup generated by a subset of S is parabolic; it is called **standard parabolic subgroup** of W. Any parabolic subgroup is conjugate to some standard parabolic subgroup. Let R be the set of roots of W, and let Q be the set of roots of H, that is the set of roots for which the corresponding reflection lies in H; by Steinberg's theorem it can be seen that a reflection subgroup H which is not parabolic is characterized by the fact that Q is not closed under linear combinations in R.

It is a theorem discovered by Deodhar [Deo89] and Dyer [Dye90] independently at the same time that a reflection subgroup H of a Coxeter group has a canonical set of fundamental roots even if it is not parabolic: a fundamental system of roots for H is given by the positive roots $t \in Q$ such that the subset of R of roots whose sign is changed by the reflection with root t meets Q in the single element t. This is used by the routine `ReflectionSubgroup` to determine the root system Q of a reflection subgroup H.

```
gap> W := CoxeterGroup( "G", 2 );
CoxeterGroup("G",2)
gap> W.roots[4];
[ 1, 2 ]
gap> H := ReflectionSubgroup( W, [ 2, 4 ] );
ReflectionSubgroup(CoxeterGroup("G",2), [ 2, 3 ])
gap> PrintDiagram( H );          # not a parabolic subgroup
~A2 2 - 3
```

We see that the result of the above algorithm is that `W.roots[2]` and `W.roots[3]` form a system of simple roots in H.

The line containing the Dynkin diagram of H introduces a convention: we use the notation `"~A"` to denote a root subsystem of type `"A"` generated by short roots.

We now point the differences which occur when considering complex reflection groups. First, a subgroup generated by a subset of the standard generators need not always be parabolic, if W needs more than $\dim V$ reflections to be generated. The type of a reflection subgroup is not known a priori, but CHEVIE will try to determine it if you call `ReflectionType` or any operation which needs the classification on the constructed subgroup. However there are no canonical way to choose the generators; CHEVIE will try to choose generators giving the same Cartan matrix as the one for the standard group of the same type defined by CHEVIE; failing that, it will at least try to find generators which satisfy the appropriate braid relations.

The record for a reflection subgroup contains additional components the most important of which is `rootInclusion` which gives the positions of the roots of H in the roots of W:

```
gap> H.rootInclusion;
[ 2, 3, 4, 8, 9, 10 ]
```

The inverse (partial) map is stored in `H.rootRestriction`.

If H is a standard parabolic subgroup of a Coxeter group W then the length function on H (with respect to its set of generators) is the restriction of the length function on W. This need not no longer be true for arbitrary reflection subgroups of W:

```
gap> CoxeterLength( W, H.generators[2] );
3
gap> CoxeterLength( H, H.generators[2] );
1
```

In GAP3, finite reflection groups W are represented as permutation groups on a set of roots. Consequently, a reflection subgroup $H \subseteq W$ is a permutation subgroup, i.e., its elements are represented as permutations of the roots of the *parent* group. This has to be kept in mind when working with reduced expressions and functions like `CoxeterWord`, and `EltWord`.

Reduced words in simple reflections of H:

```
gap> el := CoxeterWords( H );
[ [  ], [ 2 ], [ 3 ], [ 2, 3 ], [ 3, 2 ], [ 2, 3, 2 ] ]
```

Reduced words in the generators of H:

```
gap> el1 := List( el, x -> H.rootRestriction{ x } );
[ [  ], [ 1 ], [ 2 ], [ 1, 2 ], [ 2, 1 ], [ 1, 2, 1 ] ]
```

Permutations on the roots of W:

```
gap> el2 := List( el, x -> EltWord( H, x ) );
[ (), ( 1, 5)( 2, 8)( 3, 4)( 7,11)( 9,10),
  ( 1,12)( 2, 4)( 3, 9)( 6, 7)( 8,10),
  ( 1, 5,12)( 2,10, 3)( 4, 9, 8)( 6, 7,11),
  ( 1,12, 5)( 2, 3,10)( 4, 8, 9)( 6,11, 7),
  ( 2, 9)( 3, 8)( 4,10)( 5,12)( 6,11) ]
```

Reduced words in the generators of W:

```
gap> List( el2, x -> CoxeterWord( W, x ) );
[ [  ], [ 2 ], [ 1, 2, 1 ], [ 2, 1, 2, 1 ], [ 1, 2, 1, 2 ],
  [ 2, 1, 2, 1, 2 ] ]
```

Another basic result about reflection subgroups of Coxeter groups is that each coset of H in W contains a unique element of minimal length. Since a coset is a subset of W, the length of elements is taken with respect to the roots of W. See 88.3.

In many applications it is useful to know the decomposition of the irreducible characters of W when we restrict them from W to a reflection subgroup H. In order to apply the usual GAP3 functions for inducing and restricting characters and computing scalar products, we need to know the fusion map for the conjugacy classes of H into those of W. This is done, as usual, with the GAP3 function `FusionConjugacyClasses`, which calls a special implementation for Coxeter groups. The decomposition of induced characters into irreducibles then is a simple matter of combining some functions which already exist in GAP3. The package CHEVIE provides a function `InductionTable` which performs this job.

```
gap> W := CoxeterGroup( "G", 2 );;
gap> W.roots[4];
[ 1, 2 ]
gap> H := ReflectionSubgroup( W, [ 2, 4 ] );;
gap> Display( InductionTable( H, W ) );
Induction from ~A2 to G2
          |111 21 3
_____________________
phi{1,0}   |  .   . 1
phi{1,6}   |  1   . .
phi{1,3}'  |  .   . 1
phi{1,3}'' |  1   . .
phi{2,1}   |  .   1 .
phi{2,2}   |  .   1 .
```

We have similar functions for the j-induction and the J-induction of characters. These operations are obtained by truncating the induced characters by using the a-invariants and b-invariants associated with the irreducible characters of W (see 88.6 and 88.7).

88.1 ReflectionSubgroup

`ReflectionSubgroup(` W, r `)`

Returns the reflection subgroup of the real or complex reflection group W generated by the reflections with roots specified by r. r is a list of indices specifying a subset of the roots of W.

A reflection subgroup H of W is a permutation subgroup, and otherwise has the same fields as W, with some new ones added which express the relationship with the parent W:

`rootInclusion`
: the indices of the roots in the roots of W

`parentN`
: the number of positive roots of W (for Coxeter groups)

`rootRestriction`
: a list of length `2*H.parentN` with entries in positions `H.rootInclusion` bound to `[1..2*H.N]`.

A reflection group which is not a subgroup actually also contains these fields, set to the trivial values:`rootInclusion = [ 1 .. 2*W.N ]`, `parentN = W.N` and `rootRestriction = [ 1 .. 2*W.N ]`.

With these fields, the method `IsLeftDescending(H,w,i)` is written (where w is given as a permutation of the roots of the parent)

`H.rootInclusion[i]^w>H.parentN`

`ReflectionSubgroup` returns a subgroup of the parent group of the argument (like the GAP3 function `Subgroup`).

```
gap> W := CoxeterGroup( "F", 4 );;
gap> H := ReflectionSubgroup( W, [ 1, 2, 11, 20 ] );
ReflectionSubgroup(CoxeterGroup("F",4), [ 1, 2, 9, 16 ])
gap> ReflectionName( H );  # not a parabolic subgroup
"D4"
gap> H.rootRestriction;
[ 1, 2,,, 5,,,, 3,, 6,,, 8,, 4,, 7,, 9,, 10, 11, 12, 13, 14,,, 17,,,,
  15,, 18,,, 20,, 16,, 19,, 21,, 22, 23, 24 ]
gap> ReflectionSubgroup( H, [ 1, 2, 6 ] );
ReflectionSubgroup(CoxeterGroup("F",4), [ 1, 2, 3 ])
```

88.2 Functions for reflection subgroups

All functions for Reflection groups are actually defined for reflection subgroups, provided their reflection type is known (this is automatic in the Coxeter case, otherwise use `ReflectionType` as mentioned in introduction). The generators for the subgroups are labeled according to the corresponding number of the root they represent in the parent group. This affects the labeling given by all functions dealing with words and generators, e.g., `PrintDiagram` or `EltWord`.

```
gap> W := CoxeterGroup( "F", 4 );
CoxeterGroup("F",4)
gap> H := ReflectionSubgroup( W, [ 10, 11, 12 ] );
ReflectionSubgroup(CoxeterGroup("F",4), [ 10, 11, 12 ])
gap> PrintDiagram( H );
B2 10 <=< 11
~A1 12
gap> LongestCoxeterWord( H );
[ 10, 11, 10, 11, 12 ]
```

Note that for the functions `ReflectionType`, `ReflectionName` and `PrintDiagram` for Coxeter subgroups, an irreducible subsystem which consists of short roots in a system which has longer roots (i.e., type `"B"`, `"C"`, `"G"` or `"F"`) is labeled as type `"~A"`.

`ReducedInRightCoset(` H `,` w `)`

this function works in a more general context for reflection subgroups of finite Coxeter groups than for general Coxeter groups. The only condition is that w is a permutation of the roots of the parent group of H, which leaves invariant the set of roots of H and thus induces an automorphism of H. `ReducedInRightCoset` returns the unique element in the right coset $H.w$ which sends all roots of H to positive roots.

```
gap> W := CoxeterGroup("F", 4 );;
gap> H := ReflectionSubgroup( W, [ 1, 2, 9, 16 ] );;
gap> PrintDiagram( H );
D4 9
    \
     1 - 16
    /
   2
gap> w := EltWord( W, [ 3, 2, 3, 4, 3, 2 ] );;
gap> f := ReducedInRightCoset( H, w );;
gap> CoxeterWord( W, f );
[ 4, 3 ]
gap> H.rootInclusion{[ 1 ..4 ]};
[ 1, 2, 9, 16 ]
```

The triality automorphism of D_4 is induced by `f`:

```
gap> OnTuples( H.rootInclusion{[ 1 .. 4 ]}, f );
[ 1, 9, 16, 2 ]
```

88.3 ReducedRightCosetRepresentatives

`ReducedRightCosetRepresentatives(` W, H `[,`l`])`

returns a list of reduced elements in the Coxeter group W which are distinguished representatives for the right cosets of the reflection subgroup H in W. The distinguished representative in the coset $H.w$ is the unique element in the coset which sends all roots of H to positive roots (the element returned by `ReducedInRightCoset`). It is also the element of minimal length in the coset. The representatives are returned in order of increasing length.

```
gap> W := CoxeterGroup( "B", 3 );;
gap> H := ReflectionSubgroup(W, [ 2, 3 ]);;
gap> List( ReducedRightCosetRepresentatives( W, H ),
>                                 x-> CoxeterWord( W, x ) );
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 1 ], [ 1, 2, 3 ], [ 1, 2, 1, 3 ],
  [ 1, 2, 1, 3, 2 ], [ 1, 2, 1, 3, 2, 1 ] ]
```

If a third argument l is given, it should be an integer or a list of integers, and only the representatives whose CoxeterLength is in l are returned. This form is the only one which makes sense for infinite Coxeter groups.

```
gap> W:=Affine(CoxeterGroup("A",2));
Affine(CoxeterGroup("A",2))
gap> H:=ReflectionSubgroup(W,[1]);
ReflectionSubgroup(Affine(CoxeterGroup("A",2)), [ 1 ])
gap> List(ReducedRightCosetRepresentatives(W,H,[0..3]),
>                                 x-> CoxeterWord( W, x ) );
[ [  ], [ 3 ], [ 2 ], [ 3, 1 ], [ 2, 1 ], [ 2, 3 ], [ 3, 2 ],
  [ 2, 1, 3 ], [ 3, 1, 2 ], [ 2, 3, 2 ], [ 2, 3, 1 ], [ 3, 2, 1 ] ]
```

88.4 PermCosetsSubgroup

`PermCosetsSubgroup(` W, H `)`

returns the list of permutations induced by the standard generators of the Coxeter group W on the cosets of the Coxeter subgroup H. The cosets are in the order determined by the result of the function `ReducedRightCosetRepresentatives(` W, H `)`.

```
gap> W := CoxeterGroup( "F", 4 );;
gap> PermCosetsSubgroup( W, ReflectionSubgroup( W, [ 1, 2, 3 ] ) );
[ ( 4, 5)( 6, 7)( 8,10)(16,18)(17,20)(19,21),
  ( 3, 4)( 7, 9)(10,12)(14,16)(15,17)(21,22),
  ( 2, 3)( 4, 6)( 5, 7)( 9,11)(12,14)(13,15)(17,19)(20,21)(22,23),
  ( 1, 2)( 6, 8)( 7,10)( 9,12)(11,13)(14,15)(16,17)(18,20)(23,24) ]
```

88.5 StandardParabolic

`StandardParabolic(` W, H `)`

returns an element w of W which conjugates the reflection subgroup H of W to a standard parabolic subgroup (that is, a reflection subgroup generated by a subset of the generators of W), if such a w exists. Otherwise returns false.

The returned element w is thus such that `H^w` is a standard parabolic subgroup of W.

```
gap> W:=CoxeterGroup("E",6);;
gap> R:=ReflectionSubgroup(W,[20,30,19,22]);
ReflectionSubgroup(CoxeterGroup("E",6), [ 1, 9, 19, 20 ])
gap> StandardParabolic(W,R);
( 1, 4,49,12,10)( 2,54,62, 3,19)( 5,17,43,60, 9)( 6,21,34,36,20)
( 7,24,45,41,53)( 8,65,50,15,22)(11,32,31,27,28)(13,48,46,37,40)
(14,51,58,44,29)(16,23,35,33,30)(18,26,39,55,38)(42,57,70,72,56)
(47,68,67,63,64)(52,59,71,69,66)
gap> R^last;
ReflectionSubgroup(CoxeterGroup("E",6), [ 4, 5, 2, 6 ])
gap> R:=ReflectionSubgroup(W,[1,2,3,5,6,35]);;
gap> ReflectionName(R);
"A2<1,3>xA2<2,35>xA2"
gap> StandardParabolic(W,R);
false
```

88.6 jInductionTable for Macdonald-Lusztig-Spaltenstein induction

`jInductionTable(` H, W `)`

computes the decomposition into irreducible characters of the reflection group W of the j-induced of the irreducible characters of the reflection subgroup H. The j-induced of χ is the sum of the irreducible components of the induced of χ which have same b-function (see 87.7) as χ. In the table the rows correspond to the characters of the parent group, the columns to those of the subgroup. What is returned is actually a record with several fields:`scalar` contains the induction table proper, and there is a `Display` method. The other fields contain labeling information taken from the character tables of H and W when it exists.

```
gap> W := CoxeterGroup( "D", 4);;
gap> H := ReflectionSubgroup( W, [ 1, 3 ] );;
gap> Display( jInductionTable( H, W ) );
j-Induction from A2 to D4
      |111 21 3
________________
11+   |  .  . .
11-   |  .  . .
1.111 |  .  . .
.1111 |  .  . .
11.2  |  .  . .
1.21  |  1  . .
.211  |  .  . .
2+    |  .  . .
2-    |  .  . .
.22   |  .  . .
1.3   |  .  1 .
.31   |  .  . .
.4    |  .  . 1
```

88.7 JInductionTable

`JInductionTable(` H, W `)`

`JInductionTable` computes the decomposition into irreducible characters of the reflection group W of the J-induced of the irreducible characters of the reflection subgroup H. The J-induced of χ is the sum of the irreducible components of the induced of χ which have same a-function (see 87.10) as χ. In the table the rows correspond to the characters of the parent group, the columns to those of the subgroup. What is returned is actually a record with several fields:`scalar` contains the induction table proper, and there is a `Display` method. The other fields contain labeling information taken from the character tables of H and W when it exists.

```
gap> W := CoxeterGroup( "D", 4 );;
gap> H := ReflectionSubgroup( W, [ 1, 3 ] );;
gap> Display( JInductionTable( H, W ) );
J-Induction from A2 to D4
      |111 21 3
________________
11+   |  .  . .
11-   |  .  . .
1.111 |  .  . .
.1111 |  .  . .
11.2  |  1  . .
1.21  |  1  . .
.211  |  .  . .
2+    |  .  . .
2-    |  .  . .
.22   |  .  . .
```

```
1.3    |  .  1 .
.31    |  .  . .
.4     |  .  . 1
```

Chapter 89

Garside and braid monoids and groups

Garside monoids are a general class of monoids whose most famous examples are the braid and dual braid monoids. The CHEVIE implementation of these last monoids is in the framework of a general implementation of Garside monoids.

To define them we first need to introduce some vocabulary about divisibility in monoids. A **left divisor** of x is a d such that there exists y with $x = dy$ (and then we say that x is a **right multiple** of d). The divisor d is **proper** if $y \neq 1$. We say that x is an **atom** if it has no proper left divisor apart from 1. A **left gcd** of x and y is a common left divisor d of x and y such that any other common left divisor is a right multiple of d. Similarly a **right lcm** of x and y is a common multiple which is a left divisor of any other common multiple. We say that a monoid M is left (resp. right) cancellable if an equality $dx = dy$ (resp. $xd = yd$) implies $x = y$.

We call **Garside** a monoid M which is:

- left and right cancellable.
- generated by its atoms, which are finite in number.
- such that any element has only finitely many divisors.
- admits left and right gcds and lcms.
- admits a **Garside element**, which is an element Δ whose set of left and right divisors coincide and generate M.

Garside elements are not unique, but there is a unique minimal one (for divisibility); we assume such an element has been chosen. Then the divisors of Δ are called the **simples** of M. A Garside monoid embeds into its group of fractions, which is called a **Garside group** (it may be that a Garside group has several distinct Garside structures, as we will see is the case for Braid groups of finite Coxeter groups).

CHEVIE also implements **locally Garside** monoids, which are monoids where lcms do not always exist, but exist when any common multiple exists; the set of simples is then not defined using a Garside element, but by the condition that they contain the atoms and are closed under lcms and taking divisors (see [BDM02]); since it is not ensured by the existence of Δ, one has to add the condition that any element is divisible by finitely many simples (but the number of simples can be infinite). The main example is the braid monoid of an infinite Coxeter group. It is not known if these monoids embed in their group of fractions (though that has been proved for braid monoids of Coxeter groups by Paris [Par02]) and thus computing in the monoid does not help for computing in the group (only the monoid is implemented in CHEVIE).

What allows computation inside Garside and locally Garside monoids, and Garside groups, is the fact that they admit normal forms — these normal forms where first exhibited for braid monoids by Deligne [Del72], who extended previous work of Brieskorn, Saito [BS72] and Garside [Gar69]:

(i) Let M be a locally Garside monoid and let $b \in M$. Then there is a unique maximal left simple divisor $\alpha(b)$ of b, called the **head** of b — any other simple dividing b on the left divides $\alpha(b)$ on the left.

(ii) Assume M is a Garside monoid, Δ is its Garside element and G is its group of fractions. Then, given any element $x \in G$, there is some power Δ^i such that $\Delta^i x \in M$.

A consequence of (i) is that any element has a canonical decomposition as a product of simples, called its left-greedy normal form. If we define $\omega(x)$ by $x = \alpha(x)\omega(x)$, then the normal form of x is $\alpha(x)\alpha(\omega(x))\alpha(\omega^2(x))\ldots$ We use the normal form to represent elements of M, and when M is Garside (ii) to represent elements of G:given $x \in G$ we compute the smallest power of Δ such that $\Delta^i x \in M$, and we represent x by the couple $(i, \Delta^i x)$. We are thus reduced to the case where $x \in M$, not divisible by Δ, where we represent x by the sequence of simples which constitutes its normal form.

We now describe Artin-Tits braid monoids. Let (W, S) be a Coxeter system, that is W has presentation

$$\langle s \in S \mid s^2 = 1, \quad \underbrace{sts\cdots}_{m(s,t)\text{ factors}} = \underbrace{tst\cdots}_{m(s,t)\text{ factors}} \quad \text{for all } s, t \in S\rangle$$

for some Coxeter matrix $\{m_{s,t}\}_{s,t\in S}$. The braid group B associated to (W, S) is the group defined by the presentation

$$\langle \mathbf{s} \in \mathbf{S} \mid \quad \underbrace{\mathbf{sts}\cdots}_{m(s,t)\text{ factors}} = \underbrace{\mathbf{tst}\cdots}_{m(s,t)\text{ factors}} \quad \text{for all } \mathbf{s}, \mathbf{t} \in \mathbf{S}\rangle$$

The **positive** braid monoid B^+ associated to W is the monoid defined by the presentation above — it identifies to the submonoid of B generated by $\mathbf{S}$ by the result of Paris mentioned above. This monoid is locally Garside, with set of simples in bijection with elements of W and atoms the elements of $\mathbf{S}$; we will denote by $\mathbf{W}$ the set of simples, and by $\mathbf{w} \mapsto w$ the bijection between simples and elements of W. The group W has a length defined in terms of reduced expressions (see `CoxeterLength`). Similarly, having only homogeneous relations, B^+ has a natural length function. Then $\mathbf{W}$ can be characterized as the subset of the elements of B^+ of the same length as their image in W.

If W is finite, then B^+ is Garside with Garside element the element of $\mathbf{W}$ whose image is the longest element of W. A finite Coxeter group is also a reflection group in a real vector space, thus in its complexified V, and B has also a topological definition as the fundamental group of the space V^{reg}/W, where V^{reg} is the set of elements of V which are fixed by no non-identity element of S; however, we will not use this here.

Given a Coxeter group W,

```
gap> W:=CoxeterGroup("A",4);;M:=BraidMonoid(W);
BraidMonoid(CoxeterGroup("A",4))
```

constructs the associated braid monoid, and then the function `M.B` constructs elements of the braid monoid (or group when W is finite) from a list of generators. This function is directly available from W as `Braid(W)`. Here is an example:

```
gap> W:=CoxeterGroup("A",4);;B:=Braid(W);;
gap> w:=B(1,2,3,4);
1234
gap> w^3;
121321432.343
gap> CoxeterWord(W,GarsideAlpha(w^3));
[ 1, 2, 1, 3, 2, 1, 4, 3, 2 ]
gap> w^4;
w0.232432
gap> w^-1;
(1234)^-1
```

As seen in the fourth line above, the function `GarsideAlpha(b)` returns the simple $\alpha(b) \in \mathbf{W}$, which is returned as an element of W.

How an element of a Garside group is printed is controlled by the record `CHEVIE.PrintGarside`. The user can change the way elements of Garside monoids and groups are printed whenever she wants during a GAP3 session by changing this record. When you load the CHEVIE package, `PrintGarside` is initialized to the empty record. Then elements are printed as fractions $a^{-1}b$ where a and b have no left common divisor. Each of a and b is printed using its left-greedy normal form, that is a maximal power of the Garside element followed the rest. One can print the entire element in the left-greedy normal from by setting the `Greedy` field in `PrintGarside`; with the same w as above we have:

```
gap> CHEVIE.PrintGarside:=rec(Greedy:=true);;
gap> w^-1;
w0^-1.232432
```

Finally, if the field `GAP` in the `PrintGarside` record is set, the element is printed in a form which after assigning `B:=Braid(W);` can be input back into GAP3:

```
gap> CHEVIE.PrintGarside:=rec(GAP:=true);;
gap> w;
B(1,2,3,4)
gap> w ^ 3;
B(1,2,1,3,2,1,4,3,2,3,4,3)
gap> w^-1;
B(1,2,3,4)^-1
gap> CHEVIE.PrintGarside:=rec(GAP:=true,Greedy:=true);;
```

```
gap> w^-1;
B([2,3,2,4,3,2],-1)
gap> CHEVIE.PrintGarside:=rec();;
```

In general elements of a Garside monoid are displayed thus as a list of their constituting atoms.

We now describe the dual braid monoid. For that, we first give a possible approach to construct Garside monoids. Given a group W and a set S of generators of W as a monoid, we define the length $l(w)$ as the minimum number of elements of S needed to write w. We then define left divisors of x as the d such that there exists y with $x = dy$ and $l(d) + l(y) = l(x)$. We say that $w \in W$ is balanced if its set of left and right divisors coincide, is a lattice (where upper and lower bounds are lcms and gcds) and generates W. Then we have:

suppose w is balanced and let $[1, w]$ be its set of divisors (an interval for the partial order defined by divisibility). Then the monoid M with generators $[1, w]$ and relations $xy = z$ whenever $xy = z$ holds in W and $l(x)+l(y) = l(z)$ is Garside, with simples $[1, w]$ and atoms S.

The Artin-Tits braid monoid can be obtained in this fashion by taking for S the Coxeter generators, in which case l is the Coxeter length, and taking for w the longest element of W. The dual monoid, constructed by Birman, Ko and Lee for type A and by Bessis for all well-generated complex reflection groups, is obtained in a similar way, by taking this time for S the set of all reflections, and for w a Coxeter element; then l is the `ReflectionLength`, see 84.13; (this is for Coxeter groups; for well-generated complex reflection groups S has to be restricted to only those reflections which divide w for the reflection length); the simples are in bijection with $[1, w]$, a subset of W of cardinality the generalized Catalan numbers. Monoids M constructed this way from an interval in a group, are called **interval monoids**. An interval monoid has naturally an inverse morphism from M to W, called `EltBraid` which is the quotient map from the interval monoid to W which sends back simple braids to $[1, w]$.

A last notable notion is **reversible** monoids. Since in CHEVIE we store only left normal forms, it is easy to compute left lcms and gcds, but hard to compute right ones. But this becomes easy to do if the monoid has an operation `a->Reverse(a)`, which has the property that `a` is a left divisor of `b` if and only if `Reverse(a)` is a right divisor of `Reverse(b)`. This holds for Artin-Tits and dual braid monoids; Artin-Tits monoids have a reverse operation which consists of reversing a word, written as a list of atoms. The dual monoid also has a reverse operation defined in the same way, but this operation changes monoid: it goes from the dual monoid for the Coxeter element w to the dual monoid for the Coxeter element w^{-1}. The operations `RightLcm` and `RightGcd`, as well quite a few algorithms have faster implementations if the monoid has a reverse operation.

We have implemented in CHEVIE functions to solve the conjugacy problem and compute centralizers in Garside groups, following the work of Franco, Gebhardt and Gonzalez-Meneses ([GGM10] and [FGM03]).

We say that w and w', elements of a monoid M are conjugate in M if there exists $x \in M$ such that $wx = xw'$; if M satisfies the Öre conditions, it has a group of fractions where this becomes $x^{-1}wx = w'$, the usual definition of conjugacy. A special case which is even closer to conjugacy in the group is if there exists $y \in M$ such that $w = xy$ and $w' = yx$. This relation is not transitive in general, but we call **cyclic conjugacy** the transitive closure of this relation, a restricted form of conjugacy.

The next observation is that if w, w' are conjugate in the group of fractions of the Garside monoid M then they are conjugate in M, since if $wx = xw'$ then there is a power Δ^i which is central and such that $x\Delta^i \in M$. Then $wx\Delta^i = x\Delta^i w'$ is a conjugation in M.

The crucial observation for solving the conjugacy problem is to introduce $\inf(w) = \sup\{i \text{ such that } \Delta^i \text{ divides} w\}$ and $\sup(w) = \inf\{i \text{ such that } w \text{ divides} \Delta^i\}$, and to notice that the number of conjugates of w with same inf and sup as w is finite. Further, a theorem of Birman shows that the maximum inf and minimum sup in a conjugacy class can be achieved simultaneously; the elements achieving this are called the super summit set of w. Thus a way to determine if two elements are conjugate is to find a representative of both of them in their super summit set, and then solve conjugacy within that set. This can also be used to compute the centralizer of an element:if we consider the super summit set as the objects of a category whose morphisms are the conjugations by simple elements, the centralizer is given by the endomorphisms of the given object.

We illustrate this on an example

```
gap> w:=B(2,1,4,1,4);
214.14
gap> ConjugacySet(w,"SS"); #  super summit set
[ 1214.4, 214.14, 124.24, 1343.1, 14.124, 143.13, 24.214, 134.14,
  13.134, 14.143 ]
gap> RepresentativeConjugation(w,B(1,4,1,4,3));
(1)^-1.21321432
gap> w^B(-1,2,1,3,2,1,4,3,2);
14.143
gap> CentralizerGenerators(w);
[ 4, 321432.213243, 21.1 ]
```

There is a faster solution to the conjugacy problem given in [GGM10]:for each $b \in M$, they define a particular simple left divisor of b, its **preferred prefix** such that the operation **sliding** which cyclically conjugates b by its preferred prefix, is eventually periodic, and the period is contained in the super summit set of x. We say that x is in its sliding circuit if some iterated sliding of x is equal to x. The set of sliding circuits in a given conjugacy class is smaller than the super summit set, thus allows to solve the conjugacy problem faster. Continuing from the above example,

```
gap> CoxeterWord(W,PreferredPrefix(w));
[ 2, 1 ]
gap> w^B(PreferredPrefix(w));
1214.4
gap> last^B(PreferredPrefix(last));
1214.4
gap> ConjugacySet(w,"SC"); #  set of sliding circuits
[ 1214.4, 1343.1 ]
```

Finally, we have implemented Hao Zheng's algorithm to extract roots in a Garside monoid:

```
gap> W:=CoxeterGroup("A",3);; M:=BraidMonoid(W);
BraidMonoid(CoxeterGroup("A",3))
gap> pi:=M.B(M.delta)^2;
w0.w0
```

```
gap> GetRoot(pi,2);
w0
gap> GetRoot(pi,3);
1232
gap> GetRoot(pi,4);
132
```

89.1 Operations for (locally) Garside monoid elements

We illustrate with braids basic operations on elements of a locally Garside monoid or a Garside group. Thus we suppose first we have defined two elements `a`, `b` as

```
gap> W := CoxeterGroup( "A", 2 );;
gap> a := Braid( W )( [1] );
1
gap> b := Braid( W )( [2] );
2
```

All examples below are with `CHEVIE.PrintOption("Garside","Greedy")`.

b1 `*` *b2*

The multiplication of two elements of the same locally Garside monoid or Garside group is defined.

```
gap> a * b;
12
```

b1 `^` *i*

An element can be raised to an integral, positive power (or negative power if the monoid is Garside, which is the case here since W is finite). Here `w0` is how the fundamental element Δ prints in the case of braids.

```
gap> ( a * b ) ^ 4;
w0.w0.12
gap> ( a * b ) ^ -1;
(12)^-1
```

b1 `^` *b2*

This is defined if the monoid is Garside and returns $b_1^{-1}b_2b_1$.

```
gap> a ^ b;
(2)^-1.12
```

b1 `/` *b2*

This is defined if the monoid is Garside and returns $b_1b_2^{-1}$.

```
gap> a / b;
(12)^-1.21
```

`Format(` *b*, *option* `)`

`String(` *b* `)`

`Print(` *b* `)`

`String` returns a display form of the element b, and `Print` prints the result of `String`. The way elements are printed depends on the value of the record `CHEVIE.PrintGarside`. If if it is `rec(GAP =true)`, the elements are printed in a form which can be read in back by a function `B()` which accepts a list of atoms (for braids, `Braid(W)` returns such a function). If it is `rec(Greedy =true)` (resp. `rec()`) the left-greedy (resp. fraction) normal form (as explained in the introduction) is printed:

```
gap> CHEVIE.PrintGarside:=rec(GAP:=true);;
gap> ( a * b ) ^ -1;
B(1,2)^-1
gap> CHEVIE.PrintGarside:=rec(Greedy:=true);;
gap> ( a * b ) ^ -1;
w0^-1.2
gap> CHEVIE.PrintGarside:=rec();;
gap> ( a * b ) ^ -1;
(12)^-1
```

The function `Format(` b`,` *option*`)` returns the element formatted in a string the same way it would be printed with `PrintGarside` set to the corresponding option. `String` is equivalent to `Format(`b`)` so always formats its argument as `Print` does after `CHEVIE.PrintGarside =rec()`.

`GetRoot(` b`,` n `)`

Returns the n-th root of b.

89.2 Records for(locally) Garside monoids

This section is rather technical and describes an internal representation which is not yet completely fixed and thus might still change. We describe here how a Garside or locally Garside monoid with finitely many atoms is specified in CHEVIE, as a particular kind of record. If someone uses the information below to construct a new kind of Garside monoid, it is thus advisable to contact me (Jean Michel) to discuss possible changes.

To construct a locally Garside monoid one creates a record `M` containing the following fields holding data and operations, and then calls `CompleteGarsideRecord(M)`. The simples can be arbitrary objects (for interval monoids they should be elements of a group), the following operations should just be defined on them.

`M.atoms`
: the list of simples which are atoms of M.

`M.identity`
: the identity element of M, a simple.

`M.IsLeftDescending(s,i)`
: tells whether `M.atoms[i]` divides on the left the simple s.

`M.IsRightAscending(s,i)`
: tells whether the product of the simple s by `M.atoms[i]` is still simple.

`M.Product(s,t)`
: returns the product of the simples s and t. It does not have to be defined in all cases, but should at least be defined when `t=M.atom[i]` and `M.IsRightAscending(s,i)`.

`M.LeftQuotient(s,t)`
: returns the quotient `s^-1*t` for simples s and t. It does not have to be defined in all cases, but should at least be defined when `s=M.atom[i]` and `M.IsLeftDescending(t,i)`.

`M.RightQuotient(s,t)`
: returns the quotient `s/t` for simples s and t

The source code for `BraidMonoid` and `DualBraidMonoid` provide examples. The above functions are sufficient for multiplication, and most operations on locally Garside monoids, like `LeftGcd`, `GarsideAlpha`, etc... to be defined; however see below for conjugacy.

In the case the monoid is an interval monoid, which is specified by giving a second argument `rec(interval:=true)` to `CompleteGarsideMonoid`, then the functions `Product`, `LeftQuotient`, `RightQuotient` are automatically defined (since in that case simples are element of a group, they are defined by the group operations).

For `ReversedBraid`, `RightGcd`, `RightLcm`, `M.RightGcdSimples` to work (see below) and the Franco and González-Meneses conjugacy and centralizer algorithms to be defined, one needs in addition either:

a function `M.ReverseSimple` to be defined, which defines the `Reverse` operation (if `M` is an interval monoid and `M.delta^2=M.identity`, this is just `x->x^-1` thus is then automatically defined),

or the symmetric routines `M.IsRightDescending`, `M.IsLeftAscending` to be defined, and one needs that `Product(M.atoms[i],s)` be defined for `s` simple such that `M.IsLeftAscending(s,i)`, and that `M.RightQuotient(s,M.atoms[i])` be defined when `M.IsRightDescending(s,i)`.

For the monoid constructed to be Garside one should in addition define the following data and operations:

`M.delta`
: the fundamental element Δ.

`M.DeltaAction(s,i)`
: Let f be the automorphism induced on simples by Δ (such that $\Delta s = f(s)\Delta$). The function returns $f^i(s)$.

`M.stringDelta`
: how Δ should be printed in normal forms.

If the monoid is an interval monoid, `DeltaAction` is automatically defined as conjugation by `M.delta` in the group to which simples belong.

`CompleteGarsideRecord` uses internally the test `IsBound(M.delta)` to detect if the monoid is Garside.

Some additional fields and methods are added by `CompleteGarsideRecord` if not present; they are only added if not present since often the user could define more efficient or more appropriate versions for a particular kind of monoid (this is the case for braid monoids, for instance). These fields and methods are :

`M.AtomListSimple(s)`
: returns the list of atoms of which the simple s is the product. This is mostly used for display purposes, so the individual representation for atoms may be any kind of object. However it is useful for the function `AsWord` that atoms be represented by positive integers (if `M.AtomListSimple` is not pre-defined, `CompleteGarsideRecord`

defines a default version where an atom is represented by its index in the list `M.atoms`). An existing situation where the representation of an atom is not by its index in the list of atoms is for braid monoids of Coxeter subgroups (where the index in the list of generators of the parent is used — see `.reflectionLabels`); also in this case the pre-defined function is faster than the default one would be.

`M.RightComplementToDelta(s)`
: for Garside monoids. Given a simple s returns the simple t such that $st = \Delta$. Again one may often be able to pre-define a faster function than the default one.

`M.LeftComplementToDelta(s)`
: for Garside monoids. Given a simple s returns the simple t such that $ts = \Delta$. The default version reads `DeltaAction(RightComplementToDelta(s),1)`.

For interval monoids, fast versions of `RightComplementToDelta` and `LeftComplementToDelta` are automatically defined.

`M.LeftGcdSimples(`$a_1, \ldots, a_n$`)`
: returns the simple which is the left gcd of the simples $a_1, \ldots, a_n$.

`M.LeftLcmSimples(`$a_1, \ldots, a_n$`)`
: for Garside monoids. Returns the simple which is the left lcm of the simples $a_1, \ldots, a_n$.

In addition `CompleteGarsideRecord` defines `M.RightGcdSimples` and (for Garside monoids) `M.RightLcmSimples` if the methods `M.IsRightDescending`, `M.IsLeftAscending`, `M.LeftMultiple`, `M.RightQuotient` have been defined.

89.3 GarsideWords

`GarsideWords(` M `,` l `)`

M should be a (locally) Garside monoid which has an additive length function (that is, a product of l atoms is not equal to any product of less than l atoms). `GarsideWords(`M`,` l`)` returns the list of elements of length l in M.

```
gap>  M := BraidMonoid(CoxeterGroup( "A", 2 ));;
gap>  GarsideWords( M, 4 );
[ 21.1.1, 21.12, w0.2, 2.21.1, 2.2.21, 2.2.2.2, w0.1, 1.1.1.1,
  1.1.12, 1.12.2, 12.21, 12.2.2 ]
```

89.4 Presentation

`Presentation(`M`)`

M should be a Garside monoid. `Presentation` returns a presentation of the corresponding Garside group (the presentation is as given in theorem 4.1 of [DP99]).

```
gap>  M := DualBraidMonoid(CoxeterGroup( "A", 3 ));;
gap>  p:=Presentation(M);DisplayPresentation(p);
<< presentation with 6 gens and 15 rels of total length 62 >>
1: ab=da
2: ac=ca
3: ec=cb
4: bd=da
```

```
5: bd=ab
6: cd=fc
7: ae=fa
8: be=cb
9: be=ec
10: de=ed
11: ef=fa
12: df=fc
13: df=cd
14: ef=ae
15: def=acb
gap>  ShrinkPresentation(p);DisplayPresentation(p);
#I there are 3 generators and 4 relators of total length 26
#I there are 3 generators and 3 relators of total length 16
1: ab=ba
2: cbc=bcb
3: cac=aca
```

89.5 ShrinkGarsideGeneratingSet

`ShrinkGarsideGeneratingSet(`*b*`)`

The list b is a list of elements of the same Garside group G. This function tries to find another set of generators of the subgroup of G generated by the elements of b, of smaller total length (the length being counted as returned by the function `AsWord`).

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> b:=[B(1)^3,B(2)^3,B(-2,-1,-1,2,2,2,2,1,1,2),B(1,1,1,2)];
[ 1.1.1, 2.2.2, (1.12)^-1.2.2.2.21.12, 1.1.12 ]
gap>  ShrinkGarsideGeneratingSet(b);
[ 2, 1 ]
```

89.6 locally Garside monoid and Garside group elements records

Now, we describe elements of a (locally) Garside monoid which are records with 3 fields:

`elm`
: the list of simples in the left-greedy normal form.

`operations`
: points to GarsideEltOps.

`monoid`
: points to the record describing the corresponding Garside or locally Garside monoid.

And a fourth field if the monoid is Garside:

`pd`
: the power of Δ involved in the greedy normal form.

`CompleteGarsideRecord` adds to (locally) Garside monoid records `M` a function `M.Elt` which can be used to build such elements. The syntax is

`M.Elt(`s `[,`pd`])`

which defines an element with `.elm=`s and `.pd=`pd (if the monoid is Garside but pd is not given it is initialized to `0`). The user must only give valid normal forms in s, otherwise unpredictable errors may occur. For example, Δ should be entered as `M.Elt([],1)` and the identity as `M.Elt([])`.

89.7 AsWord

`AsWord(` b `)`

b should be a locally Garside monoid or Garside group element. `AsWord` then returns a description of b as a list of the atoms of which it is a product (as returned by `AtomListSimple`). If b is in the group but not the monoid, it is represented in fraction normal form where as a special convention the inverses of the atoms are represented by negating the corresponding integer.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2, 1, 2, 1, 1)*Braid(W)(2,2)^-1;
(21)^-1.1.12.21
gap> AsWord( b );
[ -1, -2, 1, 1, 2, 2, 1 ]
```

89.8 GarsideAlpha

`GarsideAlpha(` b `)`

b should be an element of a (locally) Garside monoid. `GarsideAlpha` returns the simple $\alpha(b)$ (for braids this is an element of the corresponding Coxeter group).

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid( W )(2, 1, 2, 1, 1);
121.1.1
gap> CoxeterWord(W,GarsideAlpha( b ));
[ 1, 2, 1 ]
```

89.9 LeftGcd

`LeftGcd(` $a_1,\ldots,a_n$ `)`

$a_1,\ldots,a_n$ should be elements of the same (locally) Garside monoid M. Let d be the greatest common left divisor of $a_1,\ldots,a_n$; then `LeftGcd` returns the list `[d,d^-1*`$a_1,\ldots,$`d^-1*`a_n`]`.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2,1,2)^2;
121.121
gap> LeftGcd(b,Braid(W)(3,2)^2);
[ 2, 121.21, 32.2 ]
```

89.10 LeftLcm

`LeftLcm(` $a_1,\ldots,a_n$ `)`

$a_1,\ldots,a_n$ should be elements of the same Garside monoid M. Let m be the least common left multiple of $a_1,\ldots,a_n$; then `LeftGcd` returns the list `[m,m*`a_1^{-1}`,...,m*`a_n^{-1}`]`.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2,1,2)^2;
121.121
gap> LeftLcm(b,Braid(W)(3,2)^2);
[ w0.121, 123, 23.321 ]
```

89.11 ReversedWord

`ReversedWord(` b `)`

b should be an element of a (locally) Garside monoid which has a `reverse` operation (see the end of the introduction). The function returns the result of the reverse operation applied to b.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2,1);
21
gap> ReversedWord(b);
12
```

89.12 RightGcd

`RightGcd(` $a_1,\ldots,a_n$ `)`

$a_1,\ldots,a_n$ should be elements of the same (locally) Garside monoid M which has a `reverse` operation (see the end of the introduction). Let d be the greatest common right divisor of $a_1,\ldots,a_n$; then `RightGcd` returns the list `[d,`a_1`*d^-1,...,`a_n`*d^-1]`.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2,1,2)^2;
121.121
gap> RightGcd(b,Braid(W)(3,2)^2);
[ 2.2, 12.21, 23 ]
```

89.13 RightLcm

`RightLcm(` $a_1,\ldots,a_n$ `)`

$a_1,\ldots,a_n$ should be elements of the same Garside monoid M which has a `reverse` operation (see the end of the introduction). Let m be the least common right multiple of $a_1,\ldots,a_n$; then `RightLcm` returns the list `[m,`a_1^{-1}`*m,...,`a_n^{-1}`*m]`.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)(2,1,2)^2;
121.121
gap> RightLcm(b,Braid(W)(3,2)^2);
[ w0.w0, 321.123, 12321.321 ]
```

89.14 AsFraction

`AsFraction(` *b* `)`

Let *b* be an element of the Garside group *G*. `AsFraction` returns a pair `[x,y]` of two elements of *M* with no non-trivial common left divisor and such that `b=x^-1*y`.

```
gap>  W := CoxeterGroup( "A", 3 );;
gap>  b := Braid(W)( 2, 1, -3, 1, 1);
(23)^-1.321.1.1
gap> AsFraction(b);
[ 23, 321.1.1 ]
```

89.15 LeftDivisorsSimple

`LeftDivisorsSimple(` *M*`,` *s* `[,`*i*`])`

Returns all the left divisors of the simple element *s* of the (locally) Garside monoid *M*, as a list of lists, where the *i+1*th list holds the divisors of length *i* in the atoms. If a third argument *i* is given, returns only the list of divisors of length *i*.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> M:=BraidMonoid(W);
BraidMonoid(CoxeterGroup("A",3))
gap> List(LeftDivisorsSimple(M,EltWord(W,[1,3,2])),x->List(x,M.B));
[ [ . ], [ 3, 1 ], [ 13 ], [ 132 ] ]
gap> M:=DualBraidMonoid(W);
DualBraidMonoid(CoxeterGroup("A",3),[ 1, 3, 2 ])
gap> List(LeftDivisorsSimple(M,EltWord(W,[1,3,2])),x->List(x,M.B));
[ [ . ], [ 3, 2, 5, 1, 4, 6 ], [ 45, 25, 13, 34, 12, 15 ], [ c ] ]
```

`Concatenation(LeftDivisorsSimple(M,M.delta))` returns all simples of the monoid *M*.

89.16 LeftDivisors

`LeftDivisors(` *b*`[,` *i*`] )`

returns all left divisors of Garside element *b* (of length *i* if specified).

```
gap> B:=DualBraid(CoxeterGroupSymmetricGroup(4));
function ( arg ) ... end
gap> LeftDivisors(B(1,5,4,3));
[ ., 5, 1, 1.4, 1.4.3, 1.4.2, 6, 15, 15.4, 15.4.3 ]
gap> LeftDivisors(B(1,5,4,3),1);
[ 5, 1, 6 ]
```

89.17 EltBraid

`EltBraid(` *b* `)`

This function is defined only if *b* is an element of an interval monoid, for instance a braid. It returns the image of *b* in the group of which the monoid is an interval monoid. For instance it gives the projection of a braid in an Artin monoid back to the Coxeter group.

```
gap>  W := CoxeterGroupSymmetricGroup( 4 );;
gap>  b := Braid( W )(2, 1, 2, 1, 1);
121.1.1
gap> p := EltBraid( b );
(1,3)
gap> CoxeterWord( W, p );
[ 1, 2, 1 ]
```

89.18 The Artin-Tits braid monoids and groups

`BraidMonoid(`*W*`)`

Returns (as a Garside or locally Garside monoid record) the Artin-Tits braid monoid of the Coxeter group W. The monoid is Garside if and only if W is finite; in which case elements of the resulting monoid can be used as elements of a group.

89.19 Construction of braids

`Braid(` *W* `)(` *s1* `, .. ,` *sn* `)`

`Braid(` *W* `)(` *list* `[,` *pd* `])`

`Braid(` *W* `)(` *w* `[,` *pd* `])`

Let W be a Coxeter group and let w be an element of W or a sequence $s_1, .., s_n$ of integers representing a (non necessarily reduced) word in the generators of W. The calls above return the element of the braid monoid of W defined by w. In the second form the *list* is a list of s_i as in the first form. If *pd* (a positive or negative integer) is given (which is allowed only when W is finite), the resulting element is multiplied in the braid group by $\mathbf{w}_0^{pd}$. The result of `Braid(W)` is a braid-making function, which can be assigned to make conveniently braid elements as in the example below. This function can also be obtained as `BraidMonoid(W).B`.

```
gap> W := CoxeterGroup( "A", 3 );;
gap> B := Braid( W );
function ( arg ) ... end
gap> B( W.generators[1] );
1
gap> B( 2, 1, 2, 1, 1 );
121.1.1
gap> CHEVIE.PrintGarside:=rec(Greedy:=true);;
gap> B( [ 2, 1, 2, 1, 1 ], -1 );
w0^-1.121.1.1
```

As a special case (to follow usual conventions for entering braids) a negative integer in a given list representing a word in the generators is taken as representing the inverse of a generator.

```
gap> CHEVIE.PrintGarside:=rec();;
gap> B( -1, -2, -3, 1, 1 );
(321)^-1.1.1
```

89.20 Operations for braids

`Frobenius(` *WF* `)(`*b*`)`

If `WF` is a Coxeter coset associated to the Coxeter group W, the function `Frobenius(WF)` returns the associated automorphism of the braid monoid of W.

```
gap> W:=CoxeterGroup("D",4);WF:=CoxeterCoset(W,(1,2,4));
CoxeterGroup("D",4)
3D4
gap> B:=Braid(W);;b:=B(1,3);
13
gap> Frobenius(WF)(b);
43
gap> Frobenius(WF)(b,-1);
23
```

`BrieskornNormalForm(` *b* `)`

If b is an element of the braid monoid of the Coxeter group `W`, this function returns the Brieskorn normal form of b, which is defined as the concatenation of the Brieskorn normal form for the terms of the normal form of b (see 83.16).

89.21 GoodCoxeterWord

`GoodCoxeterWord(` *W*`,` *w* `)`

Let W be a Coxeter group with associated braid monoid B^+. `GoodCoxeterWord` checks if the element w of W (given as sequence of generators of W) represents a "good element" in the sense of Geck-Michel [GM97] of the braid monoid, i.e., if $\mathbf{w}^d$ (where d is the order of the element w in W, and $\mathbf{w}$ is the element of $\mathbf{W}$ with image w) is a product of (the braid elements corresponding to) longest elements in a decreasing chain of parabolic subgroups of W. If this is true, then a list of couples, the corresponding subsets of the generators with their multiplicities in the chain, is returned. Otherwise, `false` is returned.

Good elements have nice properties with respect to their eigenvalues in irreducible representations of the Hecke-Iwahori algebra associated to W. The representatives in the component `classtext` of `ChevieClassInfo(`*W*`)` are all good elements of minimal length in their class.

```
gap> W := CoxeterGroup( "F", 4 );;
gap> w:=[ 2, 3, 2, 3, 4, 3, 2, 1, 3, 4 ];;
gap> GoodCoxeterWord( W, w );
[ [ [ 1, 2, 3, 4 ], 2 ], [ [ 3, 4 ], 4 ] ]
gap> OrderPerm( EltWord( W, w ) );
6
gap> Braid( W )( w ) ^ 6;
w0.w0.343.343.343.343
gap>  GoodCoxeterWord( W, [ 3, 2, 3, 4, 3, 2, 1, 3, 4, 2 ] );
false
```

89.22 BipartiteDecomposition

`BipartiteDecomposition(`*W*`)`

Returns a bipartite decomposition `[L,R]` of the indices of the generators of the reflection group `W`, such that `ReflectionSubgroup(W,L)` and `ReflectionSubgroup(W,R)` are abelian subgroups (and `W=ReflectionSubgroup(W,Concatenation(L,R))`). Gives an error if no such decomposition is possible.

```
gap> BipartiteDecomposition(CoxeterGroup("E",8));
[ [ 1, 4, 6, 8 ], [ 3, 2, 5, 7 ] ]
```

89.23 DualBraidMonoid

`DualBraidMonoid(`W `[,` c`])`

Returns (as a Garside monoid record) the dual braid monoid of the **finite** and **well-generated** complex reflection group W associated to the Coxeter element c of W. If W is a Coxeter group, c can be omitted and a particular one is then chosen, the element `EltWord(W,Concatenation(BipartiteDecomposition(W)))`.

```
gap> DualBraidMonoid(CoxeterGroup("A",4));
DualBraidMonoid(CoxeterGroup("A",4),[ 1, 3, 2, 4 ])
gap> M:=DualBraidMonoid(CoxeterGroup("A",4),[1,2,3,4]);
DualBraidMonoid(CoxeterGroup("A",4),[ 1, 2, 3, 4 ])
```

For Coxeter groups, the dual monoid contains an operation `.ToOrdinary` which converts simples to elements of the ordinary braid monoid of W. To go on from the above example, we compute the list of ordinary braids corresponding to each simple of length 2 of the dual monoid

```
gap> List(LeftDivisorsSimple(M,M.delta,2),M.ToOrdinary);
[ 34, 24, 23, (34)^-1.2343, (3)^-1.234, (4)^-1.234, 14, 13,
  (4)^-1.134, (2)^-1.124, (23)^-1.1232, (2343)^-1.123243, 12,
  (2)^-1.123, (234)^-1.12324, (3)^-1.123, (23)^-1.1234,
  (234)^-1.12343, (24)^-1.1234, (34)^-1.1234 ]
```

89.24 DualBraid

`B:=DualBraid(` W `[,` c`])`

then

`B(` *s1* `, .. ,` *sn* `)`

`B(` *list* `[,` *pd* `])`

`B(` w `[,` *pd* `])`

Let W be a well generated complex reflection group and c be a Coxeter element of W (if W is a Coxeter group and no c is given a particular one is chosen by making the product of elements in a partition of the Coxeter diagram in two sets where elements in each commute pairwise). The result of `DualBraid` is a dual braid-making function for the dual monoid determined by W and c: let w be an element of W or a sequence $s_1,..,s_n$ of integers representing a list of reflections of W. The calls to `B` above return the element of the dual braid monoid of W defined by w. In the second form the *list* is a list of s_i as in the first form. If *pd* (a positive or negative integer) is given, the resulting element is multiplied in the braid group by c^{pd}.

```
gap> W := CoxeterGroup( "A", 3 );;
gap> B := DualBraid( W );
function ( arg ) ... end
gap> B( W.reflections[4] );
4
gap> B( 2, 1, 2, 1, 1 );
12.1.1.1
gap> B( [ 2, 1, 2, 1, 1 ], -1 );
(3)^-1.5.5.5
```

As a special case (to follow usual conventions for entering braids) a negative integer in a given list representing a word in the generators is taken as representing the inverse of a generator.

```
gap> B( -1, -2, -3, 1, 1 );
(25.1)^-1.1.1
```

The function `B` can also be obtained by making the calls `M:=DualBraidMonoid(` *W* `[,` *c*`])` and `B =M.B`.

89.25 Operations for dual braids

`EltBraid` has the same meaning as for ordinary braids.

89.26 ConjugacySet

`ConjugacySet(`*b*`[,`*F*`][,`*type*`])`

b should be an element of a Garside group. By default, or if the *type* given is `"SC"`, computes the set of sliding circuits of *b*. If *type* is `"SS"`, computes the super summit set of *b*. If *type* is `"Cyc"`, computes the cyclic conjugacy class of *b*. Finally, if type is `"Pos"`, computes the set of all positive elements conjugate to *b*.

If an argument F is given it should be the Frobenius of a Reflection coset attached to the same group to which the Garside monoid is attached. Then the same computations are effected but relative to F-conjugacy.

```
gap> W:=CoxeterGroup("A",4);;w:=Braid(W)(4,3,3,2,1);
43.321
gap> ConjugacySet(w);
[ 32143, 21324 ]
gap> ConjugacySet(w,"SC");
[ 32143, 21324 ]
gap> ConjugacySet(w,"SS");
[ 32143, 13243, 21432, 21324 ]
gap> ConjugacySet(w,"Cyc");
[ 43.321, 3.3214, 32143, 2143.3, 143.32, 213.34, 13.324, 13243,
  1243.3, 123.34 ]
gap> ConjugacySet(w,"Pos");
[ 43.321, 3.3214, 32143, 2143.3, 21432, 143.32, 213.34, 1432.2,
  21324, 13.324, 432.21, 1324.2, 13243, 324.21, 124.23, 1243.3,
  24.213, 12.234, 123.34, 2.2134 ]
```

```
gap> W:=CoxeterGroup("D",4);;
gap> F:=Frobenius(CoxeterCoset(W,(1,2,4)));
function ( arg ) ... end
gap> w:=Braid(W)(4,4,4);
4.4.4
gap> ConjugacySet(w);
[ 4.4.4, 3.3.3, 1.1.1, 2.2.2 ]
gap> ConjugacySet(w,F);
[ 124 ]
```

89.27 CentralizerGenerators

`CentralizerGenerators(`*b*`[,`*F*`][,`*type*`])`

b should be an element of a Garside group. The function returns a list of generators of the centralizer of *b*. The computation is done by computing the endomorphisms of the object *b* in the category of its sliding circuits. If an argument *type* is given, the computation is done in the corresponding category — see 89.26. The main use of this is to compute the centralizer in the category of cyclic conjugacy by giving `"Cyc"` as the type.

If an argument *F* is given it should be the Frobenius of a Reflection coset attached to the same group to which the Garside monoid is attached. Then the *F*-centralizer is computed.

```
gap> W:=CoxeterGroup("D",4);;
gap> w:=Braid(W)(4,4,4);
4.4.4
gap> CentralizerGenerators(w);
[ 4, 2, (1)^-1.34.431, 34.43, (32431)^-1.132431, 1, (2)^-1.34.432,
  (31432)^-1.231432 ]
gap> ShrinkGarsideGeneratingSet(last);
[ 4, 2, 1, 34.43, (3243)^-1.13243 ]
gap> CentralizerGenerators(w,"Cyc");
[ 4 ]
gap> F:=Frobenius(CoxeterCoset(W,(1,2,4)));
function ( arg ) ... end
gap> CentralizerGenerators(w,F);
[ 312343123, 124 ]
```

89.28 RepresentativeConjugation

`RepresentativeConjugation(`*b*`,`*b1*`[,`*F*`][,`*type*`])`

b and *b1* should be elements of the same Garside group. The function returns `false` if they are not conjugate, and an element *a* such that `b^a=b1` if they are conjugate. The computation is done by computing the set of the sliding circuits of *b* and check if it is the same as the set of sliding circuits of *b1*. If an argument *type* is given, the computation is done in the corresponding category — see 89.26. The main use of this is to compute if *b* and *b1* are related by cyclic conjugacy by giving `"Cyc"` as the type.

If an argument *F* is given it should be the Frobenius of a Reflection coset attached to the same group to which the Garside monoid is attached. Then *F*-conjugacy is used for the computations.

```
gap> W:=CoxeterGroup("D",4);;B:=Braid(W);
function ( arg ) ... end
gap> b:=B(2,3,1,2,4,3);b1:=B(1,4,3,2,2,2);
231243
1432.2.2
gap> RepresentativeConjugation(b,b1);
(134312.23)^-1
gap> b^last;
1432.2.2
gap> RepresentativeConjugation(b,b1,"Cyc");
232.2
gap> b^last;
1432.2.2
gap> RepresentativeConjugation(b,b1,F);
false
gap> c:=B(3,2,2,3,3,4);
32.23.34
gap> F:=Frobenius(CoxeterCoset(W,(1,2,4)));
function ( arg ) ... end
gap> a:=RepresentativeConjugation(b,c,F);
(13)^-1.23.31
gap> a^-1*b*F(a);
32.23.34
gap> a:=RepresentativeConjugation(b,c,F,"Cyc");
2312431.312343.324.23.31
gap> a^-1*b*F(a);
32.23.34
```

Chapter 90

Cyclotomic Hecke algebras

The cyclotomic Hecke algebras (Hecke algebras for complex reflection groups) are deformations of the group algebras, generalizing those for real reflection groups (see the next chapter on Iwahori-Hecke algebras).

Their general definition is as a quotient of the algebra of the braid group. We assume now that W is a **finite** reflection group in the complex vector space V since the theory for infinite groups has not yet been investigated in sufficient generality. The **braid group** associated is the fundamental group Π^1 of the space $(V - \bigcup_{H\in\mathcal{H}} H)/W$, where $\mathcal{H}$ is the set of reflection hyperplanes of W. This group is generated by **braid reflections**, elements which by the natural map from the braid group to the reflection group project to distinguished reflections. All braid reflections which map to a given W-orbit of reflections are conjugate. For each such orbit let $\mathbf{s}$ be a representative of the orbit, let e be the order of the image of $\mathbf{s}$ in W, and let $u_{\mathbf{s},0},\ldots,u_{\mathbf{s},e-1}$ be indeterminates. The generic Hecke algebra is the $\mathbb{Z}[u_{\mathbf{s},i}^{\pm 1}]_{\mathbf{s},i}$-algebra quotient of the braid group algebra by the relations $(\mathbf{s}-u_{\mathbf{s},0})\ldots(\mathbf{s}-u_{\mathbf{s},e-1})=0$, and in general a cyclotomic Hecke algebra is any algebra obtained from this generic algebra by specializing some of the parameters.

The quotient of the Hecke algebra obtained by $u_{\mathbf{s},i}\mapsto$`E(e)^i` is isomorphic to the group algebra of W. It is actually conjectured that over a suitable ring (such as the algebraic closure of the field of fractions $\mathbb{Q}(u_{\mathbf{s},i})_{\mathbf{s},i}$) the Hecke algebra is itself isomorphic to the group algebra of W over the same ring (this conjecture has been proven for imprimitive groups and most exceptional groups of rank 2 or 3, see [MM10a] for references; in addition it is well known to hold for real reflection groups; in the missing cases the ingredient lacking is to show that the dimension of the Hecke algebra is `Size(W)`).

The cyclotomic Hecke algebras can also been defined in terms of presentations. The braid group is presented by homogeneous relations, called **braid relations**, described in [BMR98] and [BM04] (some were obtained using the VKCURVE package of GAP3). Further, these relations are such that the reflection group is presented by the same relations, plus relations describing the order of the generating reflections, called the **order relations**. This allows to define the Hecke algebra by the same presentation as W, with the order relations replaced by a deformed version. Specifically, for each orbit of reflection hyperplanes of W, let us chose a **distinguished** reflection s of W, that is a reflection with a non-trivial eigenvalue of minimal argument (i.e., of the form `E(e)` where e is the order of s; then any reflection around an hyperplane of the same orbit is a conjugate of a power of s). Let then $u_{s,0},\ldots,u_{s,e-1}$ be indeterminates. The generic Hecke algebra is the $\mathbb{Z}[u_{\mathbf{s},i}^{\pm 1}]_{\mathbf{s},i}$-algebra H with generators T_s in bijection with the generators of W, presented by the braid relations and the deformed order relations $(T_s-u_{\mathbf{s},0})\ldots(T_s-u_{\mathbf{s},e-1})=0$ for each s as above.

Ariki, Koike and Malle have computed character tables for some of these algebras, including all those for 2-dimensional reflection groups, see [BM93] and [Mal96]; CHEVIE contains models of each representation and character tables for real reflection groups, for imprimitive groups and for primitive groups of dimension 2 and 3 (these last representations have been computed in [MM10a]) and for G_{29} and G_{33}. Further there are some partial lists of representations and partial character tables for the remaining groups G_{31}, G_{32} and G_{34}.

A refinement of the conjecture that H has the same dimension as W is that there exists a set $\{b_w\}_{w\in W}$ of elements of the Braid group such that $b_1=1$ and b_w maps to w by the natural quotient map, such that their images T_w form a basis of the Hecke algebra. It is further conjectured that these can be chosen such that the linear form t defined by $t(T_w)=0$ if $w\neq 1$ and $t(1)=1$ is a symmetrizing form for the symmetric algebra H. This is well known for all real reflection groups and has been proved in [MM98] for imprimitive reflection groups and in [MM10a] for most primitive groups of dimension 2 and 3. Then for each irreducible character χ of H we define the **Schur element** S_χ associated to χ by the condition that for any element T of H we have $t(T)=\sum_\chi \chi(T)/S_\chi$. It can be shown that the Schur elements are Laurent polynomials, and they do not depend on the choice of a basis having the above property. Malle has the computed these Schur elements, assuming the above conjectures; they are in the CHEVIE data.

90.1 Hecke

`Hecke(` G`,` *para* `)`

`Hecke(` *rec* `)`

returns the cyclotomic Hecke algebra corresponding to the complex reflection group G (see the introduction). The following forms are accepted for *para*: if *para* is a single value, it is replicated to become a list of same length as the number of generators of W. Otherwise, *para* should be a list of the same length as the number of generators of W, with possibly unbound entries (which means it can also be a list of lesser length). There should be at least one entry bound for each orbit of reflections, and if several entries are bound for one orbit, they should all be identical. Now again, an entry for a reflection of order e can be either a single value or a list of length `e`. If it is a list, it is interpreted as the list $[u_0,\ldots,u_{e-1}]$ of parameters for that reflection. If it is a single value q, it is interpreted as the partly specialized list of parameters `[q,E(e),...,E(e-1)]` (thus the convention is upwardly compatible with that for Coxeter groups, and `Hecke(G,1)` is the group algebra of G over the cyclotomic field $\mathbb{Q}(\{E(e)\}_e)$ where e runs over the orders of the generating reflections).

```
gap> G := ComplexReflectionGroup(4);
ComplexReflectionGroup(4)
gap> v := X( Cyclotomics );; v.name := "v";;
gap> CH := Hecke( G, v );
Hecke(G4,v)
gap> CH.parameter;
[ [ v, E(3), E(3)^2 ], [ v, E(3), E(3)^2 ] ]
```

Here the single parameter `v` is interpreted as `[v,v]` which is in turn interpreted according to the above rules as `[[v,E(3),E(3)^2],[v,E(3),E(3)^2]]`.

The second form of the function `Hecke` takes as an argument a record which has a field `hecke` and returns the value of this field. This is used to return the Hecke algebra of objects derived from Hecke algebras, such as Hecke elements in various bases.

90.2 Operations for cyclotomic Hecke algebras

`Group`
: returns the complex reflection group from which the cyclotomic Hecke algebra was generated.

`Print`
: prints the cyclotomic Hecke algebra in a compact form. use `FormatGAP` for a form which can be read back into GAP3.

```
gap> G := ComplexReflectionGroup( 4 );
ComplexReflectionGroup(4)
gap> v := X( Cyclotomics );; v.name := "v";;
gap> CH := Hecke( G, v );
Hecke(G4,v)
gap> FormatGAP(CH);
"Hecke(ComplexReflectionGroup(4),v)"
```

`CharTable`
: returns the character table for some types of cyclotomic Hecke algebras, namely those of imprimitive type and the primitive reflection groups numbered `G(4)` to `G(30)` in the Shephard-Todd classification, as well as `G(33)`. This is a record with exactly the same components as for the corresponding complex reflection group but where the component `irreducibles` contains the values of the irreducible characters of the algebra on certain basis elements T_w where w runs over the elements in the component `classtext`. Thus, the values are now polynomials in the parameters of the algebra. There are partial tables for the remaining groups `G(31)`, `G(32)`, `G(34)`.

```
gap> Display( CharTable( CH ) );
H(G4)

          2 3     3   2     1        1      1            1
          3 1     1   .     1        1      1            1

            .     z 212    12      z12      1           1z
         2P .     .   z     1        1    z12          z12
         3P .     z 212     z        .      .            z
         5P .     z 212    1z        1    z12           12

phi{1,0}    1   v^6 v^3   v^2      v^8      v          v^7
phi{1,4}    1     1   1  E3^2     E3^2     E3           E3
phi{1,8}    1     1   1    E3       E3   E3^2         E3^2
phi{2,5}    2    -2   .     1       -1     -1            1
phi{2,3}    2 -2v^3   . E3^2v -E3^2v^4 v+E3^2 -v^4-E3^2v^3
```

```
phi{2,1}    2 -2v^3   .   E3v   -E3v^4   v+E3   -v^4-E3v^3
phi{3,2}    3  3v^2  -v     .        .    v-1      v^3-v^2
```

90.3 SchurElements

`SchurElements(` H `)`

returns the list Schur elements for the (cyclotomic) Hecke algebra H (see the introduction for their definition).

```
gap> v:=X(Cyclotomics);;v.name:="v";;
gap> H:=Hecke(ComplexReflectionGroup(4),v);
Hecke(G4,v)
gap> SchurElements(H);
[ v^8 + 2*v^7 + 3*v^6 + 4*v^5 + 4*v^4 + 4*v^3 + 3*v^2 + 2*v + 1,
  (2*E(3)-2*E(3)^2) + (-2*E(3)-10*E(3)^2)*v^(-1) + 12*v^(-2) + (
    -10*E(3)-2*E(3)^2)*v^(-3) + (-2*E(3)+2*E(3)^2)*v^(-4),
  (-2*E(3)+2*E(3)^2) + (-10*E(3)-2*E(3)^2)*v^(-1) + 12*v^(-2) + (
    -2*E(3)-10*E(3)^2)*v^(-3) + (2*E(3)-2*E(3)^2)*v^(-4),
  2 + 2*v^(-1) + 4*v^(-2) + 2*v^(-3) + 2*v^(-4),
  (-2*E(3)-E(3)^2)*v^3 + (-4*E(3)-2*E(3)^2)*v^2 + 3*v + (
    -2*E(3)-4*E(3)^2) + (-E(3)-2*E(3)^2)*v^(-1),
  (-E(3)-2*E(3)^2)*v^3 + (-2*E(3)-4*E(3)^2)*v^2 + 3*v + (
    -4*E(3)-2*E(3)^2) + (-2*E(3)-E(3)^2)*v^(-1),
  v^2 + 2*v + 2 + 2*v^(-1) + v^(-2) ]
gap> List(last,CycPol);
[ P2^2P3P4P6, 2ER(-3)v^-4P2^2P'3P'6, -2ER(-3)v^-4P2^2P"3P"6,
  2v^-4P3P4, (3-ER(-3))/2v^-1P2^2P'3P"6, (3+ER(-3))/2v^-1P2^2P"3P'6,
  v^-2P2^2P4 ]
```

90.4 SchurElement

`SchurElement(` H, *phi* `)`

returns the Schur element (see `SchurElements`) of the Cyclotomic Hecke algebra H for the irreducible character of H of parameter *phi* (see `CharParams` in section 103);

```
gap> v := X( Cyclotomics );; v.name := "v";;
gap> W:=ComplexReflectionGroup(4);;
gap> H := Hecke( W, v);
Hecke(G4,v)
gap> SchurElement( H, [ [ 2, 5] ] );
2 + 2*v^(-1) + 4*v^(-2) + 2*v^(-3) + 2*v^(-4)
```

90.5 FactorizedSchurElements

`FactorizedSchurElements(` H `)`

Let H be a (cyclotomic) Hecke algebra for the complex reflection group W, whose parameters are all (Laurent) monomials in some variables $x_1, \ldots, x_n$, and let K be the field of definition of W. Then Maria Chlouveraki has shown that the Schur elements of H then take the particular form $M\prod_\Phi \Phi(M_\Phi)$ where Φ runs over a list of K-cyclotomic polynomials, and M and M_Φ are (Laurent) monomials (in possibly some fractional powers) of the variables x_i. The function `FactorizedSchurElements` returns a data structure which shows this factorization. In CHEVIE, the parameters of H must be `Mvp` (see 112.1).

