GAP --- A Tutorial

Release 4.15.1, 2025-10-18

The GAP Group

The GAP Group Email: support@gap-system.org
Homepage: https://www.gap-system.org

Copyright

Copyright © (1987-2025) for the core part of the GAP system by the GAP Group.

Most parts of this distribution, including the core part of the GAP system are distributed under the terms of the GNU General Public License, see https://www.gnu.org/licenses/gpl.html or the file GPL in the etc directory of the GAP installation.

More detailed information about copyright and licenses of parts of this distribution can be found in (**Reference: Copyright and License**).

GAP is developed over a long time and has many authors and contributors. More detailed information can be found in (**Reference: Authors and Maintainers**).

Contents

1 Preface **5**
- 1.1 The GAP System . . . 5
- 1.2 Further Information about GAP . . . 7

2 A First Session with GAP **9**
- 2.1 Starting and Leaving GAP . . . 9
- 2.2 Loading Source Code from a File . . . 10
- 2.3 The Read Evaluate Print Loop . . . 10
- 2.4 Constants and Operators . . . 12
- 2.5 Variables versus Objects . . . 14
- 2.6 Objects vs. Elements . . . 16
- 2.7 About Functions . . . 16
- 2.8 Help . . . 17
- 2.9 Further Information introducing the System . . . 18

3 Lists and Records **20**
- 3.1 Plain Lists . . . 20
- 3.2 Identical Lists . . . 23
- 3.3 Immutability . . . 24
- 3.4 Sets . . . 25
- 3.5 Ranges . . . 26
- 3.6 For and While Loops . . . 27
- 3.7 List Operations . . . 29
- 3.8 Vectors and Matrices . . . 30
- 3.9 Plain Records . . . 32
- 3.10 Further Information about Lists . . . 33

4 Functions **34**
- 4.1 Writing Functions . . . 34
- 4.2 If Statements . . . 35
- 4.3 Local Variables . . . 36
- 4.4 Recursion . . . 37
- 4.5 Further Information about Functions . . . 38

5 Groups and Homomorphisms **39**
5.1 Permutation groups . . . 39
5.2 Actions of Groups . . . 42
5.3 Subgroups as Stabilizers . . . 46
5.4 Group Homomorphisms by Images . . . 51
5.5 Nice Monomorphisms . . . 54
5.6 Further Information about Groups and Homomorphisms . . . 56

6 Vector Spaces and Algebras **57**
6.1 Vector Spaces . . . 57
6.2 Algebras . . . 60
6.3 Further Information about Vector Spaces and Algebras . . . 66

7 Domains **67**
7.1 Domains as Sets . . . 67
7.2 Algebraic Structure . . . 68
7.3 Notions of Generation . . . 68
7.4 Domain Constructors . . . 69
7.5 Forming Closures of Domains . . . 69
7.6 Changing the Structure . . . 69
7.7 Subdomains . . . 70
7.8 Further Information about Domains . . . 71

8 Operations and Methods **72**
8.1 Attributes . . . 72
8.2 Properties and Filters . . . 73
8.3 Immediate and True Methods . . . 75
8.4 Operations and Method Selection . . . 75

References **78**

Index **79**

Chapter 1

Preface

Welcome to GAP. This preface serves not only to introduce this manual, "the GAP Tutorial", but also as an introduction to the system as a whole.

GAP stands for *Groups, Algorithms and Programming*. The name was chosen to reflect the aim of the system, which is introduced in this tutorial manual. Since that choice, the system has become somewhat broader, and you will also find information about algorithms and programming for other algebraic structures, such as semigroups and algebras.

In addition to this manual, there are *GAP Reference Manual* containing detailed documentation of the mathematical functionality of GAP, and *HPC-GAP Reference Manual* documenting a multi-threaded version of GAP.

There is also a document `CHANGES.md` in the root directory on the most essential changes from previous GAP releases. A lot of the functionality of the system and a number of contributed extensions are provided as "GAP packages" and each of these has its own manual.

Subsequent sections of this preface explain the structure of the system and list sources of further information about GAP.

1.1 The GAP System

GAP is a *free*, *open* and *extensible* software package for computation in discrete abstract algebra. The terms "free" and "open" describe the conditions under which the system is distributed -- in brief, it is *free of charge* (except possibly for the immediate costs of delivering it to you), you are *free to pass it on* within certain limits, and all of the workings of the system are *open for you to examine and change*. Details of these conditions can be found in Section (**Reference: Copyright and License**).