```
gap> x:=Mvp("x");;y:=Mvp("y");;
gap> H:=Hecke(ComplexReflectionGroup(4),[[1,x,y]]);
Hecke(G4,[[1,x,y]])
gap> FactorizedSchurElements(H);
[ x^-4y^-4P1P6(x)P1P6(y)P2(xy), P1P6(x)P1P6(xy^-1)P2(x^2y^-1),
  -x^-4y^5P1P6(y)P2(xy^-2)P1P6(xy^-1),
  -x^-1yP1(x)P1(y)P6(xy^-1)P2(xy),
  -x^-4yP1(x)P6(y)P1(xy^-1)P2(x^2y^-1),
  x^-1y^-1P6(x)P1(y)P2(xy^-2)P1(xy^-1),
  x^-2yP2(xy^-2)P2(xy)P2(x^2y^-1) ]
```

90.6 FactorizedSchurElement

`FactorizedSchurElement(` *H* `,` *phi* `)`

returns the FactorizedSchur element (see `FactorizedSchurElements`) of the Cyclotomic Hecke algebra H for the irreducible character of H of parameter *phi* (see `CharParams` in section 103);

```
gap> W:=ComplexReflectionGroup(4);;
gap> H := Hecke( W, [[1,x,y]]);
Hecke(G4,[[1,x,y]])
gap> FactorizedSchurElement( H, [ [ 2, 5] ] );
-x^-1yP1(x)P1(y)P6(xy^-1)P2(xy)
```

90.7 Functions and operations for FactorizedSchurElements

In CHEVIE, a `FactorizedSchurElement` representing a Schur element of the form $M\prod_\Phi \Phi(M_\Phi)$ is a record with a field `.factor` which holds the monomial M, and a field `.vcyc` which holds a list of record describing each factor in the product. An element of `.vcyc` representing a term $\Phi(M_\Phi)$ is itself record with fields `.monomial` holding M_Φ, and a field `.pol` holding a `CycPol` (see 106.2) representing Φ. A monomial is an `Mvp` with a single term.

The arithmetic operations `*` and `/` work for `FactorizedSchurElements`:

```
gap> H:=Hecke(W,x);
Hecke(G4,x)
gap> p:=FactorizedSchurElement( H, [ [ 2, 5] ] );
2x^-4P3P4(x)
gap> l:=FactorizedSchurElements(H);;
gap> List(l,x->l[1]/x);
[ 1, -ER(-3)/6x^4P"3P4P"6(x), ER(-3)/6x^4P'3P4P'6(x), 1/2x^4P2^2P6(x),
  (3+ER(-3))/6xP"3P4P'6(x), (3-ER(-3))/6xP'3P4P"6(x), x^2P3P6(x) ]
```

They also have `Print` and `String` methods, as well as the following methods:

`Value` this function works as for `Mvps`, and partially or completely evaluates the given element keeping as much as possible the factorized form.

```
gap> W:=ComplexReflectionGroup(4);;
gap> H := Hecke( W, [[1,x,y]]);
Hecke(G4,[[1,x,y]])
gap> p:=FactorizedSchurElement( H, [ [ 2, 5] ] );
-x^-1yP1(x)P1(y)P6(xy^-1)P2(xy)
gap> Value(p,["x",E(3)]);
(3-ER(-3))/2y^-1P1P2P'6^2(y)
gap> Value(last,["y",2]);
-9ER(-3)/2
```

`Expand` this function expands the element, converting it to an `Mvp`.

```
gap> Expand(p);
1-x^-1y+x^-1y^2-xy^-1+2xy-xy^3-2x^2-2y^2+x^2y^-1+x^2y^2+x^3+y^3-x^3y
```

90.8 LowestPowerGenericDegrees for cyclotomic Hecke algebras

`LowestPowerGenericDegrees(` *H* `)`

H should be an Hecke algebra all of whose parameters are monomials in the same indeterminate. `LowestPowerGenericDegrees` returns a list holding, for each character χ, the opposite of the valuation of the Schur element of χ (for an Hecke algebra of a Coxeter group this is Lusztig's a-function). One should note that this function first computes explicitly the Schur elements, so for a one-parameter algebra, `LowestPowerGenericDegrees(Group(H))` may be much faster.

```
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> H:=Hecke(ComplexReflectionGroup(6),[q^2,q^4]);
Hecke(G6,[q^2,q^4])
gap> LowestPowerGenericDegrees(H);
[ 0, 10, 10, 2, 28, 28, 18, 4, 4, 18, 4, 4, 6, 12 ]
```

90.9 HighestPowerGenericDegrees for cyclotomic Hecke algebras

`HighestPowerGenericDegrees(` *H* `)`

H should be an Hecke algebra all of whose parameters are monomials in the same indeterminate. `HighestPowerGenericDegrees` returns a list holding, for each character χ, the degree of the Poincar'e polynomial minus the degree of the Schur element of χ (for an Hecke algebra of a Coxeter group this is Lusztig's A-function). One should note that this function first computes explicitly the Schur elements, so for a one-parameter algebra, `HighestPowerGenericDegrees(Group(H))` may be much faster.

```
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> H:=Hecke(ComplexReflectionGroup(6),[q^2,q^4]);
```

```
Hecke(G6,[q^2,q^4])
gap> HighestPowerGenericDegrees(H);
[ 0, 38, 38, 22, 44, 44, 42, 32, 32, 42, 32, 32, 34, 36 ]
```

90.10 HeckeCentralMonomials

`HeckeCentralMonomials(` *HW* `)`

Returns the scalars by which the central element T_π acts on irreducible representations of HW. Here, for an irreducible group, π is the generator of the center of the pure braid group, which is also $z^{|Z|}$ where z is the generator of the center of the braid group and $|Z|$ the order of the center of W. In the case of an Iwahori-Hecke algebra, T_π is thus $T_{w_0}^2$.

```
gap> v := X( Cyclotomics );; v.name := "v";;
gap> H := Hecke( CoxeterGroup( "H", 3 ),  v );;
gap> HeckeCentralMonomials( H );
[ v^0, v^30, v^12, v^18, v^10, v^10, v^20, v^20, v^15, v^15 ]
```

90.11 Representations for cyclotomic Hecke algebras

`Representations(` *H* `[,` *l* `])`

This function returns the list of representations of the algebra H. Each representation is returned as a list of the matrix images of the generators. This function is only partially implemented for the Hecke algebras of the groups G_{31}, G_{32} and G_{34}:we have 50 representations out of 59 for type G_{31}, 30 representations out of 102 for type G_{32} and 38 representations out of 169 for type G_{34}.

If there is a second argument l, it must be a list of indices (resp. a single index), and only the representations with these indices (resp. that index) in the list of all representations are returned.

```
gap> W:=ComplexReflectionGroup(4);;
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> H:=Hecke(W,q);
Hecke(G4,q)
gap> Representations(H);
[ [ [ [ q ] ], [ [ q ] ] ], [ [ [ E(3)*q^0 ] ], [ [ E(3)*q^0 ] ] ],
  [ [ [ E(3)^2*q^0 ] ], [ [ E(3)^2*q^0 ] ] ],
  [ [ [ E(3)*q^0, 0*q^0 ], [ -E(3)*q^0, E(3)^2*q^0 ] ],
      [ [ E(3)^2*q^0, E(3)^2*q^0 ], [ 0*q^0, E(3)*q^0 ] ] ],
  [ [ [ q, 0*q^0 ], [ -q, E(3)^2*q^0 ] ],
      [ [ E(3)^2*q^0, E(3)^2*q^0 ], [ 0*q^0, q ] ] ],
  [ [ [ q, 0*q^0 ], [ -q, E(3)*q^0 ] ],
      [ [ E(3)*q^0, E(3)*q^0 ], [ 0*q^0, q ] ] ],
  [ [ [ E(3)^2*q^0, 0*q^0, 0*q^0 ],
          [ (E(3)^2)*q + (E(3)^2), E(3)*q^0, 0*q^0 ],
          [ E(3)*q^0, q^0, q ] ],
      [ [ q, -q^0, E(3)*q^0 ], [ 0*q^0, E(3)*q^0,
              (-E(3)^2)*q + (-E(3)^2) ], [ 0*q^0, 0*q^0, E(3)^2*q^0 ]
         ] ] ]
```

The models implemented for imprimitive types $G(de, e, n)$ for $n > 2$ and $de > 1$, excepted for $G(3, 3, 3), G(3, 3, 4), G(3, 3, 5)$ and $G(4, 4, 3)$, involve rational fractions, so work only with `Mvp` parameters for H.

```
gap> W:=ComplexReflectionGroup(6,6,3);;
gap> H:=Hecke(W,Mvp("x"));
Hecke(G663,x)
gap> Representations(H,6);
[ [ [ -1, 0, 0 ], [ 0, -1/2+1/2x, -1/2-1/2x ],
      [ 0, -1/2-1/2x, -1/2+1/2x ] ],
  [ [ -1, 0, 0 ], [ 0, -1/2+1/2x, 1/2+1/2x ],
      [ 0, 1/2+1/2x, -1/2+1/2x ] ],
  [ [ (-x+x^2)/(1+x), (1+x^2)/(1+x), 0 ],
      [ 2x/(1+x), (-1+x)/(1+x), 0 ], [ 0, 0, -1 ] ] ]
```

90.12 HeckeCharValues for cyclotomic Hecke algebras

`HeckeCharValues(` H, w`)`

Let W be the group for which H is a Hecke algebra. w should be a word in the generators of W. The function returns the values of the irreducible characters of H on the image in H of the braid group element defined by the word w, whenever possible. It first test if w is a known representative of a conjugacy class in `ChevieClassInfo(W).classtext`. Then, if W is a Coxeter group, it returns `HeckeCharValues` on the element of the `"T"` basis defined by w. Finally, it tries to compute the matrix of w in the various representations using `Representations`.

```
gap> W:=ComplexReflectionGroup(4);;
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> H:=Hecke(W,q);
Hecke(G4,q)
gap> HeckeCharValues(H,[1,2,1,2,1,2]);
[ q^6, q^0, q^0, -2*q^0, -2*q^3, -2*q^3, 3*q^2 ]
```

Chapter 91

Iwahori-Hecke algebras

In this chapter we describe functions for dealing with Iwahori-Hecke algebras associated to Coxeter groups.

Let W, S be a Coxeter system, where W is generated by S and denote by $m_{s,t}$ the order of the product st for $s, t \in S$. Let R be a commutative ring with 1 and for $s \in S$ let $u_{s,0}, u_{s,1}$ be elements in R such that $u_{s,0} = u_{t,0}$ and $u_{s,1} = u_{t,1}$ whenever $s, t \in S$ are conjugate in W (this is the same as requiring that $u_{s,i} = u_{t,i}$ whenever $m_{s,t}$ is odd). The corresponding Iwahori-Hecke algebra with parameters $\{u_{s,i}\}$ is a deformation of the group algebra of W over R. More precisely, $H = H(W, R, \{u_{s,i}\})$ is the unitary associative R-algebra generated by elements $\{T_s\}_{s\in S}$ subject to the relations:

$$\begin{array}{rcll} (T_s - u_{s,0})(T_s - u_{s,1}) & = & 0 & \text{for all } s \text{ (the quadratic relations)} \\ T_sT_tT_s\cdots & = & T_tT_sT_t\cdots & \text{with } m_{s,t} \text{ factors on each side (the braid relations).} \end{array}$$

If $u_{s,0} = 1$ and $u_{s,1} = -1$ for all s then the quadratic relations become $T_s^2 = 1$ and the deformation of the group algebra is trivial.

Since the generators T_s satisfy the braid relations, the algebra H is in fact a quotient of the group algebra of the braid group associated with W. It follows that, if $w = s_1 \cdots s_m = t_1 \cdots t_m$ are two reduced expressions of $w \in W$ as products of elements of S, then the corresponding products of the generators T_{s_i} respectively T_{t_j} will give the same element of H, which we may therefore denote by T_w. We have $T_1 = 1$.

If one of $u_{s,0}$ or $u_{s,1}$ is invertible in R, for example $u_{s,1}$, then by changing the generators to $-T_s/u_{s,1}$, and setting $q_s = -u_{s,0}/u_{s,1}$, the braid relations do no change (since when $m_{s,t}$ is odd we have $u_{s,i} = u_{t,i}$) but the quadratic relations become $(T_s - q_s)(T_s + 1) = 0$. This last form is the most common form considered in the literature. Another common form in the context of Kazhdan-Lusztig theory is obtained by scaling the generators as $-T_s/\sqrt{-u_{s,0}u_{s,1}}$, giving rise to the quadratic relations $(T_s - v_s)(T_s + v_s^{-1}) = 0$ where $v_s = \sqrt{q_s}$. The general form of parameters provided by CHEVIE is a special case of general cyclotomic Hecke algebras, and can be useful in many contexts.

The second form above, or for some algebras the character table in the first form, require a square root of $-u_{s,0}u_{s,1}$. CHEVIE provides a way to specify it with the field `.rootParameter` which can be given when constructing the algebra. If not given a root is automatically extracted when needed by the function `RootParameter`. Note however that sometimes an explicit choice of root is necessary which cannot be automatically determined.

There is a universal choice for R and $\{u_{s,i}\}$: Let $\{u_{s,i}\}_{s\in S,i\in\{0,1\}}$ be indeterminates such that $u_{s,i} = u_{t,i}$ whenever $m_{s,t}$ is odd, and let $A_0 = \mathbb{Z}[u_{s,i}]_{s,i}$ be the corresponding polynomial ring. Then $H_0 := H(W, A_0, \{u_{s,i}\})$ is called the **generic Iwahori-Hecke algebra** associated with W. Another algebra $H(W, R, \{v_{s,i}\})$ can be obtained by specialization from H_0: There is a unique ring homomorphism $f : A_0 \to R$ such that $f(u_{s,i}) = v_{s,i}$ for all i. Then we can view R as an A_0-module via f and we can identify $H(W, R, \{v_{s,i}\}) = R\otimes_{A_0} H_0$.

The elements $\{T_w \mid w \in W\}$ actually form an R-basis of H if one of the $u_{s,i}$ is invertible for all s. The structure constants in that basis is obtained as follows. To multiply T_v by T_w, choose a reduced expression for v, say $v = s_1 \cdots s_k$ and apply inductively the formula:

$$T_sT_w = \begin{cases} T_{sw} & \text{if } l(sw) = l(w) + 1 \\ -u_{s,0}u_{s,1}T_{sw} + (u_{s,0} + u_{s,1})T_w & \text{if } l(sw) = l(w) - 1. \end{cases}$$

If all s we have $u_{s,0} = q$, $u_{s,1} = -1$ then we call the corresponding algebra the one-parameter or Spetsial Iwahori-Hecke algebra associated with W; it can be obtained with the simplified call `Hecke(W,q)`. Certain invariants of the irreducible characters of this algebra play a special role in the representation theory of the underlying finite Coxeter groups, namely the a- and A-invariants already occured in chapter 87 (see 87.10, 88.7). For basic properties of Iwahori-Hecke algebras and their relevance to the representation theory of finite groups of Lie type, see for example [CR87], Sections 67 and 68.

In the following example, we compute the multiplication table for the 0-Iwahori–Hecke algebra associated with the Coxeter group of type A_2.

```
gap> W := CoxeterGroup( "A", 2 );
CoxeterGroup("A",2)
```

One-parameter algebra with $q = 0$:

```
gap> H := Hecke( W, 0 );
Hecke(A2,0)
```

Create the T-basis:

```
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> el := CoxeterWords( W );
[ [  ], [ 2 ], [ 1 ], [ 2, 1 ], [ 1, 2 ], [ 1, 2, 1 ] ]
```

Multiply any two T-basis elements:

```
gap> PrintArray(List(el,x->List(el,y->T(x)*T(y))));
[[     T(),       T(2),       T(1),     T(2,1),     T(1,2),  T(1,2,1)],
 [    T(2),      -T(2),     T(2,1),    -T(2,1),   T(1,2,1), -T(1,2,1)],
 [    T(1),     T(1,2),      -T(1),   T(1,2,1),    -T(1,2), -T(1,2,1)],
 [  T(2,1),   T(1,2,1),    -T(2,1), -T(1,2,1), -T(1,2,1),  T(1,2,1)],
 [  T(1,2),    -T(1,2),   T(1,2,1), -T(1,2,1), -T(1,2,1),  T(1,2,1)],
 [T(1,2,1), -T(1,2,1), -T(1,2,1),  T(1,2,1),  T(1,2,1), -T(1,2,1)]]
```

Thus, we work with algebras with arbitrary parameters. We will see that this also works on the level of characters and representations.

91.1 Hecke for Coxeter groups

`Hecke(` *W* `[,` *parameter*`, [`*rootparameter*`]] )`

Constructs the Iwahori-Hecke algebra H of the given Coxeter group. The following forms are accepted for *parameter*: if *parameter* is a single value, it is replicated to become a list of same length as the number of generators of W. Otherwise, *parameter* should be a list of the same length as the number of generators of W, with possibly unbound entries (which means it can also be a list of lesser length). There should be at least one entry bound for each orbit of reflections, and if several entries are bound for one orbit, they should all be identical. Now again, an entry for a reflection can be either a single value or a list of length 2. If it is a list, it is interpreted as the list $[u_0, u_1]$ of parameters for that reflection. If it is a single value q, it is interpreted as `[q,-1]`.

If *parameter* are not given, they are assumed to be equal to 1. The Iwahori-Hecke algebra then degenerates to the group algebra of the Coxeter group. Thus both `Hecke(W)` and `Hecke(W,1)` specify the group algebra of W.

rootparameter is used to specify a square root of $-u_0u_1$ (a square root of q when $[u_0, u_1]$ are `[q,-1]`). It is usually a list like *parameter* with at least one bound entry per orbit of reflection, or it can be a single value which is replicated to become a list of same length as the number of generators of W. If not given then *rootparameter* is computed upon need by calling the function `RootParameter` (see 91.3).

```
gap> W := CoxeterGroup( "B", 3 );
CoxeterGroup("B",3)
gap> u := X( Rationals );; u.name := "u";;
```

One parameter algebra without and with specifying square roots:

```
gap> H := Hecke( W, u );
Hecke(B3,u)
gap> H := Hecke( W, u^2, u );
Hecke(B3,u^2,u)
gap> H := Hecke( W, [ u^6, u^4, u^4 ], [ u^3, -u^2, -u^2 ] );
Hecke(B3,[u^6,u^4,u^4],[u^3,-u^2,-u^2])
```

The parameters do not have to be indeterminates:

```
gap> H := Hecke( W, 9, 3 );
Hecke(B3,9,3)
gap> H := Hecke( W, [ u^6, u^4, u^8] );
Error, parameters should be equal for conjugate reflections 3 and 2 in
function ( arg ) ... end( CoxeterGroup("B",3), [ u^6, u^4, u^8 ]
 ) called from
function ( arg ) ... end( CoxeterGroup("B",3), [ u^6, u^4, u^8 ]
 ) called from
Hecke( W, [ u ^ 6, u ^ 4, u ^ 8 ] ) called from
main loop
brk>
```

91.2 Operations and functions for Iwahori-Hecke algebras

All operations for cyclotomic Hecke algebras are defined for Iwahori-Hecke algebras, in particular :

`Group`
: returns the Coxeter group from which the Hecke algebra was generated.

`Print`
: prints the Hecke algebra in a compact form. use `FormatGAP` for a form which can be read back into GAP3.

`SchurElements`
: see 90.4 and 90.3.

`CharTable`
: returns the character table of the Hecke algebra. This is a record with exactly the same components as for the corresponding finite Coxeter group but where the component `irreducibles` contains the values of the irreducible characters of the algebra on basis elements T_w where w runs over the elements in the component `classtext`. Thus, the value are now polynomials in the parameters of the algebra. For more details see the chapter 92.

`Basis`
: the `T` basis is described in the section below. Other bases are described in chapter 93.

91.3 RootParameter

`RootParameter(`H`, `i`)`

H should be an Iwahori-Hecke algebra. If its parameters are u_0, u_1 for the i-th generating reflection, this function returns a square root of $-u_0u_1$. These roots are necessary for certain operations on the algebra, like the character values of algebras of type E_7, E_8, or two-parameter G_2. If `rootparameters` have been given at the time of the definition of the Hecke algebra (see 91.1) then they are returned. If they had not been specified, then if $u_0u_1 = -1$ then 1 is returned, else the square root is computed as `GetRoot(`$-u_0u_1$`)` (see 103.7). It is useful to specify explicit square roots, since `GetRoot` does not work in all cases, or may yield the negative of the desired root and is generally inconsistent with respect to various specializations (there cannot exist any function `GetRoot` which will commute with arbitrary specializations).

```
gap> W:=CoxeterGroup("A",2);;
gap> q:=X(Rationals);;q.name:="q";;
gap> H:=Hecke(W,q^2,-q);
Hecke(A2,q^2,-q)
gap> RootParameter(H,1);
-q
gap> H:=Hecke(W,q^2);
Hecke(A2,q^2)
gap> RootParameter(H,1);
q
```

```
gap> H:=Hecke(W,3);
Hecke(A2,3)
gap> RootParameter(H,1);
-E(12)^7+E(12)^11
```

`RootParameter(`H`, `w`)`

If w is an element of `Group(H)` then the function returns the product of the `RootParameter`s for the reflections in a reduced expression for w.

```
gap> H:=Hecke(W,q^2,-q);
Hecke(A2,q^2,-q)
gap> RootParameter(H,LongestCoxeterElement(W));
-q^3
```

91.4 HeckeSubAlgebra

`HeckeSubAlgebra( `H`, `r` )`

Given an Hecke Algebra H and a set of reflections of `Group(`H`)` given as their index in the reflections of `Parent(Group(H))` (see 88.1), return the Hecke sub-algebra generated by the T_s corresponding to these reflections. The reflections must be generating reflections if the Hecke algebra is not the group algebra of W.

As for Subgroup, a subalgebra of a subalgebra is given as a subalgebra of the parent algebra.

```
gap> u := X( Rationals );; u.name := "u";;
gap> H := Hecke( CoxeterGroup( "B", 2 ), u );
Hecke(B2,u)
gap> HeckeSubAlgebra( H, [ 1, 4 ] );
Hecke(B2,u)
gap> HeckeSubAlgebra( H, [ 1, 7 ] );
Error, Generators of a sub-Hecke algebra should be simple reflections \
in
function ( H, subW ) ... end( Hecke(B2,u), [ 1, 7 ] ) called from
HeckeSubAlgebra( H, [ 1, 7 ] ) called from
main loop
brk>
```

91.5 Construction of Hecke elements of the T basis

`Basis( `H`, "T")`

Let H be a Iwahori-Hecke algebra. The function `Basis(H,"T")` returns a function which can be used to make elements of the usual `T` basis of the algebra. It is convenient to assign this function with a shorter name when computing with elements of the Hecke algebra. In what follows we assume that we have done the assignment:

```
gap> T := Basis( H, "T" );
function ( arg ) ... end
```

`T( `w` )`

Here w is an element of the Coxeter group `Group(H)`. This call returns the basis element T_w of H.

`T(` *elts*`,` *coeffs*`)`

In this form, `elts` is a list of elements of `Group(H)` and `coeffs` a list of coefficients which should be of the same length `k`. The element `Sum([1..k],i->coeffs[i]*T(elts[i]))` of H is returned.

`T(` *list* `)`

`T(` *s1*`, ..,` *sn* `)`

In the above two forms, the GAP3 list *list* or the GAP3 list `[`*s1*`,..,`*sn*`]` represents the Coxeter word for an element w of `Group(H)`. The basis element T_w is returned (actually the call works even if the word `[`*s1*`,..,`*sn*`]` is not reduced and the element $T_{s_1}\ldots T_{s_n}$ is returned also in that case).

```
gap> W := CoxeterGroup( "B", 3 );;
gap> u := X( Rationals );; u.name := "u";;
gap> H := Hecke( W, u );;
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> T( 1, 2 ) = T( [ 1, 2 ] );
true
gap> T( 1, 2 ) = T( EltWord( W, [ 1, 2 ] ) );
true
gap> T(1,1);
uT()+(u-1)T(1)
gap> l := [ [], [ 1, 2, 3 ], [ 1 ], [ 2 ], [ 3 ] ];;
gap> pl := List( l, i -> EltWord( W, i ) );;
gap> h := T( pl, [ u^100, 1/u^20, 1, -5, 0 ] );
u^100T()+T(1)-5T(2)+u^-20T(1,2,3)
gap> h.elm;
[ (), ( 1, 4)( 2,11)( 3, 5)( 8, 9)(10,13)(12,14)(17,18),
  ( 1,10)( 2, 6)( 5, 8)(11,15)(14,17),
  ( 1,16,13,10, 7, 4)( 2, 8,12,11,17, 3)( 5, 9, 6,14,18,15) ]
gap> h.coeff;
[ u^100, -5, 1, u^(-20) ]
```

The last two lines show that a Hecke element is represented internally by a list of elements of W and the corresponding list of coefficients of the basis elements in H.

The way elements of the Iwahori-Hecke algebra are printed depends on `CHEVIE.PrintHecke` . If it is set to `CHEVIE.PrintHecke:=rec(GAP:=true)`, they are printed in a way which can be input back in GAP3. When you load CHEVIE, the record `PrintHecke` initially set to `rec()`. To go on from the above example:

```
gap> CHEVIE.PrintHecke:=rec(GAP:=true);;
gap> h;
u^100*T()+T(1)-5*T(2)+u^-20*T(1,2,3)
gap> CHEVIE.PrintHecke:=rec();;
gap> h;
u^100T()+T(1)-5T(2)+u^-20T(1,2,3)
```

91.6 Operations for Hecke elements of the T basis

All examples below are with `CHEVIE.PrintHecke=""`.

`Hecke(` *a* `)` returns the Hecke algebra of which a is an element.

a `*` *b* The multiplication of two elements given in the `T` basis of the same Iwahori-Hecke algebra is defined, returning a Hecke element expressed in the `T` basis.

```
gap> q := X( Rationals );; q.name := "q";;
gap> H := Hecke( CoxeterGroup( "A", 2 ), q );
Hecke(A2,q)
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> ( T() + T( 1 ) ) * ( T() + T( 2 ) );
T()+T(1)+T(2)+T(1,2)
gap> T( 1 ) * T( 1 );
qT()+(q-1)T(1)
gap> T( 1, 1 ); # the same
qT()+(q-1)T(1)
```

a `^` *i* An element of the `T` basis with a coefficient whose inverse is still a Laurent polynomial in q can be raised to an integral, positive or negative, power, returning another element of the algebra. An arbitrary element of the algebra can only be raised to a positive power.

```
gap> ( q * T( 1, 2 ) ) ^ -1;
(q^-1-2q^-2+q^-3)T()+(-q^-2+q^-3)T(1)+(-q^-2+q^-3)T(2)+q^-3T(2,1)
gap> ( T( 1 ) + T( 2 ) ) ^ -1;
Error, inverse implemented only for single T_w in
h.operations.inverse( h ) called from
<rec1> ^ <rec2> called from
main loop
brk>
gap> ( T( 1 ) + T( 2 ) ) ^ 2;
2qT()+(q-1)T(1)+(q-1)T(2)+T(1,2)+T(2,1)
```

a `/` *b* This is equivalent to `a*b^-1`.

a `+` *b*

a `-` *b* Elements of the algebra expressed in the `T` basis can be added or subtracted, giving other elements of the algebra.

```
gap> T( 1 ) + T();
T()+T(1)
gap> T( 1 ) - T( 1 );
0
```

`Print(` *a* `)` prints the element a, using the form initialized in CHEVIE.PrintHecke.

`String(` *a* `)` provides a string containing the same result that is printed with `Print`.

`Coefficient(` *a*, *w* `)` Returns the coefficient of the Hecke element a on the basis element T_w. Here w can be given either as a Coxeter word or as an element of the

Coxeter group.

`AlphaInvolution(` a `)` This implements the involution on the algebra defined by $T_w \mapsto T_{w^{-1}}$.

```
gap> AlphaInvolution( T( 1, 2 ) );
T(2,1)
```

`BetaInvolution(` a `)` This is only defined when the Kazhdan-Lusztig bases of the Hecke algebra can be defined. It implements the involution on the algebra defined by $x \mapsto \overline{x}$ on coefficients and and $T_w \mapsto q_{w_0}^{-1} T_{w_0 w}$.

`AltInvolution(` a `)` This is only defined when the Kazhdan-Lusztig bases of the Hecke algebra can be defined. It implements the involution on the algebra defined by $x \mapsto \overline{x}$ on coefficients and $T_s \mapsto -q_s T_s$. Essentially it corresponds to tensoring with the sign representation.

`Frobenius(`WF`)(` a `)` The Frobenius of a Coxeter Coset associated to `Group(Hecke(`a`))` can be applied to a. For more details see chapter 96.7.

```
gap> W:=CoxeterGroup("D",4);WF:=CoxeterCoset(W,(1,2,4));
CoxeterGroup("D",4)
3D4
gap> H:=Hecke(W,X(Rationals));
Hecke(D4,q)
gap> T:=Basis(H,"T");
function ( arg ) ... end
gap> Frobenius(WF)(T(1));
T(4)
gap> Frobenius(WF)(T(1),-1);
T(2)
```

`Representation(`a`)(` n `)` Returns the image of a in the representation n of `H =Hecke(`a`)`. n can be a representation of H, or an integer specifying the n-th representation of H.

```
gap> Representation(T(1,2),2);
[ [ -q, 0*q^0, 0*q^0 ], [ 0*q^0, -q, 0*q^0 ],
  [ q^2 - q, -q + 1, q^0 ] ]
gap> r:=Representations(H,2);
[ [ [ q - 1, -q^0, 0*q^0 ], [ -q, 0*q^0, 0*q^0 ],
      [ -q^2 + q, q - 1, -q^0 ] ],
  [ [ 0*q^0, q^0, 0*q^0 ], [ q, q - 1, 0*q^0 ],
      [ 0*q^0, 0*q^0, -q^0 ] ],
  [ [ -q^0, 0*q^0, 0*q^0 ], [ 0*q^0, 0*q^0, q^0 ],
      [ 0*q^0, q, q - 1 ] ],
  [ [ 0*q^0, q^0, 0*q^0 ], [ q, q - 1, 0*q^0 ],
      [ 0*q^0, 0*q^0, -q^0 ] ] ]
gap> Representation(T(1,2),r);
[ [ -q, 0*q^0, 0*q^0 ], [ 0*q^0, -q, 0*q^0 ],
  [ q^2 - q, -q + 1, q^0 ] ]
```

91.7 HeckeClassPolynomials

`HeckeClassPolynomials(` *h* `)`

returns the class polynomials of the Hecke element h of the Hecke algebra H with respect to representatives *reps* of minimal length in the conjugacy classes of the Coxeter group *Group(H)*. Such minimal length representatives are given by the function `ChevieClassInfo(Group(H)).classtex`
These polynomials have the following property. Given the class polynomials p corresponding to h and the matrix X of the values of the irreducible characters of the Iwahori-Hecke algebra on T_w (for w in *reps*), then the product `X*p` is the list of values of the irreducible characters on the element h of the Iwahori-Hecke algebra.

```
gap> u := X( Rationals );; u.name := "u";;
gap> W := CoxeterGroup( "A", 3 );
CoxeterGroup("A",3)
gap> H := Hecke( W, u );;
gap> h := Basis( H, "T" )( LongestCoxeterElement( W ) );
T(1,2,1,3,2,1)
gap> cp := HeckeClassPolynomials( h );
[ 0*u^0, 0*u^0, u^2, u^3 - 2*u^2 + u, u^3 - u^2 + u - 1 ]
gap> CharTable( H ).irreducibles * cp;
[ u^0, -u^2, 2*u^3, -u^4, u^6 ]
```

So, the entries in this list are the values of the irreducible characters on the basis element corresponding to the longest element in the Coxeter group.

The class polynomials were introduced in [GP93].

91.8 HeckeCharValues

`HeckeCharValues(` *h* `[,`*irreds*`])`

h is an element of an Iwahori-Hecke algebra (expressed in any basis) and *irreds* is a set of irreducible characters of the algebra (given as vectors). `HeckeCharValues` returns the values of *irreds* on the element h (the method used is to convert to the `T` basis, and then use `HeckeClassPolynomials`). If *irreds* is not given, all character values are returned.

```
gap> q := X( Rationals );; q.name := "q";;
gap> H := Hecke( CoxeterGroup( "B", 2 ), q ^ 2, q );;
gap> HeckeCharValues( Basis( H, "C'" )( 1, 2, 1 ) );
[ -q - q^(-1), q + q^(-1), 0*q^0, q^3 + 2*q + 2*q^(-1) + q^(-3),
  0*q^0 ]
```

See also 91.7.

91.9 Specialization from one Hecke algebra to another

`Specialization(` *H1* `,` *H2* `,` *f* `)`

H1 and *H2* should be Hecke algebras of the same group, and f should be a function which applied to the parameters of *H1* gives those of *H2*. The result is a function which can transport Hecke elements from *H1* to *H2* by the specialization map induced by f. As an example below, we compute the so-called "Kazhdan-Lusztig basis" of the symmetric group by specializing that of the Hecke algebra.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> W:=CoxeterGroup("A",2);H1:=Hecke(W,q^2);H2:=Hecke(W);
CoxeterGroup("A",2)
Hecke(A2,q^2)
Hecke(A2)
gap> T:=Basis(H1,"T");Cp:=Basis(H1,"C'");
function ( arg ) ... end
# warning: C'basis: q chosen as 2nd root of q^2
function ( arg ) ... end
gap> f:=function(x)
>  if IsPolynomial(x) then return Value(x,1);
>  else return x; fi;
> end;
function ( x ) ... end
gap> s:=Specialization(H1,H2,f);
function ( t ) ... end
gap> CoxeterWords(W);
[ [  ], [ 2 ], [ 1 ], [ 2, 1 ], [ 1, 2 ], [ 1, 2, 1 ] ]
gap> List(last,x->s(T(Cp(x))));
[ T(), T()+T(2), T()+T(1), T()+T(1)+T(2)+T(2,1), T()+T(1)+T(2)+T(1,2),
  T()+T(1)+T(2)+T(1,2)+T(2,1)+T(1,2,1) ]
```

Note that in the above example we specialize `T(Cp(x))`, not `Cp(x)`: since Kazhdan-Lusztig bases do not, strictly speaking, exist for the symmetric group algebra, it does not make sense to specialize them so they have no `Specialization` method. Rather, one has to convert to basis `T` first, which has a specialization method.

91.10 CreateHeckeBasis

`CreateHeckeBasis(`*basis*`,` *ops*`,` *heckealgebraops*`)`

creates a new basis for Hecke algebras in which to do computations. (The design of this function has benefited from conversation with Andrew Mathas, the author of the package `Specht`).

The first argument *basis* must be a unique (different from that used for other bases) character string. The second argument *ops* is a record which should contain at least two fields, *ops*`.T` and *ops*`.(`*basis*`)` which should contain :

ops`.T` a function which takes an element in the basis *basis* and converts it to the `T` basis.

ops`.(`*basis*`)` a function which takes an element in the `T` basis and converts it to the *basis* basis.

The third arguments should be the field `.operations` of some Hecke algebra. After the call to `CreateHeckeBasis`, a new field `(`*basis*`)` is added to `heckealgebraops` which contains a function to create elements of the *basis* basis. Thus all Hecke algebras which have the same field `.operations` will have the new basis.

The elements of the new basis will have the standard operations for Hecke elements: `+`, `-`, `*`, `^`, `=`, `Print`, `Coefficient`, plus all extra operations that the user may have specified in *ops*. It is thus possible to create a new basis which has extra operations. In addition, for any already created basis *y* of the algebra, the function `(`*y*`)` will have the added capability to convert elements from the *basis* basis to the *y* basis. If the user has provided a field *ops*`.(`*y*`)` , the function found there will be used. Otherwise, the function *ops*`.T` will be used to convert our *basis* element to the `T` basis, followed by calling the function `(`*y*`)` which was given in *ops* at the time the *y* basis was created, to convert to the *y* basis. The following forms of the Basis function will be accepted (as for the `T` basis):

`Basis(` *H* `,` *basis* `)(` *w* `)`

`Basis(` *H* `,` *basis* `)(` *elts* `,` *coeffs* `)`

`Basis(` *H* `,` *basis* `)(` *list* `)`

`Basis(` *H* `,` *basis* `)(` *s1* `, ..` `,` *sn* `)`

One should note, however that for the last two forms only reduced expressions will be accepted in general.

Below is an example where the basis $t_w = q^{-l(w)/2}T_w$ is created and used; we assume the equal parameter case. As an example of an extra operation in *ops*, we have given a method for `BetaInvolution`, which just dos the map $w \mapsto ww_0$ for the t basis. If methods for one of `BetaInvolution`, `AltInvolution` are given they will be automatically called by the generic functions with the same name.

In order to understand the following code, one has to recall that an arbitrary Hecke element is a record representing a sum of basis elements; it contains a list of Coxeter group elements in the component `elm` and the corresponding list of coefficients in the component `coeff`. For efficiency reasons, it is desirable to describe how to convert to another base such general Hecke elements and not just one basis element T_w or t_w.

```
gap> CreateHeckeBasis("t", rec(
>   T := h->Basis( Hecke(h), "T" )(h.elm, List( [1 .. Length( h.elm )],
>      i->RootParameter(Hecke(h),h.elm[i])^-1*h.coeff[i])),
>   t := h->Basis( Hecke(h), "t" )(h.elm, List( [1 .. Length( h.elm )],
>      i->RootParameter(Hecke(h), h.elm[i])*h.coeff[i])),
>   BetaInvolution := h->Basis(Hecke(h),"t")(List(h.elm,
>    x->x*LongestCoxeterElement(Group(Hecke(h)))),h.coeff)),
>    H.operations);
```

Now we setup the algebra using $v = q^{1/2}$ and use the basis `t`.

```
gap> v := X( Rationals );; v.name := "v";;
gap> H := Hecke( CoxeterGroup( "A", 3 ), v ^ 2, v );;
gap> h := Basis( H, "t" )( 3, 1, 2 );
t(1,3,2)
gap> h1 := Basis( H, "T")( h );
v^-3T(1,3,2)
gap> h2 := Basis( H, "t" )( h1 );
t(1,3,2)
gap> BetaInvolution( h2 );
t(2,1,3)
```

Parameterized bases

One can defined parameterized bases, that is bases whose behavior depend on some parameter(s). As an example we will define the basis $t(i)_w = q^{il(w)}T_w$, where we assume that $2i$ is an integer. We first write the example and then comment its differences from the previous case.

```
gap> CreateHeckeBasis( "ti", rec(
> T := h->Basis( Hecke(h), "T" )( h.elm, List( [1 .. Length( h.elm )],
>  i->Hecke(h).rootParameter[1]^(-CoxeterLength(
>          Group( Hecke(h) ), h.elm[i])*2*h.i)*h.coeff[i] ) ),
> extraFields:=["i"],
> ti := function(h,extra) return Basis( Hecke(h), "ti" )
>    ( h.elm, List( [1 .. Length( h.elm )],
>    i->Hecke(h).rootParameter[1]^(CoxeterLength(
>        Group( Hecke(h) ), h.elm[i])*2*extra.i)*h.coeff[i]), extra);
>    end,
> BetaInvolution:=h->Basis(Hecke(h),"ti")(
>   H.operations.T.BetaInvolution(Basis(Hecke(h),"T")(h)),
>   HeckeEltOps.GetExtra(h)),
> FormatBasis:=h->SPrint("t[",h.i,"]")),
> H.operations );
```

Here each Hecke element `h` has a field `h.i` holding the value of `i`. This is used in converting the element to basis `T`. The field `i` must be copied from one object to the other during various operations on Hecke elements. For CHEVIE to know which fields must be copied, the list of such fields must be declared, which is done by the above assignment `extraFields:=["i"]`. A record containing the extra fields must be passed as the last argument of each call of the `Basis` function, so it knows which information to put in the built Hecke elements; thus the call take one of the forms

`Basis(` *H* `,` *basis* `)(` *w* `,` *extra* `)`

`Basis(` *H* `,` *basis* `)(` *elts* `,` *coeffs* `,` *extra* `)`

`Basis(` *H* `,` *basis* `)(` *list* `,` *extra* `)`

`Basis(` *H* `,` *basis* `)(` *s1* `, .. ,` *sn* `,` *extra* `)`

The function `ti` to convert to the basis `ti` requires also a record argument containing the extra information, which it passes to the second form above. The function `BetaInvolution` illustrates how one can extract this extra information record from a Hecke element using the function `HeckeEltOps.GetExtra(h))`; we also used a different method to compute the `BetaInvolution`, delegating the computation to basis `T` instead of doing it directly. Finally the function `FormatBasis`, if given, produces a parameterized printing of a basis element. Here is how one can use the above definitions:

```
gap> v := X( Rationals );; v.name := "v";;
gap> H := Hecke( CoxeterGroup( "A", 3 ), v ^ 2, v );;
gap> t:=function(arg) Add(arg,rec(i:=1/2));
>  return ApplyFunc(Basis(H,"ti"),arg);end;
function ( arg ) ... end
```

Here the function `t` does exactly the same as `Basis(H,"t")` in the previous example, excepted for the printing of Hecke elements

```
gap> h := t( 3, 1, 2 );
t[1/2](1,3,2)
gap> h1 := Basis( H, "T")( h );
v^-3T(1,3,2)
gap> h2 := t( h1 );
t[1/2](1,3,2)
gap> BetaInvolution( h2 );
t[1/2](2,1,3)
```

The point is that one can now just as easily define similar functions for other values of `i`.

Chapter 92

Representations of Iwahori-Hecke algebras

Let W, S be a finite Coxeter system and $H = H(W, R, \{u_{s,i}\}_{s\in S, i\in\{0,1\}})$ a corresponding Iwahori-Hecke algebra over the ring R as defined in chapter 91. We shall now describe functions for dealing with representations and characters of H.

The fact that we know a presentation of H makes it easy to check that a list of matrices $M_s \in R^{d\times d}$ for $s \in S$ gives rise to a representation: there is a (unique) representation $\rho : H \to R^{d\times d}$ such that $\rho(T_s) = M_s$ for all $s \in S$, if and only if the matrices M_s satisfy the same relations as those for the generators T_s of H.

A general approach for the construction of representations is in terms of W-graphs, see [KL79, p.165]. Any such W-graph carries a representation of H. Note that in this approach, it is necessary to know the square roots of the parameters of H. The simplest example, the standard W-graph defined in [KL79, Ex. 6.2] yields a "deformation"of the natural reflection representation of W. This can be produced in CHEVIE using the function `HeckeReflectionRepresentation`.

Another possibility to construct W-graphs is by using the Kazhdan-Lusztig theory of left cells (see [KL79]); see the following chapter for more details.

In a similar way as the ordinary character table of the finite Coxeter group W is defined, one also has a character table for the Iwahori-Hecke algebra H in the case when the ground ring A is a field such H is split and semisimple. The generic choice for such a ground ring is the rational function field $K = \overline{\mathbb{Q}}(v_s)_{s\in S}$ where the parameters of the corresponding algebra H_K satisfy $-u_{s,0}/u_{s,1} = v_s^2$ for all s.

By Tits' Deformation Theorem (see [CR87, Sec. 68], for example), the algebra H_K is (abstractly) isomorphic to the group algebra of W over K. Moreover, we have a bijection between the irreducible characters of H_K and W, given as follows. Let χ be an irreducible character of H_K. Then we have $\chi(T_w) \in A$ where $A = \overline{\mathbb{Z}}[v_s]_{s\in S}$ and $\overline{\mathbb{Z}}$ denotes the ring of algebraic integers in $\overline{\mathbb{Q}}$. We can find a ring homomorphism $f : A \to \overline{\mathbb{Q}}$ such that $f(a) = a$ for all $a \in \overline{\mathbb{Z}}$ and $f(v_s) = 1$ for $s \in S$. Then it turns out that the function $\chi_f : w \mapsto f(\chi(T_w))$ is an irreducible character of W, and the assignment $\chi \mapsto \chi_f$ defines a bijection between the irreducible characters of H_K and W.

Now this bijection does depend on the choice of f. But one should keep in mind that this only plays a role in the case where W is a non-crystallographic Coxeter group. In all other cases, as is well-known, the character table of W is rational; moreover, the values of the irreducible characters of H_K on basis elements T_w lie in the ring $\mathbb{Z}[v_s]_{s\in S}$.

The character table of H_K is defined to be the square matrix $(\chi(T_w))$ where χ runs over the irreducible characters of H_K and w runs over a set of representatives of *minimal length* in the conjugacy classes of W. The character tables of Iwahori-Hecke algebras (in this sense) are known for all types: the table for type A was first computed by Starkey (see the description of his work in [Car86]); then different descriptions with different proofs were given in [Ram91] and [Pfe94]. The tables for the non crystallographic types $I_2(m)$, H_3, H_4 can be constructed from the explicit matrix representations given in [CR87, Sec. 67C], [Lus81] and [AL82], respectively. For the classical types B and D see [HR94] and [Pfe96]. The tables for the remaining exceptional types were computed in [Gec94], [Gec95] and [GM97].

If H is an Iwahori-Hecke algebra over an arbitrary ground ring R as above, then the GAP3 function `CharTable` applied to the corresponding record returns a character table record which is build up in exactly the same way as for the finite Coxeter group W itself but where the record component `irreducibles` contains the character values which are obtained from those of the generic multi-parameter algebra H_K by specializing the indeterminates v_i to the variables in `rootParameter`.

92.1 HeckeReflectionRepresentation

`HeckeReflectionRepresentation(` W `)`

returns a list of matrices which give the reflection representation of the Iwahori-Hecke algebra corresponding to the Coxeter group W. The function `Hecke` must have been applied to the record W.

```
gap> v:= X( Rationals );;  v.name := "v";;
gap> H := Hecke(CoxeterGroup( "B", 2) , v^2, v);
Hecke(B2,v^2,v)
gap> ref:= HeckeReflectionRepresentation( H );
[ [ [ -v^0, 0*v^0 ], [ -v^2, v^2 ] ],
  [ [ v^2, -2*v^0 ], [ 0*v^0, -v^0 ] ] ]
gap> H := Hecke( CoxeterGroup( "H", 3 ));;
gap> HeckeReflectionRepresentation( H );
[ [ [ -1, 0, 0 ], [ -1, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 1, E(5)+2*E(5)^2+2*E(5)^3+E(5)^4, 0 ], [ 0, -1, 0 ],
      [ 0, -1, 1 ] ], [ [ 1, 0, 0 ], [ 0, 1, -1 ], [ 0, 0, -1 ] ] ]
```

92.2 CheckHeckeDefiningRelations

`CheckHeckeDefiningRelations(` H `,` t `)`

returns true or false, according to whether a given set t of elements corresponding to the standard generators of *Group(H)* defines a representation of the Hecke algebra H or not.

```
gap> H := Hecke(CoxeterGroup( "F", 4 ));;
```

```
gap> r := HeckeReflectionRepresentation( H );;
gap> CheckHeckeDefiningRelations( H, r );
true
gap> CheckHeckeDefiningRelations(H,List([1..4],i->Basis(H,"T")(i)));
true
```

92.3 CharTable for Hecke algebras

`CharTable( H )`

`CharTable` returns the character table record of the Iwahori-Hecke algebra H. This is basically the same as the character table of a Coxeter group described earlier with the exception that the component `irreducibles` contains the matrix of the values of the irreducible characters of the generic Iwahori-Hecke algebra specialized at the parameters in the component `parameter` of H. Thus, if all these parameters are equal to $1 \in \mathbb{Q}$ then the component `irreducibles` just contains the ordinary character table of the underlying Coxeter group.

The function `CharTable` first recognizes the type of H, then calls special functions for each type involved in H and finally forms the direct product of all these tables.

```
gap> W := CoxeterGroup( "G", 2 );;
gap> u := X( Rationals );;  u.name := "u";;
gap> v := X( LaurentPolynomialRing( Rationals ) );; v.name := "v";;
gap> u := u * v^0;;
gap> H := Hecke( W, [ u^2, v^2 ], [ u, v ] );
Hecke(G2,[u^2,v^2],[u,v])
gap> Display( CharTable( H ) );
H(G2)

              2  2     2     2      1       1        2
              3  1     .     .      1       1        1

                A0   ~A1    A1     G2      A2   A1+~A1
             2P A0    A0    A0     A2      A2       A0
             3P A0   ~A1    A1 A1+~A1      A0   A1+~A1

phi{1,0}         1   v^2   u^2 u^2v^2  u^4v^4   u^6v^6
phi{1,6}         1    -1    -1      1       1        1
phi{1,3}'        1   v^2    -1   -v^2     v^4     -v^6
phi{1,3}''       1    -1   u^2   -u^2     u^4     -u^6
phi{2,1}         2 v^2-1 u^2-1    -uv -u^2v^2  2u^3v^3
phi{2,2}         2 v^2-1 u^2-1     uv -u^2v^2 -2u^3v^3
```

As mentioned before, the record components `classparam`, `classnames` and `irredinfo` contain canonical labels and parameters for the classes and the characters (see the introduction to chapter 87 and also 87.4). For direct products, sequences of such canonical labels of the individual types are given.

We can also have character tables for algebras where the parameters are not necessarily indeterminates:

```
gap> H1 := Hecke( W, [ E(6)^2, E(6)^4 ],[ E(6), E(6)^2 ] );
Hecke(G2,[E3,E3^2],[-E3^2,E3])
gap> ct := CharTable( H1 );
CharTable( "H(G2)" )
gap> Display( ct );
H(G2)

             2  2            2             2      1    1      2
             3  1            .             .      1    1      1

               A0          ~A1            A1     G2   A2 A1+~A1
            2P A0           A0            A0     A2   A2     A0
            3P A0          ~A1            A1 A1+~A1   A0 A1+~A1

phi{1,0}        1         E3^2            E3      1    1      1
phi{1,6}        1           -1            -1      1    1      1
phi{1,3}'       1         E3^2            -1  -E3^2   E3     -1
phi{1,3}''      1           -1            E3    -E3 E3^2     -1
phi{2,1}        2 (-3-ER(-3))/2 (-3+ER(-3))/2      1   -1     -2
phi{2,2}        2 (-3-ER(-3))/2 (-3+ER(-3))/2     -1   -1      2

gap> RankMat( ct.irreducibles );
5
```

The last result tells us that the specialized character table is no more invertible.

Character tables of Iwahori–Hecke algebras were introduced in [GP93]; see also the introduction to this chapter for further information about the origin of the various tables.

92.4 Representations for Hecke algebras

`Representations(` H, l `)`

This function returns the list of representations of the Iwahori-Hecke algebra H. Each representation is returned as a list of the matrix images of the generators T_s.

If there is a second argument, it can be a list of indices (or a single integer) and only the representations with these indices (or that index) in the list of all representations are returned.

```
gap> W:=CoxeterGroup("I",2,5);
CoxeterGroup("I",2,5)
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> H:=Hecke(W,q);
Hecke(I2(5),q)
gap> Representations(H);
[ [ [ [ q ] ], [ [ q ] ] ], [ [ [ -q^0 ] ], [ [ -q^0 ] ] ],
  [ [ [ -q^0, q^0 ], [ 0*q^0, q ] ],
      [ [ q, 0*q^0 ], [ (-E(5)-2*E(5)^2-2*E(5)^3-E(5)^4)*q, -q^0 ] ] ],
  [ [ [ -q^0, q^0 ], [ 0*q^0, q ] ], [ [ q, 0*q^0 ], [ (-2*E(5)-E(5)^2
               -E(5)^3-2*E(5)^4)*q, -q^0 ] ] ] ]
```

The models implemented for types B_n and D_n involve rational fractions, thus work only with algebras whose parameters are `Mvp`s.

```
gap> W:=CoxeterGroup("B",3);
CoxeterGroup("B",3)
gap> H:=Hecke(W,Mvp("x"));
Hecke(B3,x)
gap> Representations(H,2);
[ [ [ -1, 0, 0 ], [ 0, x, 0 ], [ 0, 0, x ] ],
  [ [ (-x+x^2)/(1+x), (1+x^2)/(1+x), 0 ],
      [ 2x/(1+x), (-1+x)/(1+x), 0 ], [ 0, 0, -1 ] ],
  [ [ -1, 0, 0 ], [ 0, -1/2+1/2x, 1/2+1/2x ],
      [ 0, 1/2+1/2x, -1/2+1/2x ] ] ]
```

92.5 PoincarePolynomial

`PoincarePolynomial(` H `)`

The Poincaré polynomial of the Hecke algebra H, which is equal to `SchurElements(`H`)[`ind`]` where *ind* is the position of the 1-dimensional index representation in the character table of H, that is, the representation which maps T_s to the corresponding parameter $u_{s,0}$.

```
gap> q := X( Rationals );; q.name := "q";;
gap> W := CoxeterGroup( "G", 2 );; H := Hecke( W, q );
Hecke(G2,q)
gap> PoincarePolynomial( H );
q^6 + 2*q^5 + 2*q^4 + 2*q^3 + 2*q^2 + 2*q + 1
```

92.6 SchurElements for Iwahori-Hecke algebras

`SchurElements(` H `)`

returns the list of constants S_χ arising from the Schur relations for the irreducible characters χ of the Iwahori-Hecke algebra H, that is $\delta_{w,1} = \sum_\chi \chi(T_w)/S_\chi$ where δ is the Kronecker symbol.

The element S_χ also equal to P/D_χ where P is the Poincare polynomial and D_χ is the generic degree of χ. Note, however, that this only works if $D_\chi \neq 0$. (We can have $D_\chi = 0$ if the parameters of H are suitably chosen roots of unity, for example.) The ordering of the Schur elements corresponds to the ordering of the characters as returned by the function `CharTable`.

```
gap> u := X( Rationals );; u.name := "u";;
gap> v := X( LaurentPolynomialRing( Rationals ) );; v.name := "v";;
gap> W := CoxeterGroup("G",2);;
gap> schur := SchurElements( Hecke( W, [ u ^ 2, v ^ 2 ]));
# warning: u*v chosen as 2nd root of (u^2)*v^2
[ (u^6 + u^4)*v^6 + (u^6 + 2*u^4 + u^2)*v^4 + (u^4 + 2*u^2 + 1)*v^
    2 + (u^2 + 1), (1 + u^(-2)) + (1 + 2*u^(-2) + u^(-4))*v^(
    -2) + (u^(-2) + 2*u^(-4) + u^(-6))*v^(-4) + (u^(-4) + u^(-6))*v^(
    -6), (u^(-4) + u^(-6))*v^6 + (u^(-2) + 2*u^(-4) + u^(-6))*v^4 + (
    1 + 2*u^(-2) + u^(-4))*v^2 + (1 + u^(-2)),
```

```
    (u^2 + 1) + (u^4 + 2*u^2 + 1)*v^(-2) + (u^6 + 2*u^4 + u^2)*v^(
      -4) + (u^6 + u^4)*v^(-6), (2*u^0)*v^2 + (-2*u + 2*u^(-1))*v + (
      2*u^2 - 2 + 2*u^(-2)) + (2*u - 2*u^(-1))*v^(-1) + (2*u^0)*v^(-2),
    (2*u^0)*v^2 + (2*u - 2*u^(-1))*v + (2*u^2 - 2 + 2*u^(-2)) + (-2*u +
      2*u^(-1))*v^(-1) + (2*u^0)*v^(-2) ]
```

The Poincaré polynomial is just the Schur element corresponding to the trivial (or index) representation:

```
gap> schur[PositionId(W)];
(u^6 + u^4)*v^6 + (u^6 + 2*u^4 + u^2)*v^4 + (u^4 + 2*u^2 + 1)*v^
2 + (u^2 + 1)
```

(note that the trivial character is not always the first character, which is why we use `PositionId`) For further information about generic degrees and connections with the representation theory of finite groups of Lie type, see [BC72] and [Car85].

92.7 SchurElement for Iwahori-Hecke algebras

`SchurElement(` *H* `,` *phi* `)`

returns the Schur element (see `Schur Elements for Iwahori-Hecke algebras`) of the Iwahori-Hecke algebra *H* for the irreducible character of *H* of parameter *phi* (see `CharParams` in section 103).

```
gap> u := X( Rationals );; u.name := "u";;
gap> v := X( LaurentPolynomialRing( Rationals ) );; v.name := "v";;
gap> H := Hecke( CoxeterGroup( "G", 2 ), [ u , v]);
Hecke(G2,[u,v])
gap> SchurElement( H, [ [ 1, 3, 1] ] );
(u^(-2) + u^(-3))*v^3 + (u^(-1) + 2*u^(-2) + u^(-3))*v^2 + (1 + 2*u^(
-1) + u^(-2))*v + (1 + u^(-1))
```

92.8 GenericDegrees

We do not have a function for the generic degrees of an Iwahori-Hecke algebra since they are not always defined (for example, if the parameters of the algebra are roots of unity). If they are defined, they can be computed with the command:

`List( SchurElements(` *H* `), x -> PoincarePolynomial(` *H* `) / x );`

(See 92.5 and 90.4.)

92.9 LowestPowerGenericDegrees for Hecke algebras

`LowestPowerGenericDegrees(` *H* `)`

H should be an Iwahori-Hecke algebra all of whose parameters are monomials in the same indeterminate. `LowestPowerGenericDegrees` returns a list holding the a-function for all irreducible characters of this algebra, that is, for each character χ, the valuation of the Schur element of χ. The ordering of the result corresponds to the ordering of the characters in `CharTable(H)`. One should note that this function first computes explicitly the Schur elements, so for a one-parameter algebra, `LowestPowerGenericDegrees(Group(H))` may be much faster.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> H:=Hecke(CoxeterGroup("B",4),[q^2,q]);
Hecke(B4,[q^2,q,q,q])
gap> LowestPowerGenericDegrees(H);
[ 7, 6, 7, 12, 20, 3, 5, 3, 7, 6, 13, 2, 3, 10, 1, 4, 2, 7, 0, 3 ]
```

92.10 HeckeCharValuesGood

`HeckeCharValuesGood(` H`,` w `)`

Let H be a Hecke algebra for the Coxeter group `CoxeterGroup(`H`)`, let w be a good element of `CoxeterGroup(`H`)` in the sense of [GM97] (the representatives of conjugacy classes stored in CHEVIE are such elements), and let d be the order of w.

`HeckeCharValuesGood` computes the values of the irreducible characters of the Iwahori-Hecke algebra HW on T_w^d. The point is that the character table of the Hecke algebra is not used, and that all the eigenvalues of T_w^d are monomials in `H.parameters`, so this can be used to find the absolute value of the eigenvalues of T_w, a step towards computing the character table of the Hecke algebra.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> H:=Hecke(CoxeterGroup("B",4),[q^2,q]);
Hecke(B4,[q^2,q,q,q])
gap> HeckeCharValuesGood( H, [ 1, 2, 3 ] );
[ q^12, 4*q^12, 3*q^12 + 3*q^8, 3*q^8 + 1, q^0, 2*q^18 + q^12,
  6*q^12, 2*q^18 + 3*q^16 + 3*q^12, 3*q^12 + 3*q^8 + 2*q^6,
  3*q^16 + 3*q^8, 2*q^6 + 1, 2*q^18, 3*q^16 + 3*q^12, 2*q^6,
  q^24 + 2*q^18, 4*q^12, q^24 + 3*q^16, q^12 + 2*q^6, q^24, q^12 ]
```

Chapter 93

Kazhdan-Lusztig polynomials and bases

Let $\mathcal{H}$ be the Iwahori-Hecke algebra of a Coxeter system (W, S), with quadratic relations $(T_s - u_{s,0})(T_s - u_{s,1}) = 0$ for $s \in S$. If $-u_{s,0}u_{s,1}$ has a square root, we can scale the basis T_s to $-T_s/\sqrt{-u_{s,0}u_{s,1}}$ to get a new basis t_s with quadratic relations $(t_s - v_s)(t_s + v_s^{-1}) = 0$ where $v_s = \sqrt{-u_{s,0}/u_{s,1}}$. The most general case when Kazhdan-Lusztig bases and polynomials can be defined is when the parameters v_s belong to a totally ordered abelian group Γ (where the group law is multiplication of parameters), see [Lus83]. We set $\Gamma^+ = \{\gamma \in \Gamma \mid \gamma\, 0\}$ and $\Gamma^- = \{\gamma^{-1} \mid \gamma \in \Gamma^+\} = \{\gamma \in \Gamma \mid \gamma < 0\}$.

Thus we assume $\mathcal{H}$ defined over the ring $\mathbb{Z}[\Gamma]$, the group algebra of Γ over $\mathbb{Z}$, and the quadratic relations of $\mathcal{H}$ associate to each $s \in S$ a $v_s \in \Gamma^+$ such that $(t_s - v_s)(t_s + v_s^{-1}) = 0$. We also set $q_s = v_s^2$ and define the basis $T_s = v_s t_s$ with quadratic relations $(T_s - q_s)(T_s + 1) = 0$; we extend the notation to define an element $q_w \in \Gamma_+$ by setting $q_w = q_{s_1} \ldots q_{s_n}$ if $w = s_1 \ldots s_n$ is a reduced expression for some $w \in W$, and denote $q_w^{(1/2)} = v_{s_1} \ldots v_{s_n}$.

We define the bar involution on $\mathcal{H}$ by linearity:on $\mathbb{Z}[\Gamma]$ we define it by $\overline{\sum_{\gamma\in\Gamma} a_\gamma \gamma} = \sum_{\gamma\in\Gamma} a_\gamma \gamma^{-1}$ and we extend it to $\mathcal{H}$ by $\overline{T}_s = T_s^{-1}$. Then the Kazhdan-Lusztig basis C'_w is defined as the only basis of $\mathcal{H}$ stable by the bar involution and congruent to t_w modulo $\sum_{w\in W} \Gamma_- t_w$.

The basis C'_w can be computed as follows. We define elements $R_{x,y}$ of $\mathbb{Z}[\Gamma]$ by $T_y^{-1} = \sum_x \overline{R_{x,y^{-1}}} q_x^{-1} T_x$. We then define inductively the Kazhdan-Lusztig polynomials (in this general context we should say the Kazhdan-Lusztig elements of $\mathbb{Z}[\Gamma]$, which belong to the subalgebra of $\mathbb{Z}[\Gamma]$ generated by the q_s) by $P_{x,w} = \tau_{\leq (q_w/q_x)^{1/2}}(\sum_{x<y\leq w} R_{x,y} P_{y,w})$ where τ is the truncation : $\tau_{\leq\nu} \sum_{\gamma\in\Gamma} a_\gamma\gamma = \sum_{\gamma\leq\nu} a_\gamma\gamma$; the induction is thus on decreasing x for the Bruhat order and starts at $P_{w,w} = 1$. We have then $C'_w = \sum_y q_w^{-1/2} P_{y,w} T_y$.

CHEVIE can compute Kazhdan-Lusztig polynomials, left cells, and the various Kazhdan-Lusztig bases of Iwahori-Hecke algebras (see [KL79]). More facilities are implemented for the one-parameter case when all v_s have a common value v.

There is a separate function to compute one-parameter kazhdan-Lusztig polynomials. From a computational point of view, even this case is quite a challenge. It seems that the best approach still is by using the recursion formula in the original article [KL79] (which deals with the one-parameter case, where the above recursion simplifies). One can first run a number of standard checks on a given pair of elements to see if the computation of the corresponding polynomial can be reduced to a similar computation for elements of smaller length, for example. One such check involves the notion of critical pairs (cf. [Alv87]):We say that a pair of elements $w_1 \leq w_2 \in W$ is critical if $\mathcal{L}(w_2) \subseteq \mathcal{L}(w_1)$ and $\mathcal{R}(w_2) \subseteq \mathcal{R}(w_1)$, where $\mathcal{L}$ and $\mathcal{R}$ denote the left and right descent set, respectively. Now if $y \leq w \in W$ are arbitrary elements then there always exists a critical pair (z, w) with $y \leq z \leq w$ and then $P_{y,w} = P_{z,w}$. Given two elements y and w, such a critical pair is found by the function `CriticalPair`. Whenever the polynomial corresponding to a critical pair is computed then this pair and the polynomial are stored in the field `.klpol` of the record of the underlying Coxeter group.

A good example to see how long the programs will take for computations in big Coxeter groups is the following:

```
gap> W:=CoxeterGroup("B",5);;
gap> LeftCells(W);;
```

which takes 10 seconds cpu time on 3Ghz computer. The computation of all Kazhdan-Lusztig polynomials for type F_4 takes a bit more than 1 minute. Computing the Bruhat order is a bottleneck for these computations; they can be speeded up by a factor of two if one does:

```
gap> ReadChv("contr/brbase");
gap> BaseBruhat(W);;
```

after which the computation of the Bruhat order will be speeded up by a large factor.

However, Alvis' computation of the Kazhdan–Lusztig polynomials of the Coxeter group of type H_4 in a computer algebra system like GAP3 would still take many hours. For such applications, it is probably more efficient to use a special purpose program like the one provided by F. DuCloux [DuC91].

The code for the Kazhdan-Lusztig bases `C`, `D` and their primed versions has been written by Andrew Mathas around 1994, who also contributed to the initial implementation and to the design of the programs dealing with Kazhdan-Lusztig bases. He also implemented some other bases, such as the Murphy basis which can be found in the contributions directory (see also his `Specht` package). The extension to the unequal parameters case has been written by F.Digne and J.Michel around 1999.

The other Kazhdan-Lusztig bases are computed in CHEVIE in terms of the C' basis.

CHEVIE is able to define automatically the bar and truncation operations on $\mathbb{Z}(\Gamma)$ when all parameters are powers of the same indeterminate q, with total order on Γ by the power of q, or when the parameters are monomials in some `Mvp`s, with the lexicographic order. The bar involution is evaluating a Laurent polynomial at the inverse of the variables, and truncation is keeping terms of smaller degree than that of ν. It is possible to use arbitrary groups Γ by doing the following steps: first, define the Hecke algebra `H`. Then, before defining any of the Kazhdan-Lusztig bases, write functions `H.Bar(p)`, `H.PositivePart(p)` and `H.NegativePart(p)` which perform the operations respectively $\sum_{\gamma\in\Gamma} a_\gamma\gamma \mapsto \sum_{\gamma\in\Gamma} a_\gamma\gamma^{-1}$, $\sum_{\gamma\in\Gamma} a_\gamma\gamma \mapsto \sum_{\gamma\geq 1} a_\gamma\gamma$ and $\sum_{\gamma\in\Gamma} a_\gamma\gamma \mapsto \sum_{\gamma\leq 1} a_\gamma\gamma$ on elements `p` of $\mathbb{Z}[\Gamma]$. It is then possible to define Kahzdan-Lusztig bases and the operations above will be used internally by CHEVIE to compute them.

93.1 KazhdanLusztigPolynomial

`KazhdanLusztigPolynomial(` W, y, w`)`

returns the coefficients of the Kazhdan-Lusztig polynomial $P_{y,w}(q)$ attached to the elements y and w of the Coxeter group W and to `Hecke(W,q)`. If one prefers to give as input just two Coxeter words, one can define a new function as follows (for example):

```
gap> klpol := function( W, x, y)
>   return KazhdanLusztigPolynomial(W, EltWord(W, x), EltWord(W, y));
>   end;
function ( W, x, y ) ... end
```

We use this function in the following example where we compute the polynomials $P_{1,w}$ for all elements w in the Coxeter group of type A_3.

```
gap> q := X( Rationals );; q.name := "q";;
gap> W := CoxeterGroup( "B", 3 );;
gap> el := CoxeterWords( W );
[ [  ], [ 3 ], [ 2 ], [ 1 ], [ 3, 2 ], [ 2, 1 ], [ 2, 3 ], [ 1, 3 ],
  [ 1, 2 ], [ 2, 1, 2 ], [ 3, 2, 1 ], [ 2, 3, 2 ], [ 2, 1, 3 ],
  [ 1, 2, 1 ], [ 1, 3, 2 ], [ 1, 2, 3 ], [ 3, 2, 1, 2 ],
  [ 2, 1, 2, 3 ], [ 2, 3, 2, 1 ], [ 2, 1, 3, 2 ], [ 1, 2, 1, 2 ],
  [ 1, 3, 2, 1 ], [ 1, 2, 1, 3 ], [ 1, 2, 3, 2 ], [ 3, 2, 1, 2, 3 ],
  [ 2, 1, 2, 3, 2 ], [ 2, 3, 2, 1, 2 ], [ 2, 1, 3, 2, 1 ],
  [ 1, 3, 2, 1, 2 ], [ 1, 2, 1, 2, 3 ], [ 1, 2, 1, 3, 2 ],
  [ 1, 2, 3, 2, 1 ], [ 2, 3, 2, 1, 2, 3 ], [ 2, 1, 2, 3, 2, 1 ],
  [ 2, 1, 3, 2, 1, 2 ], [ 1, 3, 2, 1, 2, 3 ], [ 1, 2, 1, 2, 3, 2 ],
  [ 1, 2, 1, 3, 2, 1 ], [ 1, 2, 3, 2, 1, 2 ], [ 2, 1, 2, 3, 2, 1, 2 ],
  [ 2, 1, 3, 2, 1, 2, 3 ], [ 1, 2, 3, 2, 1, 2, 3 ],
  [ 1, 2, 1, 2, 3, 2, 1 ], [ 1, 2, 1, 3, 2, 1, 2 ],
  [ 2, 1, 2, 3, 2, 1, 2, 3 ], [ 1, 2, 1, 2, 3, 2, 1, 2 ],
  [ 1, 2, 1, 3, 2, 1, 2, 3 ], [ 1, 2, 1, 2, 3, 2, 1, 2, 3 ] ]
gap> List( el, w -> Polynomial(Rationals,klpol( W, [], w )));
[ q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0, q^0,
  q^0, q^0, q^0, q^0, q^0, q^0, q + 1, q^0, q^0, q^0, q^0, q + 1,
  q^0, q^0, q + 1, q^0, q^0, q + 1, q + 1, q^0, q + 1, q^0, q + 1,
  q^0, q^2 + 1, q + 1, q^2 + q + 1, q + 1, q + 1, q^0, q^0, q^2 + 1,
  q^0, q + 1, q^0 ]
```

Kazhdan–Lusztig polynomials for critical pairs are stored in the record component `klpol` of W, which allows the function to work much faster after the first time it is called.

93.2 CriticalPair

`CriticalPair(` W, y, w `)`

Given elements y and w in the Coxeter group W the function `CriticalPair` returns the longest element in the double coset $W_{\mathcal{L}(w)}yW_{\mathcal{R}(w)}$; it is such that the Kazhdan–Lusztig polynomials $P_{z,w}$ and $P_{y,w}$ are equal.

```
gap> W := CoxeterGroup( "F", 4 );
```

```
CoxeterGroup("F",4)
gap> w := LongestCoxeterElement( W ) * W.generators[1];;
gap> CoxeterLength( W, w );
23
gap> y := EltWord( W, [ 1, 2, 3, 4 ] );;
gap> cr := CriticalPair( W, y, w );;
gap> CoxeterWord( W, cr);
[ 2, 3, 2, 1, 3, 4, 3, 2, 1, 3, 2, 3, 4, 3, 2, 3 ]
gap> KazhdanLusztigPolynomial( W, y, w);
[ 1, 0, 0, 1 ]
gap> KazhdanLusztigPolynomial( W, cr, w);
[ 1, 0, 0, 1 ]
```

93.3 KazhdanLusztigCoefficient

`KazhdanLusztigCoefficient(` W`,` y`,` w`,` k `)`

returns the coefficient of q^k in the Kazhdan-Lusztig polynomial $P_{y,w}$ attached to the elements y and w of the Coxeter group W and to `Hecke(W,q)`.

```
gap> W := CoxeterGroup( "B", 4 );;
gap> y := [ 1, 2, 3, 4, 3, 2, 1 ];;
gap> py := EltWord( W, y );
( 1,28)( 2,15)( 4,27)( 6,16)( 7,24)( 8,23)(11,20)(12,17)(14,30)(18,31)
(22,32)
gap> x := [ 1 ];;
gap> px := EltWord( W, x );
( 1,17)( 2, 8)( 6,11)(10,14)(18,24)(22,27)(26,30)
gap> Bruhat( W, px, py );
true
gap> List([0..3],i->KazhdanLusztigCoefficient( W, px, py, i ) );
[ 1, 2, 1, 0 ]
```

So the Kazhdan-Lusztig polynomial corresponding to x and y is $1 + 2q + q^2$.

93.4 KazhdanLusztigMue

`KazhdanLusztigMue(` W`,` y`,` w `)`

given elements y and w in the Coxeter group W, this function returns the coefficient of degree $(l(w) - l(y) - 1)/2$ of the Kazhdan-Lusztig polynomial $P_{y,w}$.

Of course, the result of this function could also be obtained by

`KazhdanLusztigCoefficient(W,y,w,(CoxeterLength(W,w)-CoxeterLength(W,y)-1)/2)`

but there are some speed-ups compared to this general function.

93.5 LeftCells

`LeftCells(` W `[,` i`])`

returns a list of records describing left cells of W for `Hecke(W,q)`. The program uses precomputed data(see [GH14]) for exceptional types and for type A, so is quite fast for these types (it takes 32 seconds to compute the 101796 left cells for type E_8). For other types, left cells are computed from first principles, thus computing many Kazhdan-Lusztig polynomials. It takes 10 seconds to compute the left cells of D_5, for example.

```
gap> W := CoxeterGroup( "G", 2 );;
gap> LeftCells(W);
[ LeftCell<G2: duflo= character=phi{1,0}>,
  LeftCell<G2: duflo=2 character=phi{2,1}+phi{1,3}'+phi{2,2}>,
  LeftCell<G2: duflo=1 character=phi{2,1}+phi{1,3}''+phi{2,2}>,
  LeftCell<G2: duflo=1,2 character=phi{1,6}> ]
```

Printing such a record displays the character afforded by the left cell and its Duflo involution; the Duflo involution r is printed as a subset `I` of `[1..W.N]` such that `r=LongestCoxeterElement(ReflectionSubg` see 85.9.

If a second argument i is given, the program returns only the left cells which are in the i-th two-sided cell, that is whose character is in the i-th family of W (see 98.15).

```
gap> LeftCells(W,1);
[ LeftCell<G2: duflo=2 character=phi{2,1}+phi{1,3}'+phi{2,2}>,
  LeftCell<G2: duflo=1 character=phi{2,1}+phi{1,3}''+phi{2,2}> ]
```

93.6 LeftCell

`LeftCell(` W, x`)`

returns a record describing the left cell of W for `Hecke(W,q)` containing element x.

```
gap> W := CoxeterGroup( "E", 8 );;
gap> LeftCell(W, Random(W));
LeftCell<E8: duflo=6,11,82,120 character=phi{2268,30}+phi{1296,33}>
```

93.7 Functions for LeftCells

`Size(` *cell*`)`

Returns the number of elements of the cell.

```
gap> W:=CoxeterGroup( "H", 3);;
gap> c := LeftCells( W );;
gap> List( c, Size);
[ 1, 6, 5, 8, 5, 6, 1, 5, 8, 5, 5, 6, 6, 5, 8, 5, 5, 8, 5, 6, 6, 5 ]
```

`Elements(` *cell*`)`

Returns the list of elements of the cell.

The operations `in` and `=` are defined for left cells.

`Representation(` *cell*, H `)`

returns a list of matrices giving the representation of `Hecke(W,v^2,v)` on the left cell c.

```
gap> v := X( Cyclotomics ) ;; v.name := "v";;
gap> H := Hecke(W, v^2, v );
Hecke(H3,v^2,v)
gap> Representation(c[3],H);
```

```
[ [ [ -v^0, 0*v^0, 0*v^0, 0*v^0, 0*v^0 ],
    [ 0*v^0, -v^0, 0*v^0, 0*v^0, v ],
    [ 0*v^0, 0*v^0, -v^0, v, v ],
    [ 0*v^0, 0*v^0, 0*v^0, v^2, 0*v^0 ],
    [ 0*v^0, 0*v^0, 0*v^0, 0*v^0, v^2 ] ],
  [ [ -v^0, v, 0*v^0, 0*v^0, 0*v^0 ],
    [ 0*v^0, v^2, 0*v^0, 0*v^0, 0*v^0 ],
    [ 0*v^0, 0*v^0, v^2, 0*v^0, 0*v^0 ],
    [ 0*v^0, 0*v^0, v, -v^0, 0*v^0 ],
    [ 0*v^0, v, v, 0*v^0, -v^0 ] ],
  [ [ v^2, 0*v^0, 0*v^0, 0*v^0, 0*v^0 ],
    [ v, -v^0, 0*v^0, 0*v^0, 0*v^0 ],
    [ 0*v^0, 0*v^0, -v^0, v, 0*v^0 ],
    [ 0*v^0, 0*v^0, 0*v^0, v^2, 0*v^0 ],
    [ 0*v^0, 0*v^0, 0*v^0, 0*v^0, -v^0 ] ] ]
```

`Character(c)`

Returns a list l such that the character of W afforded by the left cell c is `Sum(CharTable(W).irreducibles{l})`.

```
gap> Character(c[13]);
[ 6, 5 ]
```

See also `WGraph` below. When `Character(c)` has been computed, then `c.a` also has been bound which holds the common value of Lusztig's a-function (see 87.10) for

- The elements of `c`.
- The irreducible constituents of `Character(c)`.

93.8 W-Graphs

Let H be the 1-parameter Hecke algebra with parameter q associated to the Coxeter system (W, S). A W-graph encodes a representation of H of the kind which is constructed by Kazhdan-Lusztig theory. It consists of a basis V (the vertices of the graph) of a real vector space with a function $x \mapsto I(x)$ from V to the subsets of S and a function $\mu : V^2 \to \mathbb{R}$ (the nonzero values are thus labels for the edges of the graph). This defines the representation of H given in the basis V by the formulae

$$T_s(x) = \begin{cases} -x & \text{if } s \in I(x) \\ qx + \sum_{\{y\in V \mid s\in I(y)\}} \sqrt{q}\mu(y,x)y & \text{otherwise.} \end{cases}$$

There are two points to W-graphs. First, they describe nice, sparse, integral representations of H (and thus of W also). Second, they can be stored very compactly; for example, for the representation of dimension 7168 of the Hecke algebra of type E_8, a naive implementation would take more than a gigabyte. The corresponding W-graph takes 500KB.

W-graphs are represented in CHEVIE as a pair.

- The first element is a list describing C; its elements are either a set I(x), or an integer n specifying to repeat the previous element n more times.

- The second element is a list which specifies μ. We first describe the μ-list for symmetric W-graphs (when $\mu(x,y)=\mu(y,x)$). There is one element of the μ-list for each non-zero value m taken by μ, which consists of a pair whose first element is m and whose second element is a list of lists; if `l` is one of these lists each pair `[l[1],l[i]]` represents an edge `x=l[1],y=l[i]` such that $\mu(x,y)=\mu(y,x)=m$. For non-symmetric W-graphs, the first element of each pair in the μ-list is a pair `[m,n]` and each edge `[x,y]` obtained from the lists in the second element has to be interpreted as $\mu(x,y)=m$ and $\mu(y,x)=n$.

Here is an example of graph for a Coxeter group, and the corresponding representation. Here `v` is a variable representing the square root of `q`.

```
gap> W:=CoxeterGroup("H",3);;
gap> WGraph(W,3);
[ [ [ 2 ], [ 1, 2 ], [ 1, 3 ], [ 1, 3 ], [ 2, 3 ] ],
  [ [ -1, [ [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 5 ] ] ] ] ]
gap> WGraphToRepresentation(3,last,Mvp("v"));
[ [ [ v^2, 0, 0, 0, 0 ], [ 0, -1, 0, 0, 0 ], [ -v, 0, -1, 0, -v ],
      [ 0, 0, 0, -1, -v ], [ 0, 0, 0, 0, v^2 ] ],
  [ [ -1, 0, -v, 0, 0 ], [ 0, -1, 0, -v, 0 ], [ 0, 0, v^2, 0, 0 ],
      [ 0, 0, 0, v^2, 0 ], [ 0, 0, -v, -v, -1 ] ],
  [ [ v^2, 0, 0, 0, 0 ], [ 0, v^2, 0, 0, 0 ], [ -v, 0, -1, 0, 0 ],
      [ 0, -v, 0, -1, 0 ], [ 0, 0, 0, 0, -1 ] ] ]
```

93.9 WGraph

`WGraph(` *W* `,` *i* `)`

W should be a finite Coxeter group. Returns the W-graph for the i-th representation of the one-parameter Hecke algebra of W (or the i-th representation of W). For the moment this is only implemented for irreducible groups of exceptional type E, F, G, H.

`WGraph(` *c* `)`

c should be a left cell for the one-parameter Hecke algebra of a finite Coxeter group W. Returns the corresponding W-graph.

93.10 WGraphToRepresentation

`WGraphToRepresentation (`*r*`,` *graph*`,` *v*`)`

graph should be a W-graph for some finite Coxeter group W of semisimple rank r. The function returns the r matrices defining the representation defined by *graph* of the Hecke algebra for W with parameters -1 and v^2.

```
gap> W:=CoxeterGroup("H",3);;
gap> g:=WGraph(W,3);
[ [ [ 2 ], [ 1, 2 ], [ 1, 3 ], [ 1, 3 ], [ 2, 3 ] ],
  [ [ -1, [ [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 5 ] ] ] ] ]
gap> WGraphToRepresentation(3,g,Mvp("v"));
[ [ [ v^2, 0, 0, 0, 0 ], [ 0, -1, 0, 0, 0 ], [ -v, 0, -1, 0, -v ],
```

```
        [ 0, 0, 0, -1, -v ], [ 0, 0, 0, 0, v^2 ] ],
    [ [ -1, 0, -v, 0, 0 ], [ 0, -1, 0, -v, 0 ], [ 0, 0, v^2, 0, 0 ],
        [ 0, 0, 0, v^2, 0 ], [ 0, 0, -v, -v, -1 ] ],
    [ [ v^2, 0, 0, 0, 0 ], [ 0, v^2, 0, 0, 0 ], [ -v, 0, -1, 0, 0 ],
        [ 0, -v, 0, -1, 0 ], [ 0, 0, 0, 0, -1 ] ] ]
```

`WGraphToRepresentation(`*H*`, `*graph*`)`

H should be a one-parameter Hecke algebra for a finite Coxeter group. The function returns the matrices of the representation defined by *graph* of H.

```
gap> H:=Hecke(W,[[Mvp("v"),-Mvp("v")^-1]]);
Hecke(H3,[[v,-v^-1]])
gap> WGraphToRepresentation(H,g);
[ [ [ v, 0, 0, 0, 0 ], [ 0, -v^-1, 0, 0, 0 ], [ 1, 0, -v^-1, 0, 1 ],
      [ 0, 0, 0, -v^-1, 1 ], [ 0, 0, 0, 0, v ] ],
  [ [ -v^-1, 0, 1, 0, 0 ], [ 0, -v^-1, 0, 1, 0 ], [ 0, 0, v, 0, 0 ],
      [ 0, 0, 0, v, 0 ], [ 0, 0, 1, 1, -v^-1 ] ],
  [ [ v, 0, 0, 0, 0 ], [ 0, v, 0, 0, 0 ], [ 1, 0, -v^-1, 0, 0 ],
      [ 0, 1, 0, -v^-1, 0 ], [ 0, 0, 0, 0, -v^-1 ] ] ]
```

93.11 Hecke elements of the C basis

`Basis( `*H*`, "C")`

returns a function which gives the C-basis of the Iwahori-Hecke algebra H. The parameters of H should be powers of a single indeterminate or `Mvp`s (see the introduction). This basis is defined as follows (see e.g. [Lus85, (5.1)]). Let W be the underlying Coxeter group. For $x, y \in W$ let $P_{x,y}$ be the corresponding Kazhdan–Lusztig polynomial. If $\{T_w \mid w \in W\}$ denotes the usual T-basis, then $C_x := \sum_{y\leq x}(-1)^{l(x)-l(y)}P_{y,x}(q^{-1})q_x^{1/2}q_y^{-1}T_y$ for $x \in W$. For example, we have $C_s = q_s^{-1/2}T_s - q_s^{1/2}T_1$ for $s \in S$. Thus, the transformation matrix between the T-basis and the C-basis is lower unitriangular, with powers of v along the diagonal. In the one-parameter case (all q_s are equal to v^2) the multiplication rules for the C basis are given by:

$$C_s \cdot C_x = \begin{cases} -(v+v^{-1})C_x & \text{, if } sx < x \\ C_{sx} + \sum_y \mu(y,x)C_y & \text{, if } sx > x \end{cases}$$

where the sum is over all y such that $y < x$, $l(y) \not\equiv l(x) \bmod 2$ and $sy < y$. The coefficient $\mu(y,x)$ is the coefficient of degree $(l(x)-l(y)-1)/2$ in the Kazhdan–Lusztig polynomial $P_{x,y}$.

```
gap> W := CoxeterGroup( "B", 3 );;
gap> v := X( Rationals );; v.name := "v";;
gap> H := Hecke( W, v^2, v );
Hecke(B3,v^2,v)
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> C := Basis( H, "C" );
function ( arg ) ... end
gap> T( C( 1 ) );
```

```
-vT()+v^-1T(1)
gap> C( T( 1 ) );
v^2C()+vC(1)
```

We can also compute character values on elements in the C-basis as follows:

```
gap> ref := HeckeReflectionRepresentation( H );;
gap> c := CharRepresentationWords(ref, ChevieClassInfo(W).classtext);
[ 3*v^0, 2*v^2 - 1, v^8 - 2*v^4, -3*v^12, 2*v^2 - 1, v^4,
  v^4 - 2*v^2, -v^6, v^4 - v^2, 0*v^0 ]
gap> List(ChevieClassInfo(W).classtext, i->HeckeCharValues(C(i),c));
[ 3*v^0, -v - v^(-1), 0*v^0, 0*v^0, -v - v^(-1), 2*v^0, 0*v^0, 0*v^0,
  v^0, 0*v^0 ]
```

93.12 Hecke elements of the primed C basis

`Basis(` H`, "C'")`

returns a function which gives the C'-basis of the Iwahori-Hecke algebra H (see [Lus85, (5.1)]) The parameters of H should be powers of a single indeterminate or monomials in `Mvp`'s (see the introduction). This basis is defined by

$$C'_x := \sum_{y \leq x} P_{y,x} q_x^{-1/2} T_y \quad \text{for } x \in W.$$

We have $C'_x = (-1)^{l(x)}\mathrm{Alt}(C_x)$ for all $x \in W$ (see `AltInvolution` in section 91.6).

```
gap>  v := X( Rationals );; v.name := "v";;
gap>  H := Hecke( CoxeterGroup( "B", 2 ), [v ^4, v^2] );;
gap>  h := Basis( H, "C'" )( 1 );
# warning: C' basis: v^2 chosen as 2nd root of v^4
C'(1)
gap>  h2 := h * h;
(v^2+v^-2)C'(1)
gap>  Basis( H, "T" )( h2 );
(1+v^-4)T()+(1+v^-4)T(1)
gap> Basis(H,"C'")(last);
(v^2+v^-2)C'(1)
```

93.13 Hecke elements of the D basis

`Basis(` H`, "D")`

returns a function which gives the D-basis of the (one parameter generic) Iwahori-Hecke algebra H (see [Lus85, (5.1)]) of the finite Coxeter group W. This can be defined by

$$D_x := v^{-N} C'_{xw_0} T_{w_0} \text{ for every } x \in W,$$

where N denotes the number of positive roots in the root system of W and w_0 is the longest element of W. The D-basis is dual to the C-basis with respect to the non-degenerate form $H \times H \to \mathbb{Z}[v, v^{-1}]$, $(h_1, h_2) \mapsto \tau(h_1 \cdot h_2)$ where $\tau : H \to \mathbb{Z}[v, v^{-1}]$ is the linear form such that $\tau(T_1) = 1$ and $\tau(T_x) = 0$ for $x \neq 1$. We have $D_x = \beta(C'_{w_0x})$ for all $x \in W$ (see `BetaInvolution` in section 91.6).

```
gap> W := CoxeterGroup( "B", 2 );;
gap> v := X( Rationals );; v.name := "v";;
gap> H := Hecke( W, v^2, v );
Hecke(B2,v^2,v)
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> D := Basis( H, "D" );
function ( arg ) ... end
gap> D( T( 1 ) );
vD(1)-v^2D(1,2)-v^2D(2,1)+v^3D(1,2,1)+v^3D(2,1,2)-v^4D(1,2,1,2)
gap> BetaInvolution( D( 1 ) );
C'(2,1,2)
```

93.14 Hecke elements of the primed D basis

`Basis(` H`, "D'")`

returns a function which gives the D'-basis of the (one parameter generic) Iwahori-Hecke algebra H of the finite Coxeter group W (see [Lus85, (5.1)]). This can be defined by

$$D'_x := v^{-N} C_{xw_0} T_{w_0} \text{ for every } x \in W,$$

where N denotes the number of positive roots in the root system of W and w_0 is the longest element of W. The D'-basis is basis dual to the C'-basis with respect to the non-degenerate form $H \times H \to \mathbb{Z}[v, v^{-1}]$, $(h_1, h_2) \mapsto \tau(h_1 \cdot h_2)$ where $\tau : H \to \mathbb{Z}[v, v^{-1}]$ is the linear form such that $\tau(T_1) = 1$ and $\tau(T_x) = 0$ for $x \neq 1$. We have $D'_x = \mathrm{Alt}(D_x)$ for all $x \in W$ (see `AltInvolution` in section 91.6).

```
gap> W := CoxeterGroup( "B", 2 );;
gap> v := X( Rationals );; v.name := "v";;
gap> H := Hecke( W, v^2, v );
Hecke(B2,v^2,v)
gap> T := Basis( H, "T" );
function ( arg ) ... end
gap> Dp := Basis( H, "D'" );
function ( arg ) ... end
gap> AltInvolution( Dp( 1 ) );
D(1)
gap> Dp( 1 )^3;
(v+2v^-1-5v^-5-9v^-7-8v^-9-4v^-11-v^-13)D'()+(v^2+2+v^-2)D'(1)
```

93.15 Asymptotic algebra

`AsymptoticAlgebra(` W`,` i`)`

The asymptotic algebra A associated to the algebra $\mathcal{H}$ =`Hecke(W,q)` is an algebra with basis $\{t_x\}_{x\in W}$ and structure constants $t_x t_y = \sum_z \gamma_{x,y,z} t_z$ given by:let $h_{x,y,z}$ be the coefficient of

C_xC_y on C_z. Then $h_{x,y,z} = \gamma_{x,y,z^{-1}}q^{a(z)/2}$+lower terms, where $q^{a(z)/2}$ is the maximum over x, y of the degree of $h_{x,y,z}$.

The algebra A is the direct product of the subalgebras $A_{\mathcal{C}}$ generated by the elements $\{t_x\}_{x\in\mathcal{C}}$, where $\mathcal{C}$ runs over the two-sided cells of W. If $\mathcal{C}$ is the i-th two-sided cell of W, the command `AsymptoticAlgebra(W,i)` returns the algebra $A_{\mathcal{C}}$. Note that the function `a(z)` is constant over a two-sided cell, equal to the common value of the `a`-function attached to the characters of the two-sided cell (see `Character` for left cells).

```
gap> W:=CoxeterGroup("G",2);;
gap> A:=AsymptoticAlgebra(W,1);
Asymptotic algebra dim.10
gap> b:=A.basis;
[ t(2), t(12), t(212), t(1212), t(21212), t(1), t(21), t(121),
  t(2121), t(12121) ]
gap> List(b,x->b*x);
[ [ t(2), t(12), t(212), t(1212), t(21212), 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, t(12), t(2)+t(212), t(12)+t(1212), t(212)+t(21212),
      t(1212) ],
  [ t(212), t(12)+t(1212), t(2)+t(212)+t(21212), t(12)+t(1212),
      t(212), 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, t(1212), t(212)+t(21212), t(12)+t(1212),
      t(2)+t(212), t(12) ],
  [ t(21212), t(1212), t(212), t(12), t(2), 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, t(1), t(21), t(121), t(2121), t(12121) ],
  [ t(21), t(1)+t(121), t(21)+t(2121), t(121)+t(12121), t(2121), 0,
      0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, t(121), t(21)+t(2121), t(1)+t(121)+t(12121),
      t(21)+t(2121), t(121) ],
  [ t(2121), t(121)+t(12121), t(21)+t(2121), t(1)+t(121), t(21), 0,
      0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, t(12121), t(2121), t(121), t(21), t(1) ] ]
```

93.16 Lusztigaw

`Lusztigaw(` W, w`)`

For w an element of the Coxeter groups W, this function returns the coefficients on the irreducible characters of the virtual Character $\dashv_w$ defined in [Lus85, 5.10.2]. This character has the property that the corresponding almost character is integral and positive.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> Lusztigaw(W,Reflection(W,1));
[ 0, 0, 1, 0, 1, 1 ]
gap> last*List([1..NrConjugacyClasses(W)],i->AlmostCharacter(W,i));
[G2]=<phi{1,3}'>+<phi{2,1}>+<phi{2,2}>
```

93.17 LusztigAw

`LusztigAw(` W, w`)`

For w an element of the Coxeter groups W, this function returns the coefficients on the irreducible characters of the virtual Character $\mathcal{A}_w$ defined in [Lus85, 5.10.2]. This character has the property that the corresponding almost character is integral and positive.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> LusztigAw(W,Reflection(W,1));
[ 0, 0, 0, 1, 1, 1 ]
gap> last*List([1..NrConjugacyClasses(W)],i->AlmostCharacter(W,i));
[G2]=<phi{1,3}''>+<phi{2,1}>+<phi{2,2}>
```

Chapter 94

Parabolic modules for Iwahori-Hecke algebras

Let H be the Hecke algebra of the Coxeter group W with Coxeter generating set S, and let I be a subset of S. Let χ be a one-dimensional character of the parabolic subalgebra H_I of H. Then $H \otimes_{H_I} \chi$ (the induced representation of χ from H_I to H) is naturally a H-module, with a natural basis $MT_w = T_w \otimes 1$ indexed by the reduced-I elements of W (i.e., those elements w such that $l(ws) > l(w)$ for any $s \in I$).

The module action of an generator T_s of H which satisfies the quadratic relation $(T_s - p_s)(T_s - q_s) = 0$ is given in this basis by:

$$T_s \cdot MT_w = \begin{cases} \chi(T_{w^{-1}sw})MT_w, & \text{if } sw \text{ is not reduced-}I \text{ (then } w^{-1}sw \in I\text{).} \\ -p_s q_s MT_{sw} + (p_s + q_s)MT_w, & \text{if } sw < w \text{ is reduced-}I. \\ MT_{sw}, & \text{if } sw > w \text{ is reduced-}I. \end{cases}$$

Kazhdan-Lusztig bases of an Hecke module are also defined in the same circumstances when Kazhdan-Lusztig bases of the algebra can be defined, but only the case of the base `C'` for χ the sign character has been implemented for now.

94.1 Construction of Hecke module elements of the MT basis

`ModuleBasis(` H`, "MT"[,` I `[,`*chi*`]] )`

H should be an Iwahori-Hecke algebra of a Coxeter group W with Coxeter generating set S, I should be a subset of S (specified by a list of the names of the generators in I), and *chi* should be a one-dimensional character of the parabolic subalgebra of H determined by I, represented by the list of its values on $\{T_s\}_{s \in I}$ (if *chi* takes the same value on all generators of H_I it can be represented by a single value).

The result is a function which can be used to make elements of the `MT` basis of the Hecke module associated to I and *chi*.

If omitted, I is assumed to be the first `W.semiSimpleRank-1` generators of W (this makes sense for an affine Weyl group where they generate the corresponding linear Weyl group), and *chi* is taken to be equal to -1 (which specifies the sign character of H).

It is convenient to assign this function with a shorter name when computing with elements of the Hecke module. In what follows we assume that we have done the assignment:

```
gap> W:=CoxeterGroup("A",2);;Wa:=Affine(W);;
gap> q:=X(Rationals);;q.name:="q";;
gap> H:=Hecke(Wa,q);
Hecke(~A2,q)
gap> MT:=ModuleBasis(H,"MT");
function ( arg ) ... end
```

`MT(` *w* `)`

Here w is an element of the Coxeter group `Group(H)`. The basis element MT_w is returned if w is reduced-I, and otherwise an error is signaled.

`MT(` *elts*`,` *coeffs*`)`

In this form, `elts` is a list of elements of `Group(H)` and `coeffs` a list of coefficients which should be of the same length `k`. The element `Sum([1..k],i->coeffs[i]*MT(elts[i]))` is returned.

`MT(` *list* `)`

`MT(` *s1*`, ..,` *sn* `)`

In the above two forms, the GAP3 list *list* or the GAP3 list `[`*s1*`,..,`*sn*`]` represents the Coxeter word for an element w of `Group(H)`. The basis element MT_w is returned if w is reduced-I, and otherwise an error is signaled.

The way elements of the Hecke module are printed depends on `CHEVIE.PrintHecke`. If `CHEVIE.PrintHecke=rec(GAP =true)`, they are printed in a way which can be input back in GAP3. When you load CHEVIE, the `PrintHecke` is initially set to `rec()`.

94.2 Construction of Hecke module elements of the primed MC basis

`ModuleBasis(` *H*`, "MC'"[,` *I*`] )`

H should be an Iwahori-Hecke algebra with all parameters a power of the same indeterminate of a Coxeter group W with Coxeter generating set S and I should be a subset of S (specified by a list of the names of the generators in I). The character *chi* does not have to be specified since in this case only *chi*`=-1` has been implemented.

If omitted, I is assumed to be the first `W.semiSimpleRank-1` generators of W (this makes sense for an affine Weyl group where they generate the corresponding linear Weyl group).

The result is a function which can be used to make elements of the `MC'` basis of the Hecke module associated to I and the sign character. In this particular case, the MC' basis can be defined for an reduced-I element w in terms of the MT basis by $MC'_w = C'_w MT_1$.

```
gap> H:=Hecke(Wa,q^2);
Hecke(~A2,q^2)
gap> MC:=ModuleBasis(H,"MC'");
# warning      MC'basis      q chosen as 2nd root of q^2
function ( arg ) ... end
```

94.3 Operations for Hecke module elements

`+`, `-`

one can add or subtract two Hecke module elements.

Basis(*x*)

this call will convert Hecke module element x to basis *Basis*. With the same initializations as in the previous sections, we have:

```
gap> MT:=ModuleBasis(H,"MT");;
gap> MC(MT(1,2,3));
-MC'()+qMC'(3)-q^2MC'(1,3)-q^2MC'(2,3)+q^3MC'(1,2,3)
```

`*`

one can multiply on the left an Hecke module element by an element of the corresponding Hecke algebra. With the same initializations as in the previous sections, we have:

```
gap> H:=Hecke(Wa,q);
Hecke(~A2,q)
gap> MT:=ModuleBasis(H,"MT");;
gap> T:=Basis(H,"T");
function ( arg ) ... end
gap> T(1)*MT(1,2,3);
qMT(2,3)+(q-1)MT(1,2,3)
```

94.4 CreateHeckeModuleBasis

`CreateHeckeModuleBasis(`*basis*`,` *ops*`,` *algebraops*`)`

This function is completely parallel to the function `CreateHeckeBasis`. See the description of this last function. The only difference is that it is not *ops*`.T` which is required to be bound, but *ops*`.MT` which should contain a function which takes an element in the basis *basis* and converts it to the `MT` basis.

Chapter 95

Reflection cosets

Let $W \subset \mathrm{G}L(V)$ be a complex reflection group on the vector space V. Let ϕ be an element of $\mathrm{G}L(V)$ which normalizes W. Then the coset $W\phi$ is called a reflection coset.

A reference for these cosets is [BMM99]; the main motivation is that in the case where W is a rational reflection group (a Weyl group) such cosets, that we will call **Weyl cosets**, model rational structures on finite reductive groups. Finally, when W is a so-called **Spetsial** group, they are the basic object for the construction of a **Spetses**, which is an object attached to a complex reflection group from which one can derive combinatorially some attributes shared with finite reductive groups, like unipotent degrees, etc....

We say that a reflection coset is irreducible if W is irreducible. A general coset is a direct product of **descents of scalars**, which is the case where ϕ is transitive on the irreducible components of W. The irreducible cosets have been classified in [BMM99]: up to multiplication of ϕ by a scalar, there is usually only one or two possible cosets for a given irreducible group.

In CHEVIE we deal only with **finite order** cosets, that is, we assume there is a (minimal) integer δ such that $(W\phi)^\delta = W\phi$. Then the group generated by W and ϕ is finite, of order $\delta|W|$.

A subset C of a $W\phi$ is called a **conjugacy class** if one of the following equivalent conditions is fulfilled:

- C is the orbit of an element in $W\phi$ under the conjugation action of W.
- C is a conjugacy class of $\langle W, \phi\rangle$ contained in $W\phi$.
- The set $\{w \in W \mid \; w\phi \in C\}$ is a ϕ-conjugacy class of W (two elements $v, w \in W$ are called ϕ-conjugate, if and only if there exists $x \in W$ with $v = xw\phi(x^{-1})$).

An irreducible character of $\langle W, \phi\rangle$ has some non-zero values on $W\phi$ if and only if its restriction to W is irreducible. Further, two characters χ_1 and χ_2 which have same irreducible restriction to W differ by a character of the cyclic group $\langle\phi\rangle$ (which identifies to the quotient $\langle W, \phi\rangle/W$). A set containing one extension to $\langle W, \phi\rangle$ of each ϕ-invariant character of W is called a **set of irreducible characters of** $W\phi$. Two such characters are orthogonal for the scalar product on the class functions on $W\phi$ given by

$$\langle\chi, \psi\rangle := \frac{1}{|\,W\,|}\sum_{w\in W} \chi(w\phi)\overline{\psi(w\phi)}.$$

For rational groups (Weyl groups), Lusztig has defined a canonical choice of a set of irreducible characters for $W\phi$ (called the **preferred extensions**), but for more general reflection cosets we have made some rather arbitrary choices, which however have the property that their values lie in the smallest possible field.

The **character table** of $W\phi$ is the table of values of a set of irreducible characters on the conjugacy classes.

A **subcoset** $Lw\phi$ of $W\phi$ is given by a reflection subgroup L of W and an element w of W such that $w\phi$ normalizes L.

We then have a natural notion of **restriction** of class functions on $W\phi$ to class functions on $Lw\phi$ as well as of **induction** in the other direction. These maps are adjoint with respect to the scalar product defined above (see [BMM99]).

In CHEVIE the most general construction of a reflection coset is by starting from a reflection datum, and giving in addition the matrix `phiMat` of the map $\phi : V \to V$ (see the command `ReflectionCoset`). However, at present, general cosets are only implemented for groups represented as permutation groups on a set of roots, and it is required that the automorphism given preserves this set up to a scalar (it is allowed that these scalars depend on the pair of an irreducible component and its image). If it also allowed to specify ϕ by the permutation it induces on the roots; in this case it is assumed that ϕ acts trivially on the orthogonal of the roots, but the roots could be those of a parent group, generating a larger space. Thus in any case we have a permutation representation of $\langle W, \phi\rangle$ and we consider the coset to be a set of permutations.

Reflection cosets are implemented in CHEVIE by a record which points to a reflection group record and has additional fields holding `phiMat` and the corresponding permutation `phi`. In the general case, on each component of W which is a descent of scalars, `phiMat` will permute the components and differ by a scalar on each component from an automorphism which preserves the roots. In this case, we have a permutation `phi` and a `scalar` which is stored for that component.

The most common situation where cosets with non-trivial `phi` arise is as sub-cosets of reflection groups. Here is an “exotic”example, see the next chapter for more classical examples involving Coxeter groups.

```
gap> W:=ComplexReflectionGroup(14);
ComplexReflectionGroup(14)
gap> PrintDiagram(W);
G14 1--8--2(3)
gap> R:=ReflectionSubgroup(W,[2,4]);
ReflectionSubgroup(ComplexReflectionGroup(14), [ 2, 4 ])
gap> PrintDiagram(R);
G5(ER(6)) 2(3)==4(3)
gap> Rphi:=ReflectionCoset(R,W.1);
2G5(ER(6))<2,4>
gap> PrintDiagram(Rphi);
phi acts as (2,4) on the component below
G5(ER(6)) 2(3)==4(3)
gap> ReflectionDegrees(Rphi);
[ [ 6, 1 ], [ 12, -1 ] ]
```

The last line shows for each reflection degree the corresponding **factor** of the coset, which is the scalar by which ϕ acts on the corresponding fundamental reflection invariant. The factors characterize the coset.

The variable `CHEVIE.PrintSpets` determines if a coset is printed in an abbreviated form which describes its type, as above (`G5` twisted by 2, with a Cartan matrix which differs from the standard one by a factor of $\sqrt{6}$), or in a form which could be input back in GAP3. The above example was for the default value `CHEVIE.PrintSpets=rec()`. With the same data we have:

```
gap> CHEVIE.PrintSpets:=rec(GAP:=true);;
gap> Rphi;
Spets(ReflectionSubgroup(ComplexReflectionGroup(14), [ 2, 4 ]), (1,3)(\
2,4)(5,9)(6,10)(7,11)(8,12)(13,21)(14,22)(15,23)(16,24)(17,25)(18,26)(\
19,27)(20,28)(29,41)(30,42)(31,43)(32,44)(33,45)(34,46)(35,47)(36,48)(\
37,49)(38,50)(39,51)(40,52)(53,71)(54,72)(55,73)(56,74)(57,75)(58,76)(\
59,77)(60,78)(62,79)(64,80)(65,81)(66,82)(67,69)(68,70)(83,100)(84,101\
)(85,102)(87,103)(89,99)(90,97)(91,98)(92,96)(93,104)(94,95)(105,113)(\
106,114)(109,111)(110,112)(115,118)(116,117)(119,120))
gap> CHEVIE.PrintSpets:=rec();;
```

95.1 ReflectionCoset

`ReflectionCoset(` *W* `[,` *phiMat* `] )`

`ReflectionCoset(` *W* `[,` *phiPerm*`] )`

This function returns a reflection coset as a GAP3 object. The argument W must be a reflection group (created by `ComplexReflectionGroup`, `CoxeterGroup`, `PermRootGroup` or `ReflectionSubgroup`). In the first form the argument *phiMat* must be an invertible matrix with `Rank(`W`)` rows, which normalizes the parent of W (if any) as well as W. In the second form *phiPerm* is a permutation which describes the images of the roots under *phi* (only the image of the roots corresponding to the generating reflections need be given, since they already determine a unique *phiMat*). This second form is only allowed if the semisimple rank of W equals the rank (i.e., the roots are a basis of V). If there is no second argument the default for *phiMat* is the identity matrix, so the result is the trivial coset equal to W itself.

`ReflectionCoset` returns a record from which we document the following components:

`isDomain`, `isFinite`
: true

`group`
: the group W

`phiMat`
: the matrix acting on V which represents ϕ.

`phi`
: the permutation on the roots of W induced by `phiMat`.

```
gap> W := CoxeterGroup("A",3);;
gap> Wphi := ReflectionCoset( W, (1,3));
```

```
2A3
gap> m:=MatXPerm(W,(1,3));
[ [ 0, 0, 1 ], [ 0, 1, 0 ], [ 1, 0, 0 ] ]
gap> ReflectionCoset( W,m);
2A3
```

95.2 Spets

`Spets` is a synonym for `ReflectionCoset`. See 95.1.

95.3 ReflectionSubCoset

`ReflectionSubCoset(` *WF*`,` *r*`, [`*w*`] )`

Returns the reflection subcoset of the reflection coset *WF* generated by the reflections specified by *r*. *r* is a list of indices specifying a subset of the roots of *W* where *W* is the reflection group `Group(`*WF*`)`. If specified, *w* must be an element of *W* such that `w*WF.phi` normalizes up to scalars the subroot system generated by *r*. If absent, the default value for *w* is `()`. If the subroot system is not normalized then `false` is returned, with a warning message if `InfoChevie=Print`.

```
gap> W:=ComplexReflectionGroup(14);
ComplexReflectionGroup(14)
gap> Wphi:=ReflectionCoset(W);
G14
gap> ReflectionSubCoset(Wphi,[2,4],W.1);
2G5(ER(6))<2,4>
gap> WF:=ReflectionCoset(CoxeterGroup("A",4),(1,4)(2,3));
2A4
gap> ReflectionSubCoset(WF,[2,3]);
2A2<2,3>.(q-1)(q+1)
gap> ReflectionSubCoset(WF,[1,2]);
# I permutation for F0 must normalize set of roots.
false
```

95.4 SubSpets

`SubSpets` is a synonym for `ReflectionSubCoset`. See 95.3.

95.5 Functions for Reflection cosets

`Group(` *WF* `)`
: returns the reflection group of which *WF* is a coset.

Quite a few functions defined for domains, permutation groups or reflection groups have been implemented to work with reflection Cosets.

`Size`, `Rank`, `SemisimpleRank`
: these functions use the corresponding functions for `Group(` *WF* `)`. `Elements`, `Random`, `Representative`, `in`

these functions use the corresponding functions for `Group(` *WF* `)` and multiply the result by `WF.phi`.

`ConjugacyClasses(` *WF* `)`
returns the conjugacy classes of the coset *WF* (see the introduction of this Chapter). Let W be `Group(`*WF*`)`. Then the classes are defined to be the W-orbits on $W\phi$, where W acts by conjugation (they coincide with the $W\phi$-orbits, $W\phi$ acting by the conjugation); by the translation $w \mapsto w\phi^{-1}$ they are sent to the ϕ-conjugacy classes of W.

`PositionClass(` *WF* `,` *x* `)`
for any element x in *WF* this returns the number `i` such that x is an element of `ConjugacyClasses(`*WF*`)[i]` (to work fast, the classification of reflection groups is used).

`FusionConjugacyClasses(` *WF1* `,` *WF* `)`
works in the same way as for groups. See the section `ReflectionSubCoset`.

`Print(` *WF* `)`
if *WF*`.name` is bound then this is printed, else this function prints the coset in a form which can be input back into GAP3.

`InductionTable(` *HF* `,` *WF* `)`
works in the same way as for groups. It gives the induction table from the Reflection subcoset *HF* to the Reflection coset *WF*. If $Hw\phi$ is a Reflection subcoset of $W\phi$, restriction of characters is defined as restriction of functions from $W\phi$ to $Hw\phi$, and induction as the adjoint map for the natural scalar product $\langle f, g\rangle = \frac{1}{|W|}\sum_{v\in W} f(v\phi)\overline{g}(v\phi)$.

```
gap> W := CoxeterGroup( "A", 4 );;
gap> Wphi := ReflectionCoset( W, (1,4)(2,3) );
2A4
gap> Display(InductionTable(ReflectionSubCoset(Wphi,[2,3 ]),Wphi));
Induction from 2A2<2,3>.(q-1)(q+1) to 2A4
      |111 21 3
________________
11111 |  1  . .
2111  |  .  1 .
221   |  1  . .
311   |  1  . 1
32    |  .  . 1
41    |  .  1 .
5     |  .  . 1
```

`InductionTable` and `FusionConjugacyClasses` work only between cosets. If the parent coset is the trivial coset it should still be given as a coset and not as a group:

```
gap> Wphi:=ReflectionCoset(W);
A4
gap> L:=ReflectionSubCoset(Wphi,[2,3],LongestCoxeterElement(W));
A2<2,3>.(q-1)(q+1)
gap> InductionTable(L,W);
Error, A2<2,3>.(q-1)(q+1) is a coset but CoxeterGroup("A",4) is not in
```

```
S.operations.FusionConjugacyClasses( S, R ) called from
FusionConjugacyClasses( u, g ) called from
InductionTable( L, W ) called from
main loop
brk>
gap> InductionTable(L,Wphi);
InductionTable(A2<2,3>.(q-1)(q+1), A4)
```

`ReflectionName( WF )`

returns a string which describes the isomorphism type of the group $W \rtimes \langle F\rangle$, associated to WF, as described in the introduction of this Chapter. An orbit of $\phi = WF$.`phi` on the components is put in brackets if of length k greater than 1, and is preceded by the order of phi^k on it, if this is not 1. For example `"2(A2xA2)"` denotes 2 components of type A_2 permuted by ϕ, and such that phi^2 induces the non-trivial diagram automorphism on any of them, while `3D4` denotes an orbit of length 1 on which *phi* is of order 3.

```
gap> W:=ReflectionCoset(CoxeterGroup("A",2,"G",2,"A",2),(1,5,2,6));
2(A2xA2)<1,2,5,6>xG2<3,4>
gap> ReflectionName( W );
"2(A2xA2)<1,2,5,6>xG2<3,4>"
```

`PrintDiagram( WF )`

this is a purely descriptive routine (as was already the case for finite Reflection groups themselves). It prints the Dynkin diagram of `ReflectionGroup(`WF`)` together with the information how WF`.phi` acts on it. Going from the above example:

```
gap> PrintDiagram( W );
phi permutes the next 2 components
phi^2 acts as (1,2) on the component below
A2 1 - 2
A2 5 - 6
G2 3 >>> 4
```

`ChevieClassInfo( WF )`, see the explicit description in 95.10.

`CharParams( WF )`

This returns appropriate labels for the characters of the ReflectionCoset. `CharNames` has also a special version for cosets.

`GenericOrder( WF, q )`

Returns the generic order of the associated algebraic group (for a Weyl coset) or Spetses, using the generalized reflection degrees. We also have `TorusOrder(WF,i,q)` which is the same as `GenericOrder(SubSpets(WF,[],Representative(ConjugacyClasses(WF)[i])))`.

Note that some functions for elements of a Reflection group work naturally for elements of a Reflection coset: `EltWord`, `ReflectionLength`, `ReducedInRightCoset`, etc...

95.6 ChevieCharInfo for reflection cosets

`ChevieCharInfo( WF )`

`ChevieCharInfo` gives for a reflection coset WF a record similar to what it gives for the corresponding group W, excepted that some fields which do not make sense are omitted, and that two fields record information allowing to relate characters of the coset to that of the group:

`charRestriction`
: records for each character of *WF* the index of the character of *W* of which it is an extension.

`nrGroupClasses`
: records `NrConjugacyClasses(Group(WF))`.

```
gap> ChevieCharInfo(RootDatum("3D4"));
rec(
  extRefl := [ 1, 5, 4, 6, 2 ],
  charparams := [ [ [ [  ], [ 4 ] ] ], [ [ [  ], [ 1, 1, 1, 1 ] ] ],
      [ [ [  ], [ 2, 2 ] ] ], [ [ [ 1, 1 ], [ 2 ] ] ],
      [ [ [ 1 ], [ 3 ] ] ], [ [ [ 1 ], [ 1, 1, 1 ] ] ],
      [ [ [ 1 ], [ 2, 1 ] ] ] ],
  charRestrictions := [ 13, 4, 10, 5, 11, 3, 6 ],
  nrGroupClasses := 13,
  b := [ 0, 12, 4, 4, 1, 7, 3 ],
  B := [ 0, 12, 8, 8, 5, 11, 9 ],
  charnames := [ ".4", ".1111", ".22", "11.2", "1.3", "1.111", "1.21"
     ],
  positionId := 1,
  positionDet := 2,
  a := [ 0, 12, 3, 3, 1, 7, 3 ],
  A := [ 0, 12, 9, 9, 5, 11, 9 ] )
```

95.7 ReflectionType for reflection cosets

`ReflectionType(` *WF* `)`

returns the type of the Reflection coset *WF*. This consists of a list of records, one for each orbit of *WF*`.phi` on the irreducible components of the Dynkin diagram of `Group(`*WF*`)`, which have two fields:

`orbit`
: is a list of types of the irreducible components in the orbit. These types are the same as returned by the function `ReflectionType` for an irreducible untwisted reflection group. The components are ordered according to the action of *WF*`.phi`, so *WF*`.phi` maps the generating permutations with indices in the first type to indices in the second type in the same order as stored in the type, etc ...

`phi`
: if k is the number of irreducible components in the orbit, this is the permutation which describes the action of *WF*`.phi`^{k} on the simple roots of the first irreducible component in the orbit.

```
gap> W:=ReflectionCoset(CoxeterGroup("A",2,"A",2), (1,3,2,4));
2(A2xA2)
```

```
gap> ReflectionType( W );
[ rec(orbit := [ rec(rank    := 2,
      series  := "A",
      indices := [ 1, 2 ]), rec(rank    := 2,
      series  := "A",
      indices := [ 3, 4 ]) ],
      twist := (1,2)) ]
```

95.8 ReflectionDegrees for reflection cosets

`ReflectionDegrees(` *WF* `)`

Let W be the Reflection group corresponding to the Reflection coset WF, and let V be the vector space of dimension `W.rank` on which W acts as a reflection group. Let $f_1, \ldots, f_n$ be the basic invariants of W on the symmetric algebra SV of V; they can be chosen so they are eigenvectors of the matrix `WF.phiMat`. The corresponding eigenvalues are called the **factors** of ϕ acting on V; they characterize the coset — they are equal to 1 for the trivial coset. The **generalized degrees** of WF are the pairs formed of the reflection degrees and the corresponding factor.

```
gap> W := CoxeterGroup( "E", 6 );; WF := ReflectionCoset( W );
E6
gap> phi := EltWord( W,[6,5,4,2,3,1,4,3,5,4,2,6,5,4,3,1]);;
gap> HF := ReflectionSubCoset( WF, [ 2..5 ], phi );;
gap> PrintDiagram( HF );
phi acts as (2,3,5) on the component below
D4 2
    \
     4 - 5
    /
   3
gap> ReflectionDegrees( HF );
[ [ 1, E(3) ], [ 1, E(3)^2 ], [ 2, 1 ], [ 4, E(3) ], [ 6, 1 ],
  [ 4, E(3)^2 ] ]
```

95.9 Twistings

`Twistings(` *W*`,` *L* `)`

W should be a Reflection group record or a Reflection coset record, and L should be a reflection subgroup of W (or of `Group(`W`)` for a coset), or a sublist of the generating reflections of W (resp. `Group(W)`), in which case the call is the same as `Twistings(`W`,ReflectionSubgroup(`W`,`L`))` (resp. `Twistings(`W`,ReflectionSubgroup(Group(`W`),`L`)))`.

The function returns a list of representatives, up to W-conjugacy, of reflection sub-cosets of W whose reflection group is L.

```
gap> W:=ComplexReflectionGroup(3,3,4);
ComplexReflectionGroup(3,3,4)
gap> Twistings(W,[1..3]);
[ G333.(q-1), 3'G333<1,2,3,76>.(q-E3^2), 3G333<1,2,3,76>.(q-E3) ]
```

95.10 ChevieClassInfo for Reflection cosets

`ChevieClassInfo(` *WF* `)`

returns information about the conjugacy classes of the Reflection coset *WF*. The result is a record with three components: `classtext` contains a list of reduced words for the representatives in `ConjugacyClasses(`*WF*`)`, `classnames` contains corresponding names for the classes, and `classparams` gives corresponding parameters for the classes.

```
gap> W:=ReflectionCoset(ComplexReflectionGroup(14));
G14
gap> Rphi:=ReflectionSubCoset(W,[2,4],Group(W).1);
2G5(ER(6))<2,4>
gap> ChevieClassInfo(Rphi);
rec(
  classtext :=
   [ [  ], [ 2, 4, 4, 2, 4, 4, 2 ], [ 2, 4, 4, 2, 4, 4, 2, 2 ],
      [ 2 ], [ 2, 2, 4, 2, 2 ], [ 2, 2, 4, 4, 2, 2 ], [ 2, 4 ],
      [ 2, 4, 2 ], [ 4, 2, 4, 2 ] ],
  classes := [ 12, 6, 6, 6, 12, 6, 6, 6, 12 ],
  orders := [ 2, 24, 24, 24, 6, 8, 24, 8, 6 ],
  classnames := [ "", "1221221", "12212211", "1", "11211", "112211",
      "12", "121", "2121" ] )
```

95.11 CharTable for Reflection cosets

`CharTable(` *WF* `)`

This function returns the character table of the Reflection coset *WF* (see also the introduction of this Chapter). We call "characters"of the Reflection coset WF with corresponding Reflection group W the restriction to $W\phi$ of a set containing one extension of each ϕ-invariant character of W to the semidirect product of W with the cyclic group generated by ϕ. The choice of extension is always the same for a given coset, but rather arbitrary in general; for Weyl cosets it is the "preferred extension"of Lusztig.

The returned record contains almost all components present in the character table of a Reflection group. But if ϕ is not trivial then there are no components `powermap` (since powers of elements in the coset need not be in the coset) and `orders` (if you really need them, use `MatXPerm` to determine the order of elements in the coset).

```
gap> W := ReflectionCoset( CoxeterGroup( "D", 4 ), (1,2,4) );
3D4
gap> Display( CharTable( W ) );
3D4

        2  2   2     2      2  2      3      3
        3  1   1     1      .  .      1      1

          C3 ~A2 C3+A1 ~A2+A1 F4 ~A2+A2 F4(a1)

.4         1   1     1      1  1      1      1
```

```
.1111    -1   1     1     -1  1      1       1
.22       .   2     2      . -1     -1      -1
11.2      .   .     .      . -1      3       3
1.3       1   1    -1     -1  .     -2       2
1.111    -1   1    -1      1  .     -2       2
1.21      .   2    -2      .  .      2      -2
```

Chapter 96

Coxeter cosets

Let R be a root system in the real vector space V as in Chapter 85. We say that $F_0 \in \mathrm{GL}(V)$ is an **automorphism of** R if it permutes R and is of finite order (finite order is automatic if R generates V). It follows by [Bou68, chap. VI, §1.1, lemme 1] that the dual $F_0^* \in \mathrm{GL}(V^\vee)$ permutes the coroots $R^\vee \subset V^\vee$; thus F_0 normalizes the reflection group W associated to R, that is $w \mapsto F_0 w F_0^{-1}$ is an automorphism of W. Thus (see 95) we get a reflection coset WF_0, called here a **Coxeter coset**.

The motivation for introducing Coxeter cosets comes from automorphisms of algebraic reductive groups, in particular non-split reductive groups over finite fields. Let us, as in 86 fix a connected reductive algebraic group $\mathbf{G}$. We assume $\mathbf{G}$ is a group over an algebraic closure $\overline{\mathcal{F}}_q$ of a finite field $\mathcal{F}_q$, defined over $\mathcal{F}_q$, which corresponds to be given a Frobenius endomorphism F so that the finite group of rational points $\mathbf{G}(\mathcal{F}_q)$ identifies to the subgroup $\mathbf{G}^F$ of fixed points under F.

Let $\mathbf{T}$ be a maximal torus of $\mathbf{G}$, and Φ (resp. $\Phi^\vee$) be the roots (resp. coroots) of G with respect to $\mathbf{T}$ in the character group $X(\mathbf{T})$ (resp. the group of one-parameter subgroups $Y(\mathbf{T})$). As explained in 86 then $\mathbf{G}$ is determined up to isomorphism by $(X(\mathbf{T}), \Phi, Y(\mathbf{T}), \Phi^\vee)$ and in CHEVIE this corresponds to give a rational reflection group $W = N_{\mathbf{G}}(\mathbf{T})/\mathbf{T}$ acting on the vector space $V = \mathbb{Q} \otimes X(\mathbf{T})$ together with a root system.

If $\mathbf{T}$ is F-stable the Frobenius endomorphism F acts also naturally on $X(T)$ and defines thus an endomorphism of V, which is of the form qF_0, where $F_0 \in \mathrm{GL}(V)$ is of finite order and normalizes W. We get thus a Coxeter coset $WF_0 \subset \mathrm{GL}(V)$. The data $(X(\mathbf{T}), \Phi, Y(\mathbf{T}), \Phi^\vee, F_0)$, and the integer q completely determine up to isomorphism the associated **reductive finite group $\mathbf{G}^F$**. Thus these data is a way of representing in CHEVIE the essential information which determines a finite reductive group. Indeed, all properties of Chevalley groups can be computed from that datum: symbols representing characters, conjugacy classes, and finally the whole character table of $\mathbf{G}^F$.

It turns out that an interesting part of the objects attached to this datum depends only on (V, W, F_0): the order of the maximal tori, the "fake degrees", the order of $\mathbf{G}^F$, symbols representing unipotent characters, Deligne-Lusztig induction in terms of "almost characters", the Fourier matrix relating characters and almost characters, etc... (see, e.g., [BMM93]). It is thus possible to extend their construction to non-crystallographic groups (or even to more general complex reflection groups, see 95.2); this is why we did not include a root system in the definition of a reflection coset. However, unipotent conjugacy classes for instance depend on the root system thus do not exist in general.

We assume now that $\mathbf{T}$ is contained in an F-stable Borel subgroup of $\mathbf{G}$. This defines an order on the roots, and there is a unique element $\phi \in WF_0$, the **reduced element** of the coset, which preserves the set of positive roots. It thus defines a **diagram automorphism**, that is an automorphism of the Coxeter system (W, S). This element is stored in the component `.phi` of the coset record. It may be defined without mentioning the roots, as follows:$(W, F_0(S))$ is another Coxeter system, thus conjugate to S by a unique element of W, thus there is a unique element $\phi \in WF_0$ which stabilizes S (a proof follows from [Bou68, Theoreme 1, chap. V, §3]). We consider thus cosets of the form $W\phi$ where ϕ stabilizes S. The coset $W\phi$ is completely defined by the permutation `.phi` when $\mathbf{G}$ is semi-simple — equivalently when Φ generates V; in this case we just need to specify `phi` to define the coset.

There is a slight generalisation of the above setup, covering in particular the case of the Ree and Suzuki groups. We consider $\mathbf{G}^F$ where F not a Frobenius endomorphism, but an isogeny such that some power F^n is a Frobenius endomorphism. Then F still defines an endomorphism of V which normalizes W; we define a real number q such that F^n is attached to an $\mathcal{F}_{q^n}$-structure. Then we still have $F = qF_0$ where F_0 is of finite order but q is no more an integer. Thus $F_0 \in \mathrm{GL}(V \otimes \mathbb{R})$ but $F_0 \notin \mathrm{GL}(V)$. For instance, for the Ree and Suzuki groups, F_0 is an automorphism of order 2 of W, which is of type G_2, B_2 or F_4, and $q = \sqrt{2}$ for B_2 and F_4 and $q = \sqrt{3}$ for G_2 To get this, we need to start from root systems for G_2, B_2 or F_4 where all the roots have the same length. This kind of root system is **not** crystallographic. Such more general root systems also exist for all finite Coxeter groups such as the dihedral groups and H_3 and H_4. We will call here **Weyl cosets** the cosets corresponding to rational forms of algebraic groups, which include thus some non-rational roots systems for B_2, G_2 and F_4.

Conjugacy classes and irreducible characters of Coxeter cosets are defined as for general reflection cosets. For irreducible characters of Weyl cosets, in CHEVIE we choose (following Lusztig) for each ϕ-stable character of W a particular extension to a character of $W \rtimes \langle\phi\rangle$, which we will call the **preferred extension**. The **character table** of the coset $W\phi$ is the table of the restrictions to $W\phi$ of the preferred extensions (See also the section below `CharTable for Coxeter cosets`). The question of finding the conjugacy classes and character table of a Coxeter coset can be reduced to the case of irreducible root systems R.

- The automorphism ϕ permutes the irreducible components of W, and $W\phi$ is a direct product of cosets where ϕ permutes cyclically the irreducible components of W. The preferred extension is defined to be the direct product of the preferred extension in each of these situations.

- Assume now that $W\phi$ is a descent of scalars, that is the decomposition in irreducible components $W = W_1 \times \cdots \times W_k$ is cyclically permuted by ϕ. Then there are natural bijections from the ϕ-conjugacy classes of W to the ϕ^k-conjugacy classes of W_1 as well as from the ϕ-stable characters of W to the ϕ^k-stable characters of W_1, which reduce the definition of preferred extensions on $W\phi$ to the definition for $W_1\phi^k$.

- Assume now that W is the Coxeter group of an irreducible root system. ϕ permutes the simple roots, hence induces a graph automorphism on the corresponding Dynkin diagram. If $\phi = 1$ then conjugacy classes and characters coincide with those of the Coxeter group W.

- The nontrivial cases for crystallographic roots systems are (the order of ϕ is written as left exponent to the type): ${}^2A_n$, ${}^2D_n$, ${}^3D_4$, ${}^2E_6$.
- For non-crystallographic root systems where all the roots have the same length the additional cases ${}^2B_2$, ${}^2G_2$, ${}^2F_4$ and ${}^2I_2(k)$ arise.
- In case ${}^3D_4$ the group $W \rtimes \langle\phi\rangle$ can be embedded into the Coxeter group of type F_4, which induces a labeling for the conjugacy classes of the coset. The preferred extension is chosen as the (single) extension with rational values.
- In case ${}^2D_n$ the group $W \rtimes \langle\phi\rangle$ is isomorphic to a Coxeter group of type B_n. This induces a canonical labeling for the conjugacy classes of the coset and allows to define the preferred extension in a combinatorial way using the labels (pairs of partitions) for the characters of the Coxeter group of type B_n.
- In the remaining crystallographic cases ϕ identifies to $-w_0$ where w_0 is the longest element of W. So, there is a canonical labeling of the conjugacy classes and characters of the coset by those of W. The preferred extensions are defined by describing the signs of the character values on $-w_0$.

In GAP3 the most general construction of a Coxeter coset is by starting from a Coxeter datum specified by the matrices of `simpleRoots` and `simpleCoroots`, and giving in addition the matrix `F0Mat` of the map $F_0 : V \to V$ (see the commands `CoxeterCoset` and `CoxeterSubCoset`). As for Coxeter groups, the elements of $W\phi$ are uniquely determined by the permutation they induce on the set of roots R. We consider these permutations as `Elements` of the Coxeter coset.

Coxeter cosets are implemented in GAP3 by a record which points to a Coxeter datum record and has additional fields holding `F0Mat` and the corresponding element `phi`. Functions on the coset (for example, `ChevieClassInfo`) are about properties of the group coset $W\phi$; however, most definitions for elements of untwisted Coxeter groups apply without change to elements in $W\phi$: e.g., if we define the length of an element $w\phi \in W\phi$ as the number of positive roots it sends to negative ones, it is the same as the length of w, i.e., ϕ is of length 0, since ϕ has been chosen to preserve the set of positive roots. Similarly, the `CoxeterWord` describing $w\phi$ is the same as the one for w, etc...

We associate to a Coxeter coset $W\phi$ a **twisted Dynkin diagram**, consisting of the Dynkin diagram of W and the graph automorphism induced by ϕ on this diagram (this specifies the group $W \rtimes \langle F\rangle$, mentioned above, up to isomorphism). See the functions `ReflectionType`, `ReflectionName` and `PrintDiagram` for Coxeter cosets.

Below is an example showing first how to **not** define, then how to define, the Weyl coset for a Suzuki group:

```
gap> 2B2:=CoxeterCoset(CoxeterGroup("B",2),(1,2));
# I transposed of matrix for F0 must normalize set of coroots of parent.
false
gap> 2B2:=CoxeterCoset(CoxeterGroup("Bsym",2),(1,2));
2Bsym2
```

```
gap> Display(CharTable(2B2));
2Bsym2

     2 1      2     2

              1   121

2.     1      1     1
.11    1     -1    -1
1.1    . -ER(2) ER(2)
```

A **subcoset** $Hw\phi$ of $W\phi$ is given by a reflection subgroup H of W and an element w of W such that $w\phi$ induces an automorphism of the root system of H. For algebraic groups, this corresponds to a rational form of a reductive subgroup of maximal rank. For example, if $W\phi$ corresponds to the algebraic group $\mathbf{G}$ and H is the trivial subgroup, the coset $Hw\phi$ corresponds to a maximal torus $\mathbf{T}_w$ of type w.

```
gap> CoxeterSubCoset(2B2,[],Group(2B2).1);
(q^2-ER(2)q+1)
```

A subgroup H which is a parabolic subgroup corresponds to a rational form of a Levi subgroup of $\mathbf{G}$. The command `Twistings` gives all rational forms of such a Levi.

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> Twistings(W,[1]);
[ ~A1.(q-1), ~A1.(q+1) ]
gap> Twistings(W,[2]);
[ A1<2>.(q-1), A1<2>.(q+1) ]
```

Notice how we distinguish between subgroups generated by short roots and by long roots. A general H corresponds to a reductive subgroup of maximal rank. Here we consider the subgroup generated by the long roots in B_2, which corresponds to a subgroup of type $SL_2 \times SL_2$ in SP_4, and show its possible rational forms.

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> Twistings(W,[2,4]);
[ A1<2>xA1<4>, (A1xA1)<2,4> ]
```

96.1 CoxeterCoset

`CoxeterCoset(` *W* `[,` *FMat* `] )`

`CoxeterCoset(` *W* `[,` *FPerm*`] )`

This function returns a Coxeter coset as a GAP3 object. The argument W must be a Coxeter group (created by `CoxeterGroup` or `ReflectionSubgroup`). In the first form the argument *F0Mat* must be an invertible matrix of dimension `Rank(`*W*`)`, representing an automorphism F of the root system of the parent of W. In the second form *FPerm* is a permutation of the roots of the parent group of W; it is assumed that the corresponding *FMat* acts trivially on the orthogonal of these roots. A shortcut is accepted if W has same rank as semisimple rank and *FPerm* preserves its simple roots: one need only give the induced permutation of the simple roots.

If there is no second argument the default for *F0Mat* is the identity matrix.

`CoxeterCoset` returns a record from which we document the following components:

`isDomain`, `isFinite`
: true

`reflectionGroup`
: the Coxeter group W

`F0Mat`
: the matrix acting on V which represents the unique element ϕ in WF_0 which preserves the positive roots.

`phi`
: the permutation of the roots of W induced by `F0Mat` (also the element of smallest length in the Coset WF_0).

In the first example we create a Coxeter coset corresponding to the general unitary groups $GU_3(q)$ over finite fields with q elements.

```
gap> W := RootDatum("gl",3);;
gap> gu3 := CoxeterCoset( W, -IdentityMat( 3 ) );
2A2.(q+1)
gap> F4 := CoxeterGroup( "F", 4 );;
gap> D4 := ReflectionSubgroup( F4, [ 1, 2, 16, 48 ] );;
gap> PrintDiagram( D4 );
D4 9
    \
     1 - 16
    /
   2
gap> CoxeterCoset( D4, MatXPerm(D4,(2,9,16)) );
3D4<9,16,1,2>
gap> CoxeterCoset( D4, (2,9,16));
3D4<9,16,1,2>
```

96.2 CoxeterSubCoset

`CoxeterSubCoset(` *WF*`,` *r*`[,` *w*`])`

Returns the reflection subcoset of the Coxeter coset *WF* generated by the reflections with roots specified by r. r is a list of indices specifying a subset of the roots of W where W is the Coxeter group `CoxeterGroup(`*WF*`)`. If specified, w must be an element of W such that `w*WF.phi` normalizes the subroot system generated by r. If absent, the default value for w is `()`. It is an error, if `w*WF.phi` does not normalize the subsystem.

```
gap> CoxeterSubCoset( CoxeterCoset( CoxeterGroup( "A", 2 ), (1,2) ),
>                                                          [ 1 ] );
Error, must give w, such that w * WF.phi normalizes subroot system.
```

```
 in
CoxeterSubCoset( CoxeterCoset( CoxeterGroup( "A", 2 ), (1,2) ), [ 1 ]
 ) called from
main loop
brk>
gap> f4coset := CoxeterCoset( CoxeterGroup( "F", 4 ) );
F4
gap> w := RepresentativeOperation( CoxeterGroup( f4coset ),
>                       [ 1, 2, 9, 16 ], [ 1, 9, 16, 2], OnTuples );;
gap> 3d4again := CoxeterSubCoset( f4coset, [ 1, 2, 9, 16], w );
3D4<9,16,1,2>
gap> PrintDiagram( 3d4again );
phi acts as ( 2, 9,16) on the component below
D4 9
    \
     1 - 2
    /
   16
```

96.3 Functions on Coxeter cosets

All functions for reflection cosets are implemented for Coxeter cosets.

This includes `Group(` *WF* `)` which returns the Coxeter group of which *WF* is a coset; the functions `Elements`, `Random`, `Representative`, `Size`, `in`, Rank, SemisimpleRank, which use the corresponding functions for `Group(` *WF* `)`; `ConjugacyClasses(` *WF* `)`, which returns the ϕ-conjugacy classes of W; the corresponding `PositionClass(` *WF* `,` *x* `)` and `FusionConjugacyClasses(` *HF*`,` *WF* `)`, `InductionTable(` *HF*`,` *WF* `)`.

For Weyl coset associated to a finite reductive group $\mathbf{G}^F$, the characters of the coset correspond to unipotent Deligne-Lusztig characters, and the induction from a subcoset corresponding to a Levi subgroup corresponds to the Lusztig induction of the Deligne-Lusztig characters (see more details in the next chapter). Here are some examples:

Harish-Chandra induction in the basis of almost characters:

```
gap> WF := CoxeterCoset( CoxeterGroup( "A", 4 ), (1,4)(2,3) );
2A4
gap> Display( InductionTable( CoxeterSubCoset( WF, [ 2, 3 ] ), WF ) );
Induction from 2A2<2,3>.(q-1)(q+1) to 2A4
      |111 21 3
________________
11111 |  1  . .
2111  |  .  1 .
221   |  1  . .
311   |  1  . 1
32    |  .  . 1
41    |  .  1 .
5     |  .  . 1
```

Lusztig induction from a diagonal Levi:

```
gap> HF := CoxeterSubCoset( WF, [1, 2],
>               LongestCoxeterElement( CoxeterGroup( WF ) ) );;
gap> Display( InductionTable( HF, WF ) );
Induction from 2A2.(q+1)^2 to 2A4
      |111 21  3
-----------------
11111 | -1  .  .
2111  | -2 -1  .
221   | -1 -2  .
311   |  1  2 -1
32    |  . -2  1
41    |  .  1 -2
5     |  .  .  1
```

A descent of scalars:

```
gap> W := CoxeterCoset( CoxeterGroup( "A", 2, "A", 2 ), (1,3)(2,4) );
(A2xA2)
gap> Display( InductionTable( CoxeterSubCoset( W, [ 1, 3 ] ), W ) );
Induction from (A1xA1)<1,3>.(q-1)(q+1) to (A2xA2)
    |11 2
----------
111 | 1 .
21  | 1 1
3   | . 1
```

`Print( WF )`, `ReflectionName( WF )` and `PrintDiagram( WF )` show the isomorphism type of the reductive group $\mathbf{G}^F$. An orbit of $\phi = WF$.`phi` on the components is put in brackets if of length k greater than 1, and is preceded by the order of ϕ^k on it, if this is not 1. For example `"2(A2xA2)"` denotes 2 components of type A_2 permuted by ϕ, and such that ϕ^2 induces the non-trivial diagram automorphism on any of them, while `3D4` denotes an orbit of length 1 on which ϕ is of order 3.

```
gap> W := CoxeterCoset( CoxeterGroup( "A", 2, "G", 2, "A", 2 ),
>                                                     (1,5,2,6) );
2(A2xA2)<1,2,5,6>xG2<3,4>
gap> ReflectionName( W );
"2(A2xA2)<1,2,5,6>xG2<3,4>"
gap> W := CoxeterCoset( CoxeterGroup( "A", 2, "A", 2 ), (1,3,2,4) );
2(A2xA2)
gap> PrintDiagram( W );
phi permutes the next 2 components
phi^2 acts as (1,2) on the component below
A2 1 - 2
A2 3 - 4
```

`ChevieClassInfo( WF )`, see the explicit description in 96.5.

`ChevieCharInfo` returns additional information on the irreducible characters, see 95.6.

Finally, some functions for elements of a Coxeter group work naturally for elements of a Coxeter coset: `CoxeterWord`, `EltWord`, `CoxeterLength`, `LeftDescentSet`, `RightDescentSet`, `ReducedInRightCoset`, etc... These functions take the same value on $w\phi \in W\phi$ that they take on $w \in W$.

96.4 ReflectionType for Coxeter cosets

`ReflectionType(` *WF* `)`

returns the type of the Coxeter coset *WF*. This consists of a list of records, one for each orbit of *WF*`.phi` on the irreducible components of the Dynkin diagram of `CoxeterGroup(`*WF*`)`, which have two fields:

`orbit`
: is a list of types of the irreducible components in the orbit. These types are the same as returned by the function `ReflectionType` for an irreducible untwisted Coxeter group (see `ReflectionType` in chapter 85): a couple `[`*type*`,`*indices*`]` (a triple for type $I_2(n)$). The components are ordered according to the action of *WF*`.phi`, so *WF*`.phi` maps the generating permutations with indices in the first type to indices in the second type in the same order as stored in the type, etc ...

`phi`
: if k is the number of irreducible components in the orbit, this is the permutation which describes the action of *WF*`.phi`^{k} on the simple roots of the first irreducible component in the orbit.

```
gap> W := CoxeterCoset( CoxeterGroup( "A", 2, "A", 2 ), (1,3,2,4) );
2(A2xA2)
gap> ReflectionType( W );
[ rec(orbit := [ rec(rank    := 2,
      series  := "A",
      indices := [ 1, 2 ]), rec(rank    := 2,
      series  := "A",
      indices := [ 3, 4 ]) ],
      twist := (1,2)) ]
```

96.5 ChevieClassInfo for Coxeter cosets

`ChevieClassInfo(` *WF* `)`

returns information about the conjugacy classes of the Coxeter coset *WF*. The result is a record with three components: `classtext` contains a list of reduced words for the representatives in `ConjugacyClasses(`*WF*`)`, `classnames` contains corresponding names for the classes, and `classparams` gives corresponding parameters for the classes. Let W be the Coxeter group `CoxeterGroup(`*WF*`)`. In the case where $-1 \notin W$, i.e., $\phi = -w_0$, they are obtained by multiplying by w_0 a set of representatives of *maximal* length of the classes of W.

```
gap> W := CoxeterGroup( "D", 4 );;
gap> ChevieClassInfo( CoxeterCoset( W, (1,2,4) ) );
rec(
  classtext := [ [ 1 ], [  ], [ 1, 2, 3, 1, 2, 3 ], [ 3 ], [ 1, 3 ],
      [ 1, 2, 3, 1, 2, 4, 3, 2 ], [ 1, 2, 3, 2 ] ],
```

```
    classnames := [ "C_3", "\\tilde A_2", "C_3+A_1", "\\tilde A_2+A_1",
        "F_4", "\\tilde A_2+A_2", "F_4(a_1)" ],
    classparams :=
     [ [ "C_3" ], [ "\\tilde A_2" ], [ "C_3+A_1" ], [ "\\tilde A_2+A_1"
          ], [ "F_4" ], [ "\\tilde A_2+A_2" ], [ "F_4(a_1)" ] ] )
```

96.6 CharTable for Coxeter cosets

`CharTable(` *WF* `)`

This function returns the character table of the Coxeter coset *WF* (see also the introduction of this Chapter). We call "characters"of the Coxeter coset WF with corresponding Coxeter group W the restriction to $W\phi$ of a set containing one extension of each ϕ-invariant character of W to the semidirect product of W with the cyclic group generated by ϕ. (We choose, following Lusztig, in each case one non-canonical extension, called the preferred extension.)