The system is "extensible" in that you can write your own programs in the GAP language, and use them in just the same way as the programs which form part of the system (the "library"). Indeed, we actively support the contribution, refereeing and distribution of extensions to the system, in the form of "GAP packages". Further details of this can be found in Chapter (**Reference: Using and Developing GAP Packages**), and on our website.

Development of GAP began at Lehrstuhl D für Mathematik, RWTH-Aachen, under the leadership of Joachim Neubüser in 1985. Version 2.4 was released in 1988 and version 3.1 in 1992. In 1997 coordination of GAP development, now very much an international effort, was transferred to St Andrews. A complete internal redesign and almost complete rewrite of the system was completed over the following years and version 4.1 was released in July 1999. A sign of the further internationalization of the project was the GAP 4.4 release in 2004, which has been coordinated from Colorado

State University, Fort Collins.

More information on the motivation and development of GAP to date, can be found on our website in a section entitled "Some History of GAP": https://www.gap-system.org/Doc/History/history.html.

For those readers who have used an earlier version of GAP, an overview of the changes from GAP 4.4 and a brief summary of changes from earlier versions is given in `CHANGES.md` file in the main directory.

The system that you are getting now consists of a "core system" and a number of packages. The core system consists of four main parts.

1. A *kernel*, written in C, which provides the user with
   - automatic dynamic storage management, which the user needn't bother about when 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 GAP language, an untyped imperative programming language with functions as first class objects and some extra built-in data types such as permutations and finite field elements. The language supports a form of object-oriented programming, similar to that supported by languages like C++ and Java but with some important differences.
   - a small set of system functions allowing the GAP programmer to handle files and execute external programs in a uniform way, regardless of the particular operating system in use.
   - a set of programming tools for testing, debugging, and timing algorithms.
   - a "read-eval-view" style user interface.
2. A much larger *library of GAP functions* that implement algebraic and other algorithms. Since this is written entirely in the GAP language, the GAP language is both the main implementation language and the user language of the system. Therefore a user can, as easily as the original programmers, investigate and vary algorithms of the library and add new ones to it, first for their own use and eventually for the benefit of all GAP users.
3. A *library of group theoretical data* which contains various libraries of groups, including the library of small groups (containing all groups of order at most 2000, except those of order 1024) and others. Large libraries of ordinary and Brauer character tables and Tables of Marks are included as packages.
4. The *documentation*. This is available as on-line help, as printable files in PDF format and as HTML for viewing with a Web browser.

Also included with the core system are some test files and a few small utilities which we hope you will find useful.

GAP *packages* are self-contained extensions to the core system. A package contains GAP code and its own documentation and may also contain data files or external programs to which the GAP code provides an interface. These packages may be loaded into GAP using the `LoadPackage` (**Reference: LoadPackage**) command, and both the package and its documentation are then available just as if they were parts of the core system. Some packages may be loaded automatically, when GAP is started, if they are present. Some packages, because they depend on external programs, may only

be available on the operating systems where those programs are available (usually UNIX). You should note that, while the packages included with this release are the most recent versions ready for release at this time, new packages and new versions may be released at any time and can be easily installed in your copy of GAP.

With GAP there are two packages (the library of ordinary and Brauer character tables, and the library of tables of marks) which contain functionality developed from parts of the GAP core system. These have been moved into packages for ease of maintenance and to allow new versions to be released independently of new releases of the core system. The library of small groups should also be regarded as a package, although it does not currently use the standard package mechanism. Other packages contain functionality which has never been part of the core system, and may extend it substantially, implementing specific algorithms to enhance its capabilities, providing data libraries, interfaces to other computer algebra systems and data sources such as the electronic version of the Atlas of Finite Group Representations; therefore, installation and usage of packages is recommended.

Further details about GAP packages can be found in chapter (**Reference: Using and Developing GAP Packages**), and on the GAP website here: https://www.gap-system.org/Packages/packages.html.