The returned record contains almost all components present in the character table of a Coxeter group. But if ϕ is not trivial then there are no components `powermap` (since powers of elements in the coset need not be in the coset) and `orders` (if you really need them, use `MatXPerm` to determine the order of elements in the coset).

```
gap> W := CoxeterCoset( CoxeterGroup( "D", 4 ), (1,2,4) );
3D4
gap> Display( CharTable( W ) );
3D4

       2  2   2     2      2  2      3      3
       3  1   1     1      .  .      1      1

          C3 ~A2 C3+A1 ~A2+A1 F4 ~A2+A2 F4(a1)

.4         1   1     1      1  1      1      1
.1111     -1   1     1     -1  1      1      1
.22        .   2     2      . -1     -1     -1
11.2       .   .     .      . -1      3      3
1.3        1   1    -1     -1  .     -2      2
1.111     -1   1    -1      1  .     -2      2
1.21       .   2    -2      .  .      2     -2
```

96.7 Frobenius

`Frobenius(` *WF* `)(` *o* `[,` *i*`])`

Given a Coxeter coset *WF*, `Frobenius(`*WF*`)` returns a function which makes *WF*`.phi` act on its argument which is some object for which this action has been defined. If *o* is a list, it applies recursively `Frobenius` to each element of the list. If it is a permutation or an integer, it returns *o*`^(`*WF*`.phi^-1)`. If a second argument *i* is given, it applies `Frobenius` raised to the *i*-th power (this is convenient for instance to apply the inverse of `Frobenius`). Finally, for an arbitrary object defined by a record, it looks if the method *o*`.operations.Frobenius` is defined and if so calls this function with arguments *WF*, *o* and *i* (with `i=1` if it was omitted).

Such an action of the Frobenius is defined for instance for braids, Hecke elements and semisimple elements.

```
gap> W:=CoxeterGroup("E",6);;WF:=CoxeterCoset(W,(1,6)(3,5));
2E6
gap> T:=Basis(Hecke(Group(WF)),"T");;Frobenius(WF)(T(1));
T(6)
gap> B:=Braid(W);; Frobenius(WF)(B(1,2,3));
265
gap> s:=SemisimpleElement(W,[1..6]/6);
<1/6,1/3,1/2,2/3,5/6,0>
gap> Frobenius(WF)(s);
<0,1/3,5/6,2/3,1/2,1/6>
gap> W:=CoxeterGroup("D",4);WF:=CoxeterCoset(W,(1,2,4));
CoxeterGroup("D",4)
3D4
gap> B:=Braid(W);;b:=B(1,3);
13
gap> Frobenius(WF)(b);
43
gap> Frobenius(WF)(b,-1);
23
```

96.8 GraphAutomorphisms

`GraphAutomorphisms(` *t* `)`

t should be the reflection type of a Coxeter group. The function returns the group of all Graph automorphisms of *t*.

```
gap> W:=CoxeterGroup("D",4,"D",4);
CoxeterGroup("D",4,"D",4)
gap> GraphAutomorphisms(ReflectionType(W));
Group( (1,5)(2,6)(3,7)(4,8), (1,2), (1,4) )
gap> Size(last);
72
```

96.9 Twistings for Coxeter cosets

`Twistings(` *W* `)`

W should be a Coxeter group record which is not a proper reflection subgroup of another reflection group. The function returns all `CoxeterCosets` representing twisted forms of algebraic groups of type *W*.

```
gap> Twistings(CoxeterGroup("A",3,"A",3));
[ A3xA3, A3x2A3, 2A3xA3, 2A3x2A3, (A3xA3), 2(A3xA3),
  2(A3xA3)<1,2,3,6,5,4>, (A3xA3)<1,2,3,6,5,4> ]
gap> Twistings(CoxeterGroup("D",4));
```

```
[ D4, 2D4<2,4,3,1>, 2D4, 3D4, 3'D4<1,4,3,2>, 2D4<1,4,3,2> ]
gap> Twistings(RootDatum("so",8));
[ D4, 2D4 ]
```

`Twistings(` *W* `,` *L* `)`

W should be a Coxeter group record or a Coxeter coset record, and *L* should be a reflection subgroup of *W* (or of `Group(`*W*`)` for a coset), or a sublist of the generating reflections of *W* (resp. `Group(W)`), in which case the call is the same as `Twistings(`*W*`,ReflectionSubgroup(`*W*`,`*L*`))` (resp. `Twistings(`*W*`,ReflectionSubgroup(Group(`*W*`),`*L*`)))`.

The function returns the list, up to *W*-conjugacy, of Coxeter sub-cosets of *W* whose Coxeter group is *L* — In term of algebraic groups, it corresponds to representatives of the possible twisted forms of the reductive subgroup of maximal rank *L*. In the case that *W* represents a coset $W\phi$, the subgroup *L* must be conjugate to $\phi(L)$ for a rational form to exist. If $w\phi$ normalizes *L*, then the rational forms are classified by the the ϕ-classes of $N_W(L)/L$.

```
gap> W:=CoxeterGroup("E",6);
CoxeterGroup("E",6)
gap> WF:=CoxeterCoset(W,(1,6)(3,5));
2E6
gap> L:=ReflectionSubgroup(W,[2..5]);
ReflectionSubgroup(CoxeterGroup("E",6), [ 2, 3, 4, 5 ])
gap> Twistings(W,L);
[ D4<2,3,4,5>.(q-1)^2, 3D4<2,3,4,5>.(q^2+q+1),
  2D4<2,3,4,5>.(q-1)(q+1) ]
gap> Twistings(WF,L);
[ 2D4<2,5,4,3>.(q-1)(q+1), 3'D4<2,5,4,3>.(q^2-q+1),
  D4<2,3,4,5>.(q+1)^2 ]
```

96.10 RootDatum for Coxeter cosets

`RootDatum(`*type*`[,`*rank*`])`

The function `RootDatum` can be used to get Coxeter cosets corresponding to known types of algebraic groups. The twisted types known are `"2B2"`, `"suzuki"`, `"2E6"`, `"2E6sc"`, `"2F4"`, `"2G2"`, `"ree"`, `"2I"`, `"3D4"`, `"triality"`, `"3D4sc"`, `"3gpin8"`, `"pso-"`, `"so-"`, `"spin-"`, `"gpin-"`, `"psu"`, `"su"`, `"u"`.

```
gap> RootDatum("su",4);
2A3
```

The above call is same as `CoxeterCoset(CoxeterGroup("A",3,"sc"),(1,3))` or `CoxeterCoset(RootDatum("sl`

96.11 Torus for Coxeter cosets

`Torus(`*M*`)`

M should be an integral matrix of finite order. `Torus` returns the coset `WF` of the trivial Coxeter group such that `WF.FOMat=`*M*. This corresponds to an algebraic torus **T** of rank `Length(M)`, with an isogeny which acts by *M* on $X(\mathbf{T})$.

```
gap> m:=[[0,-1],[1,-1]];
[ [ 0, -1 ], [ 1, -1 ] ]
```

```
gap> Torus(m);
(q^2+q+1)
```

`Torus(`W`,i)`

This returns the Torus twisted by the i-th conjugacy class of W. For Coxeter groups or cosets it is the same as `Twistings(W,[])[i]`.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> Twistings(W,[]);
[ (q-1)^3, (q-1)^2(q+1), (q-1)(q+1)^2, (q-1)(q^2+q+1), (q+1)(q^2+1) ]
gap> Torus(W,2);
(q-1)^2(q+1)
gap> W:=CoxeterCoset(CoxeterGroup("A",3),(1,3));
2A3
gap> Twistings(W,[]);
[ (q+1)^3, (q-1)(q+1)^2, (q-1)^2(q+1), (q+1)(q^2-q+1), (q-1)(q^2+1) ]
gap> Torus(W,2);
(q-1)(q+1)^2
```

96.12 StructureRationalPointsConnectedCentre

`StructureRationalPointsConnectedCentre(`G`,q)`

W should be a Coxeter group record or a Coxeter coset record, representing a finite reductive group $\mathbf{G}^F$, and q should be the prime power associated to the isogeny F. The function returns the abelian invariants of the finite abelian group $Z^0\mathbf{G}^F$ where $Z^0\mathbf{G}$ is the connected center of $\mathbf{G}$.

In the following example one determines the structure of $\mathbf{T}(\mathbb{F}_3)$ where $\mathbf{T}$ runs over all the maximal tori of SL_4.

```
gap> G:=RootDatum("sl",4);
RootDatum("sl",4)
gap> List(Twistings(G,[]),T->StructureRationalPointsConnectedCentre(T,3));
[ [ 2, 2, 2 ], [ 2, 8 ], [ 4, 8 ], [ 26 ], [ 40 ] ]
```

96.13 ClassTypes

`ClassTypes(`G `[,`p`])`

G should be a root datum or a twisted root datum representing a finite reductive group $\mathbf{G}^F$ and p should be a prime. The function returns the class types of G. Two elements of $\mathbf{G}^F$ have the same *class type* if their centralizers are conjugate. If su is the Jordan decomposition of an element x, the class type of x is determined by the class type of its semisimple part s and the unipotent class of u in $C_{\mathbf{G}}(s)$.

The function `ClassTypes` is presently only implemented for simply connected groups, where $C_{\mathbf{G}}(s)$ is connected. This section is a bit experimental and may change in the future.

`ClassTypes` returns a record which contains a list of classtypes for semisimple elements, which are represented by `CoxeterSubCosets` and contain additionnaly information on the unipotent classes of $C_{\mathbf{G}}(s)$.

The list of class types is different in bad characteristic, so the argument p should give a characteristic. If p is omitted or equal to 0, then good characteristic is assumed.

Let us give some examples

```
gap> t:=ClassTypes(RootDatum("sl",3));
ClassTypes(CoxeterCoset(RootDatum("sl",3)),good characteristic)
gap> Display(t);
ClassTypes(CoxeterCoset(RootDatum("sl",3)),good characteristic)
      Type |Centralizer
________________________
(q-1)^2    |       P1^2
(q-1)(q+1) |       P1P2
(q^2+q+1)  |         P3
A1.(q-1)   |    qP1^2P2
A2         |q^3P1^2P2P3
```

By default, only information about semisimple centralizer types is returned: the type, and its generic order.

```
gap> Display(t,rec(unip:=true));
ClassTypes(CoxeterCoset(RootDatum("sl",3)),good characteristic)
      Type |     u Centralizer
_______________________________
(q-1)^2    |              P1^2
(q-1)(q+1) |              P1P2
(q^2+q+1)  |                P3
A1.(q-1)   |    11     qP1^2P2
           |     2         qP1
A2         |   111 q^3P1^2P2P3
           |    21       q^3P1
           |     3        3q^2
           |  3_E3        3q^2
           |3_E3^2        3q^2
```

Here we have displayed information on unipotent classes, with their centralizer.

```
gap> Display(t,rec(nrClasses:=true));
ClassTypes(CoxeterCoset(RootDatum("sl",3)),good characteristic)
      Type |        nrClasses Centralizer
_________________________________________
(q-1)^2    |(4-5q+2q_3+q^2)/6        P1^2
(q-1)(q+1) |       (-q+q^2)/2        P1P2
(q^2+q+1)  |  (1+q-q_3+q^2)/3          P3
A1.(q-1)   |         -1+q-q_3     qP1^2P2
A2         |              q_3 q^3P1^2P2P3
```

Here we have added information on how many semisimple conjugacy classes of $\mathbf{G}^F$ have a given type. The answer in general involves variables of the form `q_d` which represent $\gcd(q-1,d)$.

Finally an example in bad characteristic

```
gap> t:=ClassTypes(CoxeterGroup("G",2),2);
```

```
ClassTypes(CoxeterCoset(CoxeterGroup("G",2)),char. 2)
gap> Display(t,rec(nrClasses:=true));
ClassTypes(CoxeterCoset(CoxeterGroup("G",2)),char. 2)
        Type |          nrClasses     Centralizer
__________________________________________________
(q-1)^2      |(10-8q+2q_3+q^2)/12           P1^2
(q-1)(q+1)   |         (-2q+q^2)/4          P1P2
(q-1)(q+1)   |         (-2q+q^2)/4          P1P2
(q^2-q+1)    |    (1-q-q_3+q^2)/6             P6
(q^2+q+1)    |    (1+q-q_3+q^2)/6             P3
(q+1)^2      |(-2-4q+2q_3+q^2)/12           P2^2
A1.(q-1)     |        (-1+q-q_3)/2       qP1^2P2
A1.(q+1)     |         (1+q-q_3)/2       qP1P2^2
G2           |                   1 q^6P1^2P2^2P3P6
A2<1,5>      |          (-1+q_3)/2     q^3P1^2P2P3
2A2<1,5>     |          (-1+q_3)/2     q^3P1P2^2P6
~A1<2>.(q-1) |            (-2+q)/2       qP1^2P2
~A1<2>.(q+1) |                 q/2       qP1P2^2
```

We notice that if q is a power of 2 such that $q \equiv 2 \pmod 3$, so that `q_3=1`, some class types do not exist. We can see what happens by using the function `Value` to give a specific value to `q_3`:

```
gap> Display(Value(t,["q_3",1]),rec(nrClasses:=true));
ClassTypes(CoxeterCoset(CoxeterGroup("G",2)),char. 2) q_3=1
        Type |     nrClasses     Centralizer
_____________________________________________
(q-1)^2      |(12-8q+q^2)/12           P1^2
(q-1)(q+1)   |    (-2q+q^2)/4          P1P2
(q-1)(q+1)   |    (-2q+q^2)/4          P1P2
(q^2-q+1)    |     (-q+q^2)/6            P6
(q^2+q+1)    |      (q+q^2)/6            P3
(q+1)^2      |   (-4q+q^2)/12          P2^2
A1.(q-1)     |       (-2+q)/2       qP1^2P2
A1.(q+1)     |            q/2       qP1P2^2
G2           |              1 q^6P1^2P2^2P3P6
~A1<2>.(q-1) |       (-2+q)/2       qP1^2P2
~A1<2>.(q+1) |            q/2       qP1P2^2
```

96.14 Quasi-Semisimple elements of non-connected reductive groups

We may also use Coxeter cosets to represented non-connected reductive groups of the form $\mathbf{G} \rtimes \sigma$ where $\mathbf{G}$ is a connected reductive group and σ an algebraic automorphism of $\mathbf{G}$, and more specifically the coset $\mathbf{G}.\sigma$. We may always choose $\sigma \in \mathbf{G}\cdot\sigma$ **quasi-semisimple**, which means that σ preserves a pair $\mathbf{T} \subset \mathbf{B}$ of a maximal torus and a Borel subgroup of $\mathbf{G}$. If σ is of finite order, it then defines an automorphism F_0 of the root datum $(X(\mathbf{T}), \Phi, Y(\mathbf{T}), \Phi^\vee)$, thus a Coxeter coset. We refer to [DM18] for details.

We have extended the functions for semi-simple elements to work with quasi-semisimple elements $t\sigma \in \mathbf{T}\cdot\sigma$. Here, as in [DM18], σ is a quasi-central automorphism uniquely defined by a diagram automorphism of (W, S), taking σ symplectic in type A_{2n}. We recall that a quasi-central element is a quasi-semisimple element such that the Weyl group of $C_{\mathbf{G}}(\sigma)$ is equal to W^σ; such an element always exists in the coset $\mathbf{G}\cdot\sigma$.

Here are some examples:

```
gap> WF:=RootDatum("u",6);
2A5.(q+1)
```

The above defines the coset $\mathrm{G}L_6\cdot\sigma$ where σ is the composed of transpose, inverse and the longest element of W.

```
gap> l:=QuasiIsolatedRepresentatives(WF);
[ <0,0,0,0,0,0>, <1/4,0,0,0,0,3/4>, <1/4,1/4,0,0,3/4,3/4>,
  <1/4,1/4,1/4,3/4,3/4,3/4> ]
```

we define an element $t\sigma \in \mathbf{T}\cdot\sigma$ to be quasi-isolated if the Weyl group of $C_{\mathbf{G}}(t\sigma)$ is not in any proper parabolic subgroup of W^σ. This generalizes the definition for connected groups. The above shows the elements t where $t\sigma$ runs over representatives of quasi-isolated quasi-semisimple classes of $\mathbf{G}\cdot\sigma$. The given representatives have been chosen σ-stable.

```
gap> List(l,s->Centralizer(WF,s));
[ C3<3,2,1>, B2.(q+1), (A1xA1)<1,3>xA1<2>, 2A3<3,1,2> ]
```

in the above, the groups $C_{\mathbf{G}}(t\sigma)$ are computed and displayed as extended Coxeter groups (following the same convention as for centralisers in connected reductive groups).

We define an element $t\sigma \in \mathbf{T}\cdot\sigma$ to be isolated if the Weyl group of $C_{\mathbf{G}}(t\sigma)^0$ is not in any proper parabolic subgroup of W^σ. This generalizes the definition for connected groups.

```
gap> List(l,s->IsIsolated(WF,s));
[ true, false, true, true ]
```

96.15 Centralizer for quasisemisimple elements

`Centralizer(` *WF*`,` *t*`)`

WF should be a Coxeter coset representing an algebraic coset $\mathbf{G}\cdot\sigma$, where $\mathbf{G}$ is a connected reductive group (represented by `W =Group(WF)`), and σ is a quasi-central automorphism of $\mathbf{G}$ defined by *WF*. The element t should be a semisimple element of $\mathbf{G}$. The function returns an extended reflection group describing $C_{\mathbf{G}}(t\sigma)$, with the reflection group part representing $C^0_{\mathbf{G}}(t\sigma)$, and the diagram automorphism part being those induced by $C_{\mathbf{G}}(t\sigma)/C_{\mathbf{G}}(t\sigma)^0$ on $C_{\mathbf{G}}(t\sigma)^0$.

```
gap> WF:=RootDatum("u",6);
2A5.(q+1)
gap> s:=SemisimpleElement(Group(WF),[1/4,0,0,0,0,3/4]);
<1/4,0,0,0,0,3/4>
gap> Centralizer(WF,s);
B2.(q+1)
gap> Centralizer(WF,s^0);
C3<3,2,1>
```

96.16 QuasiIsolatedRepresentatives for Coxeter cosets

`QuasiIsolatedRepresentatives(`WF`[, `p`])`

WF should be a Coxeter coset representing an algebraic coset $\mathbf{G}\cdot\sigma$, where $\mathbf{G}$ is a connected reductive group (represented by `W =Group(WF)`), and σ is a quasi-central automorphism of $\mathbf{G}$ defined by WF. The function returns a list of semisimple elements of $\mathbf{G}$ such that $t\sigma$, when t runs over this list, are representatives of the conjugacy classes of quasi-isolated quasisemisimple elements of $\mathbf{G}\cdot\sigma$ (an element $t\sigma\in\mathbf{T}\cdot\sigma$ is quasi-isolated if the Weyl group of $C_{\mathbf{G}}(t\sigma)$ is not in any proper parabolic subgroup of W^σ). If a second argument p is given, it lists only those representatives which exist in characteristic p.

```
gap> QuasiIsolatedRepresentatives(RootDatum("2E6sc"));
[ <0,0,0,0,0,0>, <0,0,0,1/2,0,0>, <0,1/2,1/4,0,1/4,0>,
  <0,2/3,0,1/3,0,0>, <0,3/4,0,1/2,0,0> ]
gap> QuasiIsolatedRepresentatives(RootDatum("2E6sc"),2);
[ <0,0,0,0,0,0>, <0,2/3,0,1/3,0,0> ]
gap> QuasiIsolatedRepresentatives(RootDatum("2E6sc"),3);
[ <0,0,0,0,0,0>, <0,0,0,1/2,0,0>, <0,1/2,1/4,0,1/4,0>,
  <0,3/4,0,1/2,0,0> ]
```

96.17 IsIsolated for Coxeter cosets

`IsIsolated(`WF`, `t`)`

WF should be a Coxeter coset representing an algebraic coset $\mathbf{G}\cdot\sigma$, where $\mathbf{G}$ is a connected reductive group (represented by `W =Group(WF)`), and σ is a quasi-central automorphism of $\mathbf{G}$ defined by WF. The element t should be a semisimple element of $\mathbf{G}$. The function returns a boolean describing whether $t\sigma$ is isolated, that is whether the Weyl group of $C_{\mathbf{G}}(t\sigma)^0$ is not in any proper parabolic subgroup of W^σ.

```
gap> WF:=RootDatum("u",6);
2A5.(q+1)
gap> l:=QuasiIsolatedRepresentatives(WF);
[ <0,0,0,0,0,0>, <1/4,0,0,0,0,3/4>, <1/4,1/4,0,0,3/4,3/4>,
  <1/4,1/4,1/4,3/4,3/4,3/4> ]
gap> List(l,s->IsIsolated(WF,s));
[ true, false, true, true ]
```

Chapter 97

Hecke cosets

"Hecke cosets" are $H\phi$ where H is a Hecke algebra of some Coxeter group W on which the reduced element ϕ acts by $\phi(T_w) = T_{\phi(w)}$. This corresponds to the action of the Frobenius automorphism on the commuting algebra of the induced of the trivial representation from the rational points of some F-stable Borel subgroup to $\mathbf{G}^F$.

```
gap> W := CoxeterGroup( "A", 2 );;
gap> q := X( Rationals );; q.name := "q";;
gap> HF := Hecke( CoxeterCoset( W, (1,2) ), q^2, q );
Hecke(2A2,q^2,q)
gap> Display( CharTable( HF ) );
H(2A2)

       2      1   1   .
       3      1   .   1

            111  21   3
      2P    111 111   3
      3P    111  21 111

111          -1   1  -1
21        -2q^3   .   q
3           q^6   1 q^2
```

Thanks to the work of Xuhua He and Sian Nie, `HeckeClassPolynomials` also make sense for these cosets. This is used to compute such character tables.

97.1 Hecke for Coxeter cosets

`Hecke(` *WF*`,` *H* `)`

`Hecke(` *WF*`,` *params* `)`

Construct a Hecke coset a Coxeter coset *WF* and an Hecke algebra associated to the CoxeterGroup of *WF*. The second form is equivalent to `Hecke(` *WF*`, Hecke(CoxeterGroup(`*WF*`),` *params*`))`.

97.2 Operations and functions for Hecke cosets

`Hecke`
: returns the untwisted Hecke algebra corresponding to the Hecke coset.

`CoxeterCoset`
: returns the Coxeter coset corresponding to the Hecke coset.

`CoxeterGroup`
: returns the untwisted Coxeter group corresponding to the Hecke coset.

`Print`
: prints the Hecke coset in a form which can be read back into GAP3.

`CharTable`
: returns the character table of the Hecke coset.

`Basis(H,"T")`
: the `T` basis.

Chapter 98

Unipotent characters of finite reductive groups and Spetses

Let $\mathbf{G}$ be a connected reductive group defined over the algebraic closure of a finite field $\mathcal{F}_q$, with corresponding Frobenius automorphism F, or more generally let F be an isogeny of $\mathbf{G}$ such that a power is a Frobenius (this covers the Suzuki and Ree groups).

If $\mathbf{T}$ is an F-stable maximal torus of $\mathbf{G}$, and $\mathbf{B}$ is a (not necessarily F-stable) Borel subgroup containing $\mathbf{T}$, we define the **Deligne-Lusztig** variety $X_{\mathbf{B}} = \{g\mathbf{B} \in \mathbf{G}/\mathbf{B} \mid g\mathbf{B} \cap F(g\mathbf{B}) \neq \emptyset\}$. This variety has a natural action of $\mathbf{G}^F$ on the left, so the corresponding **Deligne-Lusztig virtual module** $\sum_i (-1)^i H^i_c(X_{\mathbf{B}}, \overline{\mathbb{Q}}_\ell)$ also. The character of this virtual module is the **Deligne-Lusztig** character $R^{\mathbf{G}}_{\mathbf{T}}(1)$; the notation reflects the fact that one can prove that this character does not depend on the choice of $\mathbf{B}$. Actually, this character is parameterized by an F-conjugacy class of W: if $\mathbf{T}_0 \subset \mathbf{B}_0$ is an F-stable pair, there is an unique $w \in W = N_{\mathbf{G}}(\mathbf{T}_0)/\mathbf{T}_0$ such that the triple $(\mathbf{T}, \mathbf{B}, F)$ is $\mathbf{G}$-conjugate to $(\mathbf{T}_0, \mathbf{B}_0, wF)$. In this case we denote R_w for $R^{\mathbf{G}}_{\mathbf{T}}(1)$; it depends only on the F-class of w.

The **unipotent characters** of $\mathbf{G}^F$ are the irreducible constituents of the R_w. In a similar way that the unipotent classes are a building block for describing the conjugacy classes of a reductive group, the unipotent characters are a building block for the irreducible characters of a reductive group. They can be parameterized by combinatorial data that Lusztig has attached just to the coset $W\phi$, where ϕ is the finite order automorphism of $X(\mathbf{T}_0)$ such that $F = q\phi$. Thus, from the viewpoint of CHEVIE, they are objects combinatorially attached to a Coxeter coset.

A subset of the unipotent characters, the **principal series** unipotent characters, can be described in an elementary way. They are the constituents of R_1, or equivalently the characters of the virtual module defined by the cohomology of $X_{\mathbf{B}_0}$, which is the discrete variety $(\mathbf{G}/\mathbf{B}_0)^F$; the virtual module reduces to the actual module $\overline{\mathbb{Q}}_\ell[(\mathbf{G}/\mathbf{B}_0)^F]$. Thus the Deligne-Lusztig induction $R^{\mathbf{G}}_{\mathbf{T}_0}(1)$ reduces to Harish-Chandra induction, defined as follows: let $\mathbf{P} = \mathbf{U} \rtimes \mathbf{L}$ be an F-stable Levi decomposition of an F-stable parabolic subgroup of $\mathbf{G}$. Then the **Harish-Chandra** induced $R^{\mathbf{G}}_{\mathbf{L}}$ of a character χ of $\mathbf{L}^F$ is the character $\operatorname{Ind}^{\mathbf{G}^F}_{\mathbf{P}^F} \tilde{\chi}$, where $\tilde{\chi}$ is the lift to $\mathbf{P}^F$ of χ via the quotient $\mathbf{P}^F/\mathbf{U}^F = \mathbf{L}^F$; Harish-Chandra induction is a particular case of **Lusztig induction**, which is defined when $\mathbf{P}$ is not F-stable using the variety $X_{\mathbf{U}} = \{g\mathbf{U} \in \mathbf{G}/\mathbf{U} \mid g\mathbf{U} \cap F(g\mathbf{U}) \neq \emptyset\}$, and gives for an $\mathbf{L}^F$-module a virtual $\mathbf{G}^F$-module. Like ordinary induction, these functors have adjoint functors going from representations of $\mathbf{G}^F$ to representations (resp. virtual representations) of $\mathbf{L}^F$ called Harish-Chandra restriction (resp. Lusztig restriction).

The commuting algebra of $\mathbf{G}^F$-endomorphisms of $R_{\mathbf{T}_0}^{\mathbf{G}}(1)$ is an Iwahori-Hecke algebra for W^ϕ, with parameters which are some powers of q; they are all equal to q when $W^\phi = W$. Thus principal series unipotent characters correspond to characters of W^ϕ.

To understand the decomposition of Deligne-Lusztig characters, and thus unipotent characters, is is useful to introduce another set of class functions which are parameterized by irreducible characters of the coset $W\phi$. If χ is such a character, we define the associated **almost character** by:$R_\chi = |W|^{-1}\sum_{w\in W}\chi(w\phi)R_w$. The reason to the name is that these class function are close to irreducible characters:they satisfy $\langle R_\chi, R_\psi\rangle_{\mathbf{G}^F} = \delta_{\chi,\psi}$; for the linear and unitary group they are actually unipotent characters (up to sign in the latter case). They are in general sum (with rational coefficients) of a small number of unipotent characters in the same **Lusztig family** (see 98.15). The degree of R_χ is a polynomial in q equal to the fake degree of the character χ of $W\phi$ (see 95.5).

We now describe the parameterization of unipotent characters when $W^\phi = W$, thus when the coset $W\phi$ identifies with W (the situation is similar but a bit more difficult to describe in general). The (rectangular) matrix of scalar products $\langle \rho, R_\chi\rangle_{\mathbf{G}^F}$, when characters of W and unipotent characters are arranged in the right order, is block-diagonal with rather small blocks which are called **Lusztig families**.

For the characters of W a family $\mathcal{F}$ corresponds to a block of the Hecke algebra over a ring called the Rouquier ring. To $\mathcal{F}$ Lusztig associates a small group Γ (not bigger than $(\mathbb{Z}/2)^n$, or $\mathfrak{S}_i$ for $i \leq 5$) such that the unipotent characters in the family are parameterized by the pairs (x,θ) taken up to Γ-conjugacy, where $x \in \Gamma$ and θ is an irreducible character of $C_\Gamma(x)$. Further, the elements of $\mathcal{F}$ themselves are parameterized by a subset of such pairs, and Lusztig defines a pairing between such pairs which computes the scalar product $\langle \rho, R_\chi\rangle_{\mathbf{G}^F}$. For more details see 98.22.

A second parameterization of unipotent character is via **Harish-Chandra series**. A character is called **cuspidal** if all its proper Harish-Chandra restrictions vanish. There are few cuspidal unipotent characters (none in linear groups, and at most one in other classical groups). The $\mathbf{G}^F$-endomorphism algebra of an Harish-Chandra induced $\mathbb{R}_{\mathbf{L}^F}^{\mathbf{G}^F}\lambda$, where λ is a cuspidal unipotent character turns out to be a Hecke algebra associated to the group $W_{\mathbf{G}^F}(\mathbf{L}^F) := N_{\mathbf{G}^F}(\mathbf{L})/\mathbf{L}$, which turns out to be a Coxeter group. Thus another parameterization is by triples $(\mathbf{L},\lambda,\varphi)$, where λ is a cuspidal unipotent character of $\mathbf{L}^F$ and φ is an irreducible character of the **relative group** $W_{\mathbf{G}^F}(\mathbf{L}^F)$. Such characters are said to belong to the Harish-Chandra series determined by $(\mathbf{L},\lambda)$.

A final piece of information attached to unipotent characters is the **eigenvalues of Frobenius**. Let F^δ be the smallest power of the isogeny F which is a split Frobenius (that is, F^δ is a Frobenius and $\phi^\delta = 1$). Then F^δ acts naturally on Deligne-Lusztig varieties and thus on the corresponding virtual modules, and commutes to the action of $\mathbf{G}^F$; thus for a given unipotent character ρ, a submodule of the virtual module which affords ρ affords a single eigenvalue μ of F^δ. Results of Lusztig and Digne-Michel show that this eigenvalue is of the form $q^{a\delta}\lambda_\rho$ where $2a \in \mathbb{Z}$ and λ_ρ is a root of unity which depends only on ρ and not the considered module. This λ_ρ is called the eigenvalue of Frobenius attached to ρ. Unipotent characters in the Harish-Chandra series of a pair $(\mathbf{L},\lambda)$ have the same eigenvalue of Frobenius as λ.

CHEVIE contains table of all this information, and can compute Harish-Chandra and Lusztig induction of unipotent characters and almost characters. We illustrate the information on some examples:

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> uc:=UnipotentCharacters(W);
UnipotentCharacters( G2 )
gap> Display(uc);
Unipotent characters for G2
     gamma |  Deg(gamma) FakeDegree Fr(gamma)     Label
________________________________________________________
phi{1,0}   |           1          1         1
phi{1,6}   |         q^6        q^6         1
phi{1,3}'  |    1/3qP3P6        q^3         1     (1,r)
phi{1,3}'' |    1/3qP3P6        q^3         1    (g3,1)
phi{2,1}   |  1/6qP2^2P3        qP8         1     (1,1)
phi{2,2}   |  1/2qP2^2P6      q^2P4         1    (g2,1)
G2[-1]     |  1/2qP1^2P3          0        -1  (g2,eps)
G2[1]      |  1/6qP1^2P6          0         1   (1,eps)
G2[E3]     |1/3qP1^2P2^2          0        E3   (g3,E3)
G2[E3^2]   |1/3qP1^2P2^2          0      E3^2 (g3,E3^2)
```

The first column gives the name of the unipotent character; the first 6 are in the principal series so are named according to the corresponding characters of W. The last 4 are cuspidal, and named by the corresponding eigenvalue of Frobenius, which is displayed in the fourth column. In general the names of the unipotent characters come from their parameterization by Harish-Chandra series; in addition, for classical groups, they are associated to **symbols**.

The first two characters are each in a family by themselves. The last eight are in a family associated to the group $\Gamma = \mathfrak{S}_3$:the last column shows the parameters (x, θ). The second column shows the degree of the unipotent characters, which is transformed by the Lusztig Fourier matrix of the third column, which gives the degree of the corresponding almost character, or equivalently the fake degree of the corresponding character of W.

One can get more information on the Lusztig Fourier matrix of the big family by asking

```
gap> Display(uc.families[1]);
D(S3)
    label |eigen
________________________________________________________
(1,1)     |    1 1/6  1/2  1/3  1/3  1/6  1/2  1/3  1/3
(g2,1)    |    1 1/2  1/2    0    0 -1/2 -1/2    0    0
(g3,1)    |    1 1/3    0  2/3 -1/3  1/3    0 -1/3 -1/3
(1,r)     |    1 1/3    0 -1/3  2/3  1/3    0 -1/3 -1/3
(1,eps)   |    1 1/6 -1/2  1/3  1/3  1/6 -1/2  1/3  1/3
(g2,eps)  |   -1 1/2 -1/2    0    0 -1/2  1/2    0    0
(g3,E3)   |   E3 1/3    0 -1/3 -1/3  1/3    0  2/3 -1/3
(g3,E3^2) | E3^2 1/3    0 -1/3 -1/3  1/3    0 -1/3  2/3
```

One can do computations with individual unipotent characters. Here we construct the Coxeter torus, and then the identity character of this torus as a unipotent character.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> T:=ReflectionCoset(ReflectionSubgroup(W,[]),EltWord(W,[1,2]));
(q^2-q+1)
gap> u:=UnipotentCharacter(T,1);
[(q^2-q+1)]=<>
```

Then here are two ways to construct the Deligne-Lusztig character associated to the Coxeter torus:

```
gap> LusztigInduction(W,u);
[G2]=<phi{1,0}>+<phi{1,6}>-<phi{2,1}>+<G2[-1]>+<G2[E3]>+<G2[E3^2]>
gap> v:=DeligneLusztigCharacter(W,[1,2]);
[G2]=<phi{1,0}>+<phi{1,6}>-<phi{2,1}>+<G2[-1]>+<G2[E3]>+<G2[E3^2]>
gap> Degree(v);
q^6 + q^5 - q^4 - 2*q^3 - q^2 + q + 1
gap> v*v;
6
```

The last two lines ask for the degree of v, then for the scalar product of v with itself.

Finally we mention that CHEVIE can also provide unipotent characters of Spetses, as defined in [BMM14]. An example:

```
gap> Display(UnipotentCharacters(ComplexReflectionGroup(4)));
Unipotent characters for G4
   gamma |           Deg(gamma) FakeDegree Fr(gamma)    Label
_____________________________________________________________
phi{1,0} |                    1          1         1
phi{1,4} | -ER(-3)/6q^4P"3P4P"6        q^4         1  1.-E3^2
phi{1,8} |  ER(-3)/6q^4P'3P4P'6        q^8         1  -1.E3^2
phi{2,5} |          1/2q^4P2^2P6      q^5P4         1   1.E3^2
phi{2,3} |(3+ER(-3))/6qP"3P4P'6      q^3P4         1   1.E3^2
phi{2,1} |(3-ER(-3))/6qP'3P4P"6        qP4         1     1.E3
phi{3,2} |              q^2P3P6    q^2P3P6         1
Z3:2     |     -ER(-3)/3qP1P2P4          0      E3^2  E3.E3^2
Z3:11    |   -ER(-3)/3q^4P1P2P4          0      E3^2   E3.-E3
G4       |        -1/2q^4P1^2P3          0        -1 -E3^2.-1
```

98.1 UnipotentCharacters

`UnipotentCharacters(`*W*`)`

W should be a Coxeter group, a Coxeter Coset or a Spetses. The function gives back a record containing information about the unipotent characters of the associated algebraic group (or Spetses). This contains the following fields:

`group`
: a pointer to *W*

`charNames`
: the list of names of the unipotent characters.

`charSymbols`
the list of symbols associated to unipotent characters, for classical groups.

`harishChandra`
information about Harish-Chandra series of unipotent characters. This is itself a list of records, one for each pair $(\mathbf{L},\lambda)$ of a Levi of an F-stable parabolic subgroup and a cuspidal unipotent character of $\mathbf{L}^F$. These records themselves have the following fields:

`levi`
a list `l` such that $\mathbf{L}$ corresponds to `ReflectionSubgroup(W,l)`.

`cuspidalName`
the name of the unipotent cuspidal character *lambda*.

`eigenvalue`
the eigenvalue of Frobenius for λ.

`relativeType`
the reflection type of $W_{\mathbf{G}}(\mathbf{L})$;

`parameterExponents`
the $\mathbf{G}^F$-endomorphism algebra of $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$ is a Hecke algebra for $W_{\mathbf{G}}(\mathbf{L})$ with some parameters of the form q^{a_s}. This holds the list of exponents a_s.

`charNumbers`
the indices of the unipotent characters indexed by the irreducible characters of $W_{\mathbf{G}}(\mathbf{L})$.

`families`
information about Lusztig families of unipotent characters. This is itself a list of records, one for each family. These records are described in the section about families below.

```
gap> W:=CoxeterGroup("Bsym",2);
CoxeterGroup("Bsym",2)
gap> WF:=CoxeterCoset(W,(1,2));
2Bsym2
gap> uc:=UnipotentCharacters(W);
UnipotentCharacters( Bsym2 )
gap> Display(uc);
Unipotent characters for Bsym2
gamma |Deg(gamma) FakeDegree Fr(gamma) Label
_________________________________________________
11.   |    1/2qP4       q^2         1   +,-
1.1   |  1/2qP2^2       qP4         1   +,+
.11   |       q^4       q^4         1
2.    |         1         1         1
.2    |    1/2qP4       q^2         1   -,+
B2    |  1/2qP1^2         0        -1   -,-
gap> uc.harishChandra[1];
rec(
  levi := [  ],
  relativeType := [ rec(series  := "B",
          indices := [ 1, 2 ],
```

```
            rank    := 2) ],
      eigenvalue := 1,
      qEigen := 0,
      parameterExponents := [ 1, 1 ],
      charNumbers := [ 1, 2, 3, 4, 5 ],
      cuspidalName := "" )
    gap> uc.families[2];
    Family("012",[1,2,5,6])
    gap> Display(uc.families[2]);
    012
    label |eigen  +,- +,+  -,+  -,-
    --------------------------------
    +,-   |    1  1/2 1/2 -1/2 -1/2
    +,+   |    1  1/2 1/2  1/2  1/2
    -,+   |    1 -1/2 1/2  1/2 -1/2
    -,-   |   -1 -1/2 1/2 -1/2  1/2
```

98.2 Operations for UnipotentCharacters

`CharNames` returns the names of the unipotent characters. Using the version with an additional option record as the second argument, one can control the display in various ways.

```
gap> uc:=UnipotentCharacters(CoxeterGroup("G",2));
UnipotentCharacters( G2 )
gap> CharNames(uc);
[ "phi{1,0}", "phi{1,6}", "phi{1,3}'", "phi{1,3}''", "phi{2,1}",
  "phi{2,2}", "G2[-1]", "G2[1]", "G2[E3]", "G2[E3^2]" ]
gap> CharNames(uc,rec(TeX:=true));
[ "\\phi_{1,0}", "\\phi_{1,6}", "\\phi_{1,3}'", "\\phi_{1,3}''",
  "\\phi_{2,1}", "\\phi_{2,2}", "G_2[-1]", "G_2[1]", "G_2[\\zeta_3]",
  "G_2[\\zeta_3^2]" ]
```

`Display` One can control the display of unipotent characters in various ways. In the record controlling `Display`, a field `items` specifies which columns are displayed. The possible values are

`"n0"`
: The index of the character in the list of unipotent characters.

`"Name"`
: The name of the unipotent character.

`"Degree"`
: The degree of the unipotent character.

`"FakeDegree"`
: The degree of the corresponding almost character.

`"Eigenvalue"`
: The eigenvalue of Frobenius attached to the unipotent character.

`"Symbol"`
: for classical groups, the symbol attached to the unipotent character.

"Family"

The parameter the character has in its Lusztig family.

"Signs"

The signs attached to the character in the Fourier transform.

The default value is `items:=["Name","Degree","FakeDegree","Eigenvalue","Family"]`

This can be changed by setting the variable `UnipotentCharactersOps.items` which holds this default value. In addition if the field `byFamily` is set, the characters are displayed family by family instead of in index order. Finally, the field `chars` can be set, indicating which characters are to be displayed in which order.

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> uc:=UnipotentCharacters(W);
UnipotentCharacters( B2 )
gap> Display(uc);
Unipotent characters for B2
gamma |Deg(gamma) FakeDegree Fr(gamma) Label
____________________________________________
11.   |    1/2qP4        q^2         1   +,-
1.1   |  1/2qP2^2        qP4         1   +,+
.11   |       q^4        q^4         1
2.    |         1          1         1
.2    |    1/2qP4        q^2         1   -,+
B2    |  1/2qP1^2          0        -1   -,-
gap> Display(uc,rec(byFamily:=true));
Unipotent characters for B2
gamma |Deg(gamma) FakeDegree Fr(gamma) Label
____________________________________________
*.11  |       q^4        q^4         1
____________________________________________
11.   |    1/2qP4        q^2         1   +,-
*1.1  |  1/2qP2^2        qP4         1   +,+
.2    |    1/2qP4        q^2         1   -,+
B2    |  1/2qP1^2          0        -1   -,-
____________________________________________
*2.   |         1          1         1
gap> Display(uc,rec(items:=["n0","Name","Symbol"]));
Unipotent characters for B2
n^0 |gamma   Symbol
____________________
1   |  11.   (12,0)
2   |  1.1   (02,1)
3   |  .11 (012,12)
4   |   2.     (2,)
5   |   .2   (01,2)
6   |   B2   (012,)
```

98.3 UnipotentCharacter

`UnipotentCharacter(` W `,l)`

Constructs an object representing the unipotent character of the algebraic group associated to the Coxeter group or Coxeter coset W which is specified by l. There are 3 possibilities for l:if it is an integer, the l-th unipotent character of W is returned. If it is a string, the unipotent character of W whose name is l is returned. Finally, l can be a list of length the number of unipotent characters of W, which specifies the coefficient to give to each.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> u:=UnipotentCharacter(W,7);
[G2]=<G2[-1]>
gap> v:=UnipotentCharacter(W,"G2[E3]");
[G2]=<G2[E3]>
gap> w:=UnipotentCharacter(W,[1,0,0,-1,0,0,2,0,0,1]);
[G2]=<phi{1,0}>-<phi{1,3}''>+2<G2[-1]>+<G2[E3^2]>
```

98.4 Operations for Unipotent Characters

`+` Adds the specified characters.

`-` Subtracts the specified characters

`*` Multiplies a character by a scalar, or if given two unipotent characters returns their scalar product.

We go on from examples of the previous section:

```
gap> u+v;
[G2]=<G2[-1]>+<G2[E3]>
gap> w-2*u;
[G2]=<phi{1,0}>-<phi{1,3}''>+<G2[E3^2]>
gap> w*w;
7
```

`Degree`
returns the degree of the unipotent character.

```
gap> Degree(w);
q^5 - q^4 - q^3 - q^2 + q + 1
gap> Degree(u+v);
(5/6)*q^5 + (-1/2)*q^4 + (-2/3)*q^3 + (-1/2)*q^2 + (5/6)*q
```

`String` and `Print`
the formatting of unipotent characters is affected by the variable `CHEVIE.PrintUniChars`. It is a record; if the field `short` is bound (the default) they are printed in a compact form. If the field `long` is bound, they are printed one character per line:

```
gap> CHEVIE.PrintUniChars:=rec(long:=true);
rec(
  long := true )
gap> w;
```

```
[G2]=
<phi{1,0}>   1
<phi{1,6}>   0
<phi{1,3}'>  0
<phi{1,3}''> -1
<phi{2,1}>   0
<phi{2,2}>   0
<G2[-1]>     2
<G2[1]>      0
<G2[E3]>     0
<G2[E3^2]>   1
gap> CHEVIE.PrintUniChars:=rec(short:=true);;
```

`Frobenius(` *WF* `)`

If `WF` is a Coxeter coset associated to the Coxeter group W, the function `Frobenius(WF)` returns a function which does the corresponding automorphism on the unipotent characters

```
gap> W:=CoxeterGroup("D",4);WF:=CoxeterCoset(W,(1,2,4));
CoxeterGroup("D",4)
3D4
gap> u:=UnipotentCharacter(W,2);
[D4]=<11->
gap> Frobenius(WF)(u);
[D4]=<.211>
gap> Frobenius(WF)(u,-1);
[D4]=<11+>
```

98.5 UnipotentDegrees

`UnipotentDegrees(`W`,`q`)`

Returns the list of degrees of the unipotent characters of the finite reductive group (or Spetses) with Weyl group (or Spetsial reflection group) W, evaluated at q.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> q:=Indeterminate(Rationals);;q.name:="q";;
gap> UnipotentDegrees(W,q);
[ q^0, q^6, (1/3)*q^5 + (1/3)*q^3 + (1/3)*q,
  (1/3)*q^5 + (1/3)*q^3 + (1/3)*q, (1/6)*q^5 + (1/2)*q^4 + (2/3)*q^
    3 + (1/2)*q^2 + (1/6)*q, (1/2)*q^5 + (1/2)*q^4 + (1/2)*q^2 + (1/
    2)*q, (1/2)*q^5 + (-1/2)*q^4 + (-1/2)*q^2 + (1/2)*q,
  (1/6)*q^5 + (-1/2)*q^4 + (2/3)*q^3 + (-1/2)*q^2 + (1/6)*q,
  (1/3)*q^5 + (-2/3)*q^3 + (1/3)*q, (1/3)*q^5 + (-2/3)*q^3 + (1/3)*q ]
```

For a non-rational Spetses, `Indeterminate(Cyclotomics)` would be more appropriate.

98.6 Ennola

`Ennola(`W`)`

Let W be an irreducible spetsial reflection group or coset, and z the generator of the center of W, viewed as a root of unity. A property checked case-by case is that, for a unipotent character γ with polynomial generic degree $\deg\gamma(q)$ of the spets attached to W (this spets is a finite reductive group, for W a Weyl group, in which case $z=-1$ if -1 is in W), $\deg\gamma(zq)$ is equal to $\pm\deg En(\gamma)(q)$ for another unipotent character $En(\gamma)$, the Ennola transform of γ. The function returns the permutation-with-signs done by En on the unipotent degrees (as a permutation-with signs of `[1..Size(UnipotentCharacters(W))]`). The argument W must be irreducible.

The permutation-with-signs is not uniquely determined by the degrees since two of them may be equal, but is uniquely determined by some additional axioms that we do not recall here.

```
gap> Ennola(RootDatum("3D4"));
(3,-4)(5,-5)(6,-6)(7,-8)
gap> Ennola(ComplexReflectionGroup(14));
(2,43,-14,16,41,34)(3,35,40,18,-11,42)(4,-37,25,-17,-26,-36)(5,-6,-79)\
(7,-7)(8,-74)(9,-73)(10,-52,13,31,-50,29)(12,53,15,32,-51,-30)(19,71,7\
0,21,67,68,20,69,72)(22,-39,27,-33,-28,-38)(23,24,-66,-23,-24,66)(44,4\
6,49,-44,-46,-49)(45,48,47,-45,-48,-47)(54,-63,-55,-57,62,-56)(58,-65,\
-59,-61,64,-60)(75,-77)(76,-76)(78,-78)
```

The last example shows that it may happen that the order of z-Ennola (here 18) is greater than the order of z (here 6); this is related to the presence of irrationalities in the character table of the spetsial Hecke algebra of W.

98.7 CycPolUnipotentDegrees

`CycPolUnipotentDegrees(` W `)`

Taking advantage that the degrees of unipotent characters of the finite reductive group (or Spetses) with Weyl group (or Spetsial reflection group) W are products of cyclotomic polynomials, this function returns these degrees as a list of `CycPols` (see 106).

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> CycPolUnipotentDegrees(W);
[ 1, q^6, 1/3qP3P6, 1/3qP3P6, 1/6qP2^2P3, 1/2qP2^2P6, 1/2qP1^2P3,
  1/6qP1^2P6, 1/3qP1^2P2^2, 1/3qP1^2P2^2 ]
```

98.8 DeligneLusztigCharacter

`DeligneLusztigCharacter(` W, w `)`

This function returns the Deligne-Lusztig character $R_{\mathbf{T}}^{\mathbf{G}}(1)$ of the algebraic group $\mathbf{G}$ associated to the Coxeter group or Coxeter coset W. The torus $\mathbf{T}$ can be specified in 3 ways:if w is an integer, it represents the w-th conjugacy class (or ϕ-conjugacy class for a coset) of W. Otherwise w can be a Coxeter word or a Coxeter element, and it represents the class (or ϕ-class) of that element.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
```

```
gap> DeligneLusztigCharacter(W,3);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
gap> DeligneLusztigCharacter(W,W.1);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
gap> DeligneLusztigCharacter(W,[1]);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
gap> DeligneLusztigCharacter(W,[1,2]);
[G2]=<phi{1,0}>+<phi{1,6}>-<phi{2,1}>+<G2[-1]>+<G2[E3]>+<G2[E3^2]>
```

98.9 AlmostCharacter

`AlmostCharacter(` W `,` i `)`

This function returns the i-th almost unipotent character of the algebraic group $\mathbf{G}$ associated to the Coxeter group or Coxeter coset W. If χ is the i-th irreducible character of W, the i-th almost character is $R_\chi = \mathtt{W}^{-1}\sum_{w\in W}\chi(w)R^{\mathbf{G}}_{\mathbf{T}_w}(1)$, where $\mathbf{T}_w$ is the maximal torus associated to the conjugacy class (or ϕ-conjugacy class for a coset) of w.

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> AlmostCharacter(W,3);
[B2]=<.11>
gap> AlmostCharacter(W,1);
[B2]=1/2<11.>+1/2<1.1>-1/2<.2>-1/2<B2>
```

98.10 LusztigInduction

`LusztigInduction(` W `,` u `)`

u should be a unipotent character of a parabolic subcoset of the Coxeter coset W. It represents a unipotent character λ of a Levi $\mathbf{L}$ of the algebraic group $\mathbf{G}$ attached to W. The program returns the Lusztig induced $R^{\mathbf{G}}_{\mathbf{L}}(\lambda)$.

```
gap> W:=CoxeterGroup("G",2);;
gap> T:=CoxeterSubCoset(CoxeterCoset(W),[],W.1);
(q-1)(q+1)
gap> u:=UnipotentCharacter(T,1);
[(q-1)(q+1)]=<>
gap> LusztigInduction(CoxeterCoset(W),u);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
gap> DeligneLusztigCharacter(W,W.1);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
```

98.11 LusztigRestriction

`LusztigRestriction(` R `,` u `)`

u should be a unipotent character of a parent Coxeter coset W of which R is a parabolic subcoset. It represents a unipotent character γ of the algebraic group $\mathbf{G}$ attached to W, while R represents a Levi subgroup L. The program returns the Lusztig restriction ${}^*R^{\mathbf{G}}_{\mathbf{L}}(\gamma)$.

```
gap> W:=CoxeterGroup("G",2);;
```

```
gap> T:=CoxeterSubCoset(CoxeterCoset(W),[],W.1);
(q-1)(q+1)
gap> u:=DeligneLusztigCharacter(W,W.1);
[G2]=<phi{1,0}>-<phi{1,6}>-<phi{1,3}'>+<phi{1,3}''>
gap> LusztigRestriction(T,u);
[(q-1)(q+1)]=4<>
gap> T:=CoxeterSubCoset(CoxeterCoset(W),[],W.2);
(q-1)(q+1)
gap> LusztigRestriction(T,u);
[(q-1)(q+1)]=0
```

98.12 LusztigInductionTable

`LusztigInductionTable(`R`,`W`)`

R should be a parabolic subgroup of the Coxeter group W or a parabolic subcoset of the Coxeter coset W, in each case representing a Levi subgroup $\mathbf{L}$ of the algebraic group $\mathbf{G}$ associated to W. The function returns a table (modeled after `InductionTable`, see 103.13) representing the Lusztig induction $R_{\mathbf{L}}^{\mathbf{G}}$ between unipotent characters.

```
gap> W:=CoxeterGroup("B",3);;
gap> t:=Twistings(W,[1,3]);
[ ~A1xA1<3>.(q-1), ~A1xA1<3>.(q+1) ]
gap> Display(LusztigInductionTable(t[2],W));
Lusztig Induction from ~A1xA1<3>.(q+1) to B3
      |11,11 11,2 2,11 2,2
___________________________
111.  |    1   -1   -1   .
11.1  |   -1    .    1  -1
1.11  |    .    .   -1   .
.111  |   -1    .    .   .
21.   |    .    .    .   .
1.2   |    1   -1    .   1
2.1   |    .    1    .   .
.21   |    .    .    .   .
3.    |    .    .    .   1
.3    |    .    1    1  -1
B2:2  |    .    .    1  -1
B2:11 |    1   -1    .   .
```

98.13 DeligneLusztigLefschetz

`DeligneLusztigLefschetz(`h`)`

Here h is an element of a Hecke algebra associated to a Coxeter group W which itself is associated to an algebraic group $\mathbf{G}$. By results of Digne-Michel, for $g \in \mathbf{G}^F$, the number of fixed points of F^m on the Deligne-Lusztig variety associated to the element $w\phi$ of the Coxeter coset $W\phi$, have, for m sufficiently divisible, the form $\sum_\chi \chi_{q^m}(T_w\phi)R_\chi(g)$ where χ runs over the irreducible characters of $W\phi$, where R_χ is the corresponding almost character, and where χ_{q^m} is a character value of the Hecke algebra $\mathcal{H}(W\phi, q^m)$ of $W\phi$ with parameter

q^m. This expression is called the **Lefschetz character** of the Deligne-Lusztig variety. If we consider q^m as an indeterminate x, it can be seen as a sum of unipotent characters with coefficients character values of the generic Hecke algebra $\mathcal{H}(W\phi, x)$.

The function `DeligneLusztigLefschetz` takes as argument a Hecke element and returns the corresponding Lefschetz character. This is defined on the whole of the Hecke algebra by linearity. The Lefschetz character of various varieties related to Deligne-Lusztig varieties, like their completions or desingularisation, can be obtained by taking the Lefschetz character at various elements of the Hecke algebra.

```
gap> W:=CoxeterGroup("A",2);;
gap> q:=X(Rationals);;q.name:="q";;
gap> H:=Hecke(W,q);
Hecke(A2,q)
gap> T:=Basis(H,"T");
function ( arg ) ... end
gap> DeligneLusztigLefschetz(T(1,2));
[A2]=<111>-q<21>+q^2<3>
gap> DeligneLusztigLefschetz((T(1)+T())*(T(2)+T()));
[A2]=q<21>+(q^2+2q+1)<3>
```

The last line shows the Lefschetz character of the Samelson-Bott desingularisation of the Coxeter element Deligne-Lusztig variety.

We now show an example with a coset (corresponding to the unitary group).

```
gap> H:=Hecke(CoxeterCoset(W,(1,2)),q^2);
Hecke(2A2,q^2)
gap> T:=Basis(H,"T");
function ( arg ) ... end
gap> DeligneLusztigLefschetz(T(1));
[2A2]=-<11>-q<2A2>+q^2<2>
```

98.14 PermutationOnUnipotents

`PermutationOnUnipotents(`W`,`aut`)`

W is a reflection group or reflection coset representing a finite reductive group $\mathbf{G}^F$, and aut is an automorphism of $\mathbf{G}^F$ (for W a permutation group, this can be given as a permutation of the roots). The function returns the permutation of the unipotent characters of $\mathbf{G}^F$ induced by aut. This makes sense for Spetsial complex reflection groups and is implemented for them.

```
gap> WF:=RootDatum("3D4");
3D4
gap> PermutationOnUnipotents(Group(WF),WF.phi);
( 1, 7, 2)( 8,12, 9)
```

98.15 Families of unipotent characters

The blocks of the (rectangular) matrix $\langle R_\chi, \rho\rangle_{\mathbf{G}^F}$ when χ runs over $Irr(W)$ and ρ runs over the unipotent characters, are called the **Lusztig families**. When $\mathbf{G}$ is split and W is a

Coxeter group they correspond on the $Irr(W)$ side to two-sided Kazhdan-Lusztig cells — for split Spetses they correspond to Rouquier blocks of the Spetsial Hecke algebra. The matrix of scalar products $\langle R_\chi, \rho\rangle_{\mathbf{G}^F}$ can be completed to a square matrix $\langle A_{\rho'}, \rho\rangle_{\mathbf{G}^F}$ where $A_{\rho'}$ are the **characteristic functions of character sheaves** on $\mathbf{G}^F$; this square matrix is called the **Fourier matrix** of the family.

The `UnipotentCharacters` record in CHEVIE contains a field `.families`, a list of family records containing information on each family, including the Fourier matrix. Here is an example.

```
gap> W:=CoxeterGroup("G",2);;
gap> uc:=UnipotentCharacters(W);
UnipotentCharacters( G2 )
gap> uc.families;
[ Family("D(S3)",[5,6,4,3,8,7,9,10]), Family("C1",[1]),
  Family("C1",[2]) ]
gap> f:=last[1];
Family("D(S3)",[5,6,4,3,8,7,9,10])
gap> Display(f);
D(S3)
    label |eigen
-----------------------------------------------------
(1,1)     |    1 1/6  1/2  1/3  1/3  1/6  1/2  1/3  1/3
(g2,1)    |    1 1/2  1/2    0    0 -1/2 -1/2    0    0
(g3,1)    |    1 1/3    0  2/3 -1/3  1/3    0 -1/3 -1/3
(1,r)     |    1 1/3    0 -1/3  2/3  1/3    0 -1/3 -1/3
(1,eps)   |    1 1/6 -1/2  1/3  1/3  1/6 -1/2  1/3  1/3
(g2,eps)  |   -1 1/2 -1/2    0    0 -1/2  1/2    0    0
(g3,E3)   |   E3 1/3    0 -1/3 -1/3  1/3    0  2/3 -1/3
(g3,E3^2) | E3^2 1/3    0 -1/3 -1/3  1/3    0 -1/3  2/3
gap> f.charNumbers;
[ 5, 6, 4, 3, 8, 7, 9, 10 ]
gap> CharNames(uc){f.charNumbers};
[ "phi{2,1}", "phi{2,2}", "phi{1,3}''", "phi{1,3}'", "G2[1]",
  "G2[-1]", "G2[E3]", "G2[E3^2]" ]
```

The Fourier matrix is obtained by `Fourier(f)`; the field `f.charNumbers` holds the indices of the unipotent characters which are in the family. We obtain the list of eigenvalues of Frobenius for these unipotent characters by `Eigenvalues(f)`. The Fourier matrix and vector of eigenvalues satisfy the properties of **fusion data**, see below. The field `f.charLabels` is what is displayed in the column `labels` when displaying the family. It contains labels naturally attached to lines of the Fourier matrix. In the case of reductive groups, the family is always attached to the 98.22 of a small finite group and the `.charLabels` come from this construction.

98.16 Family

`Family(`*f* `[,` *charNumbers* `[,` *opt*`]])`

This function creates a new family in two possible ways.

In the first case f is a string which denotes a family known to CHEVIE. Examples are "S3", "S4", "S5" which denote the family obtained as the Drinfeld double of the symmetric group on 3,4,5 elements, or "C2" which denotes the Drinfeld double of the cyclic group of order 2.

In the second case f is already a family record.

The other (optional) arguments add information to the family record defined by the first argument. If given, the second argument becomes the field `.charNumbers`. If given, the third argument *opt* is a record whose fields are added to the resulting family record.

If *opt* has a field `signs`, this field should be a list of `1` and `-1`, and then the Fourier matrix is conjugated by the diagonal matrix of those signs. This is used in Spetses to adjust the matrix to the choice of signs of unipotent degrees.

```
gap> Display(Family("C2"));
C2
   label |eigen

-----------------------------------
(1,1)    |     1 1/2  1/2  1/2  1/2
(g2,1)   |     1 1/2  1/2 -1/2 -1/2
(1,eps)  |     1 1/2 -1/2  1/2 -1/2
(g2,eps) |    -1 1/2 -1/2 -1/2  1/2
gap> Display(Family("C2",[4..7],rec(signs:=[1,-1,1,-1])));
C2
   label |eigen signs

-----------------------------------------
(1,1)    |     1      1  1/2 -1/2 1/2 -1/2
(g2,1)   |     1     -1 -1/2  1/2 1/2 -1/2
(1,eps)  |     1      1  1/2  1/2 1/2  1/2
(g2,eps) |    -1     -1 -1/2 -1/2 1/2  1/2
```

98.17 Operations for families

`Fourier(`f`)`
: returns the Fourier matrix for the family f.

`Eigenvalues(`f`)`
: returns the list of eigenvalues of Frobenius associated to f.

`String(`f`)`, `Print(`f`)`
: give a short description of the family.

`Display(`f`)`
: displays the labels, eigenvalues and Fourier matrix for the family.

`Size(`f`)`
: how many characters are in the family.

f`*`g
: returns the tensor product of two families f and g; the Fourier matrix is the Kronecker product of the matrices for f and g, and the eigenvalues of Frobenius are the pairwise products.

`ComplexConjugate(`f`)`
: is a synonym for `OnFamily(`f`,-1)`.

98.18 IsFamily

`IsFamily(`*obj*`)`

returns `true` if *obj* is a family, and `false` otherwise.

```
gap> List(UnipotentCharacters(ComplexReflectionGroup(4)).families,IsFamily);
[ true, true, true, true ]
```

98.19 OnFamily

`OnFamily(`*f*`,`*p*`)`

f should be a family. This function has two forms.

In the first form, *p* is a permutation, and the function returns a copy of the family *f* with the Fourier matrix, eigenvalues of Frobenius, `.charLabels`, etc... permuted by *p*.

In the second form, *p* is an integer and `x->GaloisCyc(x,`*p*`)` is applied to the Fourier matrix and eigenvalues of Frobenius of the family.

```
gap> f:=UnipotentCharacters(ComplexReflectionGroup(3,1,1)).families[2];
Family("0011",[4,3,2])
gap> Display(f);
0011
label |eigen        1           2           3
________________________________________________
1     | E3^2  ER(-3)/3     ER(-3)/3    -ER(-3)/3
2     |    1  ER(-3)/3 (3-ER(-3))/6 (3+ER(-3))/6
3     |    1 -ER(-3)/3 (3+ER(-3))/6 (3-ER(-3))/6
gap> Display(OnFamily(f,(1,2,3)));
0011
label |eigen            3        1           2
________________________________________________
3     |    1 (3-ER(-3))/6 -ER(-3)/3 (3+ER(-3))/6
1     | E3^2    -ER(-3)/3  ER(-3)/3     ER(-3)/3
2     |    1 (3+ER(-3))/6  ER(-3)/3 (3-ER(-3))/6
gap> Display(OnFamily(f,-1));
'0011
label |eigen        1           2           3
________________________________________________
1     |   E3 -ER(-3)/3    -ER(-3)/3     ER(-3)/3
2     |    1 -ER(-3)/3 (3+ER(-3))/6 (3-ER(-3))/6
3     |    1  ER(-3)/3 (3-ER(-3))/6 (3+ER(-3))/6
```

98.20 FamiliesClassical

`FamiliesClassical(`*l*`)`

The list *l* should be a list of symbols as returned by the function `Symbols`, which classify the unipotent characters of groups of type `"B"`, `"C"` or `"D"`. `FamiliesClassical` returns the list of families determined by these symbols.

```
gap> FamiliesClassical(Symbols(3,1));
[ Family("0112233",[4]), Family("01123",[1,3,8]),
  Family("013",[5,7,10]), Family("022",[6]), Family("112",[2]),
  Family("3",[9]) ]
```

The above example shows the families of unipotent characters for the group B_3.

98.21 FamilyImprimitive

`FamilyImprimitive(`S`)`

S should be a symbol for a unipotent characters of an imprimitive complex reflection group `G(e,1,n)` or `G(e,e,n)`. The function returns the family of unipotent characters to which the character with symbol S belongs.

```
gap> FamilyImprimitive([[0,1],[1],[0]]);
Family("0011")
gap> Display(last);
0011
label |eigen        1            2            3
________________________________________________
1     | E3^2  ER(-3)/3    -ER(-3)/3     ER(-3)/3
2     |    1 -ER(-3)/3 (3-ER(-3))/6 (3+ER(-3))/6
3     |    1  ER(-3)/3 (3+ER(-3))/6 (3-ER(-3))/6
```

98.22 DrinfeldDouble

`DrinfeldDouble(`g`[,`opt`])`

Given a (usually small) finite group Γ, Lusztig has associated a family (a Fourier matrix, a list of eigenvalues of Frobenius) which describes the representation ring of the Drinfeld double of the group algebra of Γ, and for some appropriate small groups describes a family of unipotent characters. We do not explain the details of this construction, but explain how its final result building Lusztig's Fourier matrix, and a variant of it that we use in Spetses, from Γ.

The elements of the family are in bijection with the set $\mathcal{M}(\Gamma)$ of pairs (x,χ) taken up to Γ-conjugacy, where $x\in\Gamma$ and χ is an irreducible complex-valued character of $C_\Gamma(x)$. To such a pair $\rho=(x,\chi)$ is associated an eigenvalue of Frobenius defined by $\omega_\rho := \chi(x)/\chi(1)$. Lusztig then defines a Fourier matrix T whose coefficient is given, for $\rho=(x,\chi)$ and $\rho'=(x',\chi')$, by:

$$T_{\rho,\rho'} := \#C_\Gamma(x)^{-1}\sum_{\rho_1=(x_1,\chi_1)}\overline{\chi}_1(x)\chi(y_1)$$

where the sum is over all pairs $\rho_1\in\mathcal{M}(\Gamma)$ which are Γ-conjugate to ρ' and such that $y_1\in C_\Gamma(x)$. This coefficient also represents the scalar product $\langle\rho,\rho'\rangle_{\mathbf{G}^F}$ of the corresponding unipotent characters.

A way to understand the formula for $T_{\rho,\rho'}$ better is to consider another basis of the complex vector space with basis $\mathcal{M}(\Gamma)$, indexed by the pairs (x,y) taken up to Γ-conjugacy, where

x and y are commuting elements of Γ. This basis is called the basis of Mellin transforms, and given by:

$$(x, y) = \sum_{\chi \in Irr(C_\Gamma(x))} \chi(y)(x, \chi)$$

In the basis of Mellin transforms, the linear map T is given by $(x, y) \mapsto (x^{-1}, y^{-1})$ and the linear transformation which sends ρ to $\omega_\rho \rho$ becomes $(x, y) \mapsto (x, xy)$. These are particular cases of the permutation representation of $GL_2(\mathbb{Z})$ on the basis of Mellin transforms where $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ acts by $(x, y) \mapsto (x^a y^b, x^c y^d)$.

Fourier matrices in finite reductive groups are given by the above matrix T. But for non-rational Spetses, we use a different matrix S which in the basis of Mellin transforms is given by $(x, y) \mapsto (y^{-1}, x)$. Equivalently, the formula $S_{\rho,\rho'}$ differs from the formula for $T_{\rho,\rho'}$ in that there is no complex conjugation of χ_1; thus the matrix S is equal to T multiplied on the right by the permutation matrix which corresponds to $(x, \chi) \mapsto (x, \overline{\chi})$. The advantage of the matrix S over T is that the pair S, Ω satisfies directly the axioms for a fusion algebra (see below); also the matrix S is symmetric, while T is Hermitian.

Thus there are two variants of `DrinfeldDouble`:

`DrinfeldDouble(`g`,rec(lusztig:=true))`

returns a family containing Lusztig's Fourier matrix T, and an extra field `.perm` containing the permutation of the indices induced by $(x, \chi) \mapsto (x, \overline{\chi})$, which allows to recover S, as well as an extra field `.lusztig`, set to `true`.

`DrinfeldDouble(`g`)`

returns a family with the matrix S, which does not have fields `.lusztig` or `.perm`.

The family record `f` returned also has the fields:

`.group`
: the group Γ.

`.charLabels`
: a list of labels describing the pairs (x, χ), and thus also specifying in which order they are taken.

`.fourierMat`
: the Fourier matrix (the matrix S or T depending on the call).

`.eigenvalues`
: the eigenvalues of Frobenius.

`.xy`
: a list of pairs `[x,y]` which are representatives of the Γ-orbits of pairs of commuting elements.

`.mellinLabels`
: a list of labels describing the pairs `[x,y]`.

`.mellin`
: the base change matrix between the basis (x, χ) and the basis of Mellin transforms, so

that `f.fourierMat^(f.mellin^-1)` is the permutation matrix (for $(x, y) \mapsto (y^{-1}, x)$ or $(x, y) \mapsto (y^{-1}, x^{-1})$ depending on the call).

`.special`
the index of the special element, which is $(x, \chi) = (1, 1)$.

```
gap> f:=DrinfeldDouble(SymmetricGroup(3));
Family("D(Group((1,3),(2,3)))")
gap> Display(f);
D(Group((1,3),(2,3)))
   label |eigen
_____________________________________________________
(Id,Id)  |     1 1/6  1/6  1/3  1/2  1/2  1/3  1/3  1/3
(Id,X.2) |     1 1/6  1/6  1/3 -1/2 -1/2  1/3  1/3  1/3
(Id,X.3) |     1 1/3  1/3  2/3    0    0 -1/3 -1/3 -1/3
(2a,Id)  |     1 1/2 -1/2    0  1/2 -1/2    0    0    0
(2a,X.2) |    -1 1/2 -1/2    0 -1/2  1/2    0    0    0
(3a,Id)  |     1 1/3  1/3 -1/3    0    0  2/3 -1/3 -1/3
(3a,X.2) |    E3 1/3  1/3 -1/3    0    0 -1/3 -1/3  2/3
(3a,X.3) | E3^2 1/3  1/3 -1/3    0    0 -1/3  2/3 -1/3
gap> f:=DrinfeldDouble(SymmetricGroup(3),rec(lusztig:=true));
Family("LD(Group((1,3),(2,3)))")
gap> Display(f);
LD(Group((1,3),(2,3)))
   label |eigen
_____________________________________________________
(Id,Id)  |     1 1/6  1/6  1/3  1/2  1/2  1/3  1/3  1/3
(Id,X.2) |     1 1/6  1/6  1/3 -1/2 -1/2  1/3  1/3  1/3
(Id,X.3) |     1 1/3  1/3  2/3    0    0 -1/3 -1/3 -1/3
(2a,Id)  |     1 1/2 -1/2    0  1/2 -1/2    0    0    0
(2a,X.2) |    -1 1/2 -1/2    0 -1/2  1/2    0    0    0
(3a,Id)  |     1 1/3  1/3 -1/3    0    0  2/3 -1/3 -1/3
(3a,X.2) |    E3 1/3  1/3 -1/3    0    0 -1/3  2/3 -1/3
(3a,X.3) | E3^2 1/3  1/3 -1/3    0    0 -1/3 -1/3  2/3
```

98.23 NrDrinfeldDouble

`NrDrinfeldDouble(`g`)`

This function returns the number of elements that the family associated to the Drinfeld double of the group g would have, without computing it. The evident advantage is the speed.

```
gap> NrDrinfeldDouble(ComplexReflectionGroup(5));
378
```

98.24 FusionAlgebra

`FusionAlgebra(`f`)`

The argument f should be a family, or the Fourier matrix of a family. All the Fourier matrices S in CHEVIE are unitary, that is $S^{-1} = {}^t\overline{S}$, and have a **special** line s (the line of index s =`f.special` for a family f) such that no entry $S_{s,i}$ is equal to 0. Further, they have the property that the sums $C_{i,j,k} := \sum_l S_{i,l} S_{j,l} \overline{S}_{k,l}/S_{s,l}$ take integral values. Finally, S has the property that complex conjugation does a permutation with signs σ of the lines of S.

It follows that we can define a $\mathbb{Z}$-algebra A as follows:it has a basis b_i indexed by the lines of S, and has a multiplication defined by the fact that the coefficient of $b_i b_j$ on b_k is equal to $C_{i,j,k}$.

A is commutative, and has as unit the element b_s; the basis $\sigma(b_i)$ is dual to b_i for the linear form $(b_i, b_j) = C_{i,j,\sigma(s)}$.

```
gap> W:=ComplexReflectionGroup(4);;uc:=UnipotentCharacters(W);
UnipotentCharacters( G4 )
gap> f:=uc.families[4];
Family("RZ/6^2[1,3]",[2,4,10,9,3])
gap> A:=FusionAlgebra(f);
Fusion algebra dim.5
gap> b:=A.basis;
[ T(1), T(2), T(3), T(4), T(5) ]
gap> List(b,x->x*b);
[ [ T(1), T(2), T(3), T(4), T(5) ],
  [ T(2), -T(4)+T(5), T(1)+T(4), T(2)-T(3), T(3) ],
  [ T(3), T(1)+T(4), -T(4)+T(5), -T(2)+T(3), T(2) ],
  [ T(4), T(2)-T(3), -T(2)+T(3), T(1)+T(4)-T(5), -T(4) ],
  [ T(5), T(3), T(2), -T(4), T(1) ] ]
gap> CharTable(A);

    1        2        3   4   5

1   1  -ER(-3)   ER(-3)   2  -1
2   1        1        1   .   1
3   1       -1       -1   .   1
4   1        .        .  -1  -1
5   1   ER(-3)  -ER(-3)   2  -1
```

98.25 The d-Harish-Chandra series

d-Harish-Chandra series describe unipotent l-blocks of a finite reductive group $\mathbf{G}(\mathbb{F}_q)$ for $l|\Phi_d(q)$ (at least, when l is not too small which means mostly not a bad prime for $\mathbf{G}$). Some of the facts stated below are still partly conjectural, we do not try to distinguish precisely what has been established and what is still conjectural.

If $(\mathbf{L}, \lambda)$ is a d-cuspidal pair then the constituents of the Lusztig induced $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$ are called a d-Harish-Chandra series; they form the unipotent part of an l-block of $\mathbf{G}^F$. It is conjectured (and proven in some cases) that the $\mathbf{G}^F$-endomorphism algebra of the l-adic cohomology of the variety $\mathbf{X}$ which defines the Lusztig induction is a d-cyclotomic Hecke algebra $H_{\mathbf{G}}(\mathbf{L}, \lambda)$ for the group $W_{\mathbf{G}}(\mathbf{L}, \lambda) := N_{\mathbf{G}}(\mathbf{L}, \lambda)/\mathbf{L}$, which is a complex reflection group — here d-cyclotomic means that the parameters of $H_{\mathbf{G}}(\mathbf{L}, \lambda)$ are monomials in q and that $H_{\mathbf{G}}(\mathbf{L}, \lambda)$ specializes to the algebra of $W_{\mathbf{G}}(\mathbf{L}, \lambda)$ for $q \mapsto$`E(d)`.

It follows that the decomposition of the Lusztig induction is of the form

$$R_{\mathbf{L}}^{\mathbf{G}}(\lambda) = \sum_{\phi\in\mathrm{Irr}(W_{\mathbf{G}}(\mathbf{L},\lambda))} (-1)^{n_\phi}\phi(1)\gamma_\phi,$$

where γ_ϕ is a unipotent character of $\mathbf{G}^F$ attached to ϕ and where n_ϕ is the degree $H_c^{n_\phi}(\mathbf{X})$ where γ_ϕ occurss; and further for any ϕ we have $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)(1) = (-1)^{n_\phi}\gamma_\phi(1)S_\phi$ where S_ϕ is the Schur element of the character of $H_{\mathbf{G}}(\mathbf{L},\lambda)$ which deforms to ϕ. The function `Series` allows to explore a d-Harish-Chandra series.

```
gap> W:=RootDatum("3D4");;
gap> l:=CuspidalPairs(W,3);
[ [ 3D4, 8 ], [ (q^2+q+1)^2, 1 ] ]
gap> s:=Series(W,l[2][1],l[2][2],3);;
gap> Display(s);
E(3)-series R^{3D4}_{(q^2+q+1)^2}() H_G(L,c)=Hecke(G4,[[E3q^2,E3,E3q]])
  |      Name  WGLname eps family #
-----------------------------------
1 |  phi{1,0} phi{1,0}   1        1
2 |  phi{1,6} phi{1,4}   1        2
3 |  phi{2,2} phi{1,8}  -1        5
6 |phi{1,3}'' phi{2,5}   1        4
5 | phi{1,3}' phi{2,3}  -1        3
7 |  phi{2,1} phi{2,1}  -1        5
4 |    3D4[1] phi{3,2}   1        5
```

Above we explore the 3-series corresponding to $R_{\mathbf{T}}^{\mathbf{G}}(\mathrm{Id})$ where $\mathbf{G}$ is the triality group and $\mathbf{T}$ is the torus of type $(q^2+q+1)^2$. The group $W_{\mathbf{G}}(\mathbf{T})$ is the complex reflection group G_4. The displays shows in the column `Name` the name of the unipotent characters constituents of $R_{\mathbf{T}}^{\mathbf{G}}(\mathrm{Id})$, and in the first column the number of these characters in the list of unipotent characters. In the column `WGLname` the name of the character ϕ of $W_{\mathbf{G}}(\mathbf{T})$ corresponding to the unipotent character γ_ϕ is shown; in the column `eps` we show the sign $(-1)^{n_\phi}$. Finally in the last column we show in which family of unipotent characters is γ_ϕ.

The theory of d-Harish-Chandra series can be generalized to spetsial complex reflection groups using some axioms. We show below an example.

```
gap> W:=ComplexReflectionGroup(4);
ComplexReflectionGroup(4)
gap> l:=CuspidalPairs(W,3);
[ [ G4, 3 ], [ G4, 6 ], [ G4, 7 ], [ G4, 10 ], [ (q-1)(q-E3), 1 ] ]
gap> s:=Series(W,l[5][1],l[5][2],3);;
gap> Display(s);
E(3)-series R^{G4}_{(q-1)(q-E3)}() W_G(L,c)=Z6
  |    Name WGLname(mod 3) eps  param family #
-----------------------------------------------
1 |phi{1,0}             1   1  E3q^2        1
5 |phi{2,3}         -E3^2   1   -E3q        2
2 |phi{1,4}            E3  -1     E3        4
8 |    Z3:2            -1  -1 -E3^2q        2
9 |   Z3:11          E3^2  -1   E3^2        4
```

```
4 |phi{2,5}             -E3  -1    -E3         4
```

Above we explore the 3-series corresponding to the trivial character of the torus of type $(q-1)(q-E(3))$. For cyclic groups $W_{\mathbf{G}}(\mathbf{L},\lambda)$ we display the parameters in the table since they are associated to characters of $W_{\mathbf{G}}(\mathbf{L},\lambda)$. Finally the mention `(mod 3)` which appears in the `WGLname` column means that in this case the axioms leave an ambiguity in the correspondence between unipotent characters γ_ϕ and characters ϕ (as well as with parameters): the correspondence is known only up to a translation by 3 (in this case, the same as a global multiplication of all ϕ by -1).

Finally, we should note that if the reflection group or coset W is not defined over the integers, what counts is not cyclotomic polynomials but factors of them over the field of definition of W. In this case, one should not give as argument an integer `d` representing `E(d)` but specify a root of unity with a fraction `r/d` representing `E(d)^r`. For instance, in the above case we get a different answer with:

```
gap> CuspidalPairs(W,2/3);
[ [ G4, 2 ], [ G4, 5 ], [ G4, 7 ], [ G4, 10 ], [ (q-1)(q-E3^2), 1 ] ]
```

98.26 Series

`Series(` W`,` L`,` *cuspidal*`,` d`)`

If the reflection coset or group W corresponds to the algebraic group $\mathbf{G}$ and *cuspidal* to the d-cuspidal unipotent character λ of $\mathbf{L}$, constructs the d-series corresponding to $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$. If `s` is the result, it is a record with the following fields and functions:

`s.spets` the reflection group or coset W.

`s.levi` the subcoset L.

`s.cuspidal` the index of λ in `UnipotentCharacters(L)`.

`s.d` the value of `d` (represents a root of unity as a fraction in $[0,1[$).

`s.WGL` the group $W_{\mathbf{G}}(\mathbf{L},\lambda)$.

`s.RLG` the `UnipotentCharacter` given by $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$.

`s.eps` for each character ϕ of `s.WGL` the sign $(-1)^{n_\phi}$ in the cohomology of the variety defining `s.RLG` of the corresponding constituent γ_ϕ of `s.RLG`.

`Degree(s)` the generic degree of `s.RLG`, as a `CycPol`.

`CharNumbers(s)` the indices in `UnipotentCharacters(W)` of the constituents of `s.RLG`.

`Hecke(s)` the hecke algebra $H_{\mathbf{G}}(\mathbf{L},\lambda)$.

The function `Series` has another form

`Series(` W `[,` d `[,` *ad*`]])`

returns all the d-series of W corresponding to a d-eigenspace of dimension ad (default $d=1$, and if ad not given returns all d-series).

```
gap> W:=ComplexReflectionGroup(4);
ComplexReflectionGroup(4)
gap> Series(W,3,1);
[ E(3)-series R^{G4}_{(q-1)(q-E3)}() W_G(L,c)=Z6 ]
gap> s:=last[1];
```

```
E(3)-series R^{G4}_{(q-1)(q-E3)}() W_G(L,c)=Z6
gap> s.spets;
G4
gap> s.levi;
(q-1)(q-E3)
gap> s.cuspidal;
1
gap> s.d;
1/3
gap> Hecke(s);
Hecke(Z6,[[E3q^2,-E3q,E3,-E3^2q,E3^2,-E3]])
gap> Degree(s);
E3P1P2^2P"3P4P6
gap> s.RLG;
[G4]=<phi{1,0}>-<phi{1,4}>-<phi{2,5}>+<phi{2,3}>-<Z3:2>-<Z3:11>
gap> CharNumbers(s);
[ 1, 5, 2, 8, 9, 4 ]
gap> s.eps;
[ 1, 1, -1, -1, -1, -1 ]
gap> ReflectionName(s.WGL);
"Z6"
```

Chapter 99

Eigenspaces and d-Harish-Chandra series

Let $W\phi$ be a reflection coset on a vector space V and $Lw\phi$ a reflection subcoset where L is a parabolic subgroup (the fixator of a subspace of V). There are several interesting cases where the **relative group** $N_W(Lw\phi)/L$, or a subgroup of it normalizing some further data attached to L, is itself a reflection group.

A first example is the case where $\phi = 1$ and $w = 1$, W is the Weyl group of a finite reductive group $\mathbf{G}^F$ and the Levi subgroup $\mathbf{L}^F$ corresponding to L has a cuspidal unipotent character. Then $N_W(L)/L$ is a Coxeter group acting on the space $X(Z\mathbf{L}) \otimes \mathbb{R}$. A combinatorial characterization of such parabolic subgroups of Coxeter groups is that they are normalized by the longest element of larger parabolic subgroups (see [Lus76, 5.7.1]).

A second example is when L is trivial and $w\phi$ is a **ζ-regular element**, that is the ζ-eigenspace V_ζ of $w\phi$ contains a vector outside all the reflecting hyperplanes of W. Then $N_W(Lw\phi)/L = C_W(w\phi)$ is a reflection group in its action on V_ζ.

A similar but more general example is when V_ζ is the ζ-eigenspace of some element of the reflection coset $W\phi$, and is of maximal dimension among such possible ζ-eigenspaces. Then the set of elements of $W\phi$ which act by ζ on V_ζ is a certain subcoset $Lw\phi$, and $N_W(Lw\phi)/L$ is a reflection group in its action on V_ζ (see [LS99, 2.5]).

Finally, a still more general example, but which only occurs for Weyl groups or Spetsial reflection groups, is when $\mathbf{L}$ is a ζ-split Levi subgroup (which means that the corresponding subcoset $Lw\phi$ is formed of all the elements which act by ζ on some subspace V_ζ of V), and λ is a d-cuspidal unipotent character of $\mathbf{L}$ (which means that the multiplicity of ζ as a root of the degree of λ is the same as the multiplicity of ζ as a root of the generic order of the semi-simple part of $\mathbf{G}$); then $N_W(Lw\phi, \lambda)/L$ is a complex reflection group in its action on V_ζ.

Further, in the above cases the relative group describes the decomposition of a Lusztig induction.

When $\mathbf{G}^F$ is a finite reductive group, and λ a cuspidal unipotent character of the Levi subgroup $\mathbf{L}^F$, then the $\mathbf{G}^F$-endomorphism algebra of the Harish-Chandra induced representation $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$ is a Hecke algebra attached to the group $N_W(L)/L$, thus the dimension of the characters of this group describe the multiplicities in the Harish-Chandra induced.

Similarly, when $\mathbf{L}$ is a ζ-split Levi subgroup, and λ is a d-cuspidal unipotent character of $\mathbf{L}$ then (conjecturally) the $\mathbf{G}^F$-endomorphism algebra of the Lusztig induced $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$ is a cyclotomic Hecke algebra for to the group $N_W(Lw\phi,\lambda)/L$. The constituents of $R_{\mathbf{L}}^{\mathbf{G}}(\lambda)$ are called a ζ-Harish-Chandra series. In the case of rational groups or cosets, corresponding to finite reductive groups, the conjugacy class of $Lw\phi$ depends only on the order d of ζ, so one also talks of d-Harish-Chandra series. These series correspond to ℓ-blocks where l is a prime divisor of $\Phi_d(q)$ which does not divide any other cyclotomic factor of the order of $\mathbf{G}^F$.

The CHEVIE functions described in this chapter allow to explore these situations.

99.1 RelativeDegrees

`RelativeDegrees(`*WF* `[,`*d*`])`

Let *WF* be a reflection group or a reflection coset. Here d specifies a root of unity ζ: either d is an integer and specifies ζ `=E(d)` or is a fraction smaller a/b with $0 < a < b$ and specifies ζ `=E(b)^a`. If omitted, d is taken to be 0, specifying $\zeta = 1$. Then if V_ζ is the ζ-eigenspace of some element of *WF*, and is of maximal dimension among such possible ζ-eigenspaces, and W is the group of *WF* then $N_W(V_\zeta)/C_W(V_\zeta)$ is a reflection group in its action on V_ζ. The function `RelativeDegrees` returns the reflection degrees of this complex reflection group, which are a subset of those of W.

The point is that these degrees are obtained quickly by invariant-theoretic computations: if $(d_1,\varepsilon_1),\ldots,(d_n,\varepsilon_n)$ are the generalized degrees of *WF* they are the d_i such that $\zeta^{d_i} = \varepsilon_i$.

```
gap> W:=CoxeterGroup("E",8);
CoxeterGroup("E",8)
gap> RelativeDegrees(W,4);
[ 8, 12, 20, 24 ]
```

99.2 RegularEigenvalues

`RegularEigenvalues(`*W*`)`

Let W be a reflection group or a reflection coset. A root of unity ζ is a **regular eigenvalue** for W if some element of W has a ζ-eigenvector which lies outside of the reflecting hyperplanes. The function `RelativeDegree` returns a list describing the regular eigenvalues for W. If all the primitive n-th roots of unity are regular eigenvalues, then n is put on the result list. Otherwise the fractions a/n are added to the list for each a such that $E(n)^a$ is a primitive n-root of unity and a regular eigenvalue for W.

```
gap> W:=CoxeterGroup("E",8);;
gap> RegularEigenvalues(W);
[ 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30 ]
gap> W:=ComplexReflectionGroup(6);;
gap> L:=Twistings(W,[2])[4];
Z3[I]<2>.(q-I)
gap> RegularEigenvalues(L);
[ 1/4, 7/12, 11/12 ]
```

99.3 PositionRegularClass

`PositionRegularClass(` *WF* `[,`*d*`])`

Let *WF* be a reflection group or a reflection coset. Here d specifies a root of unity ζ: either d is an integer and specifies ζ `=E(d)` or is a fraction smaller a/b with $0 < a < b$ and specifies ζ `=E(b)^a`. If omitted, d is taken to be 0, specifying $\zeta = 1$. The root ζ should be a regular eigenvalue for *WF* (see 99.2). The function returns the index of the conjugacy class of *WF* which has a ζ-regular eigenvector.

```
gap>  W:=CoxeterGroup("E",8);;
gap> PositionRegularClass(W,30);
65
gap> W:=ComplexReflectionGroup(6);;
gap> L:=Twistings(W,[2])[4];
Z3[I]<2>.(q-I)
gap> PositionRegularClass(L,7/12);
2
```

99.4 EigenspaceProjector

`EigenspaceProjector(` *WF*`,` *w* `,`*d*`)`

Let *WF* be a reflection group or a reflection coset. Here d specifies a root of unity ζ: either d is an integer and specifies ζ `=E(d)` or is a fraction smaller a/b with $0 < a < b$ and specifies ζ `=E(b)^a`. The function returns the unique w-invariant projector on the ζ-eigenspace of w.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> w:=EltWord(W,[1..3]);
( 1,12, 3, 2)( 4,11,10, 5)( 6, 9, 8, 7)
gap> EigenspaceProjector(W,w,1/4);
[ [ 1/4+1/4*E(4), 1/2*E(4), -1/4+1/4*E(4) ],
  [ 1/4-1/4*E(4), 1/2, 1/4+1/4*E(4) ],
  [ -1/4-1/4*E(4), -1/2*E(4), 1/4-1/4*E(4) ] ]
gap> RankMat(last);
1
```

99.5 SplitLevis

`SplitLevis(` *WF* `[,` *d* `[,`*ad*`]])`

Let *WF* be a reflection group or a reflection coset. If W is a reflection group it is treated as the trivial coset `Spets(W)`.

Here d specifies a root of unity ζ: either d is an integer and specifies ζ`=E(d)` or is a fraction a/b with $0 < a < b$ and specifies ζ`=E(b)^a`. If omitted, d is taken to be 0, specifying $\zeta = 1$.

A **Levi** is a subcoset of the form W_1F_1 where W_1 is a **parabolic subgroup** of W, that is the centralizer of some subspace of V.

The function returns a list of representatives of conjugacy classes of d-split Levis of W. A d-split Levi is a subcoset of *WF* formed of all the elements which act by ζ on a given subspace V_ζ. If the additional argument ad is given, it returns only those subcosets such that the common ζ-eigenspace of their elements is of dimension ad. These notions make sense and thus are implemented for any complex reflection group.

In terms of algebraic groups, an F-stable Levi subgroup of the reductive group $\mathbf{G}$ is d-split if and only if it is the centralizer of the Φ_d-part of its center. When $d = 1$, we get the notion of a **split** Levi, which is the same as a Levi sugroup of an F-stable parabolic subgroup of $\mathbf{G}$.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> SplitLevis(W,4);
[ A3, (q+1)(q^2+1) ]
gap> 3D4:=CoxeterCoset(CoxeterGroup("D",4),(1,2,4));
3D4
gap> SplitLevis(3D4,3);
[ 3D4, A2<1,3>.(q^2+q+1), (q^2+q+1)^2 ]
gap> W:=CoxeterGroup("E",8);
CoxeterGroup("E",8)
gap> SplitLevis(W,4,2);
[ D4<3,2,4,5>.(q^2+1)^2, (A1xA1)<5,7>x(A1xA1)<2,3>.(q^2+1)^2,
  2(A2xA2)<3,1,5,6>.(q^2+1)^2 ]
gap> SplitLevis(ComplexReflectionGroup(5));
[ G5, Z3.(q-1), Z3<2>.(q-1), (q-1)^2 ]
```

99.6 CuspidalUnipotentCharacters

`CuspidalUnipotentCharacters(` *WF* `[,d])`

Let *WF* be a reflection group or a reflection coset. If W is a reflection group it is treated as the trivial coset `Spets(W)`.

A unipotent character γ of the corresponding finite reductive group $\mathbf{G}$ is d-cuspidal if its Lusztig restriction to any proper d-split Levi is zero. When $d = 0$ we recover the usal notion of cuspidal character. Equivalently the Φ_d-part of the generic degree of γ is equal to the Φ_d-part of the generic order of the adjoint group of $\mathbf{G}$. This makes sense for any Spetsial complex reflection group and is implemented for them. This also generalises to an arbitrary root of unity `E(d)^r` (which should be given as the fraction `r/d`) by asking that the generic degree of γ and the generic order of the adjoin group of $\mathbf{G}$ have zeros of the same order at `E(d)^r`.

The function returns the list of indices of unipotent characters which are d-cuspidal. If d is omitted, it is taken to be 0.

```
gap> W:=CoxeterGroup("D",4);
CoxeterGroup("D",4)
gap> CuspidalUnipotentCharacters(W);
[ 14 ]
gap> CuspidalUnipotentCharacters(W,3);
[ 1, 2, 5, 7, 8, 9, 12, 14 ]
gap> CuspidalUnipotentCharacters(ComplexReflectionGroup(4),3);
[ 3, 6, 7, 10 ]
```

99.7 CuspidalPairs

`CuspidalPairs(` W `[,` d `[,` ad `]])`

returns the pairs (LF, λ) where LF is a d-split Levi (with d-center of dimension ad if ad is given) and λ is a d-cuspidal character of LF. If d is omitted it is assumed to be 1, which means ordinary cuspidal pairs. The character λ is returned as its index amongst unipotent characters. `d` can represent an arbitrary root of unity (if integral, it represents `E(d)` and if a fraction `r/d` represents `E(d)^r`).

```
gap> CuspidalPairs(CoxeterGroup("F",4));
[ [ F4, 31 ], [ F4, 32 ], [ F4, 33 ], [ F4, 34 ], [ F4, 35 ],
  [ F4, 36 ], [ F4, 37 ], [ B2<3,2>.(q-1)^2, 6 ], [ (q-1)^4, 1 ] ]
gap> CuspidalPairs(ComplexReflectionGroup(4),3);
[ [ G4, 3 ], [ G4, 6 ], [ G4, 7 ], [ G4, 10 ], [ (q-1)(q-E3), 1 ] ]
```

Chapter 100

Unipotent classes of reductive groups

CHEVIE contains information about the unipotent conjugacy classes of a connected reductive group over an algebraically closed field k, and various invariants attached to them. The unipotent classes depend on the characteristic of k; their classification differs when the characteristic is not **good** (that is, when it divides one of the coefficients of the highest root). In good characteristic, the unipotent classes are in bijection with nilpotent orbits on the Lie algebra.

CHEVIE computes the following information for the class C of a unipotent element u:

- the centralizer $C_{\mathbf{G}}(u)$, characterized by its reductive part, its group of components $A(u) := C_{\mathbf{G}}(u)/C_{\mathbf{G}}(u)^0$, and the dimension of its radical.
- in good characteristic, the Dynkin-Richardson diagram of C.
- the Springer correspondence, attaching characters of the Weyl group or of relative Weyl groups to each character of $A(u)$.

The Dynkin-Richarson diagram is attached to a nilpotent element e of the Lie algebra $\mathfrak{g}$. By the Jacobson-Morozov theorem there exists an $\mathfrak{s}l_2$ subalgebra of $\mathfrak{g}$ containing e as the element $\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$. Let $\mathbf{S}$ be the torus $\begin{pmatrix} h & 0 \\ 0 & h^{-1} \end{pmatrix}$ of SL_2 and let $\mathbf{T}$ be a maximal torus containing $\mathbf{S}$ so that $\mathbf{S}$ is the image of a one-parameter subgroup $\sigma \in Y(\mathbf{T})$. Consider the root decomposition $\mathfrak{g} = \sum_{\alpha\in\Sigma} \mathfrak{g}_\alpha$ given by $\mathbf{T}$; then $\alpha \mapsto \langle\sigma,\alpha\rangle$ defines a linear form on Σ, determined by its value on simple roots. It is possible to choose a system of simple roots Π so that $\langle\sigma,\alpha\rangle \geq 0$ for $\alpha \in \Pi$, and then $\langle\sigma,\alpha\rangle \in \{0,1,2\}$ for any $\alpha \in \Pi$. The Dynkin diagram of Π decorated by these values $0,1,2$ is called the Dynkin-Richardson diagram of e, and in good characteristic is a complete invariant of its $\mathfrak{g}$-orbit.

Let $\mathcal{B}$ be the variety of all Borel subgroups and let $\mathcal{B}_u$ be the subvariety of Borel subgroups containing the unipotent element u. Then $\dim C_{\mathbf{G}}(u) = rank\mathbf{G} + 2\dim\mathcal{B}_u$ and in good characteristic this dimension can be computed from the Dynkin-Richardson diagram:the dimension of the class of u is the number of roots α such that $\langle\sigma,\alpha\rangle \notin \{0,1\}$.

We describe now the Springer correspondence. Indecomposable locally constant $\mathbf{G}$-equivariant sheaves on C, called **local systems**, are parameterized by irreducible characters of $A(u)$. The **ordinary** Springer correspondence is a bijection between irreducible characters of the Weyl group and a large subset of the local systems which contains all trivial local systems (those parameterized by the trivial character of $A(u)$ for each u). More generally, the **generalized** Springer correspondence associates to each local system a (unique up to $\mathbf{G}$-conjugacy) **cuspidal pair** of a Levi subgroup $\mathbf{L}$ of $\mathbf{G}$ and a local system on an unipotent class of $\mathbf{L}$, such that the set of local systems associated to a given cuspidal pair is parameterized by the characters of the relative Weyl group $W_{\mathbf{G}}(\mathbf{L}) := N_{\mathbf{G}}(\mathbf{L})/\mathbf{L}$. There are only few cuspidal pairs.

The Springer correspondence gives information on the character values of a finite reductive groups as follows: assume that k is the algebraic closure of a finite field $\mathcal{F}_q$ and that F is the Frobenius attached to an $\mathcal{F}_q$-structure of $\mathbf{G}$. Let C be an F-stable unipotent class and let $u \in C^F$; we call C the **geometric class** of u and the $\mathbf{G}^F$-classes inside C^F are parameterized by the F-conjugacy classes of $A(u)$, denoted $H^1(F, A(u))$ (most of the time we can find u such that F acts trivially on $A(u)$ and then $H^1(F, A(u))$ is just the conjugacy classes of $A(u)$). To an F-stable character φ of $A(u)$ we associate the **characteristic function** of the corresponding local system (actually associated to an extension $\tilde{\varphi}$ of ϕ to $A(u).F$); it is a class function $Y_{u,\varphi}$ on $\mathbf{G}^F$ which can be normalized so that: $Y_{u,\varphi}(u_1) = \tilde{\varphi}(cF)$ if u_1 is geometrically conjugate to u and its $\mathbf{G}^F$-class is parameterized by the F-conjugacy class cF of $A(u)$, otherwise $Y_{u,\varphi}(u_1) = 0$. If the pair u, φ corresponds via the Springer correspondence to the character χ of $W_{\mathbf{G}}(\mathbf{L})$, then $Y_{u,\varphi}$ is also denoted Y_χ. There is another important class of functions indexed by local systems:to a local system on class C is attached an intersection cohomology complex, which is a complex of sheaves supported on the closure $\overline{C}$. To such a complex of sheaves is associated its **characteristic function**, a class function of $\mathbf{G}^F$ obtained by taking the alternating trace of the Frobenius acting on the stalks of the cohomology sheaves. If Y_ψ is the characteristic function of a local system, the characteristic function of the corresponding intersection cohomology complex is denoted by X_ψ. This function is supported on $\overline{C}$, and Lusztig has shown that $X_\psi = \sum_\phi P_{\psi,\chi} Y_\chi$ where $P_{\psi,\chi}$ are integer polynomials in q and Y_χ are attached to local systems on classes lying in $\overline{C}$.

Lusztig and Shoji have given an algorithm to compute the matrix $P_{\psi,\chi}$, which is implemented in CHEVIE. The relationship with characters of $\mathbf{G}(\mathcal{F}_q)$, taking to simplify the ordinary Springer correspondence, is that the restriction to the unipotent elements of the almost character R_χ is equal to $q^{b_\chi} X_\chi$, where b_χ is $\dim \mathcal{B}_u$ for an element u of the class C such that the support of χ is $\overline{C}$. The restriction of the Deligne-Lusztig characters R_w to the unipotents are called the **Green functions** and can also be computed by CHEVIE. The values of all unipotent characters on unipotent elements can also be computed in principle by applying Lusztig's Fourier transform matrix (see the section on the Fourier matrix) but there is a difficulty in that the X_χ must be first multiplied by some roots of unity which are not known in all cases (and when known may depend on the congruence class of q modulo some small primes).

We illustrate these computations on some examples:

```
gap> W:=CoxeterGroup("A",3,"sc");
CoxeterGroup("A",3,"sc")
gap> uc:=UnipotentClasses(W);
UnipotentClasses( A3 )
```

```
gap> Display(uc);
1111<211<22<31<4
   u |D-R dBu B-C        C(u) A3() A1(2A1)/-1 .(A3)/I .(A3)/-I
______________________________________________________________
4    |222   0 222       q^3.Z4  1:4       -1:2      I:      -I:
31   |202   1 22.    q^4.(q-1)   31
22   |020   2 2.2    q^4.A1.Z2 2:22      11:11
211  |101   3 2.. q^5.A1.(q-1)  211
1111 |000   6 ...           A3 1111
```

In `CoxeterGroup("A",3,"sc")` the `"sc"` specifies that we are working with the simply connected group, that is sl_n; another syntax for the same group is `RootDatum("sl",4)`. The first column in the table gives the name of the unipotent class, which here is a partition describing the Jordan form. The partial order on unipotent classes given by Zariski closure is given before the table. The column `D-R`, displayed only in good characteristic, gives the Dynkin-Richardson diagram for each class; the column `dBu` gives the dimension of the variety $\mathcal{B}_u$. The column `B-C` gives the Bala-Carter classification of u, that is in the case of sl_4 it displays u as a regular unipotent in a Levi subgroup by giving the Dynkin-Richardson diagram of a regular unipotent (all 2's) at entries corresponding to the Levi and `.` at entries which do not correspond to the Levi. The column `C(u)` describes the group $C_{\mathbf{G}}(u)$:a power q^d describes that the unipotent radical of $C_{\mathbf{G}}(u)$ has dimension d (thus q^d rational points); then follows a description of the reductive part of the neutral component of $C_{\mathbf{G}}(u)$, given by the name of its root datum. Then if $C_{\mathbf{G}}(u)$ is not connected, the description of $A(u)$ is given using another vocabulary:a cyclic group of order 4 is given as `Z4`, and a symmetric group on 3 points would be given as `S3`.

For instance, the first class `4` has $C_{\mathbf{G}}(u)^0$ unipotent of dimension 3 and $A(u)$ equal to `Z4`, the cyclic group of order 4. The class `22` has $C_G(u)$ with unipotent radical of dimension 4, reductive part of type `A1` and $A(u)$ is `Z2`, that is the cyclic group of order 2. The other classes have $C_{\mathbf{G}}(u)$ connected. For the class `31` the reductive part of $C_G(u)$ is a torus of rank 1.

Then there is one column for each **Springer series**, giving for each class the pairs `a:b` where `a` is the name of the character of $A(u)$ describing the local system involved and `b` is the name of the character of the (relative) Weyl group corresponding by the Springer correspondence. At the top of the column is written the name of the relative Weyl group, and in brackets the name of the Levi affording a cuspidal local system; next, separated by a `/` is a description of the central character associated to the Springer series (omitted if this central character is trivial):all local systems in a given Springer series have same restriction to the center of $\mathbf{G}$. To find what the picture becomes for another algebraic group in the same isogeny class, for instance the adjoint group, one simply discards the Springer series whose central character becomes trivial on the center of $\mathbf{G}$; and each group $A(u)$ has to be quotiented by the common kernel of the remaining characters. Here is the table for the adjoint group:

```
gap> Display(UnipotentClasses(CoxeterGroup("A",3)));
1111<211<22<31<4
   u |D-R dBu B-C        C(u) A3()
__________________________________
4    |222   0 222          q^3    4
31   |202   1 22.    q^4.(q-1)   31
```

```
22   |020   2 2.2       q^4.A1   22
211  |101   3 2.. q^5.A1.(q-1)  211
1111 |000   6 ...           A3 1111
```

Here is another example:

```
gap> W:=CoxeterGroup("G",2);;
gap> Display(UnipotentClasses(W));
1<A1<~A1<G2(a1)<G2
     u |D-R dBu B-C   C(u)                  G2() .(G2)
________________________________________________________
G2     | 22   0  22    q^2                 phi{1,0}
G2(a1) | 20   1  20 q^4.S3 21:phi{1,3}' 3:phi{2,1}  111:
~A1    | 01   2  .2 q^3.A1                 phi{2,2}
A1     | 10   3  2. q^5.A1               phi{1,3}''
1      | 00   6  ..     G2                 phi{1,6}
```

which illustrates that on class `G2(a1)` there are two local systems in the principal series of the Springer correspondence, and a further cuspidal local system. Also, from the `B-C` column, we see that that class is not in a proper Levi, in which case the Bala-Carter diagram coincides with the Dynkin-Richardson diagram.

The characteristics 2 and 3 are not good for `G2`. To get the unipotent classes and the Springer correspondence in bad characteristic, one gives a second argument to the function `UnipotentClasses`:

```
gap> Display(UnipotentClasses(W,3));
1<A1,(~A1)3<~A1<G2(a1)<G2
     u |dBu   C(u)       G2() .(G2) .(G2) .(G2)
_________________________________________________
G2     |  0 q^2.Z3 1:phi{1,0}        E3: E3^2:
G2(a1) |  1 q^4.Z2 2:phi{2,1}   11:
~A1    |  2    q^6   phi{2,2}
A1     |  3 q^5.A1 phi{1,3}''
(~A1)3 |  3 q^5.A1  phi{1,3}'
1      |  6     G2   phi{1,6}
```

The function `ICCTable` gives the transition matrix between the functions X_χ and Y_ψ.

```
gap> Display(ICCTable(UnipotentClasses(W)));
Coefficients of X_phi on Y_psi for G2

            |G2 G2(a1)(21) G2(a1) ~A1 A1   1
_____________________________________________
Xphi{1,0}   | 1          0      1   1  1   1
Xphi{1,3}'  | 0          1      0   1  0 q^2
Xphi{2,1}   | 0          0      1   1  1  P8
Xphi{2,2}   | 0          0      0   1  1  P4
Xphi{1,3}'' | 0          0      0   0  1   1
Xphi{1,6}   | 0          0      0   0  0   1
```

Here the row labels and the column labels show the two ways of indexing local systems: the row labels give the character of the relative Weyl group and the column labels give the class and the name of the local system as a character of $A(u)$: for instance, `G2(a1)` is the trivial local system of the class `G2(a1)`, while `G2(a1)(21)` is the local system on that class corresponding to the 2-dimensional character of $A(u) = A_2$.

100.1 UnipotentClasses

`UnipotentClasses(` W `[,` p `])`

W should be a `CoxeterGroup` record for a Weyl group or `RootDatum` describing a reductive algebraic group $\mathbf{G}$. The function returns a record containing information about the unipotent classes of $\mathbf{G}$ in characteristic p (if omitted, p is assumed to be any good characteristic for $\mathbf{G}$). This contains the following fields:

`group`
: a pointer to W

`p`
: the characteristic of the field for which the unipotent classes were computed. It is `0` for any good characteristic.

`orderClasses`
: a list describing the Hasse diagram of the partial order induced on unipotent classes by the closure relation. That is `.orderclasses[i]` is the list of `j` such that $\overline{C}_j \supsetneq \overline{C}_i$ and there is no class C_k such that $\overline{C}_j \supsetneq \overline{C}_k \supsetneq \overline{C}_i$.

`classes`
: a list of records holding information for each unipotent class (see below).

`springerSeries`
: a list of records, each of which describes a Springer series of $\mathbf{G}$.

The records describing individual unipotent classes have the following fields:

`name`
: the name of the unipotent class.

`parameter` a parameter describing the class (for example, a partition describing the Jordan form, for classical groups).

`Au`
: the group $A(u)$.

`dynkin`
: present in good characteristic; contains the Dynkin-Richardson diagram, given as a list of 0,1,2 describing the coefficient on the corresponding simple root.

`red`
: the reductive part of $C_{\mathbf{G}}(u)$.

`dimBu`
: the dimension of the variety $\mathcal{B}_u$.

The records for classes contain additional fields for certain groups:for instance, the names given to classes by Mizuno in E_6, E_7, E_8 or by Shoji in F_4.

The records describing individual Springer series have the following fields:

`levi`
: the indices of the reflections corresponding to the Levi subgroup $\mathbf{L}$ where lives the cuspidal local system ι from which the Springer series is induced.

`relgroup`
: The relative Weyl group $N_{\mathbf{G}}(\mathbf{L}, \iota)/\mathbf{L}$. The first series is the principal series for which `.levi=[]` and `.relgroup=W`.

`locsys`
: a list of length `NrConjugacyClasses(.relgroup)`, holding in `i`-th position a pair describing which local system corresponds to the `i`-th character of $N_{\mathbf{G}}(\mathbf{L}, \iota)$. The first element of the pair is the index of the concerned unipotent class `u`, and the second is the index of the corresponding character of $A(u)$.

`Z`
: the central character associated to the Springer series, specified by its value on the generators of the centre.

```
gap> W:=CoxeterGroup("A",3,"sc");;
gap> uc:=UnipotentClasses(W);
UnipotentClasses( A3 )
gap> uc.classes;
[ UnipotentClass(1111), UnipotentClass(211), UnipotentClass(22),
  UnipotentClass(31), UnipotentClass(4) ]
gap> PrintRec(uc.classes[3]);
rec(
  name      := 22,
  Au        := CoxeterGroup("A",1),
  dimBu     := 2,
  parameter := [ 2, 2 ],
  balacarter:= [ 1, 3 ],
  dynkin    := [ 0, 2, 0 ],
  red       := ReflectionSubgroup(CoxeterGroup("A",1), [ 1 ]),
  AuAction  := A1,
  dimunip   := 4,
  dimred    := 3,
  operations:= UnipotentClassOps )
gap> uc.orderClasses;
[ [ 2 ], [ 3 ], [ 4 ], [ 5 ], [  ] ]
gap> uc.springerSeries;
[ rec(
      relgroup := A3,
      Z := [ 1 ],
      levi := [  ],
      locsys := [ [ 1, 1 ], [ 2, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5, 1 ] ] )
    , rec(
      relgroup := A1,
      Z := [ -1 ],
      levi := [ 1, 3 ],
      locsys := [ [ 3, 1 ], [ 5, 3 ] ] ), rec(
      relgroup := .,
```

```
        Z := [ E(4) ],
        levi := [ 1, 2, 3 ],
        locsys := [ [ 5, 2 ] ] ), rec(
        relgroup := .,
        Z := [ -E(4) ],
        levi := [ 1, 2, 3 ],
        locsys := [ [ 5, 4 ] ] ) ]
```

The `Display` and `Format` functions for unipotent classes accept all the options of `FormatTable`, `CharNames`. Giving the option `mizuno` (resp. `shoji`) uses the names given by Mizuno (resp. Shoji) for unipotent classes. Moreover, there is also an option `fourier` which gives the correspondence tensored with the sign character of each relative Weyl group, which is the correspondence obtained via a Fourier-Deligne transform (here we assume that p is very good, so that there is a nondegenerate invariant bilinear form on the Lie algebra, and also one can identify nilpotent orbits with unipotent classes).

Here is how to display only the ordinary Springer correspondence of the unipotent classes of `E6` using the notations of Mizuno for the classes and those of Frame for the characters of the Weyl group and of Spaltenstein for the characters of `G2` (this is convenient for checking our data with the original paper of Spaltenstein):

```
gap> uc:=UnipotentClasses(CoxeterGroup("E",6));;
gap> Display(uc,rec(columns:=[1..5],mizuno:=true,frame:=true,
> spaltenstein:=true));
1<A1<2A1<3A1<A2<A2+A1<A2+2A1<2A2+A1<A3+A1<D4(a1)<D4<D5(a1)<A5+A1<D5<E6\
(a1)<E6
A2+A1<2A2<2A2+A1
A2+2A1<A3<A3+A1
D4(a1)<A4<A4+A1<A5<A5+A1
A4+A1<D5(a1)
     u |   D-R dBu    B-C            C(u)                 E6()
_______________________________________________________________
E6     |222222   0 222222             q^6                   1p
E6(a1) |222022   1 222022             q^8                   6p
D5     |220202   2 22222.       q^9.(q-1)                  20p
A5+A1  |200202   3 200202         q^12.Z2         11:15p 2:30p
A5     |211012   4 2.2222         q^11.A1                  15q
D5(a1) |121011   4 22202.      q^13.(q-1)                  64p
A4+A1  |111011   5 2222.2      q^15.(q-1)                  60p
D4     |020200   6 .2222.         q^10.A2                  24p
A4     |220002   6 2222..   q^14.A1.(q-1)                  81p
D4(a1) |000200   7 .2202. q^18.(q-1)^2.S3 111:20s 3:80s 21:90s
A3+A1  |011010   8 22.22.   q^18.A1.(q-1)                  60s
2A2+A1 |100101   9 222.22         q^21.A1                  10s
A3     |120001  10 2.22..   q^15.B2.(q-1)                  81p'
A2+2A1 |001010  11 222.2.   q^24.A1.(q-1)                  60p'
2A2    |200002  12 2.2.22         q^16.G2                  24p'
A2+A1  |110001  13 222...   q^23.A2.(q-1)                  64p'
A2     |020000  15 2.2...    q^20.(A2xA2)       11:15p' 2:30p'
3A1    |000100  16 22..2.      q^27.A2xA1                  15q'
```

```
2A1     |100001  20 22....   q^24.B3.(q-1)                20p'
A1      |010000  25 2.....         q^21.A5                 6p'
1       |000000  36 ......              E6                 1p'
```

100.2 ICCTable

`ICCTable(`*uc*`[,`*seriesNo*`[,`*q*`]])`

This function gives the table of decompositions of the functions $X_{u,\varphi}$ in terms of the functions $Y_{u,\varphi}$. Here (u,φ) runs over the pairs where u is a unipotent element of the reductive group **G** and φ is a character of the group of components $A(u)$; such a pair describes a **G**-equivariant local system on the class C of u. The function $Y_{u,\varphi}$ is the characteristic function of this local system and $X_{u,\varphi}$ is the characteristic function of the corresponding intersection cohomology complex. The local systems can also be indexed by characters of the relative Weyl group occurring in the Springer correspondence, and since the coefficient of X_χ on Y_ψ is 0 if χ and ψ do not correspond to the same relative Weyl group (are not in the same Springer series), the table given is for a given Springer series, the series whose number is given by the argument `seriesNo` (if omitted this defaults to `seriesNo=1` which is the principal series). The decomposition multiplicities are graded, and are given as polynomials in one variable (specified by the argument q; if not given `Indeterminate(Rationals)` is assumed).

```
gap> W:=CoxeterGroup("A",3);;
gap> uc:=UnipotentClasses(W);;
gap> Display(ICCTable(uc));
Coefficients of X_phi on Y_psi for A3

      |4 31 22 211 1111
------------------------
X4    |1  1  1   1    1
X31   |0  1  1  P2   P3
X22   |0  0  1   1   P4
X211  |0  0  0   1   P3
X1111 |0  0  0   0    1
```

In the above the multiplicities are given as products of cyclotomic polynomials to display them more compactly. However the `Format` or the `Display` of such a table can be controlled more precisely.

For instance, one can ask to not display the entries as products of cyclotomic polynomials:

```
gap> Display(ICCTable(uc),rec(CycPol:=false));
Coefficients of X_phi on Y_psi for A3

      |4 31 22 211    1111
---------------------------
X4    |1  1  1   1       1
X31   |0  1  1 q+1 q^2+q+1
X22   |0  0  1   1   q^2+1
X211  |0  0  0   1 q^2+q+1
X1111 |0  0  0   0       1
```

Since `Display` and `Format` use the function 104.3, all the options of this function are also available. We can use this to restrict the entries displayed to a given sublist of the rows and columns (here the indices correspond to the number in CHEVIE of the corresponding character of the relative Weyl group of the given Springer series):

```
gap> W:=CoxeterGroup("F",4);;
gap> uc:=UnipotentClasses(W);;
gap> show:=[13,24,22,18,14,9,11,19];;
gap> Display(ICCTable(uc),rec(rows:=show,columns:=show));
Coefficients of X_phi on Y_psi for F4

            |A1+~A1 A2 ~A2 A2+~A1 ~A2+A1 B2(11) B2 C3(a1)(11)
_____________________________________________________________
Xphi{9,10}  |      1  0   0      0      0      0  0          0
Xphi{8,9}'' |      1  1   0      0      0      0  0          0
Xphi{8,9}'  |      1  0   1      0      0      0  0          0
Xphi{4,7}'' |      1  1   0      1      0      0  0          0
Xphi{6,6}'  |     P4  1   1      1      1      0  0          0
Xphi{4,8}   |    q^2  0   0      0      0      1  0          0
Xphi{9,6}'' |     P4 P4   0      1      0      0  1          0
Xphi{4,7}'  |    q^2  0  P4      0      1      0  0          1
```

The function `ICCTable` returns a record with various pieces of information which can help further computations.

.scalar

this contains the table of multiplicities $P_{\psi,\chi}$ of the X_ψ on the Y_χ. One should pay attention that by default, the table is not displayed in the same order as the stored `.scalar`, which is in order in CHEVIE of the characters in the relative Weyl group; the table is transposed, then lines and rows are sorted by `dimBu,class no,index of character in A(u)` while displayed.

.group

The group W.

.relgroup

The relative Weyl group for the Springer series.

.series

The index of the Springer series given for W.

.dimBu

The list of $\dim \mathcal{B}_u$ for each local system (u,φ) in the series.

.L

The matrix of (unnormalized) scalar products of the functions Y_ψ with themselves, that is the (ϕ,ψ) entry is $\sum_{g\in\mathbf{G}(\mathcal{F}_q)} Y_\phi(g)\overline{Y_\psi(g)}$. This is thus a symmetric, block-diagonal matrix where the diagonal blocks correspond to geometric unipotent conjugacy classes. This matrix is obtained as a by-product of Lusztig's algorithm to compute $P_{\psi,\chi}$.

100.3 XTable

`XTable(`*uc*`[,rec(classes:=true)])`

This function presents in a different way the information obtained from `ICCTable`. Let $\tilde{X}_{u,\phi} = q^{1/2(codimC-\dim Z(L))}$ where C is the class of u and $Z(\mathbf{L})$ is the center of Levi subgroup on which lives the cuspidal local system attached to the local system (u,ϕ).

Then 'XTable' gives the decomposition of the functions $\tilde{X}_{u,\phi}$ on local systems, by default. If the argument `rec(classes:=true)` is given, it gives the values of the functions $\tilde{X}_{u,\phi}$ on unipotent classes.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> Display(XTable(UnipotentClasses(W)));
Values of character sheaves tilde X_iota on local systems
              X |    1  A1 ~A1 G2(a1)(111) G2(a1)(21) G2(a1) G2

---------------------------------------------------------------
X^G2_phi{1,0}   |    1   1   1           0          0      1  1
X^G2_phi{1,6}   |  q^6   0   0           0          0      0  0
X^G2_phi{1,3}'  |  q^3   0   q           0          q      0  0
X^G2_phi{1,3}'' |  q^3 q^3   0           0          0      0  0
X^G2_phi{2,1}   |  qP8   q   q           0          0      q  0
X^G2_phi{2,2}   |q^2P4 q^2 q^2           0          0      0  0
X^._            |    0   0   0         q^2          0      0  0
```

The functions $\tilde{X}$ in the first column are decorated by putting as an exponent the relative groups $W_G(L)$.

```
gap> Display(XTable(UnipotentClasses(W),rec(classes:=true)));
Values of character sheaves tilde X_iota on unipotent classes
              X |    1  A1 ~A1 G2(a1) G2(a1)_21 G2(a1)_3 G2

-----------------------------------------------------------
X^G2_phi{1,0}   |    1   1   1      1         1        1  1
X^G2_phi{1,6}   |  q^6   0   0      0         0        0  0
X^G2_phi{1,3}'  |  q^3   0   q     2q         0       -q  0
X^G2_phi{1,3}'' |  q^3 q^3   0      0         0        0  0
X^G2_phi{2,1}   |  qP8   q   q      q         q        q  0
X^G2_phi{2,2}   |q^2P4 q^2 q^2      0         0        0  0
X^._            |    0   0   0    q^2      -q^2      q^2  0
gap> Display(XTable(UnipotentClasses(W,2)));
Values of character sheaves tilde X_iota on local systems
              X |    1  A1 ~A1 G2(a1)(111) G2(a1)(21) G2(a1) G2(11) G2

-----------------------------------------------------------------------
X^G2_phi{1,0}   |    1   1   1           0          0      1      0  1
X^G2_phi{1,6}   |  q^6   0   0           0          0      0      0  0
X^G2_phi{1,3}'  |  q^3   0   q           0          q      0      0  0
X^G2_phi{1,3}'' |  q^3 q^3   0           0          0      0      0  0
X^G2_phi{2,1}   |  qP8   q   q           0          0      q      0  0
X^G2_phi{2,2}   |q^2P4 q^2 q^2           0          0      0      0  0
X^._            |    0   0   0         q^2          0      0      0  0
X^._            |    0   0   0           0          0      0      q  0
gap> Display(XTable(UnipotentClasses(RootDatum("sl",4))));
Values of character sheaves tilde X_iota on local systems
```

```
        X | 1111 211 22(11)  22 31 4    4(I) 4(-1)   4(-I)
----------------------------------------------------------
X^A3_1111 |  q^6   0      0   0  0 0       0     0       0
X^A3_211  |q^3P3 q^3      0   0  0 0       0     0       0
X^A3_22   |q^2P4 q^2      0 q^2  0 0       0     0       0
X^A3_31   |  qP3 qP2      0   q  q 0       0     0       0
X^A3_4    |    1   1      0   1  1 1       0     0       0
X^A1_11   |    0   0    q^3   0  0 0       0     0       0
X^A1_2    |    0   0    q^2   0  0 0       0     q       0
X^._      |    0   0      0   0  0 0 q^(3/2)     0       0
X^._      |    0   0      0   0  0 0       0     0 q^(3/2)
```

A side effect of calling `XTable` with the argument `rec(classes:=true)` is to compute the cardinal of the unipotent conjugacy classes:

```
gap> t:=XTable(UnipotentClasses(CoxeterGroup("G",2)),rec(classes:=true));
XTable(CoxeterGroup("G",2),rec(classes:=true))
gap> List(t.cardClass,CycPol);
[ 1, P1P2P3P6, q^2P1P2P3P6, 1/6q^2P1^2P2^2P3P6, 1/2q^2P1^2P2^2P3P6,
  1/3q^2P1^2P2^2P3P6, q^4P1^2P2^2P3P6 ]
```

100.4 GreenTable

`GreenTable(`*uc*`[,rec(classes:=true)])`

Keeping the same notations as in the description of `XTable`, this function returns a table of the functions Q_{wF}, attached to elements $wF \in W_{\mathbf{G}}(\mathbf{L}) \cdot F$ where $W_{\mathbf{G}}(\mathbf{L})$ are the relative weyl groups attached to cuspidal local systems. These functions are defined by $Q_{wF} = \sum_{u,\phi} \tilde{\phi}(wF)\tilde{X}_{u,\phi}$. An point to note is that in the principal Springer series, when $\mathbf{T}$ is a maximal torus, the function Q_{wF} coincides with the Deligne-Lusztig character $R^{\mathbf{G}}_{\mathbf{T}_W}(1)$. As for `XTable`, by default the table gives the values of the functions on local systems. If the argument `rec(classes:=true)` is given, then it gives the values of the functions Q^{wF} on conjugacy classes.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> Display(GreenTable(UnipotentClasses(W)));
Values of Green functions Q_wF on local systems
Q^I_wF\class |          1       A1       ~A1 G2(a1)(111) G2(a1)(21) G2(a1) G2

--------------------------------------------------------------------------------
Q^G2_A0      |  P2^2P3P6    P2P3  (1+2q)P2           0          q   1+2q  1
Q^G2_~A1     | -P1P2P3P6   -P1P3        P2           0          q      1  1
Q^G2_A1      | -P1P2P3P6    P2P6       -P1           0         -q      1  1
Q^G2_G2      |P1^2P2^2P3 -P1P2^2     -P1P2           0         -q     P2  1
Q^G2_A2      |P1^2P2^2P6  P1^2P2     -P1P2           0          q    -P1  1
Q^G2_A1+~A1  |  P1^2P3P6   -P1P6 (-1+2q)P1           0         -q   1-2q  1
Q^._         |         0       0         0         q^2          0      0  0
```

The functions Q_{wF} depend only on the conjugacy class of wF, so in the first column the indices of `Q` are the names of the conjugacy classes of $W_{\mathbf{G}}(\mathbf{L})$. The exponents are the names of the groups $W_{\mathbf{G}}(\mathbf{L})$.

```
gap> Display(GreenTable(UnipotentClasses(W),rec(classes:=true)));
Values of Green functions Q_wF on unipotent classes
Q^I_wF\class |         1      A1       ~A1 G2(a1) G2(a1)_21 G2(a1)_3 G2
_______________________________________________________________________
Q^G2_A0      |  P2^2P3P6    P2P3  (1+2q)P2   1+4q      1+2q       P2  1
Q^G2_~A1     | -P1P2P3P6   -P1P3        P2   1+2q         1      -P1  1
Q^G2_A1      | -P1P2P3P6    P2P6       -P1   1-2q         1       P2  1
Q^G2_G2      |P1^2P2^2P3 -P1P2^2     -P1P2    -P1        P2     1+2q  1
Q^G2_A2      |P1^2P2^2P6  P1^2P2     -P1P2     P2       -P1     1-2q  1
Q^G2_A1+~A1  |  P1^2P3P6   -P1P6 (-1+2q)P1   1-4q      1-2q      -P1  1
Q^._         |         0       0         0    q^2      -q^2      q^2  0
gap> W:=RootDatum("sl",4);
RootDatum("sl",4)
gap> Display(GreenTable(UnipotentClasses(W)),rec(columns:=[1..8]));
Values of Green functions Q_{wF} on local systems
Q^I_wF\class |      1111           211 22(11)       22   31 4    4(I) 4(-1)
___________________________________________________________________________
Q^A3_1111    |  P2^2P3P4 (1+2q+3q^2)P2      0 (1+2q)P2 1+3q 1       0     0
Q^A3_211     | -P1P2P3P4   1+q+q^2-q^3      0       P2   P2 1       0     0
Q^A3_22      |  P1^2P3P4         -P1P4      0 1-q+2q^2  -P1 1       0     0
Q^A3_31      |P1^2P2^2P4         -P1P2      0    -P1P2    1 1       0     0
Q^A3_4       | -P1^3P2P3        P1^2P2      0      -P1  -P1 1       0     0
Q^A1_11      |         0             0  q^2P2        0    0 0       0     q
Q^A1_2       |         0             0 -q^2P1        0    0 0       0     q
Q^._         |         0             0      0        0    0 0 q^(3/2)     0
Q^._         |         0             0      0        0    0 0       0     0
```

100.5 UnipotentValues

`UnipotentValues(`*uc*`[,rec(classes:=true)])`

This function returns a table of the values of unipotent characters on local systems (by default) or on unipotent classes (if the argument `rec(classes:=true)` is given).

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> uc:=UnipotentClasses(W);
UnipotentClasses( G2 )
gap> Display(UnipotentValues(uc,rec(classes:=true)),rec(columns:=[1,4,5,6,7]));
Values of unipotent characters for G2 on unipotent classes
           |          1      G2(a1) G2(a1)_21    G2(a1)_3 G2
_____________________________________________________________
phi{1,0}   |          1           1         1           1  1
phi{1,6}   |        q^6           0         0           0  0
phi{1,3}'  |    1/3qP3P6 (5/3+1/3q)q   -1/3qP1     1/3qP1  0
phi{1,3}'' |    1/3qP3P6      1/3qP1   -1/3qP1 (2/3+1/3q)q  0
phi{2,1}   |  1/6qP2^2P3 (5/6+1/6q)q   -1/6qP1     1/6qP1  0
phi{2,2}   |  1/2qP2^2P6     -1/2qP1    1/2qP2    -1/2qP1  0
G2[-1]     |  1/2qP1^2P3     -1/2qP1    1/2qP2    -1/2qP1  0
```

```
G2[1]      |  1/6qP1^2P6 (5/6+1/6q)q   -1/6qP1      1/6qP1  0
G2[E3]     |1/3qP1^2P2^2      1/3qP1   -1/3qP1 (2/3+1/3q)q  0
G2[E3^2]   |1/3qP1^2P2^2      1/3qP1   -1/3qP1 (2/3+1/3q)q  0
gap> uc:=UnipotentClasses(W,3);
UnipotentClasses( G2,3 )
gap> Display(UnipotentValues(uc,rec(classes:=true)),rec(columns:=[1,4,5,6,7]));
Values of unipotent characters for G2 on unipotent classes
           |          1  G2(a1) G2(a1)_2    G2      G2_\zeta_3
_______________________________________________________________
phi{1,0}   |          1       1        1     1               1
phi{1,6}   |        q^6       0        0     0               0
phi{1,3}'  |    1/3qP3P6  1/3qP2  -1/3qP1 -2/3q            1/3q
phi{1,3}'' |    1/3qP3P6  1/3qP2  -1/3qP1 -2/3q            1/3q
phi{2,1}   |  1/6qP2^2P3  1/6qP2  -1/6qP1  2/3q           -1/3q
phi{2,2}   |  1/2qP2^2P6 -1/2qP1   1/2qP2     0               0
G2[-1]     |  1/2qP1^2P3 -1/2qP1   1/2qP2     0               0
G2[1]      |  1/6qP1^2P6  1/6qP2  -1/6qP1  2/3q           -1/3q
G2[E3]     |1/3qP1^2P2^2  1/3qP2  -1/3qP1  1/3q (-1-3ER(-3))/6q
G2[E3^2]   |1/3qP1^2P2^2  1/3qP2  -1/3qP1  1/3q (-1+3ER(-3))/6q
```

100.6 SpecialPieces

`SpecialPieces(`*uc*`)`

The special pieces forme a partition of the unipotent variety of a reductive group **G** which was defined the first time in [Spa82, chap. III] as the fibers of d^2, where d is a "duality map". Another definition is as the set of classes in the Zariski closure of a special class and not in the Zariski closure of any smaller special class, where a special class in the support of the image of a special character by the Springer correspondence.

Each piece is a union of unipotent conjugacy classes so is represented in CHEVIE as a list of class numbers. Thus the list of special pieces is returned as a list of lists of class numbers. The list is sorted by increasing piece dimension, while each piece is sorted by decreasing class dimension, so the special class is listed first.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> SpecialPieces(UnipotentClasses(W));
[ [ 1 ], [ 4, 3, 2 ], [ 5 ] ]
gap> SpecialPieces(UnipotentClasses(W,3));
[ [ 1 ], [ 4, 3, 2, 6 ], [ 5 ] ]
```

The example above shows that the special pieces are different in characteristic 3.

100.7 InducedLinearForm

`InducedLinearForm(`W`, `K`, h)`

This routine can be used to find the Richardson-Dynkin diagram of the class in the algebraic group **G** which contains a given unipotent class of a reductive subgroup of maximum rank **S** of **G**.

It takes a linear form on the roots of K, defined by its value on the simple roots (these values can define a Dynkin-Richardson diagram); then extends this linear form to the roots of $\mathbf{G}$ by 0 on the orthogonal of the roots of K; and finally conjugates the resulting form by an element of the Weyl group so that it takes positive values on the simple roots.

```
gap> W:=CoxeterGroup("F",4);;
gap> H:=ReflectionSubgroup(W,[1,3]);;
gap> InducedLinearForm(W,H,[2,2]);
[ 0, 1, 0, 0 ]
gap> uc:=UnipotentClasses(W);;
gap> Display(uc.classes[4]);
A1+~A1: D-R0100 C=q^18.A1xA1
```

The example above shows that the class containing the regular class of the Levi subgroup of type $A_1 \times \tilde{A}_1$ is the class `A1+~A1`.

Chapter 101

Unipotent elements of reductive groups

This chapter describes functions allowing to make computations in the unipotent radical of a Borel subgroup of a connected algebraic reductive group; the implementation of these functions was initially written by Olivier Dudas.

The unipotent radical of a Borel subgroup is the product in any order of root subgroups associated to the positive roots. We fix an order, which gives a canonical form to display elements and to compare them.

The computations use the Steinberg relations between root subgroups, which come from the choice of a Chevalley basis of the Lie algebra. The reference we follow is chapters 4 to 6 of the book [Car72b] "Simple groups of Lie type"by R.W. Carter (Wiley 1972).

We start with a root datum specified by a CHEVIE Coxeter group record `W` and build a record which contains information about the maximal unipotent subgroup of the corresponding reductive group, that is the unipotent radical of the Borel subgroup determined by the positive roots.

```
gap> W:=CoxeterGroup("E",6);; U:=UnipotentGroup(W);
UnipotentGroup(CoxeterGroup("E",6))
```

Now, if α =`W.roots[2]`, we make the element $u_\alpha(4)$ of the root subgroup u_α:

```
gap> U.Element(2,4);
u2(4)
```

If we do not specify the coefficient we make by default $u_\alpha(1)$, so we have also:

```
gap> U.Element(2)^4;
u2(4)
```

We can make more complicated elements:

```
gap> U.Element(2,4)*U.Element(4,5);
u2(4) * u4(5)
gap> U.Element(2,4,4,5);
u2(4) * u4(5)
```

If the roots are not in order the element is normalized:

```
gap> u:=U.Element(4,5,2,4);
u2(4) * u4(5) * u8(-20)
```

It is possible to display the decomposition of the roots in simple roots instead of their index:

```
gap> Display(u,rec(root:=true));
u010000(4) * u000100(5) * u010100(-20)
```

The coefficients in the root subgroups can be elements of arbitrary rings. Here is an example using `Mvps` (see 112.1):

```
gap>  W:=CoxeterGroup("E",8);; U:=UnipotentGroup(W);
UnipotentGroup(CoxeterGroup("E",8))
gap> u:=U.Element(List([1..8],i->[i,Z(2)*Mvp(SPrint("x",i))]));
u1(Z(2)^0x1) * u2(Z(2)^0x2) * u3(Z(2)^0x3) * u4(Z(2)^0x4) * u5(Z(2)^0x\
5) * u6(Z(2)^0x6) * u7(Z(2)^0x7) * u8(Z(2)^0x8)
gap> Display(u^16,rec(root:=true));
u22343210(Z(2)^0x1^2x2^2x3^3x4^4x5^3x6^2x7) *
u12343211(Z(2)^0x1x2^2x3^3x4^4x5^3x6^2x7x8) *
u12243221(Z(2)^0x1x2^2x3^2x4^4x5^3x6^2x7^2x8) *
u12233321(Z(2)^0x1x2^2x3^2x4^3x5^3x6^3x7^2x8) *
u22343211(Z(2)^0x1^2x2^2x3^3x4^4x5^3x6^2x7x8) *
u12243321(Z(2)^0x1x2^2x3^2x4^4x5^3x6^3x7^2x8) *
u12244321(Z(2)^0x1x2^2x3^2x4^4x5^4x6^3x7^2x8) *
u22343321(Z(2)^0x1^2x2^2x3^3x4^4x5^3x6^3x7^2x8) *
u12344321(Z(2)^0x1x2^2x3^3x4^4x5^4x6^3x7^2x8) *
u22344321(Z(2)^0x1^2x2^2x3^3x4^4x5^4x6^3x7^2x8) *
u23354321(Z(2)^0x1^2x2^3x3^3x4^5x5^4x6^3x7^2x8) *
u22454321(Z(2)^0x1^2x2^2x3^4x4^5x5^4x6^3x7^2x8) *
u23465432(Z(2)^0x1^2x2^3x3^4x4^6x5^5x6^4x7^3x8^2)
gap> u^32;
()
```

The above computation shows that in characteristic 2 the exponent of the unipotent group of E_8 is 32. More precisely, squaring doubles the height of the involved roots, so in the above u^{16} involves only roots of height 16 or more.

Various actions are defined on unipotent elements. Elements of the Weyl group act (through certain representatives) as long as no root subgroup is in their inversion set:

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> U:=UnipotentGroup(W);
UnipotentGroup(CoxeterGroup("G",2))
gap> u:=U.Element(1,Mvp("x"),3,Mvp("y"));
u1(x) * u3(y)
gap> u^(W.2*W.1);
u4(y) * u5(x)
gap> u^W.1;
Error, u1(x) * u3(y) should have no coefficient on root 1
 in
```

```
<rec1> ^ <rec2> called from
main loop
brk>
```

Semisimple elements act by conjugation:

```
gap> s:=SemisimpleElement(W,[E(3),2]);
<E(3),2>
gap> u^s;
u1(E3x) * u3(2E3y)
```

As well as unipotent elements

```
gap> u^U.Element(2);
u1(x) * u3(x+y) * u4(-x-2y) * u5(x+3y) * u6(3xy+x^2+3y^2)
```

101.1 UnipotentGroup

`UnipotentGroup(` W `)`

W should be a Coxeter group record representing a Weyl group. This function returns a record representing the unipotent radical $\mathbf{U}$ of a Borel subgroup of the reductive group of Weyl group W.

The result is a record with the following fields:

`weylGroup`
: contains W.

`specialPairs`
: Let $<$ be the order on the roots of W resulting from some total order on the ambient vector space (CHEVIE chooses such an order once and for all and it has nothing to do with the field `.order` of the unipotent group record). A pair (r, s) of roots is **special** if $r < s$ and $r + s$ is a root. The field `.specialPairs` contains twice the list of triples $(r, s, r+s)$ for special pairs: it contains first this list, sorted by $(r+s, r)$, then it contains a copy of the list in the order $(s, r, r+s)$. Roots in these triples are represented as their index in `Parent(W).roots`. Thanks to the repetition, each ordered pair of positive roots whose sum is a root appears exactly once in `.specialPairs`.

`chevalleyConstants`
: The Lie algebra of $\mathbf{U}$ has a **Chevalley basis** e_r indexed by roots, with the property that $[e_r, e_s] = N_{r,s}e_{r+s}$ for some integer constants $N_{r,s}$ for each pair of roots whose sum is a root. The list `chevalleyConstants`, of same length as `.specialPairs`, contains the corresponding integers $N_{r,s}$.

`commutatorConstants`
: These are the constants $C_{r,s,i,j}$ which occur in the commutator formula for two root subgroups:

$$u_s(u)u_r(t) = u_r(t)u_s(u) \prod_{i,j>0} u_{ir+js}(C_{r,s,i,j}(-t)^i u^j),$$

where the product is over all the roots of the given shape. The list `.commutatorConstants` is of the same length as `.specialPairs` and contains for each pair of roots (r, s) a list of quadruples $[i, j, ir + js, C_{r,s,i,j}]$ for all possible values of i, j for this pair.

`order`

An order on the roots, used to give a canonical form to unipotent elements by listing the root subgroups in that order. `.order` is the list of indices of roots in `Parent(W)`, listed in the desired order.

```
gap> W:=CoxeterGroup("G",2);
CoxeterGroup("G",2)
gap> U:=UnipotentGroup(W);
UnipotentGroup(CoxeterGroup("G",2))
gap> U.specialPairs;
[ [ 1, 2, 3 ], [ 2, 3, 4 ], [ 2, 4, 5 ], [ 1, 5, 6 ], [ 3, 4, 6 ],
  [ 2, 1, 3 ], [ 3, 2, 4 ], [ 4, 2, 5 ], [ 5, 1, 6 ], [ 4, 3, 6 ] ]
gap> U.chevalleyConstants;
[ 1, 2, 3, 1, 3, -1, -2, -3, -1, -3 ]
gap> U.commutatorConstants;
[ [ [ 1, 1, 3, 1 ], [ 1, 2, 4, -1 ], [ 1, 3, 5, 1 ], [ 2, 3, 6, 2 ] ],
  [ [ 1, 1, 4, 2 ], [ 2, 1, 5, 3 ], [ 1, 2, 6, -3 ] ],
  [ [ 1, 1, 5, 3 ] ], [ [ 1, 1, 6, 1 ] ], [ [ 1, 1, 6, 3 ] ],
  [ [ 1, 1, 3, -1 ], [ 2, 1, 4, -1 ], [ 3, 1, 5, -1 ],
      [ 3, 2, 6, -1 ] ],
  [ [ 1, 1, 4, -2 ], [ 2, 1, 6, -3 ], [ 1, 2, 5, 3 ] ],
  [ [ 1, 1, 5, -3 ] ], [ [ 1, 1, 6, -1 ] ], [ [ 1, 1, 6, -3 ] ] ]
```

A unipotent group record also contains functions for creating and normalizing unipotent elements.

`U.Element(`r`)`

`U.Element(`r_1`,`c_1`,..,`r_n`,`c_n`)`

In the first form the function creates the element $u_r(1)$, and in the second form the element $u_{r_1}(c_1)\ldots u_{r_n}(c_n)$

```
gap> U.Element(2);
u2(1)
gap> U.Element(1,2,2,4);
u1(2) * u2(4)
gap> U.Element(2,4,1,2);
u1(2) * u2(4) * u3(-8) * u4(32) * u5(-128) * u6(512)
```

`U.CanonicalForm(`l`[,order])`

The function takes a list of pairs `[r,c]` representing a unipotent element, where `r` is a root and `c` the corresponding coefficient, and puts it in canonical form, reordering the terms to agree with `U.order` using the commutation relations. If a second argument is given, this is used instead of `U.order`.

```
gap> U.CanonicalForm([[2,4],[1,2]]);
[ [ 1, 2 ], [ 2, 4 ], [ 3, -8 ], [ 4, 32 ], [ 5, -128 ], [ 6, 512 ] ]
gap> U.CanonicalForm(last,[6,5..1]);
[ [ 2, 4 ], [ 1, 2 ] ]
```

101.2 Operations for Unipotent elements

The arithmetic operations `*`, `/` and `^` work for unipotent elements. They also have `Print` and `String` methods.

```
gap> u:=U.Element(1,4,3,-6);
u1(4) * u3(-6)
gap> u^-1;
u1(-4) * u3(6)
gap> u:=U.Element(1,4,2,-6);
u1(4) * u2(-6)
gap> u^-1;
u1(-4) * u2(6) * u3(24) * u4(-144) * u5(864) * u6(6912)
gap> u^0;
()
gap> u*u;
u1(8) * u2(-12) * u3(24) * u4(432) * u5(6048) * u6(-17280)
gap> String(u);
"u1(4) * u2(-6)"
gap> Format(u^2,rec(root:=true));
"u10(8) * u01(-12) * u11(24) * u12(432) * u13(6048) * u23(-17280)"
```

`u^n` gives the `n`-th power of `u` when `n` is an integer and `u` conjugate by `n` when `n` is a unipotent element, a semisimple element or an element of the Weyl group.

101.3 IsUnipotentElement

`IsUnipotentElement(`u`)`

This function returns `true` if `u` is a unipotent element and `false` otherwise.

```
gap> IsUnipotentElement(U.Element(2));
true
gap> IsUnipotentElement(2);
false
```

101.4 UnipotentDecompose

`UnipotentDecompose(`w`,`u`)`

`u` should be a unipotent element and `w` an element of the corresponding Weyl group. If $\mathbf{U}$ is the unipotent radical of the Borel subgroup determined by the positive roots, and $\mathbf{U}^-$ the unipotent radical of the opposite Borel, this function decomposes u into its component in $\mathbf{U}\cap {}^w\mathbf{U}^-$ and its component in $\mathbf{U}\cap {}^w\mathbf{U}$.

```
gap> u:=U.Element(2,Mvp("y"),1,Mvp("x"));
u1(x) * u2(y) * u3(-xy) * u4(xy^2) * u5(-xy^3) * u6(2x^2y^3)
gap> UnipotentDecompose(W.1,u);
[ u1(x), u2(y) * u3(-xy) * u4(xy^2) * u5(-xy^3) * u6(2x^2y^3) ]
gap> UnipotentDecompose(W.2,u);
[ u2(y), u1(x) ]
```

101.5 UnipotentAbelianPart

`UnipotentAbelianPart(`u`)`

If $\mathbf{U}$ is the unipotent subgroup and $D(\mathbf{U})$ its derived subgroup, this function returns the projection of the unipotent element `u` on $\mathbf{U}/D(\mathbf{U})$, that is its coefficients on the simple roots.

```
gap> u:=U.Element(2,Mvp("y"),1,Mvp("x"));
u1(x) * u2(y) * u3(-xy) * u4(xy^2) * u5(-xy^3) * u6(2x^2y^3)
gap> UnipotentAbelianPart(u);
u1(x) * u2(y)
```

Chapter 102

Affine Coxeter groups and Hecke algebras

In this chapter we describe functions dealing with affine Coxeter groups and Hecke algebras.

We follow the presentation in [Kac82], §1.1 and 3.7.

A **generalized Cartan matrix** C is a matrix of integers of size $n \times n$ and of rank l such that $c_{i,i} = 2$, $c_{i,j} \leq 0$ is $i \neq j$, and $c_{i,j} = 0$ if and only if $c_{j,i} = 0$. We say that C is **indecomposable** if it does not admit any block decomposition.

Let C be a generalized Cartan matrix. For I a subset of $\{1, \ldots, n\}$ we denote by C_I the square submatrix with indices i, j taken in I. If v is a real vector of length n, we write $v > 0$ if for all $i \in \{1, \ldots, n\}$ we have $v_i > 0$. It can be shown that C is a Cartan matrix if and only if for all sets I, we have $\det C_I > 0$; or equivalently, if and only if there exists $v > 0$ such that $C.v > 0$. C is called an **affine Cartan matrix** if for all proper subsets I we have $\det C_I > 0$, but $\det C = 0$; or equivalently if there exists $v > 0$ such that $C.v = 0$.

Given an irreducible Weyl group W with Cartan matrix C, we can construct a generalized Cartan matrix $\tilde{C}$ as follows. Let α_0 be the opposed of the highest root. Then the matrix

$$\begin{pmatrix} C & C.\alpha_0 \\ \alpha_0.C & 2 \end{pmatrix}$$

is an affine Cartan matrix. The affine Cartan matrices which can be obtained in this way are those we are interested in, which give rise to affine Weyl groups.

Let $d = n - l$. A **realization** of a generalized Cartan matrix is a pair $V, V^\vee$ of vector spaces of dimension $n + d$ together with vectors $\alpha_1, \ldots, \alpha_n \in V$ (the **simple roots**), $\alpha_1^\vee, \ldots, \alpha_n^\vee \in V^\vee$ (the **simple coroots**), such that $(\alpha_i^\vee, \alpha_j) = c_{i,j}$. Up to isomorphism, a realization is obtained as follows: write $C = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}$ where C_1 is of rank l. Then take α_i to be the first n vectors in a basis of V, and take $\alpha_j^\vee$ to be given in the dual basis by the rows of the matrix

$$\begin{pmatrix} C_1 & 0 \\ C_2 & \mathrm{Id}_d \end{pmatrix}.$$

To C we associate a reflection group in the space V, generated by the **fundamental reflections** r_i given by $r_i(v) = v - (\alpha_i^\vee, v)\alpha_i$. This is a Coxeter group, called the **Affine Weyl group** $\tilde{W}$ associated to W when we start with the affine Cartan matrix associated to a Weyl group W.

The Affine Weyl group is infinite; it has one additional generator s_0 (the reflection with respect to α_0) compared to W. In GAP3 we can not use 0 as a label by default for a generator of a Coxeter group (because the default labels are used as indices, and indices start at 1 in GAP3) so we label it as `n+1` where `n` is the numbers of generators of W. The user can change that by setting the field `.reflectionsLabels` of $\tilde{W}$ to `Concatenation([1..n],[0])`. As in the finite case, we associate to the realization of $\tilde{W}$ a Dynkin diagram. We get the following diagrams:

```
gap> PrintDiagram(Affine(CoxeterGroup("A",1))); #  infinite bond
A1~  1 oo 2
gap> PrintDiagram(Affine(CoxeterGroup("A",5)));  #  for An, n not 1
       - - - 6 - - -
      /             \
A5~   1 - 2 - 3 - 4 - 5
gap> PrintDiagram(Affine(CoxeterGroup("B",4)));  #  for Bn
              5
              |
B4~   1 < 2 - 3 - 4
gap> PrintDiagram(Affine(CoxeterGroup("C",4)));  #  for Cn
C4~   1 > 2 - 3 - 4 < 5
gap> PrintDiagram(Affine(CoxeterGroup("D",6)));  #  for Dn
D6~  1          7
      \        /
       3 - 4 - 5
      /        \
     2          6
gap> PrintDiagram(Affine(CoxeterGroup("E",6)));
             7
             |
             2
             |
E6~  1 - 3 - 4 - 5 - 6
gap> PrintDiagram(Affine(CoxeterGroup("E",7)));
                 2
                 |
E7~  8 - 1 - 3 - 4 - 5 - 6 - 7
gap> PrintDiagram(Affine(CoxeterGroup("E",8)));
             2
             |
E8~  1 - 3 - 4 - 5 - 6 - 7 - 8 - 9
gap> PrintDiagram(Affine(CoxeterGroup("F",4)));
F4~  5 - 1 - 2 > 3 - 4
gap> PrintDiagram(Affine(CoxeterGroup("G",2)));
G2~  3 - 1 > 2
```

We represent in GAP3 the group $\tilde{W}$ as a matrix group in the space V.

102.1 Affine

`Affine(` W `)`

This function returns the affine Weyl corresponding to the Weyl group W.

102.2 Operations and functions for Affine Weyl groups

All matrix group operations are defined on Affine Weyl groups, as well as all functions defined for abstract Coxeter groups (in particular Hecke algebras and their Kazhdan-Lusztig bases).

The functions `Print(`W`)` and `PrintDiagram(`W`)` are also defined and print an appropriate representation of W:

```
gap> W:=Affine(CoxeterGroup("A",4));
Affine(CoxeterGroup("A",4))
gap> PrintDiagram(W);
        - - 5 - -
       /         \
A4~   1 - 2 - 3 - 4
```

The function `ReflectionLength` is also defined, using the formula of Lewis, McCammond, Petersen and Schwer.

102.3 AffineRootAction

`AffineRootAction(`W`,`w`,`x`)`

The Affine Weyl group W can be realized as affine transformations on the vector space spanned by the roots of `W.linear`. Given a vector x expressed in the basis of simple roots of `W.linear` and w in W, this function returns returns the image of x under w realized as an affine transformation.

Chapter 103

CHEVIE utility functions

The functions described below, used in various parts of the CHEVIE package, are of a general nature and should really be included in other parts of the GAP3 library. We include them here for the moment for the commodity of the reader.

103.1 SymmetricDifference

`SymmetricDifference(` S, T`)`

This function returns the symmetric difference of the sets S and T, which can be written in GAP3 as `Difference(Union(x,y),IntersectionSet(x,y)`.

```
gap> SymmetricDifference([1,2],[2,3]);
[ 1, 3 ]
```

103.2 DifferenceMultiSet

`DifferenceMultiSet(` l, s `)`

This function returns the difference of the multisets l and s. That is, l and s are lists which may contain several times the same item. The result is a list which is like l, excepted if an item occurs a times in s, the first a occurrences of this item in l have been deleted (all the occurrences if a is greater than the times it occurred in l).

```
gap> DifferenceMultiSet("ababcbadce","edbca");
"abbac"
```

103.3 Rotation

`Rotation(`l, i`)`

This function returns l rotated i steps.

```
gap> l:=[1..5];;
gap> Rotation(l,1);
[ 2, 3, 4, 5, 1 ]
gap> Rotation(l,0);
```

```
[ 1, 2, 3, 4, 5 ]
gap> Rotation(l,-1);
[ 5, 1, 2, 3, 4 ]
```

103.4 Rotations

`Rotations(`*l*`)`

This function returns the list of rotations of the list l.

```
gap> Rotations("abcd");
[ "abcd", "bcda", "cdab", "dabc" ]
gap> Rotations([1,0,1,0]);
[ [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ], [ 1, 0, 1, 0 ], [ 0, 1, 0, 1 ] ]
```

103.5 Inherit

`Inherit(`*rec1*`,`*rec2*`[,`*fields*`])`

This functions copies to the record *rec1* all the fields of the record *rec2*. If an additional argument *fields* is given, it should be a list of field names, and then only the fields specified by *fields* are copied. The function returns the modified *rec1*.

```
gap> r:=rec(a:=1,b:=2);
rec(
  a := 1,
  b := 2 )
gap> s:=rec(c:=3,d:=4);
rec(
  c := 3,
  d := 4 )
gap> Inherit(r,s);
rec(
  a := 1,
  b := 2,
  c := 3,
  d := 4 )
gap> r:=rec(a:=1,b:=2);
rec(
  a := 1,
  b := 2 )
gap> Inherit(r,s,["d"]);
rec(
  a := 1,
  b := 2,
  d := 4 )
```

103.6 Dictionary

`Dictionary()`

This function creates a dictionary data type. The created object is a record with two functions:

`Get(k)` get the value associated to key `k`; it returns `false` if there is no such key.

`Insert(k,v)` sets in the dictionary the value associated to key `k` to be `v`.

The main advantage compared to records is that keys may be of any type.

```
gap> d:=Dictionary();
Dictionary with 0 entries
gap> d.Insert("a",1);
1
gap> d.Insert("b",2);
2
gap> d.Get("a");
1
gap> d.Get("c");
false
gap> d;
Dictionary with 2 entries
```

103.7 GetRoot

`GetRoot(` *x*`,` *n* `[,` *msg*`])`

n must be a positive integer. `GetRoot` returns an n-th root of x when possible, else signals an error. If *msg* is present and `InfoChevie=Print` a warning message is printed about which choice of root has been made, after printing *msg*.

In the current implementation, it is possible to find an n-th root when x is one of the following GAP3 objects:

1- a monomial of the form `a*q^(b*n)` when we know how to find a root of a. The root chosen is `GetRoot(a,n)*q^b`.

2- a root of unity of the form `E(a)^i`. The root chosen is `E(a*n)^i`.

3- an integer, when n=2 (the root chosen is `ER(x)`) or when x is a perfect n-th power of a (the root chosen is a).

4- a product of an x of form 2- by an x of form 3-.

5- when x is a record and has a method `x.operations.GetRoot` the work is delegated to that method.

```
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> GetRoot(E(3)*q^2,2,"test");
#warning: test: E3^2q chosen as 2nd root of (E(3))*q^2
(E(3)^2)*q
gap> GetRoot(1,2,"test");
#warning: test: 1 chosen as 2nd root of 1
1
```

The example above shows that `GetRoot` is not compatible with specialization: `E(3)*q^2` evaluated at `E(3)` is `1` whose root chosen by `GetRoot` is `1`, while `(-E(3)^2)*q` evaluated at E(3) is -1. Actually it can be shown that it is not possible mathematically to define a function GetRoot compatible with specializations. This is why there is a provision in functions for character tables of Hecke algebras to provide explicit roots.

```
gap> GetRoot(8,3);
2
gap> GetRoot(7,3);
Error, : unable to compute 3-th root of 7
 in
GetRoot( 7, 3 ) called from
main loop
brk>
```

103.8 CharParams

CharParams(*G*)

G should be a group or another object which has a method CharTable, or a character table. The function CharParams tries to determine a list of labels for the characters of G. If G has a method CharParams this is called. Otherwise, if G is not a character table, its CharTable is called. If the table has a field .charparam in .irredinfo this is returned. Otherwise, the list [1..Length(G.irreducibles)] is returned.

```
gap> CharParams(CoxeterGroup("A",2));
[ [ [ 1, 1, 1 ] ], [ [ 2, 1 ] ], [ [ 3 ] ] ]
gap> CharParams(Group((1,2),(2,3)));
# W Warning          Group has no name
[ 1 .. 3 ]
```

103.9 CharName

CharName(*G*, *param*)

G should be a group and param a parameter of a character of that group (as returned by CharParams). If G has a method CharName, the function returns the result of that method, which is a string which displays nicely param (this is used by CHEVIE to nicely fill the .charNames in a CharTable – all finite reflection groups have such methods CharName).

```
gap> G:=CoxeterGroup("G", 2);
CoxeterGroup("G",2)
gap> CharParams(G);
[ [ [ 1, 0 ] ], [ [ 1, 6 ] ], [ [ 1, 3, 1 ] ], [ [ 1, 3, 2 ] ],
  [ [ 2, 1 ] ], [ [ 2, 2 ] ] ]
gap>  List(last,x->CharName(G,x));
[ "phi{1,0}", "phi{1,6}", "phi{1,3}'", "phi{1,3}''", "phi{2,1}",
  "phi{2,2}" ]
```

103.10 PositionId

PositionId(*G*)

G should be a group, a character table, an Hecke algebra or another object which has characters. `PositionId` returns the position of the identity character in the character table of G.

```
gap> W := CoxeterGroup( "D", 4 );;
gap> PositionId( W );
13
```

103.11 PermutationOnClasses

`PermutationOnClasses(` G, *aut*`)`

aut is an automorphism of the group G (for a permutation group, this could be given as a permutation normalizing G). The result is the permutation of `[1..NrConjugacyClasses(G)]` induced ny *aut*.

```
gap> WF:=RootDatum("3D4");
3D4
gap> PermutationOnClasses(Group(WF),WF.phi);
( 2, 8, 7)( 5,13,12)
```

103.12 PermutationOnCharacters

`PermutationOnCharacters(` G, *aut*`)`

aut is an automorphism of the group G (for a permutation group, this could be given as a permutation normalizing G). The result is the permutation of `[1..NrConjugacyClasses(G)]` (representing indices of irreducible characters) induced by *aut*.

```
gap> WF:=RootDatum("3D4");
3D4
gap> PermutationOnCharacters(Group(WF),WF.phi);
( 1, 7, 2)( 8,12, 9)
```

103.13 InductionTable

`InductionTable(` S, G `)`

`InductionTable` computes the decomposition of the induced characters from the subgroup S into irreducible characters of G. The rows correspond to the characters of the parent group, the columns to those of the subgroup. What is returned is actually a record with several fields: `.scalar` contains the induction table proper, and there are `Display` and `Format` methods. The other fields contain labeling information taken from the character tables of S and G when it exists.

```
gap> G := Group( [ (1,2), (2,3), (3,4) ], () );
Group( (1,2), (2,3), (3,4) )
gap> S:=Subgroup( G, [ (1,2), (3,4) ] );
Subgroup( Group( (1,2), (2,3), (3,4) ), [ (1,2), (3,4) ] )
gap> G.name := "G";; S.name := "S";; # to avoid warnings
gap> Display( InductionTable( S, G ) );
Induction from S to G
```

```
     |X.1 X.2 X.3 X.4
---------------------
X.1 |  1   .   .   .
X.2 |  .   .   .   1
X.3 |  1   .   .   1
X.4 |  .   1   1   1
X.5 |  1   1   1   .
```

```
gap> G := CoxeterGroup( "G", 2 );;
gap> S := ReflectionSubgroup( G, [ 1, 4 ] );
ReflectionSubgroup(CoxeterGroup("G",2), [ 1, 4 ])
gap> t := InductionTable( S, G );
InductionTable(ReflectionSubgroup(CoxeterGroup("G",2), [ 1, 4 ]), Coxe\
terGroup("G",2))
gap> Display( t );
Induction from A1x~A1 to G2
           |11,11 11,2 2,11 2,2
--------------------------------
phi{1,0}   |    .    .    .   1
phi{1,6}   |    1    .    .   .
phi{1,3}'  |    .    1    .   .
phi{1,3}'' |    .    .    1   .
phi{2,1}   |    .    1    1   .
phi{2,2}   |    1    .    .   1
```

The `Display` and `Format` methods take the same arguments as the `FormatTable` method. For instance to select a subset of the characters of the subgroup and of the parent group, one can call

```
gap> Display( t,rec( rows := [5], columns := [3,2] ) );
Induction from A1x~A1 to G2
         |2,11 11,2
--------------------
phi{2,1} |   1    1
```

It is also possible to get TeX and LaTeX output, see 104.3.

103.14 CharRepresentationWords

`CharRepresentationWords(` *rep* `,` *elts* `)`

given a list *rep* of matrices corresponding to generators and a list *elts* of words in the generators it returns the list of traces of the corresponding representation on the elements in *elts*.

```
gap> H := Hecke(CoxeterGroup( "F", 4 ));;
gap> r := ChevieClassInfo( Group( H ) ).classtext;;
gap> t := HeckeReflectionRepresentation( H );;
gap> CharRepresentationWords( t, r );
[ 4, -4, 0, 1, -1, 0, 1, -1, -2, 2, 0, 2, -2, -1, 1, 0, 2, -2, -1, 1,
  0, 0, 2, -2, 0 ]
```

103.15 Transversals

`Transversals(` G`[,` m`] )`

returns a pair *[orb, rep]* where *orb* is a list of the orbits of the permutation group G on `[ 1..LargestMovedPoint(` G `) ]` and *rep* is a list of list of elements of G such that `rep[i][j]` applied to `orb[i][1]` yields `orb[i][j]` for all i, j. If the optional argument m is given, then `LargestMovedPoint(` G `)` is replaced by the integer m.

```
gap> G := Group( (1,7)(2,3)(5,6)(8,9)(11,12),
>                (1,5)(2,8)(3,4)(7,11)(9,10) );;
gap> Transversals( G );
[ [ [ 1, 7, 5, 11, 6, 12 ], [ 2, 3, 8, 4, 9, 10 ] ],
  [ [ (), ( 1, 7)( 2, 3)( 5, 6)( 8, 9)(11,12),
          ( 1, 5)( 2, 8)( 3, 4)( 7,11)( 9,10),
          ( 1,11,12, 7, 5, 6)( 2, 4, 3, 8,10, 9),
          ( 1, 6, 5, 7,12,11)( 2, 9,10, 8, 3, 4),
          ( 1,12)( 2, 4)( 3, 9)( 6, 7)( 8,10) ],
      [ (), ( 1, 7)( 2, 3)( 5, 6)( 8, 9)(11,12),
          ( 1, 5)( 2, 8)( 3, 4)( 7,11)( 9,10),
          ( 1,11,12, 7, 5, 6)( 2, 4, 3, 8,10, 9),
          ( 1, 6, 5, 7,12,11)( 2, 9,10, 8, 3, 4),
          ( 1, 6)( 2,10)( 4, 8)( 5,11)( 7,12) ] ] ]
```

103.16 AbelianGenerators

`AbelianGenerators(` A`)`

A should be an abelian group or the list of its generators. The function returns a list of generators for A of orders given by `AbelianInvariants(A)`.

Chapter 104

CHEVIE String and Formatting functions

CHEVIE enhances the facilities of GAP33 for formatting and displaying objects.

First, it provides some useful string functions, such as `Replace`, and `IntListToString`.

Second, it enforces a general policy on how to format and print objects. The most basic method which should be provided for an object is the `Format` method. This is a method whose second argument is a record of options to control printing/formatting the object. When the second argument is absent, or equivalently the empty record, one has the most basic formatting, which is used to make the `Display` method of the object. When the option `GAP` is set (that is the record second argument has a field `GAP`), the output should be a form which can, as far as possible, read back in GAP3. This output is what is used by default in the methods `String` and `Print`.

In addition to the above options, most CHEVIE objects also provide the formatting options `TeX` (resp. `LaTeX`), to output strings suitable for TeX or LaTeX typesetting. The objects for which this makes sense (like polynomials) provide a `Maple` option for formatting to create output readable by Maple.

104.1 Replace

`Replace(` *s* `[,` *s1* `,` *r1* `[,` *s2* `,` *r2* `[...]]])`

Replaces in list *s* all (non-overlapping) occurrences of sublist *s1* by list *r1*, then all occurrences of *s2* by *r2*, etc...

```
gap> Replace("aabaabaabbb","aabaa","c","cba","def","bbb","ult");
"default"
```

104.2 IntListToString

`IntListToString(` *part* `[,` *brackets*`] )`

part must be a list of positive integers. If all of them are smaller than 10 then a string of digits corresponding to the entries of *part* is returned. If an entry is ≥ 10 then the elements of *part* are converted to strings, concatenated with separating commas and the result surrounded by brackets. By default `()` brackets are used. This may be changed by giving as second argument a length two string specifying another kind of brackets.

```
gap> IntListToString( [ 4, 2, 2, 1, 1 ] );
"42211"
gap> IntListToString( [ 14, 2, 2, 1, 1 ] );
"(14,2,2,1,1)"
gap> IntListToString( [ 14, 2, 2, 1, 1 ], "{}" );
"{14,2,2,1,1}"
```

104.3 FormatTable

`FormatTable(` *table*, *options* `)`

This is a general routine to format a table (a rectangular array, that is a list of lists of the same length).

The option is a record whose fields can be

`rowLabels` at least this field must be present. It will contain labels for the rows of the table.

`columnLabels` labels for the columns of the table.

`rowsLabel` label for the first column (containing the `rowLabels`).

`separators` by default, a separating line is put after the line of column labels. This option contains the indices of lines after which to put separating lines, so the default is equivalent to `.separators =[0]`.

`rows` a list of indices. If given, only the rows specified by these indices are formatted.

`columns` a list of indices. If given, only the columns specified by these indices are formatted.

`TeX` if set to `true`, TeX output is generated to format the table.

`LaTeX` `TeX` also should be set if this is used. LaTeX output is generated using the package `longtable`, so the output can be split across several pages.

`columnRepartition` This is used to specify how to split the table in several parts typeset one after the other. The variable `columnRepartition` should be a list of integers to specify how many columns to print in each part. When using plain text output, this is unnecessary as `FormatTable` can automatically split the table into parts not exceeding `screenColumns` columns, if this option is specified.

`screenColumns` As explained above, is used to split the table when doing plain text output. A good value to set it is `SizeScreen()[1]`, so each part of the table does not exceed the screen width.

```
gap> t:=IdentityMat(3);;o:=rec(rowLabels:=[1..3]);;
gap> Print(FormatTable(t,o));
1 |1 0 0
2 |0 1 0
3 |0 0 1
gap> o.columnLabels:=[6..8];;Print(FormatTable(t,o));
```

```
  |6 7 8
---------
1 |1 0 0
2 |0 1 0
3 |0 0 1
gap> o.rowsLabel:="N";;Print(FormatTable(t,o));
N |6 7 8
---------
1 |1 0 0
2 |0 1 0
3 |0 0 1
gap> o.separators:=[0,2];;Print(FormatTable(t,o));
N |6 7 8
---------
1 |1 0 0
2 |0 1 0
---------
3 |0 0 1
```

104.4 Format

`Format(` *object*`[,` *options*`] )`

`FormatGAP(` *object*`[,` *options*`] )`

`FormatMaple(` *object*`[,` *options*`] )`

`FormatTeX(` *object*`[,` *options*`] )`

`FormatLaTeX(` *object*`[,` *options*`] )`

`Format` is a general routine for formatting an object. *options* is a record of options; if not given, it is equivalent to *options*`:=rec()`. The routines `FormatGAP`, `FormatMaple`, `FormatTeX` and `FormatLaTeX` add some options (or setup a record with some options if no second argument is given); respectively they set up `GAP:=true`, `Maple:=true`, `TeX:=true`, and for `FormatLaTeX` both `TeX:=true` and `LaTeX:=true`.

If *object* is a record, `Format` looks if it has a `.operations.Format` method and then calls it. Otherwise, `Format` knows how to format in various ways:polynomials, cyclotomics, lists, matrices, booleans.

Here are some examples.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> Format(q^-3-13*q);
"-13q+q^-3"
gap> FormatGAP(q^-3-13*q);
"-13*q+q^-3"
gap> FormatMaple(q^-3-13*q);
"-13*q+q^(-3)"
gap> FormatTeX(q^-3-13*q);
"-13q+q^{-3}"
```

By default, `Format` tries to recognize cyclotomics which are in quadratic number fields. If the option `noQuadrat:=true` is given it does not.

```
gap> Format(E(3)-E(3)^2);
"ER(-3)"
gap> Format(E(3)-E(3)^2,rec(noQuadrat:=true));
"-E3^2+E3"
gap> FormatTeX(E(3)-E(3)^2,rec(noQuadrat:=true));
"-\\zeta_3^2+\\zeta_3"
gap> FormatTeX(E(3)-E(3)^2);
"\\sqrt {-3}"
gap> FormatMaple(E(3)-E(3)^2);
"sqrt(-3)"
```

Formatting of arrays gives output usable for typesetting if the `TeX` or `LaTeX` options are given.

```
gap> m:=IdentityMat(3);;
gap> Print(Format(m),"\n");
1 0 0
0 1 0
0 0 1
gap> FormatTeX(m);
"1#0#0\\cr\n0#1#0\\cr\n0#0#1\\cr\n"
gap> FormatGAP(m);
"[[1,0,0],[0,1,0],[0,0,1]]"
gap> FormatLaTeX(m);
"1#0#0\\\\\n0#1#0\\\\\n0#0#1\\\\\n"
```

Chapter 105

CHEVIE Matrix utility functions

This chapter documents various functions which enhance GAP3's ability to work with matrices.

105.1 EigenvaluesMat

`EigenvaluesMat(` *mat* `)`

mat should be a square matrix of Cyclotomics. The function returns the eigenvalues of M which are 0 or roots of unity.

```
gap> EigenvaluesMat(DiagonalMat(0,1,E(3),2,3));
[ 0, 1, E(3) ]
gap> EigenvaluesMat(PermutationMat((1,2,3,4),5));
[ 1, 1, -1, E(4), -E(4) ]
```

105.2 DecomposedMat

`DecomposedMat(` *mat* `)`

Finds if the square matrix *mat* with zeroes (or `false`) in symmetric positions admits a block decomposition.

Define a graph G with vertices `[1..Length(mat)]` and with an edge between `i` and `j` if either `mat[i][j]` or `mat[j][i]` is non-zero. `DecomposedMat` return a list of lists `l` such that `l[1]`,`l[2]`, etc.. are the vertices in each connected component of G. In other words, the matrices `mat{l[1]}{l[1]}`,`mat{l[2]}{l[2]}`, etc... are blocks of the matrix *mat*. This function may also be applied to boolean matrices where non-zero is replaced by `true`.

```
gap> m := [ [  0,  0,  0,  1 ],
>           [  0,  0,  1,  0 ],
>           [  0,  1,  0,  0 ],
>           [  1,  0,  0,  0 ] ];;
gap> DecomposedMat( m );
```

```
[ [ 1, 4 ], [ 2, 3 ] ]
gap> PrintArray( m{[ 1, 4 ]}{[ 1, 4 ]});
[[0, 1],
 [1, 0]]
```

105.3 BlocksMat

`BlocksMat(` *M* `)`

Finds if the matrix *M* admits a block decomposition.

Define a bipartite graph *G* with vertices `[1..Length(M)]`, `[1..Length(M[1])]` and with an edge between `i` and `j` if `M[i][j]` is not zero. BlocksMat returns a list of pairs of lists `I` such that `[I[1][1],I[1][2]]`, etc.. are the vertices in each connected component of *G*. In other words, `M{I[1][1]}{I[1][2]}`, `M{I[2][1]}{I[2][2]}`,etc... are blocks of `M`.

This function may also be applied to boolean matrices where non-zero is replaced by `true`.

```
gap> m:=[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 1, 0 ],
>  [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ] ];;
gap> BlocksMat(m);
[ [ [ 1, 3, 5 ], [ 1, 3 ] ], [ [ 2 ], [ 2 ] ], [ [ 4 ], [ 4 ] ] ]
gap> PrintArray(m{[1,3,5]}{[1,3]});
[[1, 0],
 [1, 1],
 [0, 1]]
```

105.4 RepresentativeDiagonalConjugation

`RepresentativeDiagonalConjugation(` *M* `,` *N* `)`

M and *N* must be square matrices. This function returns a list *d* such that `N=M^DiagonalMat(d)` if such a list exists, and false otherwise.

```
gap> M:=[[1,2],[2,1]];
[ [ 1, 2 ], [ 2, 1 ] ]
gap> N:=[[1,4],[1,1]];
[ [ 1, 4 ], [ 1, 1 ] ]
gap> RepresentativeDiagonalConjugation(M,N);
[ 1, 2 ]
```

105.5 Transporter

`Transporter(` *l1* `,` *l2* `)`

l1 and *l2* should be lists of the same length of square matrices all of the same size. The result is a basis of the vector space of matrices *A* such that for any *i* we have `A*l1[i]=l2[i]*A` — the basis is returned as a list, empty if the vector space is 0. This is useful to find whether two representations are isomorphic.

```
gap> W:=CoxeterGroup("A",3);
CoxeterGroup("A",3)
gap> Transporter(W.matgens,List(W.matgens,x->x^W.matgens[1]));
```

```
[ [ [ 1, 0, 0 ], [ -1, -1, 0 ], [ 0, 0, -1 ] ] ]
gap> W.matgens[1];
[ [ -1, 0, 0 ], [ 1, 1, 0 ], [ 0, 0, 1 ] ]
gap> Transporter([W.matgens[1]],[W.matgens[1]]);
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 1, 2, 0 ], [ 0, 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0, 1 ], [ 0, 0, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 1, 2, 0 ] ],
  [ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 1 ] ] ]
```

In the second case above, we get a base of the centralizer in matrices of `W.matgens[1]`.

105.6 ProportionalityCoefficient

`ProportionalityCoefficient(` *v*, *w* `)`

v and *w* should be two vectors of the same length. The function returns a scalar *c* such that `v=c*w` if such a scalar exists, and `false` otherwise.

```
gap> ProportionalityCoefficient([1,2],[2,4]);
1/2
gap> ProportionalityCoefficient([1,2],[2,3]);
false
```

105.7 ExteriorPower

`ExteriorPower(` *mat*, *n* `)`

mat should be a square matrix. The function returns the *n*-th exterior power of *mat*, in the basis naturally indexed by `Combinations([1..r],n)`, where `r=Length(<mat>)`.

```
gap> M:=[[1,2,3,4],[2,3,4,1],[3,4,1,2],[4,1,2,3]];
[ [ 1, 2, 3, 4 ], [ 2, 3, 4, 1 ], [ 3, 4, 1, 2 ], [ 4, 1, 2, 3 ] ]
gap> ExteriorPower(M,2);
[ [ -1, -2, -7, -1, -10, -13 ], [ -2, -8, -10, -10, -12, 2 ],
  [ -7, -10, -13, 1, 2, 1 ], [ -1, -10, 1, -13, 2, 7 ],
  [ -10, -12, 2, 2, 8, 10 ], [ -13, 2, 1, 7, 10, -1 ] ]
```

105.8 SymmetricPower

`SymmetricPower(` *mat*, *n* `)`

mat should be a square matrix. The function returns the *n*-th symmetric power of *mat*, in the basis naturally indexed by `UnorderedTuples([1..r],n)`, where `r=Length(<mat>)`.

```
gap> M:=[[1,2],[3,4]];
[ [ 1, 2 ], [ 3, 4 ] ]
gap> SymmetricPower(M,2);
[ [ 1, 2, 4 ], [ 6, 10, 16 ], [ 9, 12, 16 ] ]
```

105.9 SchurFunctor

`SchurFunctor(`*mat*`,`*l*`)`

mat should be a square matrix and *l* a partition. The result is the Schur functor of the matrix *mat* corresponding to partition *l*; for example, if `l=[n]` it returns the n-th symmetric power and if `l=[1,1,1]` it returns the 3rd exterior power. The current algorithm (from Littlewood) is rather inefficient so it is quite slow for partitions of n where $n > 6$.

```
gap> m:=CartanMat("A",3);
[ [ 2, -1, 0 ], [ -1, 2, -1 ], [ 0, -1, 2 ] ]
gap> SchurFunctor(m,[2,2]);
[ [ 10, 12, -16, 16, -16, 12 ], [ 3/2, 9, -6, 4, -2, 1 ],
  [ -4, -12, 16, -16, 8, -4 ], [ 2, 4, -8, 16, -8, 4 ],
  [ -4, -4, 8, -16, 16, -12 ], [ 3/2, 1, -2, 4, -6, 9 ] ]
```

105.10 IsNormalizing

`IsNormalizing(` *lst*`,` *mat* `)`

returns true or false according to whether the matrix *mat* leaves the vectors in *lst* as a set invariant, i.e., `Set(l * M) = Set( l )`.

```
gap> a := [ [ 1, 2 ], [ 3, 1 ] ];;
gap> l := [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ], [ 0, 0 ] ];;
gap> l * a;
[ [ 1, 2 ], [ 3, 1 ], [ 4, 3 ], [ 0, 0 ] ]
gap> IsNormalizing( l, a );
false
```

105.11 IndependentLines

`IndependentLines(` *M* `)`

Returns the smallest (for lexicographic order) subset I of `[1..Length(M)]` such that the rank of `M{I}` is equal to the rank of M.

```
gap> M:=CartanMat(ComplexReflectionGroup(31));
[ [ 2, 1+E(4), 1-E(4), -E(4), 0 ], [ 1-E(4), 2, 1-E(4), -1, -1 ],
  [ 1+E(4), 1+E(4), 2, 0, -1 ], [ E(4), -1, 0, 2, 0 ],
  [ 0, -1, -1, 0, 2 ] ]
gap> IndependentLines(M);
[ 1, 2, 4, 5 ]
```

105.12 OnMatrices

`OnMatrices(` *M* `,` *p*`)`

Effects the simultaneous permutation of the lines and columns of the matrix M specified by the permutation p.

```
gap> M:=DiagonalMat([1,2,3]);
[ [ 1, 0, 0 ], [ 0, 2, 0 ], [ 0, 0, 3 ] ]
gap> OnMatrices(M,(1,2,3));
[ [ 3, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 2 ] ]
```

105.13 PermutedByCols

`PermutedByCols(` *M* `,` *p*`)`

Effects the permutation p on the columns of matrix M.

```
gap> m:=List([0..2],i->3*i+[1..3]);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
gap> PermutedByCols(m,(1,2,3));
[ [ 3, 1, 2 ], [ 6, 4, 5 ], [ 9, 7, 8 ] ]
```

105.14 MatStab

`MatStab(`*M*`[,` *l*`])`

Fast implementation of `Stabilizer(SymmetricGroup(Length(M)),M,OnMatrices)`. The program uses sophisticated algorithms, and can handle matrices up to 80×80.

```
gap> uc:=UnipotentCharacters(ComplexReflectionGroup(34));
UnipotentCharacters( G34 )
gap> MatStab(Fourier(uc.families[20]));
Group( ( 7,38), (39,44)(40,43)(41,42) )
```

105.15 PermMatMat

`PermMatMat(` *M* `,` *N* `[,` *l1* `,` *l2*`])`

M and *N* should be symmetric matrices. `PermMatMat` returns a permutation p such that `OnMatrices(M,p)=N` if such a permutation exists, and `false` otherwise. If list arguments *l1* and *l2* are given, the permutation p should also satisfy `Permuted(l1,p)=l2`.

This routine is useful to identify two objects which are isomorphic but with different labelings. It is used in CHEVIE to identify Cartan matrices and Lusztig Fourier transform matrices with standard (classified) data. The program uses sophisticated algorithms, and can often handle matrices up to 80×80.

```
gap> M:=CartanMat("D",12);;
gap> p:=( 1,12, 7, 5, 9, 8, 3, 6)( 2,10)( 4,11);;
gap> N:=OnMatrices(M,p);;
gap> PermMatMat(M,N);
( 1,12, 7, 5, 9, 8, 3, 6)( 2,10)( 4,11)
```

105.16 RepresentativeRowColPermutation

`RepresentativeRowColPermutation(M1, M2)`

M1 and *M2* should be rectangular matrices of the same dimensions. The function returns a pair of permutations `[p1,p2]` such that `PermutedByCols(Permuted(m1,p1),p2)=Permuted(PermutedByCols(m1` if such permutations exist, and `false` otherwise.

```
gap> ct:=CharTable(CoxeterGroup("A",5));
CharTable( "A5" )
gap> ct1:=CharTable(Group((1,2,3,4,5,6),(1,2)));
CharTable( Group( (1,2,3,4,5,6), (1,2) ) )
gap> RepresentativeRowColPermutation(ct.irreducibles,ct1.irreducibles);
[ ( 1, 2, 5, 9, 8,10, 6,11)( 3, 7), ( 3, 4, 8, 5)( 7,10) ]
```

105.17 BigCellDecomposition

`BigCellDecomposition(M [, b])`

M should be a square matrix, and b specifies a block structure for a matrix of same size as M (it is a list of lists whose union is `[1..Length(M)]`). If b is not given, the trivial block structure `[[1],..,[Length(M)]]` is assumed.

The function decomposes M as a product P_1LP where P is upper block-unitriangular (with identity diagonal blocks), P_1 is lower block-unitriangular and L is block-diagonal for the block structure b. If M is symmetric then P_1 is the transposed of P and the result is the pair `[P,L]`; else the result is the triple `[P1,L,P]`. The only condition for this decomposition of M to be possible is that the principal minors according to the block structure be invertible. This routine is used when computing the green functions and the example below is extracted from the computation of the Green functions for G_2.

```
gap> q:=X(Rationals);;q.name:="q";;
gap> M:= [ [ q^6, q^0, q^3, q^3, q^5 + q, q^4 + q^2 ],
> [ q^0, q^6, q^3, q^3, q^5 + q, q^4 + q^2 ],
> [ q^3, q^3, q^6, q^0, q^4 + q^2, q^5 + q ],
> [ q^3, q^3, q^0, q^6, q^4 + q^2, q^5 + q ],
> [ q^5 + q, q^5 + q, q^4 + q^2, q^4 + q^2, q^6 + q^4 + q^2 + 1,
>    q^5 + 2*q^3 + q ],
>     [ q^4 + q^2, q^4 + q^2, q^5 + q, q^5 + q, q^5 + 2*q^3 + q,
>    q^6 + q^4 + q^2 + 1 ] ];;
gap> bb:=[ [ 2 ], [ 4 ], [ 6 ], [ 3, 5 ], [ 1 ] ];;
gap> PL:=BigCellDecomposition(M,bb);
[ [ [ q^0, 0*q^0, 0*q^0, 0*q^0, 0*q^0, 0*q^0 ],
      [ q^(-6), q^0, q^(-3), q^(-3), q^(-1) + q^(-5), q^(-2) + q^(-4)
         ], [ 0*q^0, 0*q^0, q^0, 0*q^0, 0*q^0, 0*q^0 ],
      [ q^(-3), 0*q^0, 0*q^0, q^0, q^(-2), q^(-1) ],
      [ q^(-1), 0*q^0, 0*q^0, 0*q^0, q^0, 0*q^0 ],
      [ q^(-2), 0*q^0, q^(-1), 0*q^0, q^(-1), q^0 ] ],
  [ [ q^6 - q^4 - 1 + q^(-2), 0*q^0, 0*q^0, 0*q^0, 0*q^0, 0*q^0 ],
      [ 0*q^0, q^6, 0*q^0, 0*q^0, 0*q^0, 0*q^0 ],
      [ 0*q^0, 0*q^0, q^6 - q^4 - 1 + q^(-2), 0*q^0, 0*q^0, 0*q^0 ],
      [ 0*q^0, 0*q^0, 0*q^0, q^6 - 1, 0*q^0, 0*q^0 ],
      [ 0*q^0, 0*q^0, 0*q^0, 0*q^0, q^6 - q^4 - 1 + q^(-2), 0*q^0 ],
      [ 0*q^0, 0*q^0, 0*q^0, 0*q^0, 0*q^0, q^6 - 1 ] ] ]
gap> M=TransposedMat(PL[1])*PL[2]*PL[1];
true
```

Chapter 106

Cyclotomic polynomials

Cyclotomic numbers, and cyclotomic polynomials over the rationals or some cyclotomic field, play an important role in the study of reductive groups, so they do in CHEVIE. Special facilities are provided to deal with them. The most prominent is the type `CycPol` which represents the product of a polynomial with a rational fraction in one variable with all poles or zeroes equal to 0 or roots of unity.

The advantages of representing as `CycPol` objects which can be so represented are: nice display (factorized), less storage, faster multiplication, division and evaluation. The big drawback is that addition and subtraction are not implemented!

```
gap> q:=X(Cyclotomics);;q.name:="q";;
gap> p:=CycPol(q^18 + q^16 + 2*q^12 + q^8 + q^6);
(1+q^2-q^4+q^6+q^8)q^6P8
gap> p/CycPol(q^2+q+1);
(1+q^2-q^4+q^6+q^8)q^6P3^-1P8
```

The variable in a `CycPol` will be denoted by `q`. It is usually printed as `q` but it is possible to change its name, see `Format` in 106.4.

`CycPol`s are represented internally by a record with fields:

`.coeff` a coefficient, usually a cyclotomic number, but it can also be a polynomial and actually can be any GAP3 object which can be multiplied by cyclotomic polynomials.

`.valuation` the valuation, positive or negative.

`.vcyc` a list of pairs $[e_i, m_i]$ representing a root of unity and a multiplicity m_i. Actually e_i should be a fraction p/d with $p < d$ representing `E(d)^p`. The pair represents $(q-\zeta_d^p)^{m_i}$.

So if we let `mu(e):=e->E(Denominator(e))^Numerator(e)`, a record `r` represents

`r.coeff*q^r.valuation*Product(r.vcyc,p->(q-mu(p[1]))^p[2])`.

106.1 AsRootOfUnity

`AsRootOfUnity(` c `)`

c should be a cyclotomic number. `AsRootOfUnity` returns the rational `e/n` with $0 \leq e < n$ (that is, $e/n \in \mathbb{Q}/\mathbb{Z}$) if `c=E(n)^e`, and false if c is not a root of unity. The code for this function has been provided by Thomas Breuer; we thank him for his help.

```
gap> AsRootOfUnity(-E(9)^2-E(9)^5);
8/9
gap> AsRootOfUnity(-E(9)^4-E(9)^5);
false
gap> AsRootOfUnity(1);
0
```

106.2 CycPol

`CycPol(` p `)`

In the first form `CycPol(` p `)` the argument is a polynomial:

```
gap> CycPol(3*q^3-3);
3P1P3
```

Special code makes the conversion fast if p has not more than two nonzero coefficients.

The second form is a fast and efficient way of specifying a CycPol with only positive multiplicities:p should be a vector. The first element is taken as a the `.coeff` of the CycPol, the second as the `.valuation`. Subsequent elements are rationals `i/d` (with $i < d$) representing `(q-E(d)^i)` or are integers d representing $\Phi_d(q)$.

```
gap> CycPol([3,-5,6,3/7]);
3q^-5P6(q-E7^3)
```

106.3 IsCycPol

`IsCycPol(` p `)`

This function returns `true` if p is a `CycPol` and `false` otherwise.

```
gap> IsCycPol(CycPol(1));
true
gap> IsCycPol(1);
false
```

106.4 Functions for CycPols

Multiplication `*` division `/` and exponentiation `^` work as usual, and the functions `Degree`, `Valuation` and `Value` work as for polynomials:

```
gap> p:=CycPol(q^18 + q^16 + 2*q^12 + q^8 + q^6);
(1+q^2-q^4+q^6+q^8)q^6P8
gap> Value(p,q);
q^18 + q^16 + 2*q^12 + q^8 + q^6
gap> p:=p/CycPol(q^2+q+1);
(1+q^2-q^4+q^6+q^8)q^6P3^-1P8
gap> Value(p,q);
Error, Cannot evaluate the non-Laurent polynomial CycPol (1+q^2-q^4+q^\
```

```
6+q^8)q^6P3^-1P8 in
f.operations.Value( f, x ) called from
Value( p, q ) called from
main loop
brk>
gap> Degree(p);
16
gap> Value(p,3);
431537382/13
```

The function `ComplexConjugate` conjugates `.coeff` as well as all the roots of unity making up the `CycPol`.

Functions `String` and `Print` display the d-th cyclotomic polynomial Φ_d over the rationals as `Pd`. They also display as `P'd`, `P"d`, `P"'d`, `P""d` factors of cyclotomic polynomials over extensions of the rationals:

```
gap> List(SchurElements(Hecke(ComplexReflectionGroup(4),q)),CycPol);
[ P2^2P3P4P6, 2ER(-3)q^-4P2^2P'3P'6, -2ER(-3)q^-4P2^2P"3P"6,
  2q^-4P3P4, (3-ER(-3))/2q^-1P2^2P'3P"6, (3+ER(-3))/2q^-1P2^2P"3P'6,
  q^-2P2^2P4 ]
```

If Φ_d factors in only two pieces, the one which has root `E(d)` is denoted `P'd` and the other one `P"d` . The list of commonly occuring factors is as follows (note that the conventions in [Car85], pages 489–490 are different):

```
P'3=q-E(3)
P"3=q-E(3)^2
P'4=q-E(4)
P"4=q+E(4)
P'5=q^2+(1-ER(5))/2*q+1
P"5=q^2+(1+ER(5))/2*q+1
P'6=q+E(3)^2
P"6=q+E(3)
P'7=q^3+(1-ER(-7))/2*q^2+(-1-ER(-7))/2*q-1
P"7=q^3+(1+ER(-7))/2*q^2+(-1+ER(-7))/2*q-1
P'8=q^2-E(4)
P"8=q^2+E(4)
P"'8=q^2-ER(2)*q+1
P""8=q^2+ER(2)*q+1
P""'8=q^2-ER(-2)*q-1
P"""8=q^2+ER(-2)*q-1
P'9=q^3-E(3)
P"9=q^3-E(3)^2
P'10=q^2+(-1-ER(5))/2*q+1
P"10=q^2+(-1+ER(5))/2*q+1
P'11=q^5+(1-ER(-11))/2*q^4-q^3+q^2+(-1-ER(-11))/2*q-1
P"11=q^5+(1+ER(-11))/2*q^4-q^3+q^2+(-1+ER(-11))/2*q-1
P'12=q^2-E(4)*q-1
P"12=q^2+E(4)*q-1
P"'12=q^2+E(3)^2
```

```
P""12=q^2+E(3)
P""'12=q^2-ER(3)*q+1
P"""12=q^2+ER(3)*q+1
P(7)12=q+E(12)^7
P(8)12=q+E(12)^11
P(9)12=q+E(12)
P(10)12=q+E(12)^5
P'13=q^6+(1-ER(13))/2*q^5+2*q^4+(-1-ER(13))/2*q^3+2*q^2+(1-ER(13))/2*q+1
P"13=q^6+(1+ER(13))/2*q^5+2*q^4+(-1+ER(13))/2*q^3+2*q^2+(1+ER(13))/2*q+1
P'14=q^3+(-1+ER(-7))/2*q^2+(-1-ER(-7))/2*q+1
P"14=q^3+(-1-ER(-7))/2*q^2+(-1+ER(-7))/2*q+1
P'15=q^4+(-1-ER(5))/2*q^3+(1+ER(5))/2*q^2+(-1-ER(5))/2*q+1
P"15=q^4+(-1+ER(5))/2*q^3+(1-ER(5))/2*q^2+(-1+ER(5))/2*q+1
P"'15=q^4+E(3)^2*q^3+E(3)*q^2+q+E(3)^2
P""15=q^4+E(3)*q^3+E(3)^2*q^2+q+E(3)
P""'15=q^2+((1+ER(5))*E(3)^2)/2*q+E(3)
P"""15=q^2+((1-ER(5))*E(3)^2)/2*q+E(3)
P(7)15=q^2+((1+ER(5))*E(3))/2*q+E(3)^2
P(8)15=q^2+((1-ER(5))*E(3))/2*q+E(3)^2
P'16=q^4-ER(2)*q^2+1
P"16=q^4+ER(2)*q^2+1
P'18=q^3+E(3)^2
P"18=q^3+E(3)
P'20=q^4+(-1-ER(5))/2*q^2+1
P"20=q^4+(-1+ER(5))/2*q^2+1
P"'20=q^4+E(4)*q^3-q^2-E(4)*q+1
P""20=q^4-E(4)*q^3-q^2+E(4)*q+1
P'21=q^6+E(3)*q^5+E(3)^2*q^4+q^3+E(3)*q^2+E(3)^2*q+1
P"21=q^6+E(3)^2*q^5+E(3)*q^4+q^3+E(3)^2*q^2+E(3)*q+1
P'22=q^5+(-1-ER(-11))/2*q^4-q^3-q^2+(-1+ER(-11))/2*q+1
P"22=q^5+(-1+ER(-11))/2*q^4-q^3-q^2+(-1-ER(-11))/2*q+1
P'24=q^4+E(3)^2
P"24=q^4+E(3)
P"'24=q^4-ER(2)*q^3+q^2-ER(2)*q+1
P""24=q^4+ER(2)*q^3+q^2+ER(2)*q+1
P""'24=q^4-ER(6)*q^3+3*q^2-ER(6)*q+1
P"""24=q^4+ER(6)*q^3+3*q^2+ER(6)*q+1
P(7)24=q^4+ER(-2)*q^3-q^2-ER(-2)*q+1
P(8)24=q^4-ER(-2)*q^3-q^2+ER(-2)*q+1
P(9)24=q^2+ER(-2)*E(3)^2*q-E(3)
P(10)24=q^2-ER(-2)*E(3)^2*q-E(3)
P(11)24=q^2+ER(-2)*E(3)*q-E(3)^2
P(12)24=q^2-ER(-2)*E(3)*q-E(3)^2
P'25=q^10+(1-ER(5))/2*q^5+1
P"25=q^10+(1+ER(5))/2*q^5+1
P'26=q^6+(-1-ER(13))/2*q^5+2*q^4+(1-ER(13))/2*q^3+2*q^2+(-1-ER(13))/2*q+1
P"26=q^6+(-1+ER(13))/2*q^5+2*q^4+(1+ER(13))/2*q^3+2*q^2+(-1+ER(13))/2*q+1
P'27=q^9-E(3)
```

```
P"27=q^9-E(3)^2
P'30=q^4+(1-ER(5))/2*q^3+(1-ER(5))/2*q^2+(1-ER(5))/2*q+1
P"30=q^4+(1+ER(5))/2*q^3+(1+ER(5))/2*q^2+(1+ER(5))/2*q+1
P"'30=q^4-E(3)*q^3+E(3)^2*q^2-q+E(3)
P""30=q^4-E(3)^2*q^3+E(3)*q^2-q+E(3)^2
P""'30=q^2+((-1+ER(5))*E(3)^2)/2*q+E(3)
P"""30=q^2+((-1-ER(5))*E(3)^2)/2*q+E(3)
P(7)30=q^2+((-1+ER(5))*E(3))/2*q+E(3)^2
P(8)30=q^2+((-1-ER(5))*E(3))/2*q+E(3)^2
P'42=q^6-E(3)^2*q^5+E(3)*q^4-q^3+E(3)^2*q^2-E(3)*q+1
P"42=q^6-E(3)*q^5+E(3)^2*q^4-q^3+E(3)*q^2-E(3)^2*q+1
```

Finally the function `Format(c,options)` takes the options:

`.vname` a string, the name to use for printing the variable of the `CycPol` instead of `q`.

`.expand` if set to `true`, each cyclotomic polynomial is replaced by its value before being printed.

```
gap> p:=CycPol(q^6-1);
P1P2P3P6
gap> Format(p,rec(expand:=true));
"(q-1)(q+1)(q^2+q+1)(q^2-q+1)"
gap> Format(p,rec(expand:=true,vname:="x"));
"(x-1)(x+1)(x^2+x+1)(x^2-x+1)"
```

Chapter 107

Partitions and symbols

The functions described below, used in various parts of the CHEVIE package, sometimes duplicate or have similar functions to some functions in other packages (like the SPECHT package). It is hoped that a review of this area will be done in the future.

The combinatorial objects dealt with here are **partitions**, **beta-sets** and **symbols**. A partition in CHEVIE is a decreasing list of strictly positive integers $p_1 \geq p_2 \geq \ldots p_n > 0$, represented as a GAP3 list. A beta-set is a GAP3 Set of positive integers, up to the **shift** equivalence relation. This equivalence relation is the transitive closure of the elementary equivalence of $[s_1, \ldots, s_n]$ and $[0, 1+s_1, \ldots, 1+s_n]$. An equivalence class has exactly one member which does not contain 0:it is called the normalized beta-set. To a partition $p_1 \geq p_2 \geq \ldots \geq p_n > 0$ is associated a beta-set, whose normalized representative is $p_n, p_{n-1}+1, \ldots, p_1+n-1$. Conversely, to each beta-set is associated a partition, the one giving by the above formula its normalized representative.

A symbol is a list $S = [S_1, .., S_n]$ of beta-sets, taken modulo the equivalence relation generated by two elementary equivalences: the simultaneous shift of all beta-sets, and the cyclic permutation of the list (in the particular case where $n = 2$ it is thus an unordered pair of beta-sets). This time there is a unique normalized symbol where 0 is not in the intersection of the S_i.

A basic invariant attached to symbols is the **rank**, defined as

`Sum(S,Sum)-QuoInt((Sum(S,Length)-1)*(Sum(S,Length)-Length(S)+1),2*Length(S))`

Another function attached to symbols is the **shape** `List(S,Length)`; when $n = 2$ one can assume that S_1 has at least the same length as S_2 and the difference of cardinals `Length(S[1])-Length(S[2])`, called the **defect**, is then an invariant of the symbol.

Partitions and pairs of partitions are parameters for characters of the Weyl groups of classical types, and tuples of partitions are parameters for characters of imprimitive complex reflection groups. Symbols with two lines are parameters for the unipotent characters of classical Chevalley groups, and more general symbols for the unipotent characters of Spetses associated to complex reflection groups. The rank of the symbol is the semi-simple rank of the corresponding Chevalley group or Spetses.

Symbols of rank n and defect 0 parameterize characters of the Weyl group of type D_n, and symbols of rank n and defect divisible by 4 parameterize unipotent characters of split orthogonal groups of dimension $2n$. Symbols of rank n and defect congruent to $2 \pmod 4$ parameterize unipotent characters of non-split orthogonal groups of dimension $2n$. Symbols of rank n and defect 1 parameterize characters of the Weyl group of type B_n, and finally symbols of rank n and odd defect parameterize unipotent characters of symplectic groups of dimension $2n$ or orthogonal groups of dimension $2n+1$.

107.1 Compositions

`Compositions(` n`[,`i`] )`

Returns the list of compositions of the integer n (the compositions with i parts if a second argument i is given).

```
gap> Compositions(4);
[ [ 1, 1, 1, 1 ], [ 2, 1, 1 ], [ 1, 2, 1 ], [ 3, 1 ], [ 1, 1, 2 ],
  [ 2, 2 ], [ 1, 3 ], [ 4 ] ]
gap> Compositions(4,2);
[ [ 3, 1 ], [ 2, 2 ], [ 1, 3 ] ]
```

107.2 PartBeta

`PartBeta(` b `)`

Here b is an increasing list of integers representing a beta-set. `PartBeta` returns corresponding the partition (see the introduction of the section for definitions).

```
gap> PartBeta([0,4,5]);
[ 3, 3 ]
```

107.3 ShiftBeta

`ShiftBeta(` b, n `)`

Here b is an increasing list of integers representing a beta-set. `ShiftBeta` returns the set shifted by n (see the introduction of the section for definitions).

```
gap> ShiftBeta([4,5],3);
[ 0, 1, 2, 7, 8 ]
```

107.4 PartitionTupleToString

`PartitionTupleToString(` *tuple* `)`

converts the partition tuple *tuple* to a string where the partitions are separated by a dot.

```
gap> d:=PartitionTuples(3,2);
[ [ [ 1, 1, 1 ], [  ] ], [ [ 1, 1 ], [ 1 ] ], [ [ 1 ], [ 1, 1 ] ],
  [ [  ], [ 1, 1, 1 ] ], [ [ 2, 1 ], [  ] ], [ [ 1 ], [ 2 ] ],
  [ [ 2 ], [ 1 ] ], [ [  ], [ 2, 1 ] ], [ [ 3 ], [  ] ],
  [ [  ], [ 3 ] ] ]
gap>  for i in d do
>      Print( PartitionTupleToString( i ),"   ");
>  od; Print("\n");
111.   11.1   1.11   .111   21.   1.2   2.1   .21   3.   .3
```

107.5 SymbolPartitionTuple

`SymbolPartitionTuple(` *p*, *s*`)`

returns the symbol of shape s associated to partition tuple p.

In the most general case, s is a list of positive integers of same length as p and the BetaSets for p are shifted accordingly (a constant integer may be added to s to make the shifts possible).

When s is a positive integer it is interpreted as `[s,0,0,...]` and a negative integer is interpreted as `[0,-s,-s,....]` so when p is a double partition one gets the symbol of defect s associated to p; as other uses the principal series of G(e,1,r) is `SymbolPartitionTuple(p,1)` and that of G(e,e,r) is `SymbolPartitionTuple(p,0)`.

Note. The function works also for periodic p for G(e,e,r) provided $s = 0$.

```
gap> SymbolPartitionTuple([[1,2],[1]],1);
[ [ 2, 2 ], [ 1 ] ]
gap> SymbolPartitionTuple([[1,2],[1]],0);
[ [ 2, 2 ], [ 0, 2 ] ]
gap> SymbolPartitionTuple([[1,2],[1]],-1);
[ [ 2, 2 ], [ 0, 1, 3 ] ]
```

107.6 Tableaux

`Tableaux(`*partition tuple* `or` *partition*`)`

returns the list of standard tableaux associated to the partition tuple *tuple*, that is a filling of the associated young diagrams with the numbers `[1..Sum(`*tuple*`,Sum)]` such that the numbers increase across the rows and down the columns. If the imput is a single partition, the standard tableaux for that partition are returned.

```
gap> Tableaux([[2,1],[1]]);
[ [ [ [ 2, 4 ], [ 3 ] ], [ [ 1 ] ] ],
  [ [ [ 1, 4 ], [ 3 ] ], [ [ 2 ] ] ],
  [ [ [ 1, 4 ], [ 2 ] ], [ [ 3 ] ] ],
  [ [ [ 2, 3 ], [ 4 ] ], [ [ 1 ] ] ],
  [ [ [ 1, 3 ], [ 4 ] ], [ [ 2 ] ] ],
  [ [ [ 1, 2 ], [ 4 ] ], [ [ 3 ] ] ],
  [ [ [ 1, 3 ], [ 2 ] ], [ [ 4 ] ] ],
  [ [ [ 1, 2 ], [ 3 ] ], [ [ 4 ] ] ] ]
gap> Tableaux([2,2]);
[ [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ] ]
```

107.7 DefectSymbol

`DefectSymbol(` *s* `)`

Let *s*`=[S,T]` be a symbol given as a pair of lists (see the introduction to the section). `DefectSymbol` returns the defect of s, equal to `Length(S)-Length(T)`.

```
gap> DefectSymbol([[1,2],[1,5,6]]);
-1
```

107.8 RankSymbol

`RankSymbol(` s `)`

Let s= $[S_1, .., S_n]$ be a symbol given as a tuple of lists (see the introduction to the section). `RankSymbol` returns the rank of s.

```
gap> RankSymbol([[1,2],[1,5,6]]);
11
```

107.9 Symbols

`Symbols(` n`,` d `)`

Returns the list of all two-line symbols of defect d and rank n (see the introduction for definitions). If $d = 0$ the symbols with equal entries are returned twice, represented as the first entry, followed by the repetition factor 2 and an ordinal number 0 or 1, so that `Symbols(`n`, 0)` returns a set of parameters for the characters of the Weyl group of type D_n.

```
gap> Symbols(2,1);
[ [ [ 1, 2 ], [ 0 ] ], [ [ 0, 2 ], [ 1 ] ], [ [ 0, 1, 2 ], [ 1, 2 ] ],
  [ [ 2 ], [  ] ], [ [ 0, 1 ], [ 2 ] ] ]
gap> Symbols(4,0);
[ [ [ 1, 2 ], 2, 0 ], [ [ 1, 2 ], 2, 1 ],
  [ [ 0, 1, 3 ], [ 1, 2, 3 ] ], [ [ 0, 1, 2, 3 ], [ 1, 2, 3, 4 ] ],
  [ [ 1, 2 ], [ 0, 3 ] ], [ [ 0, 2 ], [ 1, 3 ] ],
  [ [ 0, 1, 2 ], [ 1, 2, 4 ] ], [ [ 2 ], 2, 0 ], [ [ 2 ], 2, 1 ],
  [ [ 0, 1 ], [ 2, 3 ] ], [ [ 1 ], [ 3 ] ], [ [ 0, 1 ], [ 1, 4 ] ],
  [ [ 0 ], [ 4 ] ] ]
```

107.10 SymbolsDefect

`SymbolsDefect(` e`,` r`,` *def* `,` *inh*`)`

Returns the list of symbols defined by Malle for Unipotent characters of imprimitive Spetses. Returns e-symbols of rank r, defect *def* (equal to 0 or 1) and content equal to *inh* modulo e. Thus the symbols for unipotent characters of `G(d,1,r)` are given by `SymbolsDefect(d,r,0,1)` and those for unipotent characters of `G(e,e,r)` by `SymbolsDefect(e,r,0,0)`.

```
gap> SymbolsDefect(3,2,0,1);
[ [ [ 1, 2 ], [ 0 ], [ 0 ] ], [ [ 0, 2 ], [ 1 ], [ 0 ] ],
  [ [ 0, 2 ], [ 0 ], [ 1 ] ], [ [ 0, 1, 2 ], [ 1, 2 ], [ 0, 1 ] ],
  [ [ 0, 1 ], [ 1 ], [ 1 ] ], [ [ 0, 1, 2 ], [ 0, 1 ], [ 1, 2 ] ],
  [ [ 2 ], [  ], [  ] ], [ [ 0, 1 ], [ 2 ], [ 0 ] ],
  [ [ 0, 1 ], [ 0 ], [ 2 ] ], [ [ 1 ], [ 0, 1, 2 ], [ 0, 1, 2 ] ],
  [ [  ], [ 0, 2 ], [ 0, 1 ] ], [ [  ], [ 0, 1 ], [ 0, 2 ] ],
  [ [ 0 ], [  ], [ 0, 1, 2 ] ], [ [ 0 ], [ 0, 1, 2 ], [  ] ] ]
gap> List(last,StringSymbol);
[ "(12,0,0)", "(02,1,0)", "(02,0,1)", "(012,12,01)", "(01,1,1)",
  "(012,01,12)", "(2,,)", "(01,2,0)", "(01,0,2)", "(1,012,012)",
  "(,02,01)", "(,01,02)", "(0,,012)", "(0,012,)" ]
```

```
gap> SymbolsDefect(3,3,0,0);
[ [ [ 1 ], 3, 0 ], [ [ 1 ], 3, 1 ], [ [ 1 ], 3, 2 ],
  [ [ 0, 1 ], [ 1, 2 ], [ 0, 2 ] ], [ [ 0, 1 ], [ 0, 2 ], [ 1, 2 ] ],
  [ [ 0, 1, 2 ], [ 0, 1, 2 ], [ 1, 2, 3 ] ], [ [ 0 ], [ 1 ], [ 2 ] ],
  [ [ 0 ], [ 2 ], [ 1 ] ], [ [ 0, 1 ], [ 0, 1 ], [ 1, 3 ] ],
  [ [ 0 ], [ 0 ], [ 3 ] ], [ [ 0, 1, 2 ], [  ], [  ] ],
  [ [ 0, 1, 2 ], [ 0, 1, 2 ], [  ] ] ]
gap> List(last,StringSymbol);
[ "(1+)", "(1E3)", "(1E3^2)", "(01,12,02)", "(01,02,12)",
  "(012,012,123)", "(0,1,2)", "(0,2,1)", "(01,01,13)", "(0,0,3)",
  "(012,,)", "(012,012,)" ]
```

107.11 CycPolGenericDegreeSymbol

`CycPolGenericDegreeSymbol(` s `)`

Let s= $[S_1,..,S_n]$ be a symbol given as a tuple of lists (see the introduction to the section). `CycPolGenericDegreeSymbol` returns as a `CycPol` the generic degree of the unipotent character parameterized by s.

```
gap> CycPolGenericDegreeSymbol([[1,2],[1,5,6]]);
1/2q^13P5P6P7P8^2P9P10P11P14P16P18P20P22
```

107.12 CycPolFakeDegreeSymbol

`CycPolFakeDegreeSymbol(` s `)`

Let s= $[S_1,..,S_n]$ be a symbol given as a tuple of lists (see the introduction to the section). `CycPolFakeDegreeSymbol` returns as a `CycPol` the fake degree of the unipotent character parameterized by s.

```
gap> CycPolFakeDegreeSymbol([[1,5,6],[1,2]]);
q^16P5P7P8P9P10P11P14P16P18P20P22
```

107.13 LowestPowerGenericDegreeSymbol

`LowestPowerGenericDegreeSymbol(` s `)`

Let s=`[S1,..,Sn]` be a symbol given as a pair of lists (see the introduction to the section). `LowestPowerGenericDegreeSymbol` returns the valuation of the generic degree of the unipotent character parameterized by s.

```
gap> LowestPowerGenericDegreeSymbol([[1,2],[1,5,6]]);
13
```

107.14 HighestPowerGenericDegreeSymbol

`HighestPowerGenericDegreeSymbol(` s `)`

Let s=`[S1,..,Sn]` be a symbol given as a pair of lists (see the introduction to the section). `HighestPowerGenericDegreeSymbol` returns the degree of the generic degree of the unipotent character parameterized by s.

```
gap> HighestPowerGenericDegreeSymbol([[1,5,6],[1,2]]);
91
```

Chapter 108

Signed permutations

A **signed permutation** of `[1..n]` is a permutation of the set $\{-n,\ldots,-1,1,\ldots,n\}$ which preserves the pairs $[-i,i]$. It is represented internally as the images of $[1..n]$. It is printed as a product of signed cycles.

A signed permutation can be represented in two other ways which may be convenient. The first way is to replace the integers $\{1,\ldots,n,-n,\ldots,-1\}$ by $\{1,3,\ldots,2n-1,2,4,\ldots,2n\}$ to have GAP3 permutations, which form the hyperoctaedral group (see 83.2).

The second way is to represent the signed permutation by a monomial matrix with entries `1` or `-1`. If such a matrix *m* represents the signed permutation *sp*, then `l*m` is the same as `Permuted(`*l*`,`*sp*`)`.

108.1 Permuted for signed permutations

`Permuted(` *l*`,` *sp*`)`

`Permuted` returns a new list *n* that contains the elements of the list *l* permuted according to the signed permutation *sp*. If *sp* is given as a list, then *n*`[AbsInt(i^sp)] =` *l*`[`*i*`]SignInt(i^sp)`.

```
gap> p:=SignedPerm([-2,-1,-3]);
(1,-2)(3,-3)
gap> Permuted([20,30,40],p);
[ -30, -20, -40 ]
```

108.2 PermutationMat for signed permutations

`PermutationMat(` *sp* `[,`*d*`])`

This function returns the signed permutation matrix of the signed permutation *sp*. This is a matrix *m* such that `Permuted(l,sp)=l*m` for any list of numbers l. If an additional argument *d* is given the matrix is returned of that dimension.

```
gap> p:=SignedPerm([-2,-1,-3]);
(1,-2)(3,-3)
gap> PermutationMat(p);
[ [ 0, -1, 0 ], [ -1, 0, 0 ], [ 0, 0, -1 ] ]
```

108.3 SignedPerm

`SignedPerm(` *sp* `[,`*d* `or` *sgns*`])`

This function converts to a signed permutation a list, an element of the hyperoctaedral group, a signed permutation matrix, or a pair of a permutation and of a list of signs. If given an element of the hyperoctaedral group, the rank d of that group can be given as an argument, otherwise a representation of *sp* as a list is given corresponding to the smallest hyperoctaedral group to which it belongs.

```
gap> SignedPerm([[0,-1,0],[0,0,-1],[-1,0,0]]);
(1,-2,3,-1,2,-3)
gap> SignedPerm((1,4,5,2,3,6));
(1,-2,3,-1,2,-3)
gap> SignedPerm((1,2,3),[-1,-1,-1]);
(1,-2,3,-1,2,-3)
gap> SignedPerm([-2,-3,-1]);
(1,-2,3,-1,2,-3)
```

108.4 Cycles for signed permutations

`Cycles(` *sp* `)`

Returns the list of cycles of the signed permutation *sp* on $\{-n,\ldots,-1,1,\ldots,n\}$. If one cycle is the negative of another, only one of the two cycles is given.

```
gap> Cycles(SignedPerm([-2,-3,-1]));
[ [ 1, -2, 3, -1, 2, -3 ] ]
gap> Cycles(SignedPerm([-2,-1,-3]));
[ [ 1, -2 ], [ 3, -3 ] ]
gap> Cycles(SignedPerm([-2,-1,3]));
[ [ 1, -2 ] ]
```

108.5 SignedPermListList

`SignedPermListList(` *list1* `,` *list2* `)`

`SignedPermListList` returns a signed permutation that may be applied to *list1* to obtain *list2*, if there is one. Otherwise it returns `false`.

```
gap> SignedPermListList([20,30,40],[-40,-20,-30]);
(1,-3,2,-1,3,-2)
gap> Permuted([-40,-20,-30],last);
[ 20, 30, 40 ]
```

108.6 SignedMatStab

`SignedMatStab(`*M*`[,` *l*`])`

Finds the stabilizer of M in the group of signed permutations.

```
gap> uc:=UnipotentCharacters(ComplexReflectionGroup(6));
UnipotentCharacters( G6 )
```

```
gap> SignedMatStab(Fourier(uc.families[2]));
Group( (2,19)(4,-14)(5,20)(7,12), (1,-1)(2,-2)(3,-3)(4,-4)(5,-5)(6,-6)\
(7,-7)(8,-8)(9,-9)(10,-10)(11,-11)(12,-12)(13,-13)(14,-14)(15,-15)(16,\
-16)(17,-17)(18,-18)(19,-19)(20,-20)(21,-21)(22,-22), (1,3)(2,19)(4,-1\
4)(5,-5)(6,-18)(7,-7)(8,10)(11,15)(12,-12)(13,22)(16,21)(17,-17)(20,-2\
0), (1,6)(2,-19)(3,-18)(4,14)(8,16)(9,-9)(10,21)(11,-13)(15,-22), (1,1\
1)(3,15)(4,14)(6,-13)(7,-12)(8,-10)(9,-9)(16,-21)(18,22) )
gap> Size(last);
32
```

108.7 SignedPermMatMat

`SignedPermMatMat(` *M* , *N* `[,` *l1* , *l2*`])`

M and *N* should be symmetric matrices. `PermMatMat` returns a signed permutation *p* such that `OnMatrices(M,p)=N` if such a permutation exists, and `false` otherwise. If list arguments *l1* and *l2* are given, the permutation *p* should also satisfy `Permuted(l1,p)=l2`.

This routine is useful to identify two objects which are isomorphic but with different labelings. It is used in CHEVIE to identify Lusztig Fourier transform matrices with standard (classified) data. The program uses sophisticated algorithms, and can often handle matrices up to 80×80.

```
gap> f:=SubFamilyij(CHEVIE.families.X(12),1,3,(3+ER(-3))/2);
Family("RZ/12^2[1,3]")
gap> M:=Fourier(ComplexConjugate(f));;
gap> uc:=UnipotentCharacters(ComplexReflectionGroup(6));
UnipotentCharacters( G6 )
gap> N:=Fourier(uc.families[2]);;
gap> SignedPermMatMat(M,N);
(1,13)(2,19,-2,-19)(3,22)(4,-4)(5,-5)(6,-11)(7,12)(8,21,-8,-21)(9,-9)(\
10,16)(15,-18,-15,18)
gap> OnMatrices(M,last)=N;
true
```

Chapter 109

CHEVIE utility functions – Decimal and complex numbers

The original incentive for the functions described in this file was to get the ability to decide if a cyclotomic number which happens to be real is positive or negative (this is needed to tell if a root of an arbitrary Coxeter group is negative or positive). Of course, there are other uses for fixed-precision real and complex numbers, which the functions described here provide. A special feature of the present implementation is that to make evaluation of cyclotomics relatively fast, a cache of primitive roots of unity is maintained (the cached values are kept for the largest precision ever used to compute them).

We first describe a general facility to build complex numbers as pairs of real numbers. The real numbers in the pairs can be of any type that GAP3 knows about: integers, rationals, cyclotomics, or elements of any ring actually.

109.1 Complex

`Complex(` r`[,` i`] )`

In the first form, defines a complex number whose real part is r and imaginary part is i. If omitted, i is taken to be 0. There are two special cases when there is only one argument: if r is already a complex, it is returned; and if r is a cyclotomic number, it is converted to the pair of its real and imaginary part and returned as a complex.

```
gap> Complex(0,1);
I
gap> Complex(E(3));
-1/2+ER(3)/2I
gap> Complex(E(3)^2);
-1/2-ER(3)/2I
gap> x:=X(Rationals);;x.name:="x";;Complex(0,x);
xI
```

The last line shows that the arguments to `Complex` can be of any ring. Complex numbers are represented as a record with two fields, `.r` and `.i` holding the real and imaginary part respectively.

109.2 Operations for complex numbers

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for complex numbers. They also have `Print` and `String` methods.

```
gap> Complex(0,1);
I
gap> last+1;
1+I
gap> last^2;
2I
gap> last^2;
-4
gap> last+last2;
-4+2I
gap> x:=X(Rationals);;x.name:="x";;Complex(0,x);
xI
gap> last^2;
-x^2
```

Finally we should mention the `FormatGAP` method, which allows to print complex numbers in a way such that they can be read back in GAP3:

```
gap> a:=Complex(1/2,1/3);
1/2+1/3I
gap> FormatGAP(a);
"Complex(1/2,1/3)"
```

109.3 ComplexConjugate

`ComplexConjugate(` *c*`)`

This function applies complex conjugation to its argument. It knows ho to do this for cyclotomic numbers (it then just calls `GaloisCyc(`*c*`,-1)`), complex numbers, lists (it conjugates each element of the list), polynomials (it conjugates each coefficient), and can be taught to conjugate elements `x` of an arbitrary domain by defining `x.operations.ComplexConjugate`.

```
gap> ComplexConjugate(Complex(0,1));
-I
gap> ComplexConjugate(E(3));
E(3)^2
gap> x:=X(Cyclotomics);;x.name:="x";;ComplexConjugate(x+E(3));
x + (E(3)^2)
```

109.4 IsComplex

`IsComplex(` *c*`)`

This function returns `true` iff its argument is a complex number.

```
gap> IsComplex(Complex(1,0));
true
gap> IsComplex(E(4));
false
```

109.5 evalf

`evalf(` *c* `[,` *prec*`] )`

The name of this function intentionally mimics that of a Maple function. It computes a fixed-precision decimal number with *prec* digits after the decimal point approximating its argument; if not given, *prec* is taken to be 10 (this can be changed via the function `SetDecimalPrecision`, see below). Trailing zeroes are not shown on output, so the actual precision may be more than the number of digits shown.

```
gap> evalf(1/3);
0.3333333333
```

As one can see, the resulting `decimal numbers` have an appropriate Print method (which uses the String method).

```
gap> evalf(1/3,20);
0.33333333333333333333
```

`evalf` can also be applied to cyclotomic or complex numbers, yielding a complex which is a pair of decimal numbers.

```
gap> evalf(E(3));
-0.5+0.8660254038I
```

`evalf` works also for strings (the result is truncated if too precise)

```
gap> evalf(".3450000000000000000001"); # precision is 10
0.345
```

and for lists (it is applied recursively to each element).

```
gap> evalf([E(5),1/7]);
[ 0.3090169944+0.9510565163I, 0.1428571429 ]
```

Finally, an `evalf` method can be defined for elements of any domain. One has been defined in CHEVIE for complex numbers:

```
gap> a:=Complex(1/2,1/3);
1/2+1/3I
gap> evalf(a);
0.5+0.3333333333I
```

109.6 Rational

`Rational(`*d*`)`

d is a `decimal` number. The function returns the rational number which is actually represented by *d*

```
gap> evalf(1/3);
0.3333333333
gap> Rational(last);
3333333333/10000000000
```

109.7 SetDecimalPrecision

`SetDecimalPrecision(` *prec* `)`

This function sets the default precision to be used when converting numbers to decimal numbers without giving a second argument to `evalf`.

```
gap> SetDecimalPrecision(20);
gap> evalf(1/3);
0.33333333333333333333
gap> SetDecimalPrecision(10);
```

109.8 Operations for decimal numbers

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for decimal numbers, as well as the function `GetRoot` and the comparison functions `<`, `>`, etc... The precision of the result of an operation is that of the least precise number used. They can be raised to a fractional power: `GetRoot(d,n)` is equivalent to `d^(1/n)`. Decimal numbers also have `Print` and `String` methods.

```
gap> evalf(1/3)+1;
1.3333333333
gap> last^3;
2.3703703704
gap> evalf(E(3));
-0.5+0.8660254038I
gap> last^3;
1
gap> evalf(ER(2));
1.4142135624
gap> GetRoot(evalf(2),2);
1.4142135624
gap> evalf(2)^(1/2);
1.4142135624
gap> evalf(1/3,20);
0.33333333333333333333
gap> last+evalf(1);
1.3333333333
gap> last2+1;
1.33333333333333333333
```

Finally we should mention the `FormatGAP` method, which, given option `GAP`, allows to print decimal numbers in a way such that they can be read back in GAP3:

```
gap> FormatGAP(evalf(1/3));
"evalf(33333333333/100000000000,10)"
```

109.9 Pi

`Pi(` [*prec*]`)`

This function returns a decimal approximation to π, with *prec* digits (or if *prec* is omitted with the default umber of digits defined by `SetDecimalPrecision`, initially 10).

```
gap> Pi();
3.1415926536
gap> Pi(34);
3.1415926535897932384626433832795029
```

109.10 Exp

`Exp(` x`)`

This function returns the complex exponential of x. The argument should be a decimal or a decimal complex. The result has as many digits of precision as the argument.

```
gap> Exp(evalf(1));
2.7182818285
gap> Exp(evalf(1,20));
2.71828182845904523536
gap> Exp(Pi()*E(4));
-1
```

The code of `Exp` shows how easy it is to use decimal numbers.

```
gap> Print(Exp,"\n");
function ( x )
    local  res, i, p, z;
    if IsCyc( x )  then
        x := evalf( x );
    fi;
    z := 0 * x;
    res := z;
    p := 1;
    i := 1;
    while p <> z  do
        res := p + res;
        p := 1 / i * p * x;
        i := i + 1;
    od;
    return res;
end
```

109.11 IsDecimal

`IsDecimal(` x`)`

returns `true` iff x is a decimal number.

```
gap> IsDecimal(evalf(1));
true
gap> IsDecimal(evalf(E(3)));
false
```

Chapter 110

Posets and relations

Posets are represented in CHEVIE as records where at least one of the two following fields is present:

`.incidence` a boolean matrix such that `.incidence[i][j]=true` iff `i<=j` in the poset.

`.hasse` a list representing the Hasse diagram of the poset:the i-th entry is the list of indices of elements which are immediate successors (covers) of the i-th element, that is the list of j such that `i<j` and such that there is no k such that `i<k<j`.

If only one field is present, the other is computed on demand. Here is an example of use;

```
gap> P:=BruhatPoset(CoxeterGroup("A",2));
Poset with 6 elements
gap> Display(P);
<1,2<21,12<121
gap> Hasse(P);
[ [ 2, 3 ], [ 4, 5 ], [ 4, 5 ], [ 6 ], [ 6 ], [  ] ]
gap> Incidence(P);
[ [ true, true, true, true, true, true ],
  [ false, true, false, true, true, true ],
  [ false, false, true, true, true, true ],
  [ false, false, false, true, false, true ],
  [ false, false, false, false, true, true ],
  [ false, false, false, false, false, true ] ]
```

110.1 TransitiveClosure of incidence matrix

`TransitiveClosure(`M`)`

M should be a square boolean matrix representing a relation; returns a boolean matrix representing the transitive closure of this relation. The transitive closure is computed by the Floyd-Warshall algorithm, which is quite fast even for large matrices.

```
gap> M:=List([1..5],i->List([1..5],j->j-i in [0,1]));
[ [ true, true, false, false, false ],
```

```
  [ false, true, true, false, false ],
  [ false, false, true, true, false ],
  [ false, false, false, true, true ],
  [ false, false, false, false, true ] ]
gap> PrintArray(TransitiveClosure(M));
[[ true,  true,  true,  true, true],
 [false,  true,  true,  true, true],
 [false, false,  true,  true, true],
 [false, false, false,  true, true],
 [false, false, false, false, true]]
```

110.2 LcmPartitions

`LcmPartitions(`*p1*`,...,`*pn*`)` Each argument is a partition of the same set `S`, represented by a list of disjoint subsets whose union is `S`. Equivalently each argument represents an equivalence relation on `S`.

The result is the finest partition of `S` such that each argument partition refines it. It represents the `or` of the equivalence relations represented by the arguments.

```
gap> LcmPartitions([[1,2],[3,4],[5,6]],[[1],[2,5],[3],[4],[6]]);
[ [ 1, 2, 5, 6 ], [ 3, 4 ] ]
```

110.3 GcdPartitions

`GcdPartitions(`*p1*`,...,`*pn*`)` Each argument is a partition of the same set `S`, represented by a list of disjoint subsets whose union is `S`. Equivalently each argument represents an equivalence relation on `S`.

The result is the coarsest partition which refines all argument partitions. It represents the `and` of the equivalence relations represented by the arguments.

```
gap> GcdPartitions([[1,2],[3,4],[5,6]],[[1],[2,5],[3],[4],[6]]);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ] ]
```

110.4 Poset

`Poset(`*M*`)`

`Poset(`*H*`)`

Creates a poset from either an incidence matrix M such that `M[i][j]=true` if and only if `i<=j` in the poset, or a Hasse diagram H given as a list whose i-th entry is the list of indices of elements which are immediate successors (covers) of the i-th element, that is `M[i]` is the list of j such that `i<j` in the poset and such that there is no k such that `i<k<j`.

`Poset(arg)`

In this last form `arg[1]` should be a record with a field `.operations` and the functions calls `ApplyFunc(arg[1].operations.Poset,arg)`.

A poset is represented as a record with the following fields.

`.incidence` the incidence matrix.

`.hasse` the Hasse diagram.

Since the cost of computing one from the other is high, the above fields are optional (only one of them needs to be present) and the other is computed on demand.

`.size` the number of elements of the poset.

Finally, an optional field `.label` may be given for formatting or display purposes. It should be a function `label(P,i,opt)` which returns a label for the `i`-th element of the poset `P`, formatted according to the options (if any) given in the options record `opt`.

110.5 Hasse

`Hasse(`*P*`)`

returns the Hasse diagram of the poset *P*.

```
gap> p:=Poset(List([1..5],i->List([1..5],j->j mod i=0)));
Poset with 5 elements
gap> Hasse(p);
[ [ 2, 3, 5 ], [ 4 ], [  ], [  ], [  ] ]
```

110.6 Incidence

`Incidence(`*P*`)`

returns the Incidence matrix of the poset *P*.

```
gap> p:=Poset(Concatenation(List([1..5],i->[i+1]),[[]]));
Poset with 6 elements
gap> Incidence(p);
[ [ true, true, true, true, true, true ],
  [ false, true, true, true, true, true ],
  [ false, false, true, true, true, true ],
  [ false, false, false, true, true, true ],
  [ false, false, false, false, true, true ],
  [ false, false, false, false, false, true ] ]
```

110.7 LinearExtension

`LinearExtension(`*P*`)`

returns a linear extension of the poset *P*, that is a list `l` containing a permutation of the integers `[1..Size(P)]` such that if `i<j` in *P*, then `Position(l,i)<Position(l,j)`. This is also called a topological sort of *P*.

```
gap> p:=Poset(List([1..5],i->List([1..5],j->j mod i=0)));
Poset with 5 elements
gap> Display(p);
1<2<4
1<3,5
gap> LinearExtension(p);
[ 1, 2, 3, 5, 4 ]
```

110.8 Functions for Posets

The function `Size` returns the number of elements of the poset.

The functions `String` and `Print` just indicate the `Size` of the poset.

The functions `Format` and `Display` show the poset as a list of maximal covering chains, with formatting depending on their record of options. They take in account the associated partition (see 110.9) to give a more compact description where equivalent elements are listed together, separated by commas.

```
gap> p:=Poset(UnipotentClasses(ComplexReflectionGroup(28)));
Poset with 16 elements
gap> Display(p);
1<A1<~A1<A1+~A1<A2<A2+~A1<~A2+A1<C3(a1)<F4(a3)<C3,B3<F4(a2)<F4(a1)<F4
A1+~A1<~A2<~A2+A1
A2+~A1<B2<C3(a1)
```

110.9 Partition for posets

`Partition(`P`)`

returns the partition of `[1..Size(P)]` determined by the equivalence relation associated to P; that is, `i` and `j` are in the same part of the partition if the relations `i<k` and `j<k` as well are `k<i` and `k<j` are equivalent for any `k` in the poset.

```
gap> p:=Poset(List([1..8],i->List([1..8],j->i=j or (i mod 4)<(j mod 4))));
Poset with 8 elements
gap> Display(p);
4,8<1,5<2,6<3,7
gap> Partition(p);
[ [ 4, 8 ], [ 2, 6 ], [ 3, 7 ], [ 1, 5 ] ]
```

110.10 Restricted for Posets

Restricted(P,*indices*)

returns the sub-poset of P determined by *indices*, which must be a sublist of `[1..Size(P)]`.

```
gap> Display(p);
4,8<1,5<2,6<3,7
gap> Display(Restricted(p,[2..6]));
3<4<1,5<2
```

110.11 Reversed for Posets

Reversed(P)

returns the opposed poset to P.

```
gap> Display(p);
4,8<1,5<2,6<3,7
gap> Display(Reversed(p));
3,7<2,6<1,5<4,8
```

110.12 IsJoinLattice

IsJoinLattice(P)

returns true if P is a join semilattice, that is any two elements of P have a unique smallest upper bound. It returns false otherwise.

```
gap> Display(p);
4,8<1,5<2,6<3,7
gap> IsJoinLattice(p);
false
```

110.13 IsMeetLattice

IsMeetLattice(P)

returns true if P is a meet semilattice, that is any two elements of P have a unique highest lower bound. It returns false otherwise.

```
gap> Display(p);
4,8<1,5<2,6<3,7
gap> IsMeetLattice(p);
false
```

Chapter 111

The VKCURVE package

The main function of the VKCURVE package computes the fundamental group of the complement of a complex algebraic curve in $\mathbb{C}^2$, using an implementation of the Van Kampen method (see for example [Che73] for a clear and modernized account of this method).

```
gap> FundamentalGroup(x^2-y^3);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba

gap> FundamentalGroup((x+y)*(x-y)*(x+2*y));
# I there are 3 generators and 2 relators of total length 12
1: cab=abc
2: bca=abc
```

The input is a polynomial in the two variables `x` and `y`, with rational coefficients. Though approximate calculations are used at various places, they are controlled and the final result is exact.

The output is a record which contains lots of information about the computation, including a presentation of the computed fundamental group, which is what is displayed when printing the record.

Our motivation for writing this package was to find explicit presentations for generalized braid groups attached to certain complex reflection groups. Though presentations were known for almost all cases, six exceptional cases were missing (in the notations of Shephard and Todd, these cases are G_{24}, G_{27}, G_{29}, G_{31}, G_{33} and G_{34}). Since the a priori existence of nice presentations for braid groups was proved in [Bes01], it was upsetting not to know them explicitly. In the absence of any good grip on the geometry of these six examples, brute force was a way to get an answer. Using VKCURVE, we have obtained presentations for all of them.

This package was developed thanks to computer resources of the Institut de Mathématiques de Jussieu in Paris. We thank the computer support team, especially Joël Marchand, for the stability and the efficiency of the working environment.

We have tried to design this package with the novice GAP3 user in mind. The only steps required to use it are

- Run GAP33 (the package is not compatible with GAP34).

- Make sure the packages CHEVIE and VKCURVE are loaded (beware that we require the development version of CHEVIE, `http://www.math.jussieu.fr/~jmichel/chevie.html` and not the one in the GAP3.3.3.4 distribution)
- Use the function `FundamentalGroup`, as demonstrated in the above examples.

If you are not interested in the details of the algorithm, and if `FundamentalGroup` gives you satisfactory answers in a reasonable time, then you do not need to read this manual any further.

We use our own package for multivariate polynomials which is more effective, for our purposes, than the default in GAP33 (see `Mvp`). When VKCURVE is loaded, the variables `x` and `y` are pre-defined as `Mvps`; one can also use GAP3 polynomials (which will be converted to `Mvps`).

The implementation uses `Decimal` numbers, `Complex` numbers and braids as implemented in the (development version of the) package CHEVIE, so VKCURVE is dependent on this package.

To implement the algorithms, we needed to write auxiliary facilities, for instance find zeros of complex polynomials, or work with piecewise linear braids, which may be useful on their own. These various facilities are documented in this manual.

Before discussing our actual implementation, let us give an informal summary of the mathematical background. Our strategy is adapted from the one originally described in the 1930's by Van Kampen. Let C be an affine algebraic curve, given as the set of zeros in $\mathbb{C}^2$ of a non-zero reduced polynomial $P(x,y)$. The problem is to compute a presentation of the fundamental group of $\mathbb{C}^2 - C$. Consider P as a polynomial in x, with coefficients in the ring of polynomials in y

$$P = \alpha_0(y)x^n + \alpha_1(y)x^{n-1} + \ldots + \alpha_{n-1}(y)x + \alpha_n(y),$$

where the α_i are polynomials in y. Let $\Delta(y)$ be the discriminant of P or, in other words, the resultant of P and $\frac{\partial P}{\partial x}$. Since P is reduced, Δ is non-zero. For a generic value of y, the polynomial in x given by $P(x,y)$ has n distinct roots. When $y = y_j$, with j in $1,\ldots,d$, we are in exactly one of the following situations: either $P(x,y_j) = 0$ (we then say that y_j is bad), or $P(x,y_j)$ has a number of roots in x strictly smaller than n. Fix y_0 in $\mathbb{C} - \{y_1,\ldots,y_d\}$. Consider the projection $p : \mathbb{C}^2 \to \mathbb{C}, (x,y) \mapsto y$. It restricts to a locally trivial fibration with base space $B = \mathbb{C} - \{y_1,\ldots,y_d\}$ and fibers homeomorphic to the complex plane with n points removed. We denote by E the total space $p^{-1}(B)$ and by F the fiber over y_0. The fundamental group of F is isomorphic to the free group on n generators. Let $\gamma_1,\ldots,\gamma_d$ be loops in the pointed space (B,y_0) representing a generating system for $\pi_1(B,y_0)$. By trivializing the pullback of p along γ_i, one gets a (well-defined up to isotopy) homeomorphism of F, and a (well-defined) automorphism ϕ_i of the fundamental group of F, identified with the free group F_n by the choice of a generating system $f_1,\ldots,f_n$. An effective way of computing ϕ_i is by following the solutions in x of $P(x,y) = 0$, when y moves along ϕ_i. This defines a loop in the space of configuration of n points in a plane, hence an element b_i of the braid group B_n (via an identification of B_n with the fundamental group of this configuration space). Let ϕ be the Hurwitz action of B_n on F_n. All choices can be made in such a way that $\phi_i = \phi(b_i)$. The theorem of Van Kampen asserts that, if there are no bad roots of the discriminant, a presentation for the fundamental group of $\mathbb{C}^2 - C$ is

$$< f_1, \ldots, f_n \mid \forall i, j, \phi_i(f_j) = f_j >$$

A variant of the above presentation (see `VKQuotient`) can be used to deal with bad roots of the discriminant.

This algorithm is implemented in the following way.

- As input, we have a polynomial P. The polynomial is reduced if it was not.
- The discriminant Δ of P with respect to x is computed. It is a polynomial in y.
- The roots of Δ are approximated, via the following procedure. First, we reduce Δ and get Δ_{red} (generating the radical of the ideal generated by Δ). The roots $\{y_1, \ldots, y_d\}$ of Δ_{red} are separated by `SeparateRoots` (which implements Newton's method).
- Loops around these roots are computed by `LoopsAroundPunctures`. This function first computes some sort of honeycomb, consisting of a set S of affine segments, isolating the y_i. Since it makes the computation of the monodromy more effective, each inner segment is a fragment of the mediatrix of two roots of Δ. Then a vertex of one the segments is chosen as a basepoint, and the function returns a list of lists of oriented segments in S: each list of segment encodes a piecewise linear loop γ_i circling one of y_i.
- For each segment in S, we compute the monodromy braid obtained by following the solutions in x of $P(x, y) = 0$ when y moves along the segment. By default, this monodromy braid is computed by `FollowMonodromy`. The strategy is to compute a piecewise-linear braid approximating the actual monodromy geometric braid. The approximations are controlled. The piecewise-linear braid is constructed step-by-step, by computations of linear pieces. As soon as new piece is constructed, it is converted into an element of B_n and multiplied; therefore, though the braid may consist of a huge number of pieces, the function `FollowMonodromy` works with constant memory. The packages also contains a variant function `ApproxFollowMonodromy`, which runs faster, but without guarantee on the result (see below).
- The monodromy braids b_i corresponding to the loops γ_i are obtained by multiplying the corresponding monodromy braids of segments. The action of these elements of B_n on the free group F_n is computed by `BnActsOnFn` and the resulting presentation of the fundamental group is computed by `VKQuotient`. It happens for some large problems that the whole fundamental group process fails here, because the braids b_i obtained are too long and the computation of the action on F_n requires thus too much memory. We have been able to solve such problems when they occur by calling on the b_i at this stage our function `ShrinkBraidGeneratingSet` which finds smaller generators for the subgroup of B_n generated by the b_i (see the description in the third chapter). This function is called automatically at this stage if `VKCURVE.shrinkBraid` is set to `true` (the default for this variable is `false`).
- Finally, the presentation is simplified by `ShrinkPresentation`. This function is a heuristic adaptation and refinement of the basic GAP3 functions for simplifying presentations. It is non-deterministic.

From the algorithmic point of view, memory should not be an issue, but the procedure may take a lot of CPU time (the critical part being the computation of the monodromy braids by `FollowMonodromy`). For instance, an empirical study with the curves $x^2 - y^n$ suggests that the needed time grows exponentially with n. Two solutions are offered to deal with curves for which the computation time becomes unreasonable.

A global variable `VKCURVE.monodromyApprox` controls which monodromy function is used. The default value of this variable is `false`, which means that `FollowMonodromy` will be used. If the variable is set by the user to `true` then the function `ApproxFollowMonodromy` will be used instead. This function runs faster than `FollowMonodromy`, but the approximations are no longer controlled. Therefore presentations obtained while `VKCURVE.monodromyApprox` is set to `true` are not certified. However, though it is likely that there exists examples for which `ApproxFollowMonodromy` actually returns incorrect answers, we still have not seen one.

The second way of dealing with difficult examples is to parallelize the computation. Since the computations of the monodromy braids for each segment are independent, they can be performed simultaneously on different computers. The functions `PrepareFundamentalGroup`, `Segments` and `FinishFundamentalGroup` provide basic support for parallel computing.

111.1 FundamentalGroup

`FundamentalGroup(`*curve* `[,` *printlevel*`])`

curve should be an `Mvp` in x and y, or a GAP3 polynomial in two variables (which means a polynomial in a variable which is assumed to be `y` over the polynomial ring $\mathbb{Q}[x]$) representing an equation $f(x, y)$ for a curve in $\mathbb{C}^2$. The coefficients should be rationals, gaussian rationals or `Complex` rationals. The result is a record with a certain number of fields which record steps in the computation described in this introduction:

```
gap> r:=FundamentalGroup(x^2-y^3);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba

gap> RecFields(r);
[ "curve", "discy", "roots", "dispersal", "points", "segments", "loops",
  "zeros", "B", "monodromy", "basepoint", "dispersal", "braids",
  "presentation","operations" ]
gap> r.curve;
x^2-y^3
gap> r.discy;
X(Rationals)
gap> r.roots;
[ 0 ]
gap> r.points;
[ -I, -1, 1, I ]
gap> r.segments;
[ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ]
gap> r.loops;
```

```
[ [ 4, -3, -1, 2 ] ]
gap> r.zeros;
[ [ 707106781187/1000000000000+707106781187/1000000000000I,
    -707106781187/1000000000000-707106781187/1000000000000I ],
  [ I, -I ], [ 1, -1 ],
  [ -707106781187/1000000000000+707106781187/1000000000000I,
  707106781187/1000000000000-707106781187/1000000000000I ] ]
gap> r.monodromy;
[ (w0)^-1, w0, , w0 ]
gap> r.braids;
[ w0.w0.w0 ]
gap> DisplayPresentation(r.presentation);
1: bab=aba
```

Here `r.curve` records the entered equation, `r.discy` its discriminant with respect to x, `r.roots` the roots of this discriminant, `r.points`, `r.segments` and `r.loops` describes loops around these zeros as explained in the documentation of `LoopsAroundPunctures`; `r.zeros` records the zeros of $f(x, y_i)$ when y_i runs over the various `r.points`; `r.monodromy` records the monodromy along each of `r.segments`, and `r.braids` is the resulting monodromy along the loops. Finally `r.presentation` records the resulting presentation (which is what is printed by default when `r` is printed).

The second optional argument triggers the display of information on the progress of the computation. It is recommended to set the *printlevel* at 1 or 2 when the computation seems to take a long time without doing anything. *printlevel* set at 0 is the default and prints nothing; set at 1 it shows which segment is currently active, and set at 2 it traces the computation inside each segment.

```
gap> FundamentalGroup(x^2-y^3,1);
# There are 4 segments in 1 loops
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[1]:=B(-1);
# segment 1/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[2]:=B(1);
# segment 2/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[3]:=B();
# segment 3/4 Time=0sec
# The following braid was computed by FollowMonodromy in 8 steps.
monodromy[4]:=B(1);
# segment 4/4 Time=0sec
# Computing monodromy braids
# loop[1]=w0.w0.w0
# I there are 2 generators and 1 relator of total length 6
1: bab=aba
```

111.2 PrepareFundamentalGroup

`PrepareFundamentalGroup(`*curve*`, `*name*`)`

`VKCURVE.Segments(`*name*`[,`*range*`])`

`FinishFundamentalGroup(`*r*`)`

These functions provide a means of distributing a fundamental group computation over several machines. The basic strategy is to write to a file the startup-information necessary to compute the monodromy along a segment, in the form of a partially-filled version of the record returned by `FundamentalGroup`. Then the monodromy along each segment can be done in a separate process, writing again the result to files. These results are then gathered and processed by `FinishFundamentalGroup`. The whole process is illustrated in an example below. The extra argument *name* to `PrepareFundamentalGroup` is a prefix used to name intermediate files. One does first :

```
gap> PrepareFundamentalGroup(x^2-y^3,"a2");
    ----------------------------------
Data saved in a2.tmp
You can now compute segments 1 to 4
in different GAP sessions by doing in each of them:
    a2:=rec(name:="a2");
    VKCURVE.Segments(a2,[1..4]);
(or some other range depending on the session)
Then when all files a2.xx have been computed finish by
    a2:=rec(name:="a2");
    FinishFundamentalGroup(a2);
```

Then one can compute in separate sessions the monodromy along each segment. The second argument of `Segments` tells which segments to compute in the current session (the default is all). An example of such sessions may be:

```
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> VKCURVE.Segments(a2,[2]);
# The following braid was computed by FollowMonodromy in 8 steps.
a2.monodromy[2]:=a2.B(1);
# segment 2/4 Time=0.1sec
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> VKCURVE.Segments(a2,[1,3,4]);
# The following braid was computed by FollowMonodromy in 8 steps.
a2.monodromy[2]:=a2.B(1);
# segment 2/4 Time=0.1sec
```

When all segments have been computed the final session looks like

```
gap> a2:=rec(name:="a2");
rec(
  name := "a2" )
gap> FinishFundamentalGroup(a2);
1: bab=aba
```

Chapter 112

Multivariate polynomials and rational fractions

The functions described in this file were written to alleviate the deficiency of GAP3 in manipulating multi-variate polynomials. In GAP3 one can only define one-variable polynomials over a given ring; this allows multi-variate polynomials by taking this ring to be a polynomial ring; but, in addition to providing little flexibility in the choice of coefficients, this "full" representation makes for somewhat inefficient computation. The use of the `Mvp` (MultiVariate Polynomials) described here is faster than GAP3 polynomials as soon as there are two variables or more. What is implemented here is actually "Puiseux polynomials", i.e. linear combinations of monomials of the type $x_1^{a_1}\ldots x_n^{a_n}$ where x_i are variables and a_i are exponents which can be arbitrary rational numbers. Some functions described below need their argument to involve only variables to integral powers; we will refer to such objects as "Laurent polynomials"; some functions require further that variables are raised only to positive powers: we refer then to "true polynomials". Rational fractions (`RatFrac`) have been added, thanks to work of Gwenaëlle Genet (the main difficulty there was to write an algorithm for the Gcd of multivariate polynomials, a non-trivial task). The coefficients of our polynomials can in principle be elements of any ring, but some algorithms like division or Gcd require the coefficients of their arguments to be invertible.

112.1 Mvp

`Mvp(` *string s* `[,` *coeffs v*`] )`

Defines an indeterminate with name *s* suitable to build multivariate polynomials.

```
gap> x:=Mvp("x");y:=Mvp("y");(x+y)^3;
x
y
3xy^2+3x^2y+x^3+y^3
```

If a second argument (a vector of coefficients v) is given, returns `Sum([1..Length(v)],i->Mvp(s)^(i-1)*v[i])`.

```
gap> Mvp("a",[1,2,0,4]);
1+2a+4a^3
```

`Mvp(` *polynomial x*`)`

Converts the GAP3 polynomial x to an `Mvp`. It is an error if `x.baseRing.indeterminate.name` is not bound; otherwise this is taken as the name of the `Mvp` variable.

```
gap> q:=Indeterminate(Rationals);
X(Rationals)
gap> Mvp(q^2+q);
Error, X(Rationals) should have .name bound in
Mvp( q ^ 2 + q ) called from
main loop
brk>
gap> q.name:="q";;
gap> Mvp(q^2+q);
q+q^2
```

`Mvp(` *FracRat x*`)`

Returns `false` if the argument rational fraction is not in fact a Laurent polynomial. Otherwise returns that polynomial.

```
gap> Mvp(x/y);
xy^-1
gap> Mvp(x/(y+1));
false
```

`Mvp(` *elm*`,` *coeff*`)`

Build efficiently an `Mvp` from the given list of coefficients and the list *elm* describing the corresponding monomials. A monomial is itself described by a record with a field `.elm` containing the list of involved variable names and a field `.coeff` containing the list of corresponding exponents.

```
gap> Mvp([rec(elm:=["y","x"],coeff:=[1,-1])],[1]);
x^-1y
```

`Mvp(` *scalar x*`)`

A scalar is anything which is not one of the previous types (like a cyclotomic, or a finite-field-element, etc). Returns the constant multivariate polynomial whose constant term is x.

```
gap> Degree(Mvp(1));
0
```

112.2 Operations for Mvp

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for `Mvp`s. They also have `Print` and `String` methods. The operations `+`, `-`, `*` work for any inputs. `/` works only for Laurent polynomials, and may return a rational fraction (see below); if one is sure that the division is exact, one should call `MvpOps.ExactDiv` (see below).

```
gap> x:=Mvp("x");y:=Mvp("y");
x
y
gap> a:=x^(-1/2);
```

```
x^(-1/2)
gap> (a+y)^4;
x^-2+4x^(-3/2)y+6x^-1y^2+4x^(-1/2)y^3+y^4
gap> (x^2-y^2)/(x-y);
x+y
gap> (x-y^2)/(x-y);
(x-y^2)/(x-y)
gap> (x-y^2)/(x^(1/2)-y);
Error, x^(1/2)-y is not a polynomial with respect to x
 in
V.operations.Coefficients( V, v ) called from
Coefficients( q, var ) called from
MvpOps.ExactDiv( x, q ) called from
fun( arg[1][i] ) called from
List( p, function ( x ) ... end ) called from
...
brk>
```

Only monomials can be raised to a non-integral power; they can be raised to a fractional power of denominator `b` only if `GetRoot(x,b)` is defined where `x` is their leading coefficient. For an `Mvp` m, the function `GetRoot(m,n)` is equivalent to `m^(1/n)`. Raising a non-monomial Laurent polynomial to a negative power returns a rational fraction.

```
gap> (2*x)^(1/2);
ER(2)x^(1/2)
gap> (evalf(2)*x)^(1/2);
1.4142135624x^(1/2)
gap> GetRoot(evalf(2)*x,2);
1.4142135624x^(1/2)
```

The `Degree` of a monomial is the sum of the exponent of the variables. The `Degree` of an `Mvp` is the largest degree of a monomial.

```
gap> a;
x^(-1/2)
gap> Degree(a);
-1/2
gap> Degree(a+x);
1
gap> Degree(Mvp(0));
-1
```

There exists also a form of `Degree` taking as second argument a variable name, which returns the degree of the polynomial in that variable.

```
gap> p:=x/y;
xy^-1
gap> Degree(p,"x");
1
gap> Degree(p,"y");
-1
gap> Degree(p);
```

```
0
```

The `Valuation` of an `Mvp` is the minimal degree of a monomial.

```
gap> a;
x^(-1/2)
gap> Valuation(a);
-1/2
gap> Valuation(a+x);
-1/2
gap> Valuation(Mvp(0));
-1
```

There exists also a form of `Valuation` taking as second argument a variable name, which returns the valuation of the polynomial in that variable.

```
gap> Valuation(x^2+y^2);
2
gap> Valuation(x^2+y^2,"x");
0
gap> Valuation(x^2+y^2,"y");
0
```

The `Format` routine formats `Mvp`s in such a way that they can be read back in by GAP3 or by some other systems, by giving an appropriate option as a second argument, or using the functions `FormatGAP`, `FormatMaple` or `FormatTeX`. The `String` method is equivalent to `Format`, and gives a compact display.

```
gap> p:=7*x^5*y^-1-2;
-2+7x^5y^-1
gap> Format(p);
"-2+7x^5y^-1"
gap> FormatGAP(p);
"-2+7*x^5*y^-1"
gap> FormatMaple(p);
"-2+7*x^5*y^(-1)"
gap> FormatTeX(p);
"-2+7x^5y^{-1}"
```

The `Value` method evaluates an `Mvp` by fixing simultaneously the value of several variables. The syntax is `Value(x, [` *string1* `,` *value1* `,` *string2* `,` *value2* `,` ... `])`.

```
gap> p;
-2+7x^5y^-1
gap> Value(p,["x",2]);
-2+224y^-1
gap> Value(p,["y",3]);
-2+7/3x^5
gap> Value(p,["x",2,"y",3]);
218/3
```

One should pay attention to the fact that the last value is not a rational number, but a constant `Mvp` (for consistency). See the function `ScalMvp` below for how to convert such constants to their base ring.

```
gap> Value(p,["x",y]);
-2+7y^4
gap> Value(p,["x",y,"y",x]);
-2+7x^-1y^5
```

Evaluating an Mvp which is a Puiseux polynomial may cause calls to GetRoot

```
gap> p:=x^(1/2)*y^(1/3);
x^(1/2)y^(1/3)
gap> Value(p,["x",y]);
y^(5/6)
gap>  Value(p,["x",2]);
ER(2)y^(1/3)
gap>  Value(p,["y",2]);
Error, : unable to compute 3-th root of 2
 in
GetRoot( values[i], d[i] ) called from
f.operations.Value( f, x ) called from
Value( p, [ "y", 2 ] ) called from
main loop
brk>
```

The function Derivative(p,v) returns the derivative of p with respect to the variable given by the string v; if v is not given, with respect to the first variable in alphabetical order.

```
gap>  p:=7*x^5*y^-1-2;
-2+7x^5y^-1
gap> Derivative(p,"x");
35x^4y^-1
gap> Derivative(p,"y");
-7x^5y^-2
gap> Derivative(p);
35x^4y^-1
gap>  p:=x^(1/2)*y^(1/3);
x^(1/2)y^(1/3)
gap>  Derivative(p,"x");
1/2x^(-1/2)y^(1/3)
gap>  Derivative(p,"y");
1/3x^(1/2)y^(-2/3)
gap>  Derivative(p,"z");
0
```

The function Coefficients(*p*, *var*) is defined only for Mvps which are polynomials in the variable *var* . It returns as a list the list of coefficients of *p* with respect to *var*.

```
gap> p:=x+y^-1;
y^-1+x
gap> Coefficients(p,"x");
[ y^-1, 1 ]
gap> Coefficients(p,"y");
Error, y^-1+x is not a polynomial with respect to y
 in
```

```
V.operations.Coefficients( V, v ) called from
Coefficients( p, "y" ) called from
main loop
brk>
```

The same caveat is applicable to `Coefficients` as to `Value`: the result is always a list of `Mvp`s. To get a list of scalars for univariate polynomials represented as `Mvp`s, one should use `ScalMvp`.

Finally we mention the functions `ComplexConjugate` and `evalf` which are defined using for coefficients the `Complex` and `Decimal` numbers of the CHEVIE package.

```
gap> p:=E(3)*x+E(5);
E5+E3x
gap> evalf(p);
0.3090169944+0.9510565163I+(-0.5+0.8660254038I)x
gap> p:=E(3)*x+E(5);
E5+E3x
gap> ComplexConjugate(p);
E5^4+E3^2x
gap> evalf(p);
0.3090169944+0.9510565163I+(-0.5+0.8660254038I)x
gap> ComplexConjugate(last);
0.3090169944-0.9510565163I+(-0.5-0.8660254038I)x
```

112.3 IsMvp

`IsMvp(` *p* `)`

Returns `true` if p is an `Mvp` and false otherwise.

```
gap> IsMvp(1+Mvp("x"));
true
gap> IsMvp(1);
false
```

112.4 ScalMvp

`ScalMvp(` *p* `)`

If p is an `Mvp` then if p is a scalar, return that scalar, otherwise return `false`. Or if p is a list, then apply `ScalMvp` recursively to it (but return false if it contains any `Mvp` which is not a scalar). Else assume p is already a scalar and thus return p.

```
gap> v:=[Mvp("x"),Mvp("y")];
[ x, y ]
gap> ScalMvp(v);
false
gap> w:=List(v,p->Value(p,["x",2,"y",3]));
[ 2, 3 ]
gap> Gcd(w);
Error, sorry, the elements of <arg> lie in no common ring domain in
```

```
Domain( arg[1] ) called from
DefaultRing( ns ) called from
Gcd( w ) called from
main loop
brk>
gap> Gcd(ScalMvp(w));
1
```

112.5 Variables for Mvp

`Variables for Mvp(` *p* `)`

Returns the list of variables of the `Mvp` p as a sorted list of strings.

```
gap> Variables(x+x^4+y);
[ "x", "y" ]
```

112.6 LaurentDenominator

`LaurentDenominator(` *p1* `,` *p2* `, ... )`

Returns the unique monomial `m` of minimal degree such that for all the Laurent polynomial arguments *p1*, *p2*, etc... the product $m * p_i$ is a true polynomial.

```
gap> LaurentDenominator(x^-1,y^-2+x^4);
xy^2
```

112.7 OnPolynomials

`OnPolynomials(` m`,` p `[,`*vars*`] )`

Implements the action of a matrix on `Mvp`s. *vars* should be a list of strings representing variables. If v=`List(vars,Mvp)`, the polynomial p is changed by replacing in it v_i by $(v \times m)_i$. If *vars* is omitted, it is taken to be `Variables(p)`.

```
gap> OnPolynomials([[1,2],[3,1]],x+y);
3x+4y
```

112.8 FactorizeQuadraticForm

`FactorizeQuadraticForm(` p`)`

p should be an `Mvp` of degree 2 which represents a quadratic form. The function returns a list of two linear forms of which p is the product if such forms exist, otherwise it returns `false` (it returns [Mvp(1),p] if p is of degree 1).

```
gap> FactorizeQuadraticForm(x^2-y^2+x+3*y-2);
[ -1+x+y, 2+x-y ]
gap> FactorizeQuadraticForm(x^2+x+1);
[ -E3+x, -E3^2+x ]
gap> FactorizeQuadraticForm(x*y+1);
false
```

The next functions have been provided by Gwenaëlle Genet

112.9 MvpGcd

`MvpGcd(` *p1* `,` *p2* `, ...)`

Returns the Gcd of the `Mvp` arguments. The arguments must be true polynomials.

```
gap> MvpGcd(x^2-y^2,(x+y)^2);
x+y
```

112.10 MvpLcm

`MvpLcm(` *p1* `,` *p2* `, ...)`

Returns the Lcm of the `Mvp` arguments. The arguments must be true polynomials.

```
gap> MvpLcm(x^2-y^2,(x+y)^2);
xy^2-x^2y-x^3+y^3
```

112.11 RatFrac

`RatFrac(` *num* `[,`*den*`] )`

Build the rational fraction (`RatFrac`) with numerator *num* and denominator *den* (when *den* is omitted it is taken to be 1).

```
gap> RatFrac(x,y);
x/y
gap> RatFrac(x*y^-1);
x/y
```

112.12 Operations for RatFrac

The arithmetic operations `+`, `-`, `*`, `/` and `^` work for `RatFrac`s. They also have `Print` and `String` methods.

```
gap> 1/(x+1)+y^-1;
(1+x+y)/(y+xy)
gap> 1/(x+1)*y^-1;
1/(y+xy)
gap> 1/(x+1)/y;
1/(y+xy)
gap> 1/(x+1)^-2;
1+2x+x^2
```

Similarly to `Mvp`s, `RatFrac`s hav `Format` and `Value` methods

```
gap> Format(1/(x*y+1));
"1/(1+xy)"
gap> FormatGAP(1/(x*y+1));
"1/(1+x*y)"
gap> Value(1/(x*y+1),["x",2]);
1/(1+2y)
```

112.13 IsRatFrac

`IsRatFrac(` p `)`

Returns `true` if p is an `Mvp` and false otherwise.

```
gap> IsRatFrac(1+RatFrac(x));
true
gap> IsRatFrac(x);
false
```

Chapter 113

The VKCURVE functions

We document here the various functions which are used in Van Kampen's algorithm as described in the introduction.

113.1 Discy

`Discy(` *Mvp p* `)`

The input should be an `Mvp` in `x` and `y`, with rational coefficients. The function returns the discriminant of p with respect to `x` (an `Mvp` in `y`); it uses interpolation to reduce the problem to discriminants of univariate polynomials, and works reasonably fast (not hundreds of times slower than MAPLE...).

```
gap> Discy(x+y^2+x^3+y^3);
4+27y^4+54y^5+27y^6
```

113.2 ResultantMat

`ResultantMat(` *v*, *w* `)`

v and w are vectors representing coefficients of two polynomials. The function returns Sylvester matrix for these two polynomials (whose determinant is the resultant of the two polynomials). It is used internally by Discy.

```
gap>  p:=x+y^2+x^3+y^3;
x+y^2+x^3+y^3
gap>  c:=Coefficients(p,"x");
[ y^2+y^3, 1, 0, 1 ]
gap> PrintArray(ResultantMat(c,Derivative(c)));
[[      1,       0,       1, y^2+y^3,       0],
 [      0,       1,       0,       1, y^2+y^3],
 [      3,       0,       1,       0,       0],
 [      0,       3,       0,       1,       0],
 [      0,       0,       3,       0,       1]]
gap> DeterminantMat(ResultantMat(c,Derivative(c)));
4+27y^4+54y^5+27y^6
```

113.3 NewtonRoot

`NewtonRoot(`*p*`,`*initial*`,`*precision*`)`

Here *p* is a list of `Complex` rationals representing the coefficients of a polynomial. The function computes a complex rational approximation to a root of *p*, guaranteed of distance closer than *precision* (a rational) to an actual root. The first approximation used is *initial*. If *initial* is in the attraction basin of a root of *p*, the one approximated. A possibility is that the Newton method starting from *initial* does not converge (the number of iterations after which this is decided is controlled by `VKCURVE.NewtonLim`); then the function returns `false`. Otherwise the function returns a pair: the approximation found, and an upper bound of the distance between that approximation and an actual root. The upper bound returned is a power of 10, and the approximation denominator's is rounded to a power of 10, in order to return smaller-sized rational result as much as possible. The point of returning an upper bound is that it is usually better than the asked-for *precision*. For the precision estimate a good reference is [HSS01].

```
gap> p:=List([1,0,1],Complex); # p=x^2+1
[ 1, 0, 1 ]
gap> NewtonRoot(p,Complex(1,1),10^-7);
[ I, 1/1000000000 ]
# obtained precision is actually 10^-9
gap> NewtonRoot(p,Complex(1),10^-7);
false
# here Newton does not converge
```

113.4 SeparateRootsInitialGuess

`SeparateRootsInitialGuess(`*p*`, `*v*`, `*safety*`)`

Here *p* is a list of complex rationals representing the coefficients of a polynomial, and *v* is a list of approximations to roots of *p* which should lie in different attraction basins for Newton's method. The result is a list *l* of complex rationals representing approximations to the roots of *p* (each element of *l* is the root in whose attraction basin the corresponding element of *v* lies), such that if *d* is the minimum distance between two elements of *l*, then there is a root of *p* within radius $d/(2*safety)$ of any element of *l*. When the elements of *v* do not lie in different attraction basins (which is necessarily the case if *p* has multiple roots), `false` is returned.

```
gap> p:=List([1,0,1],Complex);
[ 1, 0, 1 ]
gap> SeparateRootsInitialGuess(p,[Complex(1,1),Complex(1,-1)],100);
[ I, -I ]
gap> SeparateRootsInitialGuess(p,[Complex(1,1),Complex(2,1)],100);
false # 1+I and 2+I not in different attraction basins
```

113.5 SeparateRoots

`SeparateRoots(`*p*`, `*safety*`)`

Here p is a univariate `Mvp` with rational or complex rational coefficients, or a vector of rationals or complex rationals describing the coefficients of such a polynomial. The result is a list l of complex rationals representing approximations to the roots of p, such that if d is the minimum distance between two elements of l, then there is a root of p within radius $d/(2*safety)$ of any element of l. This is not possible when p has multiple roots, in which case `false` is returned.

```
gap> SeparateRoots(x^2+1,100);
[ I, -I ]
gap> SeparateRoots((x-1)^2,100);
false
gap> SeparateRoots(x^3-1,100);
[ -1/2-108253175473/125000000000I, 1, -1/2+108253175473/125000000000I]
```

113.6 LoopsAroundPunctures

`LoopsAroundPunctures(`*points*`)`

The input is a list of complex rational numbers. The function computes piecewise-linear loops representing generators of the fundamental group of the complement of *points* in the complex line.

```
gap> LoopsAroundPunctures([Complex(0,0)]);
rec(
  points := [ -I, -1, 1, I ],
  segments := [ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  loops := [ [ 4, -3, -1, 2 ] ] )
```

The output is a record with three fields. The field `points` contains a list of complex rational numbers. The field `segments` contains a list of oriented segments, each of them encoded by the list of the positions in `points` of its two endpoints. The field `loops` contains a list of list of integers. Each list of integers represents a piecewise linear loop, obtained by concatenating the elements of `segments` indexed by the integers (a negative integer is used when the opposed orientation of the segment has to be taken).

113.7 FollowMonodromy

`FollowMonodromy(`*r*`,`*segno*`,`*print*`)`

This function computes the monodromy braid of the solution in x of an equation $P(x, y) = 0$ along a segment $[y_0, y_1]$. It is called by `FundamentalGroup`, once for each of the segments. The first argument is a global record, similar to the one produced by `FundamentalGroup` (see the documentation of this function) but only containing intermediate information. The second argument is the position of the segment in `r.segments`. The third argument is a print function, determined by the printlevel set by the user (typically, by calling `FundamentalGroup` with a second argument).

The function returns an element of the ambient braid group `r.B`.

This function has no reason to be called directly by the user, so we do not illustrate its behavior. Instead, we explain what is displayed on screen when the user sets the printlevel to 2.

What is quoted below is an excerpt of what is displayed on screen during the execution of
`gap>  FundamentalGroup((x+3*y)*(x+y-1)*(x-y),2);`

```
<1/16>    1 time=         0   ?2?1?3
<1/16>    2 time=     0.125   R2. ?3
<1/16>    3 time=   0.28125   R2. ?2
<1/16>    4 time=  0.453125   ?2R1?2
<1/16>    5 time=  0.578125   R1. ?2
======================================
=    Nontrivial braiding = 2         =
======================================
<1/16>    6 time=  0.734375   R1. ?1
<1/16>    7 time=   0.84375   . ?0.
<1/16>    8 time=  0.859375   ?1R0?1
#  The following braid was computed by FollowMonodromy in 8 steps.
monodromy[1]:=B(2);
#  segment 1/16 Time=0.1sec
```

`FollowMonodromy` computes its results by subdividing the segment into smaller subsegments on which the approximations are controlled. It starts at one end and moves subsegment after subsegment. A new line is displayed at each step.

The first column indicates which segment is studied. In the example above, the function is computing the monodromy along the first segment (out of 16). This gives a rough indication of the time left before completion of the total procedure. The second column is the number of iterations so far (number of subsegments). In our example, `FollowMonodromy` had to cut the segment into 8 subsegments. Each subsegment has its own length. The cumulative length at a given step, as a fraction of the total length of the segment, is displayed after `time=`. This gives a rough indication of the time left before completion of the computation of the monodromy of this segment. The segment is completed when this fraction reaches 1.

The last column has to do with the piecewise-linear approximation of the geometric monodromy braid. It is subdivided into sub-columns for each string. In the example above, there are three strings. At each step, some strings are fixed (they are indicated by `.` in the corresponding column). A symbol like `R5` or `?3` indicates that the string is moving. The exact meaning of the symbol has to do with the complexity of certain sub-computations.

As some strings are moving, it happens that their real projections cross. When such a crossing occurs, it is detected and the corresponding element of B_n is displayed on screen (`Nontrivial braiding =`...) The monodromy braid is the product of these elements of B_n, multiplied in the order in which they occur.

113.8 ApproxFollowMonodromy

`ApproxFollowMonodromy(`r`,`$segno$`,`pr`)`

This function computes an approximation of the monodromy braid of the solution in x of an equation $P(x, y) = 0$ along a segment $[y_0, y_1]$. It is called by `FundamentalGroup`, once for each of the segments. The first argument is a global record, similar to the one produced by `FundamentalGroup` (see the documentation of this function) but only containing intermediate information. The second argument is the position of the segment in `r.segments`. The third argument is a print function, determined by the printlevel set by the user (typically, by calling `FundamentalGroup` with a second argument).

Contrary to `FollowMonodromy`, `ApproxFollowMonodromy` does not control the approximations; it just uses a heuristic for how much to move along the segment between linear braid computations, and this heuristic may possibly fail. However, we have not yet found an example for which the result is actually incorrect, and thus the existence is justified by the fact that for some difficult computations, it is sometimes many times faster than `FollowMonodromy`. We illustrate its typical output when *printlevel* is 2.

```
VKCURVE.monodromyApprox:=true;
 FundamentalGroup((x+3*y)*(x+y-1)*(x-y),2);
```

....

```
5.3.6. ***rejected
4.3.6.<15/16>mindist=3 step=1/2 total=0 logdisc=1 ***rejected
3.3.4.<15/16>mindist=3 step=1/4 total=0 logdisc=1 ***rejected
3.3.4.<15/16>mindist=3 step=1/8 total=0 logdisc=1 ***rejected
3.3.3.<15/16>mindist=3 step=1/16 total=0 logdisc=1
3.2.3.<15/16>mindist=2.92 step=1/16 total=1/16 logdisc=1
3.3.3.<15/16>mindist=2.83 step=1/16 total=1/8 logdisc=1
3.2.3.<15/16>mindist=2.75 step=1/16 total=3/16 logdisc=1
3.3.3.<15/16>mindist=2.67 step=1/16 total=1/4 logdisc=1
======================================
=    Nontrivial braiding = 2         =
======================================
3.2.3.<15/16>mindist=2.63 step=1/16 total=5/16 logdisc=1
3.2.3.<15/16>mindist=2.75 step=1/16 total=3/8 logdisc=1
3.3.3.<15/16>mindist=2.88 step=1/16 total=7/16 logdisc=1
3.2.3.<15/16>mindist=3 step=1/16 total=1/2 logdisc=1
3.3.3.<15/16>mindist=3.13 step=1/16 total=9/16 logdisc=1
3.2.3.<15/16>mindist=3.25 step=1/16 total=5/8 logdisc=1
3.3.3.<15/16>mindist=3.38 step=1/16 total=11/16 logdisc=1
3.2.3.<15/16>mindist=3.5 step=1/16 total=3/4 logdisc=1
3.2.3.<15/16>mindist=3.63 step=1/16 total=13/16 logdisc=1
3.2.3.<15/16>mindist=3.75 step=1/16 total=7/8 logdisc=1
3.2.3.<15/16>mindist=3.88 step=1/16 total=15/16 logdisc=1 ***up
# Monodromy error=0
# Minimal distance=2.625
# Minimal step=1/16=-0.05208125+0.01041875I
# Adaptivity=10
monodromy[15]:=B(2);
# segment 15/16 Time=0.2sec
```

Here at each step the following information is displayed: first, how many iterations of the Newton method were necessary to compute each of the 3 roots of the current polynomial $f(x, y_0)$ if we are looking at the point y_0 of the segment. Then, which segment we are dealing with (here the 15th of 16 in all). Then the minimum distance between two roots of $f(x, y_0)$ (used in our heuristic). Then the current step in fractions of the length of the segment we are looking at, and the total fraction of the segment we have done. Finally, the decimal logarithm of the absolute value of the discriminant at the current point (used in the heuristic). Finally, an indication if the heuristic predicts that we should halve the step (`***rejected`) or that we may double it (`***up`).

The function returns an element of the ambient braid group `r.B`.

113.9 LBraidToWord

`LBraidToWord(`*v1*`,`*v2*`,`*B*`)`

This function converts the linear braid given by *v1* and *v2* into an element of the braid group B.

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> i:=Complex(0,1);
I
gap> LBraidToWord([1+i,2+i,3+i],[2+i,1+2*i,4-6*i],B);
1
```

The list *v1* and *v2* must have the same length, say n. The braid group B should be the braid group on n strings, in its CHEVIE implementation. The elements of *v1* (resp. *v2*) should be n distinct complex rational numbers. We use the Brieskorn basepoint, namely the contractible set $C + iV_{\mathbb{R}}$ where C is a real chamber; therefore the endpoints need not be equal (hence, if the path is indeed a loop, the final endpoint must be given). The linear braid considered is the one with affine strings connecting each point in *v1* to the corresponding point in *v2*. These strings should be non-crossing. When the numbers in *v1* (resp. *v2*) have distinct real parts, the real picture of the braid defines a unique element of B. When some real parts are equal, we apply a lexicographical desingularization, corresponding to a rotation of *v1* and *v2* by an arbitrary small positive angle.

113.10 BnActsOnFn

`BnActsOnFn(`*braid b*`,`*Free group F*`)`

This function implements the Hurwitz action of the braid group on n strings on the free group on n generators, where the standard generator σ_i of B_n fixes the generators $f_1, \ldots, f_n$, except f_i which is mapped to f_{i+1} and f_{i+1} which is mapped to $f_{i+1}^{-1} f_i f_{i+1}$.

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> b:=B(1);
1
gap> BnActsOnFn(b,FreeGroup(3));
GroupHomomorphismByImages( Group( f.1, f.2, f.3 ), Group( f.1, f.2, f.3 ),
[ f.1, f.2, f.3 ], [ f.2, f.2^-1*f.1*f.2, f.3 ] )
gap> BnActsOnFn(b^2,FreeGroup(3));
GroupHomomorphismByImages( Group( f.1, f.2, f.3 ), Group( f.1, f.2, f.3 ),
[ f.1, f.2, f.3 ], [ f.2^-1*f.1*f.2, f.2^-1*f.1^-1*f.2*f.1*f.2, f.3 ] )
```

The second input is the free group on n generators. The first input is an element of the braid group on n strings, in its CHEVIE implementation.

113.11 VKQuotient

`VKQuotient(`*braids*`,[`*bad*`])`

The input *braid* is a list of braids $b_1, \ldots, b_d$, living in the braid group on n strings. Each b_i defines by Hurwitz action an automorphism ϕ_i of the free group F_n. The function return the group defined by the abstract presentation:

$$< f_1, \ldots, f_n \mid \forall i, j, \phi_i(f_j) = f_j >$$

The optional second argument *bad* is another list of braids $c_1, \ldots, c_e$ (representing the monodromy around bad roots of the discriminant). For each c_k, we denote by ψ_k the corresponding Hurwitz automorphism of F_n. When a second argument is supplied, the function returns the group defined by the abstract presentation:

$$< f_1, \ldots, f_n, g_1, \ldots, g_k \mid \forall i, j, k, \phi_i(f_j) = f_j, \psi_k(f_j)g_k = g_k f_j >$$

```
gap> B:=Braid(CoxeterGroupSymmetricGroup(3));
function ( arg ) ... end
gap> b1:=B(1)^3; b2:=B(2);
1.1.1
2
gap> g:=VKQuotient([b1,b2]);
Group( f.1, f.2, f.3 )
gap>  last.relators;
[ f.2^-1*f.1^-1*f.2*f.1*f.2*f.1^-1, IdWord,
  f.2^-1*f.1^-1*f.2^-1*f.1*f.2*f.1, f.3*f.2^-1, IdWord, f.3^-1*f.2 ]
gap> p:=PresentationFpGroup(g);DisplayPresentation(p);
<< presentation with 3 gens and 4 rels of total length 16 >>
1: c=b
2: b=c
3: bab=aba
4: aba=bab
gap> SimplifyPresentation(p);DisplayPresentation(p);
# I there are 2 generators and 1 relator of total length 6
1: bab=aba
```

113.12 Display for presentations

`DisplayPresentation(`*p*`)`

Displays the presentation p in a compact form, using the letters `abc...` for the generators and `ABC...` for their inverses. In addition the program tries to show relations in "positive"form, i.e. as equalities between words involving no inverses.

```
gap> F:=FreeGroup(2);;
gap> p:=PresentationFpGroup(F/[F.2*F.1*F.2*F.1^-1*F.2^-1*F.1^-1]);
<< presentation with 2 gens and 1 rels of total length 6 >>
gap> DisplayPresentation(p);
1: bab=aba
```

```
gap> PrintRec(p);
rec(
  isTietze           := true,
  operations         := PresentationOps,
  generators         := [ f.1, f.2 ],
  tietze             := [ 2, 1, 6, [ f.1, f.2 ], [ 2, 1, 0, -1, -2 ],
  [ [ -2, -1, 2, 1, 2, -1 ] ], [ 6 ], [ 0 ], 0, false, 0, 0, 0, 0,
  [ 2, 1, 6 ], 0, 0, 0, 0, 0 ],
  components         := [ 1, 2 ],
  1                  := f.1,
  2                  := f.2,
  nextFree           := 3,
  identity           := IdWord,
  eliminationsLimit  := 100,
  expandLimit        := 150,
  generatorsLimit    := 0,
  lengthLimit        := infinity,
  loopLimit          := infinity,
  printLevel         := 1,
  saveLimit          := 10,
  searchSimultaneous := 20,
  protected          := 0 )
```

113.13 ShrinkPresentation

`ShrinkPresentation(`*p* `[,`*tries*`])`

This is our own program to simplify group presentations. We have found heuristics which make it somewhat more efficient than GAP3's programs `SimplifiedFpGroup` and `TzGoGo`, but the algorithm depends on random numbers so is not reproducible. The main idea is to rotate relators between calls to GAP3 functions. By default 1000 such rotations are tried (unless the presentation is so small that less rotations exhaust all possible ones), but the actual number tried can be controlled by giving a second parameter *tries* to the function. Another useful tool to deal with presentations is `TryConjugatePresentation` described in the utility functions.

```
gap> DisplayPresentation(p);
1: ab=ba
2: dbd=bdb
3: bcb=cbc
4: cac=aca
5: adca=cadc
6: dcdc=cdcd
7: adad=dada
8: Dbdcbd=cDbdcb
9: adcDad=dcDadc
10: dcdadc=adcdad
11: dcabdcbda=adbcbadcb
12: caCbdcbad=bdcbadBcb
```

```
13: cbDadcbad=bDadcbadc
14: cdAbCadBc=bdcAbCdBa
15: cdCbdcabdc=bdcbadcdaD
16: DDBcccbdcAb=cAbCdcBCddc
17: CdaBdbAdcbCad=abdcAbDadBCbb
18: bdbcabdcAADAdBDa=cbadcbDadcBDABDb
19: CbdbadcDbbdCbDDadcBCDAdBCDbdaDCDbdcbadcBCDAdBCDBBdacDbdccb
    =abdbcabdcAdcbCDDBCDABDABDbbdcbDadcbCDAdBCabDACbdBadcaDbAdd
gap> ShrinkPresentation(p);
# I there are 4 generators and 19 relators of total length 332
# I there are 4 generators and 17 relators of total length 300
# I there are 4 generators and 17 relators of total length 282
# I there are 4 generators and 17 relators of total length 278
# I there are 4 generators and 16 relators of total length 254
# I there are 4 generators and 15 relators of total length 250
# I there are 4 generators and 15 relators of total length 248
# I there are 4 generators and 15 relators of total length 246
# I there are 4 generators and 14 relators of total length 216
# I there are 4 generators and 13 relators of total length 210
# I there are 4 generators and 13 relators of total length 202
# I there are 4 generators and 13 relators of total length 194
# I there are 4 generators and 12 relators of total length 174
# I there are 4 generators and 12 relators of total length 170
# I there are 4 generators and 12 relators of total length 164
# I there are 4 generators and 12 relators of total length 162
# I there are 4 generators and 12 relators of total length 148
# I there are 4 generators and 12 relators of total length 134
# I there are 4 generators and 12 relators of total length 130
# I there are 4 generators and 12 relators of total length 126
# I there are 4 generators and 12 relators of total length 124
# I there are 4 generators and 12 relators of total length 118
# I there are 4 generators and 12 relators of total length 116
# I there are 4 generators and 11 relators of total length 100
gap> DisplayPresentation(p);
1: ba=ab
2: dbd=bdb
3: cac=aca
4: bcb=cbc
5: dAca=Acad
6: dcdc=cdcd
7: adad=dada
8: dcDbdc=bdcbdB
9: dcdadc=adcdad
10: adcDad=dcDadc
11: BcccbdcAb=dcbACdddc
```

Chapter 114

Some VKCURVE utility functions

We document here various utility functions defined by VKCURVE package and which may be useful also in other contexts.

114.1 BigNorm

`BigNorm(`c`)`

Given a `complex` number c with real part r and imaginary part j, returns a "cheap substitute" to the norm of c given by $\mathtt{r} + \mathtt{j}$.

```
gap> BigNorm(Complex(-1,-1));
2
```

114.2 DecimalLog

`DecimalLog(`r`)`

Given a rational number r, returns an integer k such that $10^k < \mathtt{r} \leq 10^{k+1}$.

```
gap> List([1,1/10,1/2,2,10],DecimalLog);
[ -1, -2, -1, 0, 1 ]
```

114.3 ComplexRational

`ComplexRational(`c`)`

c is a cyclotomic or a `Complex` number with `Decimal` or real cyclotomic real and imaginary parts. This function returns the corresponding rational complex number.

```
gap> evalf(E(3)/3);
-0.1666666667+0.2886751346I
gap> ComplexRational(last);
-16666666667/100000000000+28867513459/100000000000I
gap> ComplexRational(E(3)/3);
-1/6+28867513457/100000000000I
```

114.4 Dispersal

`Dispersal(v)`

v is a list of `complex` numbers representing points in the real plane. The result is a pair whose first element is the minimum distance between two elements of v, and the second is a pair of indices `[i,j]` such that `v[i]`, `v[j]` achieves this minimum distance.

```
gap> Dispersal([Complex(1,1),Complex(0),Complex(1)]);
[ 1, [ 1, 3 ] ]
```

114.5 ConjugatePresentation

`ConjugatePresentation(`*p* `,`*conjugation*`)`

This program modifies a presentation by conjugating a generator by another. The conjugation to apply is described by a length-3 string of the same style as the result of `DisplayPresentation`, that is `"abA"` means replace the second generator by its conjugate by the first, and `"Aba"` means replace it by its conjugate by the inverse of the first.

```
gap> F:=FreeGroup(4);;
gap> p:=PresentationFpGroup(F/[F.4*F.1*F.2*F.3*F.4*F.1^-1*F.4^-1*
> F.3^-1*F.2^-1*F.1^-1,F.4*F.1*F.2*F.3*F.4*F.2*F.1^-1*F.4^-1*F.3^-1*
> F.2^-1*F.1^-1*F.3^-1,F.2*F.3*F.4*F.1*F.2*F.3*F.4*F.3^-1*F.2^-1*
> F.4^-1*F.3^-1*F.2^-1*F.1^-1*F.4^-1]);
gap> DisplayPresentation(p);
1: dabcd=abcda
2: dabcdb=cabcda
3: bcdabcd=dabcdbc
gap> DisplayPresentation(ConjugatePresentation(p,"cdC"));
# I there are 4 generators and 3 relators of total length 36
1: cabdca=dcabdc
2: dcabdc=bdcabd
3: cabdca=abdcab
```

114.6 TryConjugatePresentation

`TryConjugatePresentation(`*p* `[,`*goal* `[,`*printlevel*`]])`

This program tries to simplify group presentations by applying conjugations to the generators. The algorithm depends on random numbers, and on tree-searching, so is not reproducible. By default the program stops as soon as a shorter presentation is found. Sometimes this does not give the desired presentation. One can give a second argument *goal*, then the program will only stop when a presentation of length less than *goal* is found. Finally, a third argument can be given and then all presentations the programs runs over which are of length less than or equal to this argument are displayed. Due to the non-deterministic nature of the program, it may be useful to run it several times on the same input. Upon failure (to improve the presentation), the program returns p.

```
gap> DisplayPresentation(p);
1: ba=ab
2: dbd=bdb
```

```
3: cac=aca
4: bcb=cbc
5: dAca=Acad
6: dcdc=cdcd
7: adad=dada
8: dcDbdc=bdcbdB
9: dcdadc=adcdad
10: adcDad=dcDadc
11: BcccbdcAb=dcbACdddc
gap> p:=TryConjugatePresentation(p);
#I there are 4 generators and 11 relators of total length 100
#I there are 4 generators and 11 relators of total length 120
#I there are 4 generators and 10 relators of total length 100
#I there are 4 generators and 11 relators of total length 132
#I there are 4 generators and 11 relators of total length 114
#I there are 4 generators and 11 relators of total length 110
#I there are 4 generators and 11 relators of total length 104
#I there are 4 generators and 11 relators of total length 114
#I there are 4 generators and 11 relators of total length 110
#I there are 4 generators and 11 relators of total length 104
#I there are 4 generators and 8 relators of total length 76
#I there are 4 generators and 8 relators of total length 74
#I there are 4 generators and 8 relators of total length 72
#I there are 4 generators and 8 relators of total length 70
#I there are 4 generators and 7 relators of total length 52
#  d->adA gives length 52
<< presentation with 4 gens and 7 rels of total length 52 >>
gap> DisplayPresentation(p);
1: ba=ab
2: dc=cd
3: aca=cac
4: dbd=bdb
5: bcb=cbc
6: adad=dada
7: aBcADbdac=dBCacbdaB
gap> TryConjugatePresentation(p,48);
#I there are 4 generators and 7 relators of total length 54
#I there are 4 generators and 7 relators of total length 54
#I there are 4 generators and 7 relators of total length 60
#I there are 4 generators and 7 relators of total length 60
#I there are 4 generators and 7 relators of total length 48
#  d->bdB gives length 48
<< presentation with 4 gens and 7 rels of total length 48 >>
gap> DisplayPresentation(last);
1: ba=ab
2: bcb=cbc
3: cac=aca
4: dbd=bdb
```

```
5: cdc=dcd
6: adad=dada
7: dAbcBa=bAcBad
```

114.7 FindRoots

`FindRoots(`*p*`,` *approx*`)`

p should be a univariate `Mvp` with cyclotomic or `Complex` rational or decimal coefficients or a list of cyclotomics or `Complex` rationals or decimals which represents the coefficients of a complex polynomial. The function returns `Complex` rational approximations to the roots of *p* which are better than *approx* (a positive rational). Contrary to the functions `SeparateRoots`, etc... described in the previous chapter, this function handles quite well polynomials with multiple roots. We rely on the algorithms explained in detail in [HSS01].

```
gap> FindRoots((x-1)^5,1/100000000000);
[ 6249999999993/6250000000000+29/12500000000000I,
  12499999999993/12500000000000-39/12500000000000I,
  12500000000023/12500000000000+11/6250000000000I,
  12500000000023/12500000000000+11/6250000000000I,
  312499999999/312500000000-3/6250000000000I ]
gap> evalf(last);
[ 1, 1, 1, 1, 1 ]
gap> FindRoots(x^3-1,1/10);
[ -1/2-108253175473/125000000000I, 1, -1/2+108253175473/125000000000I ]
gap> evalf(last);
[ -0.5-0.8660254038I, 1, -0.5+0.8660254038I ]
gap> List(last,x->x^3);
[ 1, 1, 1 ]
```

114.8 Cut

`Cut(`*string s* `[, opt])`

The first argument is a string, and the second one a record of options, if not given taken equal to `rec()`. This function prints its string argument *s* on several lines not exceeding *opt.width*; if not given *opt.width* is taken to be equal `SizeScreen[1]-2`. This is similar to how GAP3 prints strings, excepted no continuation line characters are printed. The user can specify after which characters, or before which characters to cut the string by giving fields `opt.before` and `opt.after`; the defaut is `opt.after:=","`, but some other characters can be used — for instance a good choice for printing big polynomials could be `opt.before:="+-"`. If a field `opt.file` is given, the result is appended to that file instead of written to standard output; this may be quite useful in conjunction with `FormatGAP` for dumping some GAP3 values to a file for later re-reading.

```
gap> Cut("an, example, with, plenty, of, commas\n",rec(width:=10));
an,
example,
with,
plenty,
```

```
of,
commas
gap>
```

Chapter 115

Algebra package — finite dimensional algebras

This package has been developped by Cédric Bonnafé to work with finite dimensional algebras under GAP3; it depends on the package "chevie".

Note that these programs have been mainly developed for working with Solomon descent algebras.

We start with a list of utility functions which are used in various places.

115.1 Digits

`Digits(n [, basis])`

returns the list of digits of the nonnegative integer n in basis *basis* (in basis 10 if no second argument is given).

```
gap> Digits(0); Digits(3); Digits(123); Digits(123,16);
[  ]
[ 3 ]
[ 1, 2, 3 ]
[ 7, 11 ]
```

115.2 ByDigits

`ByDigits(l [, basis])`

Does the converse of `Digits`, that is, computes an integer give the sequence of its digits (by default in basis 10; in basis *basis* if a second argument is given).

```
gap> ByDigits([2,3,4,5]);
2345
gap> ByDigits([2,3,4,5],100);
2030405
```

115.3 SignedCompositions

`SignedCompositions(n)`

computes the set of signed compositions of n that is, the set of tuples of non-zero integers $[i_1, ...i_r]$ such that $|i_1| + ... + |i_r| = n$. Note that `Length(SignedCompositions(n)) = 2*3^(n-1)`.

```
gap> SignedCompositions(3);
[ [ -3 ], [ -2, -1 ], [ -2, 1 ], [ -1, -2 ], [ -1, -1, -1 ],
  [ -1, -1, 1 ], [ -1, 1, -1 ], [ -1, 1, 1 ], [ -1, 2 ], [ 1, -2 ],
  [ 1, -1, -1 ], [ 1, -1, 1 ], [ 1, 1, -1 ], [ 1, 1, 1 ], [ 1, 2 ],
  [ 2, -1 ], [ 2, 1 ], [ 3 ] ]
```

Note that the compositions of n are obtained by the function `OrderedPartitions` in GAP3.

115.4 SignedPartitions

`SignedPartitions(`n`)`

computes the set of signed partitions of n that is, the set of tuples of integers $[i_1, \ldots, i_r, j_1, \ldots, j_s]$ such that $i_k 0$, $j_k < 0$, $|i_1|+\ldots+|i_r|+|j_1|+\ldots+|j_s| = n$, $i_1 \geq \ldots \geq i_r$ and $|j_1| \geq \ldots \geq |j_s|$.

```
gap> SignedPartitions(3);
[ [ -3 ], [ -2, -1 ], [ -1, -1, -1 ], [ 1, -2 ], [ 1, -1, -1 ],
  [ 1, 1, -1 ], [ 1, 1, 1 ], [ 2, -1 ], [ 2, 1 ], [ 3 ] ]
```

115.5 PiPart

`PiPart(`n`,`pi`)`

Let n be an integer and π a set of prime numbers. Write $n = n_1 n_2$ where no prime factor of n_2 is in π and all prime factors of n_1 are in π. Then n_1 is called the π-part of n and n_2 the π'-part of n. This function returns the π-part of n. The set π may be given as a list of primes, or as an integer in which case the set π is taken to be the list of prime factors of that integer.

```
gap> PiPart(720,2);
16
gap> PiPart(720,3);
9
gap> PiPart(720,6);
144
gap> PiPart(720,[2,3]);
144
```

115.6 CyclotomicModP

`CyclotomicModP(`z`,`p`)`

p should be a prime and z a cyclotomic number which is p-integral (that is, z times some number prime to p is a cyclotomic integer). The function returns the reduction of z mod. p, an element of some extension $\mathcal{F}_{p^r}$ of the prime field $\mathcal{F}_p$.

```
gap> CyclotomicModP(E(7),3);
Z(3^6)^104
```

115.7 PiComponent

`PiComponent(`G`,`g`,`pi`)`

Let g be an element of the finite group G and π a set of prime numbers. Write $g = g_1 g_2$ where g_1 and g_2 are both powers of g, no prime factor of the order of g_2 is in π and all prime factors of the order of g_1 are in π. Then g_1 is called the π-component of g and g_2 the π'-component of n. This function returns the π-component of g. The set π may be given as a list of primes, or as an integer in which case the set π is taken to be the list of prime factors of that integer.

115.8 PiSections

`PiSections(`G`,`pi`)`

Let π be a set of prime numbers. Two conjugacy classes of the finite group G are said to belong to the same π-section if the π-components (see 115.7) of elements of the two classes are conjugate. This function returns the partition of the set of conjugacy classes of G in π-sections, represented by the list of indices of conjugacy classes of G in each part. The set π may be given as a list of primes, or as an integer in which case the set π is taken to be the list of prime factors of that integer.

```
gap> W:=SymmetricGroup(5);
Group( (1,5), (2,5), (3,5), (4,5) )
gap> PiSections(W,2);
[ [ 1, 4, 7 ], [ 2, 5 ], [ 3 ], [ 6 ] ]
gap> PiSections(W,3);
[ [ 1, 2, 3, 6, 7 ], [ 4, 5 ] ]
gap> PiSections(W,6);
[ [ 1, 7 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ] ]
```

115.9 PiPrimeSections

`PiPrimeSections(`G`,`pi`)`

Let π be a set of prime numbers. Two conjugacy classes of the finite group G are said to belong to the same π'-section if the π'-components (see 115.7) of elements of the two classes are conjugate. This function returns the partition of the set of conjugacy classes of G in π'-sections, represented by the list of indices of conjugacy classes of G in each part. The set π may be given as a list of primes, or as an integer in which case the set π is taken to be the list of prime factors of that integer.

```
gap> W:=SymmetricGroup(5);
Group( (1,5), (2,5), (3,5), (4,5) )
gap> PiPrimeSections(W,2);
[ [ 1, 2, 3, 6 ], [ 4, 5 ], [ 7 ] ]
gap> PiPrimeSections(W,3);
[ [ 1, 4 ], [ 2, 5 ], [ 3 ], [ 6 ], [ 7 ] ]
gap> PiPrimeSections(W,6);
[ [ 1, 2, 3, 4, 5, 6 ], [ 7 ] ]
```

115.10 PRank

`PRank(`G`,`p`)`

Let p be a prime. This function returns the p-rank of the finite group G, defined as the maximal rank of an elementary abelian p-subgroup of G.

```
gap> W:=SymmetricGroup(5);
Group( (1,5), (2,5), (3,5), (4,5) )
gap> PRank(W,2);
2
gap> PRank(W,3);
1
gap> PRank(W,7);
0
```

115.11 PBlocks

`PBlocks(`G`,`p`)`

Let p be a prime. This function returns the partition of the irreducible characters of G in p-blocks, represented by the list of indices of irreducibles characters in each part.

```
gap> W:=SymmetricGroup(5);
Group( (1,5), (2,5), (3,5), (4,5) )
gap> PBlocks(W,2);
[ [ 1, 2, 5, 6, 7 ], [ 3, 4 ] ]
gap> PBlocks(W,3);
[ [ 1, 3, 6 ], [ 2, 4, 5 ], [ 7 ] ]
gap> PBlocks(W,7);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7 ] ]
```

115.12 Finite-dimensional algebras over fields

Let K be a field and let A be a K-algebra of finite dimension d. In our implementation, A must be endowed with a basis $X = (x_i)_{i\in I}$, where $I = \{i_1, ..., i_d\}$. Then A is represented by a record containing the following fields

`A.field` the field K.

`A.dimension` the dimension of A.

`A.multiplication` this is a function which associates to (k, l) the coefficients of the product $x_{i_k} x_{i_l}$ in the basis X (here, $1 \leq k, l \leq d$). If the structure constants of A are known, then it is possible to record them in `A.structureconstants`:the entry `A.structureconstants[k][l]` is equal to `A.multiplication(k,l)`. Once the function `A.multiplication` is defined, we can obtain the field `A.structureconstants` just by asking for `FDAlgebraOps.structureconstants(A)`.

`A.zero` the zero element of A.

`A.one` the unity of A.

`A.basisname` a "name"for the basis X (for instance, `A.basisname:="X"`).

`A.parameters` the parameter set I.

`A.identification` something characterizing A (this is used to test if two algebras are equal). For instance, if $A = K[G]$ is the group algebra of G, we take `A.identification =["Group algebra",G,K];`.

For convenience, the record A is often endowed with the following fields:

`A.generators` a list of generators of A.

`A.basis` the list of elements of X.

`A.vectorspace` the underlying vector space represented in GAP3 as K^d.

`A.EltToVector` the function sending an element of A to its image in `A.vectorspace` (i.e. a d-tuple of elements of K).

`A.VectorToElt` inverse function of `A.EltToVector`.

`A.type` for instance `"Group algebra"`, or `"Grothendieck ring"`...

`A.operations` This is initialized to `FDAlgebraOps` which contains quite a few operations applicable to finite-dimensional algebras, like the following:

`FDAlgebraOps.underlyingspace` once `A.dimension` is defined, this function constructs the underlying space of A. It endows the record A with the fields `A.basis`, `A.vectorspace`, `A.EltToVector`, and `A.VectorToElt`.

`FDAlgebraOps.structureconstants` computes the structure constants of A and gathers them in `A.structureconstants`.

115.13 Elements of finite dimensional algebras

An element x of A is implemented as a record containing three fields

`x.algebra` the algebra A

`x.coefficients` the list of pairs (a_k, k) such that a_k is a non-zero element of K and $x = \sum_{k=1}^{d} a_k x_{i_k}$.

`x.operations` the operations record `AlgebraEltOps` defining the operations for finite dimensional algebra elements.

115.14 Operations for elements of finite dimensional algebras

The following operations are define for elements of a finite dimensional algebra A.

`Print` this function gives a way of printing elements of A. If `A.print` is defined, it is used. Otherwise, the element x_i is printed using `A.basisname` and `A.parameters`:for instance, if `A.basisname:="BASISNAME"` and `A.parameters:=[1..d]`, then x_i is printed as `BASISNAME(i)`.

`\+` addition of elements of A.

`\-` subtraction of elements of A.

`\*` multiplication of elements of A.

`\^` powers of elements of A (negative powers are allowed for invertible elements).

`Coefficients(x)` the list of coefficients of x in `Basis(A)`.

115.15 IsAlgebraElement for finite dimensional algebras

`IsAlgebraElement(x)`

This function retuirns `true` if x is an element of a finite dimensional algebra, `false` if it is another kind of object.

```
gap> q:=X(Rationals);; q.name:="q";;
gap> A:=PolynomialQuotientAlgebra(q^2-q-1);;
gap> IsAlgebraElement(Basis(A)[1]);
true
gap> IsAlgebraElement(1);
false
```

115.16 IsAbelian for finite dimensional algebras

`IsAbelian(A)`

returns `true` if the algebra A is commutative and `false` otherwise.

```
gap> q:=X(Rationals);; q.name:="q";;
gap> A:=PolynomialQuotientAlgebra(q^2-q-1);;
gap> IsAbelian(A);
true
gap> B:=SolomonAlgebra(CoxeterGroup("A",2));;
gap> IsAbelian(B);
false
```

115.17 IsAssociative for finite dimensional algebras

`IsAssociative(A)`

returns `true` if the algebra A is associative and `false` otherwise.

```
gap> q:=X(Rationals);; q.name:="q";;
gap> A:=PolynomialQuotientAlgebra(q^2-q-1);;
gap> IsAssociative(A);
true
```

115.18 AlgebraHomomorphismByLinearity

`AlgebraHomomorphismByLinearity(A,B[,l])`

returns the linear map from A to B that sends `A.basis` to the list l (if omitted to `B.basis`). If this is not an homomorphism of algebras, the function returns an error.

```
gap> q:=X(Rationals);; q.name:="q";;
gap> A:=PolynomialQuotientAlgebra(q^4);;
gap> hom:=AlgebraHomomorphismByLinearity(A,Rationals,[1,0,0,0]);
function ( element ) ... end
gap> hom(A.class(q^4+q^3+1));
1
gap> hom2:=AlgebraHomomorphismByLinearity(A,Rationals,[1,1,1,1]);
```

```
Error, This is not a morphism of algebras in
AlgebraHomomorphismByLinearity( A, Rationals, [ 1, 1, 1, 1 ] ) called from
main loop
```

115.19 SubAlgebra for finite-dimensional algebras

`SubAlgebra(A,l)`

returns the sub-algebra B of A generated by the list l. The elements of B are written as elements of A.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> B:=SubAlgebra(A,[A.xbasis(23),A.xbasis(34)]);
SubAlgebra(SolomonAlgebra(CoxeterGroup("B",4),Rationals),
[ X(23), X(34) ])
gap> Dimension(B);
6
gap> IsAbelian(B);
false
gap> B.basis;
[ X(1234), X(23), X(34), X(2)-X(4), X(3)+X(4), X(0) ]
```

115.20 CentralizerAlgebra

`CentralizerAlgebra(A,l)`

returns the sub-algebra B of A of elements commuting with all the elements in the list l. The elements of B are written as elements of A.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> B:=CentralizerAlgebra(A,[A.xbasis(23),A.xbasis(34)]);
Centralizer(SolomonAlgebra(CoxeterGroup("B",4),Rationals),
[ X(23), X(34) ])
gap> Dimension(B);
10
gap> IsAbelian(B);
false
```

115.21 Center for algebras

`Centre(A)`

returns the center B of the algebra A. The elements of B are written as elements of A.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> B:=Centre(A);
Centre(SolomonAlgebra(CoxeterGroup("B",4),Rationals))
gap> Dimension(B);
8
gap> IsAbelian(B);
true
```

115.22 Ideals

If l is an element, or a list of elements of the algebra A, then `LeftIdeal(A,l)` (resp. `RightIdeal(A,l)`, resp. `TwoSidedIdeal(A,l)`) returns the left (resp. right, resp. two-sided) ideal of A generated by l. The result is a record containing the following fields:

`.parent` the algebra A

`.generators` the list l

`.basis` a K-basis of the ideal

`.dimension` the dimension of the ideal

`LeftTraces(A,I)`, `RightTraces(A,I)` the character afforded by the left (or right) ideal `I` (written as a list of traces of elements of the `A.basis`).

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> I:=LeftIdeal(A,[A.xbasis(234)]);
LeftIdeal(SolomonAlgebra(CoxeterGroup("B",4),Rationals),[ X(234) ])
gap> I.basis;
[ X(234), X(23)+X(34), X(24), X(2)+X(4), X(3), X(0) ]
gap> Dimension(I);
6
gap> LeftTraces(A,I);
[ 6, 18, 40, 50, 42, 64, 112, 112, 100, 136, 100, 192, 224, 224, 224,
384 ]
```

115.23 QuotientAlgebra

`QuotientAlgebra(A,I)`

A is a finite dimensional algebra, and I a two-sided ideal of A. The function returns the algebra A/I. It is also allowed than I be an element of A or a list of elements of A, in which case it is understood as the two-sided ideal generated by I.

115.24 Radical for algebras

`Radical(A)`

If the record A is endowed with the field `A.radical` (containing the radical of A) or with the field `A.Radical` (a function for computing the radical of A), then `Radical(A)` returns the radical of A (as a two-sided ideal of A). At this time, this function is available only in characteristic zero:it works for group algebras, Grothendiek rings, Solomon algebras and generalized Solomon algebras.

115.25 RadicalPower

`RadicalPower(A,n)`

returns (when possible) the n-th power of the two-sided ideal `Radical(A)`.

115.26 LoewyLength

`LoewyLength(A)`

returns (when possible) the Loewy length of A that is, the smallest natural number $n >= 1$ such that the n-th power of the two-sided ideal `Radical(A)` vanishes.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> R:=Radical(A);
TwoSidedIdeal(SolomonAlgebra(CoxeterGroup("B",4),Rationals),
[ X(13)-X(14), X(23)-X(34), X(2)-X(3), X(2)-X(4) ])
gap> Dimension(R);
4
gap> LoewyLength(A);
2
```

115.27 CharTable for algebras

`CharTable(A)`

For certain algebras, the function `CharTable` may be applied. It returns the character table of the algebra $\overline{K} \otimes_K A$:different ways of printing are used according to the type of the algebra. If A is a group algebra in characteristic zero, then `CharTable(A)` returns the character table of `A.group`. This function is available whenever K is of characteristic zero for group algebras, Grothendieck rings, Solomon algebras and generalized Solomon algebras.

```
gap> A:=GrothendieckRing(SymmetricGroup(4));
GrothendieckRing(Group( (1,4), (2,4), (3,4) ),Rationals)
gap> CharTable(A);

     X.1 X.2 X.3 X.4 X.5

MU.1   1   1   2   3   3
MU.2   1  -1   .  -1   1
MU.3   1   1   2  -1  -1
MU.4   1   1  -1   .   .
MU.5   1  -1   .   1  -1

gap> B:=SolomonAlgebra(CoxeterGroup("B",2));
SolomonAlgebra(CoxeterGroup("B",2),Rationals)
gap> CharTable(B);

     1
     2  1  2  0

12   1  .  .  .
 1   1  2  .  .
 2   1  .  2  .
 0   1  4  4  8
```

115.28 CharacterDecomposition

`CharacterDecomposition(A,char)`

Given a list *char* of elements of K (indexed by `A.basis`), then `CharacterDecomposition(A,char)` returns the decomposition of *char* into a sum of irreducible characters of A, if possible.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",3));
SolomonAlgebra(CoxeterGroup("B",3),Rationals)
gap> I:=LeftIdeal(A,[A.xbasis(13)]);
LeftIdeal(SolomonAlgebra(CoxeterGroup("B",3),Rationals),[ X(13) ])
gap> I.basis;
[ X(13), X(1), X(3), X(0) ]
gap> LeftTraces(A,I);
[ 4, 12, 20, 12, 32, 28, 28, 48 ]
gap> CharTable(A);

          1
          2  1   1  2
          3  2   3  3   1   2   0

    123   1  .   .  .   .   .   .
     12   1  2   .  .   .   .   .
     13   1  .   2  .   .   .   .
     23   1  .   .  2   .   .   .
      1   1  4   4  .   8   .   .
      2   1  2   2  4   .   4   .
      0   1  6  12  8  24  24  48

gap> CharacterDecomposition(A,LeftTraces(A,I));
[ 0, 0, 1, 0, 1, 1, 1 ]
```

115.29 Idempotents for finite dimensional algebras

`Idempotents(A)`

returns a complete set of orthogonal primitive idempotents of A. This is defined currently for Solomon algebras, quotient by polynomial algebras, group algebras and Grothendieck rings.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",2));
SolomonAlgebra(CoxeterGroup("B",2),Rationals)
gap> e:=Idempotents(A);
[ X(12)-1/2*X(1)-1/2*X(2)+3/8*X(0), 1/2*X(1)-1/4*X(0),
  1/2*X(2)-1/4*X(0), 1/8*X(0) ]
gap> Sum(e)=A.one;
true
gap> List(e, i-> i^2-i);
[ 0*X(12), 0*X(12), 0*X(12), 0*X(12) ]
gap> l:=[[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4]];;
```

```
gap> Set(List(l, i-> e[i[1]]*e[i[2]]));
[ 0*X(12) ]
```

115.30 LeftIndecomposableProjectives

`LeftIndecomposableProjectives`

returns the list of left ideals Ae, where e runs over the list `Idempotents(A)`.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",3));
SolomonAlgebra(CoxeterGroup("B",3),Rationals)
gap> proj:=LeftIndecomposableProjectives(A);;
gap> List(proj,Dimension);
[ 2, 1, 1, 1, 1, 1, 1 ]
```

115.31 CartanMatrix

`CartanMatrix(A)`

returns the Cartan matrix of A that is, the matrix $\dim Hom(P,Q)$, where P and Q run over the list `LeftIndecomposableProjectives(A)`.

```
gap> A:=SolomonAlgebra(CoxeterGroup("B",4));
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> CartanMatrix(A);

        1
        2        1  2     1  1
        3  1  2  2  3     2  3  1  2
        4  3  3  3  4  2  4  4  2  4  1  0

1234    1  .  .  .  .  .  .  .  .  .  .  .
  13    1  1  .  .  .  .  .  .  .  .  .  .
  23    1  .  1  .  .  .  .  .  .  .  .  .
 123    .  .  .  1  .  .  .  .  .  .  .  .
 234    .  .  .  .  1  .  .  .  .  .  .  .
   2    .  .  .  1  1  1  .  .  .  .  .  .
 124    .  .  .  .  .  .  1  .  .  .  .  .
 134    .  .  .  .  .  .  .  1  .  .  .  .
  12    .  .  .  .  .  .  .  .  1  .  .  .
  24    .  .  .  .  .  .  .  .  .  1  .  .
   1    .  .  .  .  .  .  .  .  .  .  1  .
   0    .  .  .  .  .  .  .  .  .  .  .  1
```

115.32 PolynomialQuotientAlgebra

An example - quotient by polynomial algebras

`PolynomialQuotientAlgebra(P)`

Given a polynomial P with coefficients in K, `A =PolynomialQuotientAlgebra(P)` returns the algebra $A = K[X]/(P(X))$. Note that the class of a polynomial Q is printed as

`Class(Q)` and that A is endowed with the field `A.class`:this function sends a polynomial Q to its image in A.

```
gap> q:=X(Rationals);; q.name:="q";;
gap> P:=1+2*q+q^3;;
gap> A:=PolynomialQuotientAlgebra(P);
Rationals[q]/(q^3 + 2*q + 1)
gap> x:=A.basis[3];
Class(q^2)
gap> x^2;
Class(-2*q^2 - q)
gap> 3*x - A.one;
Class(3*q^2 - 1)
gap> A.class(q^6);
Class(4*q^2 + 4*q + 1)
```

Group algebras

115.33 GroupAlgebra

`GroupAlgebra(G,K)`

returns the group algebra $K[G]$ of the finite group G over K. If K is not given, then the program takes for K the field of rational numbers. The i-th element in the list of elements of G is printed by default as `e(i)`. This function endows G with *G.law* containing the multiplication table of G.

115.34 Augmentation

`Augmentation(x)`

returns the image of the element x of $K[G]$ under the augmentation morphism.

```
gap> G:=SL(3,2);;
gap> A:=GroupAlgebra(G);
GroupAlgebra(SL(3,2),Rationals)
gap> A.dimension;
168
gap> A.basis[5]*A.basis[123];
e(87)
gap> (A.basis[3]-A.basis[12])^2;
e(55) - e(59) - e(148) + e(158)
gap> Augmentation(last);
0
```

Grothendieck Rings

115.35 GrothendieckRing

`GrothendieckRing(G,K)`

returns the Grothendieck ring $K \otimes ZIrrG$. The i-th irreducible ordinary character is printed as `X(i)`. This function endows G with `G.tensorproducts` containing the table of tensor products of irreducible ordinary characters of G.

115.36 Degree for elements of Grothendieck rings

`Degree(x)`

returns the image of the element x of `GrothendieckRing(G,K)` under the morphism of algebras sending a character to its degree (viewed as an element of K).

```
gap> G:=SymmetricGroup(4);
Group( (1,4), (2,4), (3,4) )
gap> Display(CharTable(G));

      2  3  2  3  .  2
      3  1  .  .  1  .

        1a 2a 2b 3a 4a
     2P 1a 1a 1a 3a 2b
     3P 1a 2a 2b 1a 4a

X.1      1  1  1  1  1
X.2      1 -1  1  1 -1
X.3      2  .  2 -1  .
X.4      3 -1 -1  .  1
X.5      3  1 -1  . -1

gap> A:=GrothendieckRing(G);
GrothendieckRing(Group( (1,4), (2,4), (3,4) ),Rationals)
gap> A.basis[4]*A.basis[5];
X(2) + X(3) + X(4) + X(5)
gap> Degree(last);
9
```

115.37 Solomon algebras

Let (W,S) be a finite Coxeter group. If w is an element of W, let $R(w) = \{s \in S \mid l(ws)l(w)\}$. If I is a subset of S, we set $Y_I = \{w \in W \mid R(w) = I\}$, $X_I = \{w \in W \mid R(w) \supset I\}$.

Note that X_I is the set of minimal length left coset representatives of W/W_I. Now, let $y_I = \sum_{w \in Y_I} w$, $x_I = \sum_{w \in X_I} w$.

They are elements of the group algebra ZW of W over Z. Now, let

$$\Sigma(W) = \oplus_{I \subset S} \mathbb{Z} y_I = \oplus_{I \subset S} \mathbb{Z} x_I.$$

This is a sub-Z-module of ZW. In fact, Solomon proved that it is a sub-algebra of ZW. Now, let $K(W)$ be the Grothendieck ring of W and let $\theta : \Sigma(W) \to K(W)$ be the map defined by $\theta(x_I) = Ind_{W_I}^W 1$. Solomon proved that this is an homomorphism of algebras. We call it the **Solomon homomorphism**.

115.38 SolomonAlgebra

`SolomonAlgebra(W,K)`

returns the Solomon descent algebra of the finite Coxeter group (W, S) over K. If $S = [s_1, ..., s_r]$, the element x_I corresponding to the subset $I = [s_1, s_2, s_4]$ of S is printed as `X(124)`. Note that `A:=SolomonAlgebra(W,K)` is endowed with the following fields:

`A.group` the group W

`A.basis` the basis $(x_I)_{I \subset S}$.

`A.xbasis` the function sending the subset I (written as a number:for instance 124 for $[s_1, s_2, s_4]$) to x_I.

`A.ybasis` the function sending the subset I to y_I.

`A.injection` the injection of A in the group algebra of W, obtained by calling `SolomonAlgebraOps.injection(A)`.

Note that `SolomonAlgebra(W,K)` endows W with the field *W.solomon* which is a record containing the following fields:

`W.solomon.subsets` the set of subsets of S

`W.solomon.conjugacy` conjugacy classes of parabolic subgroups of W (a conjugacy class is represented by the list of the positions, in `W.solomon.subsets`, of the subsets I of S such that W_I lies in this conjugacy class).

`W.solomon.mackey` essentially the structure constants of the Solomon algebra over the rationals.

```
gap> W:=CoxeterGroup("B",4);
CoxeterGroup("B",4)
gap> A:=SolomonAlgebra(W);
SolomonAlgebra(CoxeterGroup("B",4),Rationals)
gap> X:=A.xbasis;;
gap> X(123)*X(24);
2*X(2) + 2*X(4)
gap> SolomonAlgebraOps.injection(A)(X(123));
e(1) + e(2) + e(3) + e(8) + e(19) + e(45) + e(161) + e(361)
gap> W.solomon.subsets;
[ [ 1, 2, 3, 4 ], [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ],
  [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ], [ 1 ], [ 2
],
  [ 3 ], [ 4 ], [  ] ]
gap> W.solomon.conjugacy;
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ], [ 7, 8 ], [ 9, 11 ], [ 10 ],
  [ 12 ], [ 13, 14, 15 ], [ 16 ] ]
```

115.39 Generalized Solomon algebras

In this subsection, we refer to the paper [BH05].

If n is a non-zero natural number, we denote by W_n the Weyl group of type B_n and by W_{-n} the Weyl group of type A_{n-1} (isomorphic to the symmetric group of degree n). If $C=[i_1,...,i_r]$ is a **signed composition** of n, we denote by W_C the subgroup of W_n equal to $W_C=W_{i_1}x...xW_{i_r}$. This is a subgroup generated by reflections (it is not in general a parabolic subgroup of W_n). Let $X_C=\{x\in W_C \mid l(xw)\geq l(x)\forall w\in W_C\}$. Note that X_C is the set of minimal length left coset representatives of W_n/W_C. Now, let $x_C=\sum_{w\in X_C}w$. We now define $\Sigma'(W_n)=\oplus_C\mathbb{Z}x_C$, where C runs over the signed compositions of n. By [BH05], this is a subalgebra of ZW_n. Now, let Y_C be the set of elements of X_C which are not in any other X_D and let $y_C=\sum_{w\in Y_C}w$. Then $\Sigma'(W_n)=\oplus_C\mathbb{Z}y_C$. Moreover, the linear map $\theta':\Sigma'(W_n)\to K(W_n)$ defined by $\theta'(x_C)=Ind_{W_C}^{W_n}1$ is a **surjective homomorphism** of algebras (see [BH05]). We still call it the **Solomon homomorphism**.

115.40 GeneralizedSolomonAlgebra

`GeneralizedSolomonAlgebra(n,K)`

returns the generalized Solomon algebra $\Sigma'(W_n)$ defined above. If C is a signed composition of n, the element x_C is printed as `X(C)` Note that `A:=GeneralizedSolomonAlgebra(n,K)` is endowed with the following fields:

`A.group` the group `CoxeterGroup("B",n)`

`A.xbasis` the function sending the signed composition C to x_C.

`A.ybasis` the function sending the signed composition C to y_C.

`A.injection` the injection of A in the group algebra of W.

Note that `GeneralizedSolomonAlgebra(W,K)` endows W with the field `W.generalizedsolomon` which is a record containing the following fields:

`W.generalizedsolomon.signedcompositions` the set of signed compositions of n

`W.generalizedsolomon.conjugacy` conjugacy classes of reflection subgroups W_C of W (presented as sublists of `[1..2*3^(n-1)]` as in the classical Solomon algebra case).

`W.generalizedsolomon.mackey` essentially the structure constants of the generalized Solomon algebra over the rationals.

```
    gap> A:=GeneralizedSolomonAlgebra(3);
    GeneralizedSolomonAlgebra(CoxeterGroup("B",3),Rationals)
    gap> W:=A.group;
    CoxeterGroup("B",3)
    gap> W.generalizedsolomon.signedcompositions;
    [ [ 3 ], [ -3 ], [ 1, 2 ], [ 2, 1 ], [ 2, -1 ], [ -1, 2 ], [ 1, -2 ],
      [ -2, 1 ], [ -1, -2 ], [ -2, -1 ], [ 1, 1, 1 ], [ 1, -1, 1 ], [ 1, 1, -1
],
      [ -1, 1, 1 ], [ 1, -1, -1 ], [ -1, 1, -1 ], [ -1, -1, 1 ], [ -1, -1, -1
] ]
    gap> W.generalizedsolomon.conjugacy;
    [ [ 1 ], [ 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ], [ 9, 10 ], [ 11 ],
```

```
  [ 12, 13, 14 ], [ 15, 16, 17 ], [ 18 ] ]
gap> X:=A.xbasis;
function ( arg ) ... end
gap> X(2,1)*X(1,-2);
X(1,-2)+X(1,-1,1)+X(1,1,-1)+X(1,-1,-1)
```

115.41 SolomonHomomorphism

`SolomonHomomorphism(x)`

returns the image of the element x of `A=SolomonAlgebra(W,K)` or `A=GeneralizedSolomonAlgebra(n,K)` in `GrothendieckRing(W,K)` under Solomon homomorphism.

```
gap> A:=GeneralizedSolomonAlgebra(2);
GeneralizedSolomonAlgebra(CoxeterGroup("B",2),Rationals)
gap> Display(CharTable(A.group));
B2
     2   3   2   3   2   2

       11. 1.1 .11  2.  .2
    2P 11. 11. 11. 11. .11

11.      1   1   1  -1  -1
1.1      2   .  -2   .   .
.11      1  -1   1  -1   1
2.       1   1   1   1   1
.2       1  -1   1   1  -1

gap> A.basis[3]*A.basis[2];
-X(1,-1)+X(-1,1)+X(-1,-1)
gap> SolomonHomomorphism(last);
X(1)+2*X(2)+X(3)+X(4)+X(5)
```

115.42 ZeroHeckeAlgebra

`ZeroHeckeAlgebra(`W`)`

This constructs the 0-Hecke algebra of the finite Coxeter group W.

```
gap> W:=CoxeterGroup("B",2);
CoxeterGroup("B",2)
gap> A:=ZeroHeckeAlgebra(W);
ZeroHeckeAlgebra(CoxeterGroup("B",2))
gap> Radical(A);
TwoSidedIdeal(ZeroHeckeAlgebra(CoxeterGroup("B",2)),
[ T(21)-T(12), T(21)-T(212), T(21)-T(121), T(21)-T(1212) ])
```

115.43 Performance

We just present here some examples of computations with the above programs (on a usual PC:2 GHz, 256 Mo).

Constructing the group algebra of a Weyl group of type F_4 (1124 elements):4 seconds

```
gap> W:=CoxeterGroup("F",4);
CoxeterGroup("F",4)
gap> A:=GroupAlgebra(W);
GroupAlgebra(CoxeterGroup("F",4),Rationals)
gap> time;
4080
```

Constructing the Grothendieck ring of the Weyl group of type E_8 (696 729 600 elements, 112 irreducible characters):5 seconds

```
gap> W:=CoxeterGroup("E",8);
CoxeterGroup("E",8)
gap> A:=GrothendieckRing(W);
GrothendieckRing(CoxeterGroup("E",8),Rationals)
gap> time;
5950
```

Computing with the Solomon algebra of the Weyl group of type E_6 (51 840 elements)

- Constructing the algebra   less than 5 seconds
- Computing the Loewy length   1 second
- Computing the Cartan Matrix   around 12 seconds

```
gap> W:=CoxeterGroup("E",6);
CoxeterGroup("E",6)
gap> A:=SolomonAlgebra(W);
SolomonAlgebra(CoxeterGroup("E",6),Rationals)
gap> time;
4610
gap> LoewyLength(A);
5
gap> time;
1060
gap> CartanMatrix(A);

         1
         2  1  1  1  1
         3  2  2  2  3  1  1  1  1                    2
         4  3  3  3  4  2  2  2  3  1  1  1           3
         5  4  4  5  5  3  3  4  5  2  2  3  1  1     4
         6  5  6  6  6  4  5  5  6  3  5  4  2  3  1  5  0

123456   1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 12345   1  1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 12346   1  .  1  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 12356   .  .  .  1  .  .  .  .  .  .  .  .  .  .  .  .  .
 13456   .  .  .  .  1  .  .  .  .  .  .  .  .  .  .  .  .
  1234   1  1  .  .  1  1  .  .  .  .  .  .  .  .  .  .  .
```

```
    1235   2  .  1  1  .  .  1  .  .  .  .  .  .  .  .  .  .
    1245   1  .  1  .  1  .  .  1  .  .  .  .  .  .  .  .  .
    1356   .  .  .  .  .  .  .  .  1  .  .  .  .  .  .  .  .
     123   2  1  1  .  2  1  .  1  1  1  .  .  .  .  .  .  .
     125   1  1  1  1  .  .  .  .  .  .  1  .  .  .  .  .  .
     134   1  1  .  .  1  1  .  .  .  .  .  1  .  .  .  .  .
      12   2  1  1  .  1  1  .  1  1  1  .  .  1  .  .  .  .
      13   1  1  .  .  1  1  .  .  .  .  .  .  .  1  .  .  .
       1   1  1  .  .  1  1  .  .  .  .  .  1  .  .  1  .  .
    2345   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1  .
       0   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1

gap> time;
12640
```

Bibliography

[Abb89] J. A. Abbott. *On the Factorization of Polynomials over Algebraic Fields.* PhD thesis, School of Mathematical Sciences, University of Bath, September 1989.

[AL82] D. Alvis and G. Lusztig. The representations and generic degrees of the Hecke algebra of type H_4. *J. reine angew. Math.*, 336:201–212, 1982. Correction: ibid. **449**, 217–218 (1994).

[Alv87] D. Alvis. The left cells of the Coxeter group of type H_4. *J. Algebra*, 107:160–168, 1987.

[AMW82] D[avid] G. Arrell, S[anjiv] Manrai, and M[ichael] F. Worboys. A procedure for obtaining simplified defining relations for a subgroup. In Campbell and Robertson [CR82], pages 155–159.

[AR84] D[avid] G. Arrell and E[dmund] F. Robertson. A modified Todd-Coxeter algorithm. In Atkinson [Atk84], pages 27–32.

[Art68] E[mil] Artin. *Galoissche Theorie.* Verlag Harri Deutsch, Frankfurt/Main, 1968.

[Atk84] Michael D. Atkinson, editor. *Computational Group Theory, Proceedings LMS Symposium on Computational Group Theory, Durham 1982.* Academic Press, 1984.

[AW97] M. Alp and C. D. Wensley. Enumeration of cat1-groups of low order. *U.W.Bangor Preprint*, 96.05:1–17, 1997.

[Bau91] Ulrich Baum. *Existenz und effiziente Konstruktion schneller Fouriertransformationen überauflösbarer Gruppen.* Dissertation, Rheinische Friedrich Wilhelm Universität Bonn, Bonn, Germany, 1991.

[BBN+78] Harold Brown, Rolf Bülow, Joachim Neubüser, Hans Wondratschek, and Hans Zassenhaus. *Crystallographic Groups of Four-Dimensional Space.* John Wiley, New York, 1978.

[BC72] C. T. Benson and C. W. Curtis. On the degrees and rationality of certain characters of finite Chevalley groups. *Trans. Amer. Math. Soc.*, 165:251–273, 1972.

[BC76] M[ichael] J. Beetham and C[olin] M. Campbell. A note on the Todd-Coxeter coset enumeration algorithm. *Proc. Edinburgh Math. Soc. (2)*, 20:73–79, 1976.

[BC89] Richard P. Brent and Graeme L. Cohen. A new lower bound for odd perfect numbers. *Math. Comput.*, 53:431–437, 1989.

[BC92] Wieb Bosma and John [J.] Cannon. Handbook of Cayley functions. Technical report, Department of Pure Mathematics, University of Sydney, Sydney, Australia, 1992.

[BCFS91] L[azlo] Babai, G[ene] Cooperman, L[arry] Finkelstein, and 'A[kos] Seress. Nearly linear time algorithms for permutation groups with a small base. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'91), Bonn 1991*, pages 200–209. ACM Press, 1991.

[BCM81a] G. Baumslag, F.B. Cannonito, and C.F. Miller III. Computable algebra and group embeddings. *J. Algebra*, 69:186–212, 1981.

[BCM81b] G. Baumslag, F.B. Cannonito, and C.F. Miller III. Some recognizable properties of solvable groups. *Math. Z.*, 178:289–295, 1981.

[BCN89] A.E. Brouwer, A.M. Cohen, and A. Neumaier. *Distance-Regular Graphs*. Spinger, Berlin and New York, 1989.

[BDM02] D. Bessis, F. Digne, and J. Michel. Springer theory in braid groups and the Birman-Ko-Lee monoid. *Pacific J. Math.*, 205:287–309, 2002.

[Ben76] M. Benard. Schur indices and splitting fields of the unitary reflection groups. *J. Algebra*, 38:318–342, 1976.

[Ber76] T. R. Berger. Characters and derived length in groups of odd order. *J. Algebra*, 39:199–207, 1976.

[Bes01] D. Bessis. Zariski theorems and diagrams for braid groups. *Invent. Math.*, 145:487–507, 2001.

[BH78] R. Brown and P. J. Higgins. On the connection between the second relative homotopy group and some related spaces. *Proc. London Math. Soc.*, 36:193–212, 1978.

[BH05] C. Bonnafé and C. Hohlweg. Generalized descent algebra and construction of irreducible characters of hyperoctahedral groups. *Ann. Inst. Four.*, 33:2315 – 2337, 2005.

[Bis89] Thomas Bischops. Collectoren im Programmsystem GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1989.

[BM83] Gregory Butler and John McKay. The transitive groups of degree up to 11. *Communications in Algebra*, 11:863–911, 1983.

[BM93] M. Broué and G. Malle. Zyklotomische Heckealgebren. *Astérisque*, 212:119–189, 1993.

[BM04] David Bessis and Jean Michel. Explicit presentations for exceptional braid groups. *Experimental mathematics*, 13:257–266, 2004.

[BMM93] M. Broué, G. Malle, and J. Michel. Generic blocks of finite reductive groups. *Astérisque*, 212:7–92, 1993.

[BMM99] M. Broué, G. Malle, and J. Michel. Towards spetses i. *Transformation Groups*, 4:157–218, 1999.

[BMM14] M. Broué, G. Malle, and J. Michel. Split spetses for primitive reflection groups. *Astérisque*, 359:1–146, 2014.

[BMR98] M. Broué, G. Malle, and R. Rouquier. Complex reflection groups, braid groups, hecke algebras. *J. Reine Angew. Math.*, 500:127–190, 1998.

[Bon05] C. Bonnafé. Quasi-isolated elements in reductive groups 1. *Communications in Algebra*, 33:2315 – 2337, 2005.

[Bou68] N. Bourbaki. *Groupes et algèbres de Lie, Ch. 4–6.* Hermann, Paris, 1968. Masson, Paris: 1981.

[Bra89] R. J. Bradford. *On the computation of integral bases and defects of integrity.* PhD thesis, School of Mathematical Sciences, University of Bath, 1989.

[Bri71] E. Brieskorn. Die Fundamentalgruppe des Raumes der regulären Orbits einer endlichen komplexen Spiegelungsgruppe. *Invent. Math.*, 12:57–61, 1971.

[BS72] E. Brieskorn and K. Saito. Artin-Gruppen und Coxeter-Gruppen. *Invent. Math.*, 17:245–271, 1972.

[BTW93] Bernhard Beauzamy, Vilmar Trevisan, and Paul S. Wang. Polynomial factorization: Sharp bounds, Efficient algorithms. *J. Symbolic Computation*, 15:393–413, 1993.

[Bur55] W[illiam S.] Burnside. *Theory of Groups of Finite Order.* Dover Publications, New York, 1955. Unabridged republication of the second edition, published in 1911.

[But82] Gregory Butler. Computing in permutation and matrix groups II: Backtrack algorithm. *Math. Comput.*, 39:671–680, 1982.

[But85] Gregory Butler. Effective computation with group homomorphisms. *J. Symbolic Computation*, 1:143–157, 1985.

[But93] Gregory Butler. The transitive groups of degree fourteen and fifteen. *J. Symbolic Computation*, pages 413–422, 1993.

[BW95] R. Brown and C. D. Wensley. On finite induced crossed modules, and the homotopy 2-type of mapping cones. *Theory and Applications of Categories*, 1:54–71, 1995.

[BW96] R. Brown and C. D. Wensley. On the computation of induced crossed modules. *Theory and Applications of Categories*, 2:3–16, 1996.

[Cab96] Marc Cabanes, editor. *Finite Reductive Groups, Related Structures and Representations*, volume 141 of *Progress in Mathematics.* Birkhäuser, Basel, 1996.

[Cam71] Harvey A. Campbell. An extension of coset enumeration. M. Sc. thesis, McGill University, Montreal, Canada, 1971.

[Can73] John J. Cannon. Construction of defining relators for finite groups. *Discrete Math.*, pages 105–129, 1973.

[Car72a] R. W. Carter. Conjugacy classes in the Weyl group. *Compositio Math.*, 25:1–59, 1972.

[Car72b] R. W. Carter. *Simple groups of Lie type.* Wiley, New York, 1972.

[Car85] R. W. Carter. *Finite groups of Lie type: Conjugacy classes and complex characters.* Wiley, New York, 1985.

[Car86] R. W. Carter. Representation theory of the 0–Hecke algebra. *J. Algebra*, 104:89–103, 1986.

[CB93] M. Clausen and U. Baum. *Fast Fourier Transforms.* BI-Wissenschaftsverlag, Mannheim, 1993.

[CCN$^+$85] J[ohn] H. Conway, R[obert] T. Curtis, S[imon] P. Norton, R[ichard] A. Parker, and R[obert] A. Wilson. *Atlas of finite groups.* Oxford University Press, 1985.

[Cel92] Frank Celler. Kohomologie und Normalisatoren in GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992.

[Che73] D. Cheniot. Une démonstration du théorème de Zariski sur les sections hyperplanes d'une hypersurface projective et du théorème de Van Kampen sur le groupe fondamental du complémentaire d'une courbe projective plane. *Compositio Math.*, 27:141–158, 1973.

[CNW90] Frank Celler, Joachim Neubüser, and Charles R. B. Wright. Some remarks on the computation of complements and normalizers in soluble groups. *Acta Applicandae Mathematicae*, 21:57–76, 1990.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics.* Spinger, Berlin and New York, 1993.

[Con90a] S[am] B. Conlon. Calculating characters of p-groups. *J. Symbolic Computation*, 9(5 & 6):535–550, 1990.

[Con90b] S[am] B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Computation*, 9(5 & 6):551–570, 1990.

[CR82] Colin M. Campbell and Edmund F. Robertson, editors. *Groups-St. Andrews 1981, Proceedings of a conference, St. Andrews 1981*, volume 71 of *London Math. Soc. Lecture Note Series.* Cambridge University Press, 1982.

[CR87] C. W. Curtis and I. Reiner. *Methods of representation theory, vol. I, II.* John Wiley, New York, 1981/1987.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine computation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[Del72] P. Deligne. Les immeubles des groupes de tresses généralisés. *Invent. Math.*, 17:273–302, 1972.

[Deo89] V.V. Deodhar. A note on subgroups generated by reflections in Coxeter groups. *Arch. Math.*, 53:543–546, 1989.

[DG99] P. Diaconis and R. Graham. The graph of generating sets of an abelian group. *Colloq. Math.*, 80:31–38, 1999.

[Dix67] J[ohn] D. Dixon. High speed computations of group characters. *Num. Math.*, 10:446–450, 1967.

[DM18] F. Digne and J. Michel. Quasi-semisimple elements. *Proc. LMS*, 116:1301–1328, 2018.

[DP99] P. Dehornoy and L. Paris. Gaussian groups and garside groups, two generalizations of artin groups. *Proc. LMS*, 79:569–604, 1999.

[Dre69] Andreas [W. M.] Dress. A characterization of solvable groups. *Math. Z.*, 110:213–217, 1969.

[DuC91] F. DuCloux. *Coxeter Version 1.0.* Université de Lyon, France, 1991.

[Dye90] M. Dyer. Reflection subgroups of Coxeter systems. *J. Algebra*, 135:57–73, 1990.

[ECH+92] D.B.A. Epstein, J.W. Cannon, D.F. Holt, S. Levy, M.S. Paterson, and W.P. Thurston. *Word Processing and Group Theory.* Jones and Bartlett, 1992.

[Egn97a] S. Egner. *Zur Algorithmischen Zerlegungstheorie linearer Transformationen mit Symmetrie.* PhD thesis, Universität Karlsruhe, 1997.

[EGN97b] Bettina Eick, Franz G"ahler, and Werner Nickel. Computing Maximal Subgroups and Wyckoff Positions of Space Groups. *Acta Cryst A*, 1997.

[Ell84] G. Ellis. *Crossed modules and their higher dimensional analogues.* PhD thesis, University of Wales, Bangor, 1984.

[ER82] Douglas F. Elliott and K. Ramamohan Rao. *Fast Transforms — Algorithms, Analyses, Applications.* Academic Press, 1982.

[FGM03] N. Franco and J. Gonzalez-Meneses. Conjugacy problem for braid groups and garside groups. *J. Algebra*, 266:112–132, 2003.

[FJNT95] V[olkmar] Felsch, D[avid] L. Johnson, J[oachim] Neubüser, and S[ergey] V. Tsaranov. The structure of certain Coxeter groups. In C[olin] M. Campbell, T[haddeus] C. Hurley, E[dmund] F. Robertson, S[ean] J. Tobin, and J[ames] J. Ward, editors, *Groups '93 Galway / St. Andrews, Galway 1993, Volume 1*, volume 211 of *London Math. Soc. Lecture Note Series*, pages 177–190. Cambridge University Press, 1995.

[Fra82] J[ames] S. Frame. Recursive computation of tensor power components. *Bayreuther Math. Schr.*, 10:153–159, 1982.

[FS84] Volkmar Felsch and Günter Sandlöbes. An interactive program for computing subgroups. In Atkinson [Atk84], pages 137–143.

[Gar69] F. A. Garside. The braid groups and other groups. *Quart. J. Math. Oxford, 2^{nd} Ser*, 20:235–254, 1969.

[Gec94] M. Geck. On the character values of Iwahori-Hecke algebras of exceptional type. *Proc. London Math. Soc.*, 68:51–76, 1994.

[Gec95] M. Geck. *Beiträge zur Darstellungstheorie von Iwahori–Hecke Algebren*, volume 11 of *Aachener Beiträge zur Mathematik.* Verlag der Augustinus Buchhandlung, Aachen, 1995.

[GG12] I[ain] G. Gordon and S[tephen] Griffeth. Catalan numbers for complex reflection groups. *American Journal of Mathematics*, 134:1491–1502, 2012.

[GGM10] V. Gebhardt and J. Gonzalez-Meneses. Solving the conjugacy problem in Garside groups by cyclic sliding. *J. Symbolic Computation*, 45:629–656, 2010.

[GH14] M. Geck and A. Halls. Kazhdan-Lusztig cells in type E_8. *Arxiv*, 2014. to appear.

[GHL+96] M. Geck, G. Hiss, F. Lübeck, G. Malle, and G. Pfeiffer. CHEVIE-A system for computing and processing generic character tables for finite groups of lie type. *AAECC*, 7:175–210, 1996.

[Gil90] N. D. Gilbert. Derivations, automorphisms and crossed modules. *Comm. in Algebra*, 18:2703–2734, 1990.

[GK96] M. Geck and S. Kim. Bases for the Bruhat–Chevalley order on all finite Coxeter groups. Preprint, 1996.

[GKP90] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1990.

[Gla87] S[tephan] P. Glasby. *Computational Approaches to the Theory of Finite Soluble Groups.* Phd thesis, Department of Pure Mathematics, University of Sydney, Sydney, Australia, 1987.

[GM97] M. Geck and J. Michel. On "good" elements in the conjugacy classes of finite Coxeter groups and their eigenvalues on the irreducible representations of Iwahori-Hecke algebras. *Proc. London Math. Soc.*, 74:275–305, 1997.

[GP93] M. Geck and G. Pfeiffer. On the irreducible characters of Hecke algebras. *Advances in Math.*, 102:79–94, 1993.

[GS90] S[tephan] P. Glasby and M[ichael] C. Slattery. Computing intersections and normalizers in soluble groups. *J. Symbolic Computation*, 9:637–651, 1990.

[Hah83] Theo Hahn, editor. *International Tables for Crystallography, Volume A, Space-group Symmetry.* Reidel, Dordrecht, Boston, 1983.

[Hav69] George Havas. Symbolic and algebraic calculation. Basser Computing Dept., Technical Report 89, Basser Department of Computer Science, University of Sydney, Sydney, Australia, 1969.

[Hav74] George Havas. A Reidemeister-Schreier program. In M[ichael] F. Newman, editor, *Proceedings of the Second International Conference on the Theory of Groups, Canberra, 1973*, volume 372 of *Lecture Notes in Math.*, pages 347–356. Springer, Berlin, 1974.

[HER91] D.F. Holt, D.B.A. Epstein, and S. Rees. The use of knuth-bendix methods to solve the word problem in automatic groups. *J. Symbolic Computation*, 12:397–414, 1991.

[HIÖ89] Trevor [O.] Hawkes, I. M[artin] Isaacs, and M. Özaydin. On the Möbius function of a finite group. *Rocky Mountain J. Math.*, 19:1003–1034, 1989.

[HKRR84] George Havas, P[eter] E. Kenne, J[ames] S. Richardson, and E[dmund] F. Robertson. A Tietze transformation program. In Atkinson [Atk84], pages 67–71.

[HN80] George Havas and M[ichael] F. Newman. Application of computers to questions like those of Burnside. In J[ens] L. Mennicke, editor, *Burnside groups, Proceedings of a workshop, Bielefeld, Germany, 1977*, volume 806 of *Lecture Notes in Math.*, pages 211–230. Springer, Berlin, 1980.

[Holar] Derek F. Holt. The warwick automatic groups software. In *Proceedings of DIMACS Conference on Computational Group Theory, Rutgers, March 1994.*, To appear.

[How95] John M. Howie. *Fundamentals of Semigroup Theory*, volume 12 of *London Mathematical Society Monographs New Series.* Oxford University Press, 1995.

[HP89] Derek F. Holt and W[ilhelm] Plesken. *Perfect Groups.* Oxford Math. Monographs. Clarendon, Oxford, 1989.

[HR94] Derek [F.] Holt and Sarah Rees. Testing modules for irreducibility. *J. Austral. Math. Soc. Ser. A*, 57:1–16, 1994.

[HS64] Marshall Hall, Jr. and James K. Senior. *The Groups of Order* 2^n *(*$n \leq 6$*).* The Macmillan Company, New York, 1964.

[HSS01] J. Hubbard, D. Schleicher, and S. Sutherland. How to find all roots of complex polynomials by Newton's method. *Invent. Math.*, 146:1–33, 2001.

[Hug85] Hugues. On decompositions in complex imprimitive reflection groups. *Indagationes*, 88:207–219, 1985.

[Hum90] J. E. Humphreys. *Reflections groups and Coxeter groups*, volume 29 of *Cambridge studies in advanced Math.* Cambridge University Press, 1990.

[Hup67] B[ertram] Huppert. *Endliche Gruppen I*, volume 134 of *Grundlehren Math. Wiss.* Springer, Berlin, 1967.

[JLPW95] Christoph Jansen, Klaus Lux, Richard [A.] Parker, and Robert [A.] Wilson. *An Atlas of Brauer Characters*, volume 11 of *London Math. Soc. Monographs.* Clarendon, Oxford, 1995.

[Kac82] V. Kac. *Infinite dimensional Lie algebras.* Cambridge University Press, 1982.

[Ker91] Adalbert Kerber. *Algebraic Combinatorics Via Finite Group Actions.* BI-Wissenschaftsverlag, Mannheim, 1991.

[KL79] D. A. Kazhdan and G. Lusztig. Representations of Coxeter groups and Hecke algebras. *Invent. Math.*, 53:165–184, 1979.

[Kon65] T. Kondo. The characters of the weyl group of type f_4. *J. Fac. Sci. Univ. Tokyo*, 11:145–153, 1965.

[Lal79] Gérard Lallement. *Semigroups and Combinatorial Applications.* John Wiley, New York, 1979.

[Lau82] Reinhard Laue. *Zur Konstruktion und Klassifikation endlicher auflösbarer Gruppen*, volume 9 of *Bayreuther Math. Schr.* Universität Bayreuth, Bayreuth, Germany, 1982.

[LeC86] P. LeChenadec. *Canonical Forms in Finitely Presented Algebras.* London Pitman and New York, Wiley, 1986.

[Leh89a] Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany. *SOGOS - A Program System for Handling Subgroups of Finite Soluble Groups, version 5.0, User's reference manual*, 1989.

[Leh89b] Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany. *SPAS - Subgroup Presentation Algorithms System, version 2.5, User's reference manual*, 1989.

[Leo80] Jeffrey S. Leon. On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Math. Comput.*, 35:941–974, 1980.

[Leo91] S. Jeffrey Leon. Permutation Group Algorithms Based on Partitions, I: Theory and Algorithms. *Journal of Symbolic Computation*, 12:533–583, 1991.

[LGS90] C[harles] R. Leedham-Green and L[eonard] H. Soicher. Collection from the left and other strategies. *J. Symbolic Computation*, 9(5 & 6):665–675, 1990.

[Lin93] Steve Linton. *Vector Enumeration Programs, version 3*, 1993.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:513–534, 1982.

[LNS84] R[einhard] Laue, J[oachim] Neubüser, and U[lrich] Schoenwaelder. Algorithms for finite soluble groups and the SOGOS system. In Atkinson [Atk84], pages 105–135.

[Lo96] E. Lo. *A Polycyclic Quotient Algorithm.* PhD thesis, Rutgers University, 1996.

[Lod82] J. L. Loday. Spaces with finitely many non-trivial homotopy groups. *J. App. Algebra*, 24:179–202, 1982.

[LP91] Klaus Lux and Herbert Pahlings. Computational aspects of representation theory of finite groups. In G. O. Michler and C. R. Ringel, editors, *Representation theory of finite groups and finite–dimensional algebras*, volume 95 of *Progress in Mathematics*, pages 37–64. Birkhäuser, Basel, 1991.

[LPRR] S. A. Linton, G. Pfeiffer, E. F. Robertson, and N. Ruškuc. Computing transformation semigroups. in preparation.

[LPRR97] S. A. Linton, G. Pfeiffer, E. F. Robertson, and N. Ruškuc. Groups and actions in transformation semigroups. *Math. Z.*, 1997. to appear.

[LS99] G. I. Lehrer and T. Springer. Reflection multiplicities and reflection subquotients of unitary reflection groups, i. In *geometric group theory down under*, pages 181–193, 1999.

[Lus76] G. Lusztig. Coxeter orbits and eigenspaces of Frobenius. *Invent. Math.*, 34:101–159, 1976.

[Lus81] G. Lusztig. On a theorem of Benson and Curtis. *J. Algebra*, 71:490–498, 1981.

[Lus83] G. Lusztig. Left cells in Weyl groups. In *Lie groups representations*, volume 1024, pages 99–111. Springer, 1983.

[Lus85] G. Lusztig. *Characters of reductive groups over a finite field*, volume 107 of *Annals of Math. Studies.* Princeton University Press, 1985.

[Mal92] Henrique S. Malvar. *Signal Processing with Lapped Transforms.* Artech House, 1992.

[Mal96] Gunter Malle. Degrés relatifs des algèbres cyclotomiques associées aux groupes de réflexions complexes de dimension deux. In Cabanes [Cab96].

[Mal00] Gunter Malle. On the generic degrees of cyclotomic algebras. *Representation theory*, 4:342–369, 2000.

[McK90] B.D. McKay. *nauty user's guide (version 1.5), Technical report TR-CS-90-02.* Australian National University, Computer Science Department, ANU, 1990.

[McL77] D. H. McLain. An algorithm for determining defining relations of a subgroup. *Glasgow Math. J.*, 18:51–56, 1977.

[Mer96] A. Mertins. *Signaltheorie.* Teubner Verlag, 1996.

[Min93] T. Minkwitz. *Algorithmensynthese für lineare Systeme mit Symmetrie.* PhD thesis, Universität Karlsruhe, 1993.

[Min95] T. Minkwitz. Algorithms Explained by Symmetry. *Lecture Notes on Computer Science*, 900:157–167, 1995.

[Min96] T. Minkwitz. Extension of Irreducible Representations. *Applicable Algebra in Engineering, Communication and Computing*, 7:391–399, 1996.

[MM98] Gunter Malle and Andrew Mathas. Symmetric cyclotomic Hecke algebras. *J. Algebra*, 205(1):275–293, 1998.

[MM10a] Gunter Malle and Jean Michel. Constructing representations of hecke algebras for complex reflection groups. *LMS J. Comput. Math.*, 13:426–450, 2010.

[MM10b] Ivan Marin and Jean Michel. Automorphisms of complex reflection groups. *Representation theory*, 14:747–788, 2010.

[Mni92] Jürgen Mnich. Untergruppenverbände und auflösbare Gruppen in GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Math. Comput.*, 44:519–521, 1985.

[MR03] Gunter Malle and Raphaël Rouquier. Familles de caractères de groupes de réflexions complexes. *Representation theory*, 7:610–640, 2003.

[MS85] John McKay and Leonard H. Soicher. Computing Galois groups over the rationals. *J. Number Theory*, 20:273–281, 1985.

[Mur58] F[rancis] D. Murnaghan. The orthogonal and symplectic groups. Communications Series A 13, Dublin Inst. Adv. Studies, 1958.

[Neb95] Gabriele Nebe. *Endliche rationale Matrixgruppen vom Grad 24*, volume 12 of *Aachener Beiträge zur Mathematik*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, 1995. Dissertation, Lehrstuhl B für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1995.

[Neu67] Joachim Neubüser. *Die Untergruppenverbände der Gruppen der Ordnungen* ≤ 100 *mit Ausnahme der Ordnungen 64 und 96*. Habilitationsschrift, Universität Kiel, Kiel, Germany, 1967.

[Neu82] Joachim Neubüser. An elementary introduction to coset table methods in computational group theory. In Campbell and Robertson [CR82], pages 1–45.

[New77] M[ichael] F. Newman. Determination of groups of prime-power order. In R. A. Bryce, J. Cossey, and M[ichael] F. Newman, editors, *Group theory, Proc. Miniconf., Austral. Nat. Univ., Canberra, 1975*, volume 573 of *Lecture Notes in Math.*, pages 73–84. Springer, Berlin, 1977.

[NO89] M[ichael] F. Newman and E[amonn] A. O´Brien. A CAYLEY library for the groups of order dividing 128. In *Group Theory, Proceedings of the 1987 Singapore Conference, Singapore 1987*, pages 437–442. Walter de Gruyter, Berlin, New York, 1989.

[NO96] M[ichael] F. Newman and E[amonn] A. O´Brien. Application of computers to questions like those of Burnside, II. *Internat. J. Algebra Comput.*, 6:593–605, 1996.

[Nor87] K. J. Norrie. *Crossed modules and analogues of group theorems*. PhD thesis, King's College, University of London, 1987.

[Nor90] K. J. Norrie. Actions and automorphisms of crossed modules. *Bull. Soc. Math. France*, 118:129–146, 1990.

[NP95a] G[abriele] Nebe and W[ilhelm] Plesken. *Finite rational matrix groups*, volume 556 of *AMS Memoirs*. American Mathematical Society, 1995.

[NP95b] G[abriele] Nebe and W[ilhelm] Plesken. *Finite rational matrix groups of degree 16*, pages 74–144. Volume 556 of *AMS Memoirs* [NP95a], 1995.

[NPP84] J[oachim] Neubüser, H[erbert] Pahlings, and W[ilhelm] Plesken. CAS; design and use of a system for the handling of characters of finite groups. In Atkinson [Atk84], pages 195–247.

[NPW81] J[oachim] Neubüser, W[ilhelm] Plesken, and H[ans] Wondratschek. An emendatory discursion on defining crystal systems. *Match*, 10:77–96, 1981.

[O´Br90] E[amonn] A. O´Brien. The p-group generation algorithm. *J. Symbolic Computation*, 9:677–698, 1990.

[O´Br91] E[amonn] A. O´Brien. The groups of order 256. *J. Algebra*, 142, 1991.

[O´Br94] E[amonn] A. O´Brien. Isomorphism testing for p-groups. *J. Symbolic Computation*, 17 (1):133–147, 1994.

[O´Br95] E[amonn] A. O´Brien. Computing automorphism groups of p-groups. In Wieb Bosma and Alf van der Poorten, editors, *Computational Algebra Number and Number Theory*, pages 83–90. (Sydney, 1992), Kluwer Academic Publishers, Dordrecht, 1995.

[Ost86] Th[omas] Ostermann. Charaktertafeln von Sylownormalisatoren sporadischer einfacher Gruppen. Vorlesungen aus dem Fachbereich Mathematik 14, Universität Essen, Essen, Germany, 1986.

[Püs98] M. Püschel. *Konstruktive Darstellungstheorie und Algorithmengenerierung.* PhD thesis, Universität Karlsruhe, 1998.

[Pah93] Herbert Pahlings. On the Möbius function of a finite group. *Arch. Math.*, 60:7–14, 1993.

[Par02] L. Paris. Artin monoids inject in their groups. *Comment. Math. Helv.*, 77:609–637, 2002.

[Pfe94] G. Pfeiffer. Young characters on Coxeter basis elements of Iwahori-Hecke algebras and a Murnaghan-Nakayama formula. *J. Algebra*, 168:525–535, 1994.

[Pfe96] G. Pfeiffer. Character values of Iwahori-Hecke algebras of type B. In Cabanes [Cab96].

[Pil83] Günter Pilz. *Near-Rings*, volume 23 of *North-Holland Mathematics Studies.* North-Holland Publishing Company, 1983.

[Ple85] W[ilhelm] Plesken. Finite unimodular groups of prime degree and circulants. *J. Algebra*, 97:286–312, 1985.

[Ple90] W[ilhelm] Plesken. Additive decompositions of positive integral quadratic forms. The paper is available at Lehrstuhl B für Mathematik, Rheinisch Westfälische Technische Hochschule Aachen, may be it will be published in the near future, 1990.

[PN95] W[ilhelm] Plesken and G[abriele] Nebe. *Finite rational matrix groups*, pages 1–73. Volume 556 of *AMS Memoirs* [NP95a], 1995.

[Poh87] M[ichael] Pohst. A modification of the lll reduction algorithm. *J. Symbolic Computation*, 4:123–127, 1987.

[PP77] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of gl(n,z). I. the five and seven dimensional cases, II. the six dimensional case. *Math. Comput.*, 31:536–576, 1977.

[PP80] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of gl(n,z). III. the nine dimensional case, IV. remarks on even dimensions with application to n = 8, V. the eight dimensional case and a complete description of dimensions less than ten. *Math. Comput.*, 34:245–301, 1980.

[Rad68] Charles M. Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56:1107–1108, 1968.

[Ram91] A. Ram. A Frobenius formula for the characters of the Hecke algebras. *Invent. Math.*, 106:461–488, 1991.

[Ric82] R. W. Richardson. Conjugacy classes of involutions in Coxeter groups. *Bull. Aust. Math. Soc.*, 26:1–15, 1982.

[Rin93] Michael Ringe. *The C MeatAxe, Release 1.5.* Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1993.

[Rob88] E[dmund] F. Robertson. Tietze transformations with weighted substring search. *J. Symbolic Computation*, 6:59–64, 1988.

[Roy87] Gordon F. Royle. The transitive groups of degree twelve. *J. Symbolic Computation*, pages 255–268, 1987.

[Sch90] Gerhard J. A. Schneider. Dixon's character table algorithm revisited. *J. Symbolic Computation*, 9:601–606, 1990.

[Sho92] Mark W. Short. *The Primitive Soluble Permutation Groups of Degree less than 256*, volume 1519 of *Lecture Notes in Math.* Springer, Berlin and Heidelberg, 1992.

[Sim70] Charles C. Sims. Computational methods in the study of permutation groups. In John Leech, editor, *Computational Problems in Abstract Algebra, Proc. Conf. Oxford, 1967*, pages 169–183. Pergamon Press, Oxford, 1970.

[Sim94] C.C. Sims. *Computation with Finitely Presented Groups.* Cambridge University Press, 1994.

[Soi93] L[eonard] H. Soicher. GRAPE: a system for computing with graphs and groups. In L. Finkelstein and B. Kantor, editors, *Proceedings of the 1991 DIMACS Workshop on Groups and Computation*, volume 11 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 287–291. American Mathematical Society, 1993.

[Sou94] Bernd Souvignier. Irreducible finite integral matrix groups of degree 8 and 10. *Math. Comput.*, 63:335–350, 1994.

[Spa82] N. Spaltenstein. *Classes unipotentes et sous-groupes de Borel*, volume 946 of *Lecture notes in Mathematics.* Springer, Berlin and Heidelberg, 1982.

[ST54] G. C. Shephard and J. A. Todd. Finite unitary reflection groups. *Canad. J. Math.*, 6:274–304, 1954.

[Ste89] J. R. Stembridge. On the eigenvalues of representations of reflection groups and wreath products. *Pacific J. Math.*, 140:353–396, 1989.

[Tay19] J. Taylor. The structure of root data and smooth regular embeddings of reductive groups. *Proc. Edinb. Math. Soc.*, 62:523–552, 2019.

[TH95] editor Theo Hahn. *International Tables for Crystallography, Volume A, Space-group Symmetry.* Kluwer, Dordrecht, 4th edition, 1995.

[Thi87] Peter Thiemann. SOGOS III - Charaktere und Effizienzuntersuchung. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1987.

[vdW76] Robert W. van der Waall. On symplectic primitive modules and monomial groups. *Indagationes Math.*, 38:362–375, 1976.

[VL84] M[ichael] R. Vaughan-Lee. An aspect of the nilpotent quotient algorithm. In Atkinson [Atk84], pages 75–84.

[VL90a] M[ichael] R. Vaughan-Lee. Collection from the left. *J. Symbolic Computation*, 9:725–733, 1990.

[VL90b] M[ichael] R. Vaughan-Lee. *The restricted Burnside problem*, volume 5 of *London Math. Soc. Monographs*. Clarendon, Oxford, 1990.

[Whi48] J. H. C. Whitehead. On operators in relative homotopy groups. *Ann. of Math.*, 49:610–640, 1948.

[Whi49] J. H. C. Whitehead. Combinatorial homotopy ii. *Bull. Amer. Math. Soc.*, 55:453–496, 1949.

[Won95] Hans Wondratschek. Introduction to space-group symmetry. In Hahn [TH95], pages 711–735.

[Wur93] Martin Wursthorn. SISYPHOS *Computing in modular group algebras, Version 0.5.* Math. Inst. B, 3. Lehrstuhl, Universität Stuttgart, 1993.

[Zum89] Matthias Zumbroich. Grundlagen einer Arithmetik in Kreisteilungskörpern und ihre Implementation in CAS. Dimplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1989.

Index

Index