1.2 Further Information about GAP

Information about GAP is best obtained from the GAP website https://www.gap-system.org. There you will find, amongst other things:

- directions to the sites from which you can download the current GAP distribution, all accepted and deposited GAP packages, and a selection of other contributions.
- the GAP manual and an archive of the `gap-forum` mailing list, formatted for reading with a Web browser, and indexed for searching.
- information about GAP developers, and about the email addresses available for comment, discussion and support.

We would particularly ask you to note the following things:

- The GAP Forum – an email discussion forum for comments, discussions or questions about GAP. You must subscribe to the list before you can post to it, see the website for details. In particular we will announce new releases in this mailing list.
- The email address support@gap-system.org to which you are asked to send any questions or bug reports which do not seem likely to be of interest to the whole GAP Forum. Please give a (short, if possible) self-contained excerpt of a GAP session containing both input and output that illustrates your problem (including comments of why you think it is a bug) and state the type of the machine, operating system, (compiler used, if UNIX/Linux) and the version of GAP you are using (the first line after the GAP 4 banner starting `GAP, Version 4...`).
- We also ask you to send a brief message to support@gap-system.org when you install GAP.
- The correct form of citation of GAP, which we ask you use whenever you publish scientific results obtained using GAP.

It finally remains for us to wish you all pleasure and success in using GAP, and to invite your constructive comment and criticism.

The GAP Group,

2025-10-18

Chapter 2

A First Session with GAP

This tutorial introduces you to the GAP system. It is written with users in mind who have just managed to start GAP for the first time on their computer and want to learn the basic facts about GAP by playing around with some instructive examples. Therefore, this tutorial contains at many places examples consisting of several lines of input (which you should type on your terminal) followed by the corresponding output (which GAP produces as an answer to your input).

We encourage you to actually run through these examples on your computer. This will support your feeling for GAP as a tool, which is the leading aim of this tutorial. Do not believe any statement in it as long as you cannot verify it for your own version of GAP. You will learn to distinguish between small deviations of the behavior of your personal GAP from the printed examples and serious nonsense.

Since the printing routines of GAP 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 support@gap-system.org.

The examples in this tutorial should explain everything you have to know in order to be able to use GAP. The reference manual then gives a more systematic treatment of the various types of objects that GAP can manipulate. It seems desirable neither to start this systematic course with the most elementary (and most boring) structures, nor to confront you with all the complex data types before you know how they are composed from elementary structures. For this reason this tutorial wants to provide you with a basic understanding of GAP objects, on which the reference manual will then build when it explains everything in detail. So after having mastered this tutorial, you can immediately plunge into the exciting parts of GAP and only read detailed information about elementary things (in the reference manual) when you really need them.

Each chapter of this tutorial contains a section with references to the reference manual at the end.

2.1 Starting and Leaving GAP

If the program is correctly installed then you usually start GAP by simply typing gap at the prompt of your operating system followed by the RETURN key, sometimes this is also called the NEWLINE key.

Example

```
$ gap
```

GAP answers your request with its beautiful banner and then it shows its own prompt gap> asking

you for further input. (You can avoid the banner with the command line option -b; more command line options are described in Section (**Reference: Command Line Options**).)

Example

```
gap>
```

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

Example

```
gap> quit;
$
```

On some systems you could type CTRL-D to yield the same effect. In any situation GAP is ended by typing CTRL-C twice within a second. Here as always, a combination like CTRL-D means that you have to press the D key while you hold down the CTRL key.

On some systems minor changes might be necessary. This is explained in GAP installation instructions (see the INSTALL file in the GAP root directory, or the GAP website).

In most places *whitespace* characters (i.e. SPACEs, TABs and RETURNs) are insignificant for the meaning of GAP 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 GAP starts with the symbol # and continues to the end of the line. Comments are treated like whitespace by GAP. We use comments in the printed examples in this tutorial to explain certain lines of input or output.

2.2 Loading Source Code from a File

The most convenient way of creating larger pieces of GAP code is to write them to some text file. For this purpose you can simply use your favorite text editor. You can load such a file into GAP using the Read (**Reference: Read**) function:

Example

```
gap> Read("../../GAPProgs/Example.g");
```

You can either give the full absolute path name of the source file or its relative path name from the current directory.

2.3 The Read Evaluate Print Loop

GAP is an interactive system. It continuously executes a read evaluate print loop. Each expression you type at the keyboard is read by GAP, evaluated, and then the result is shown.

The interactive nature of GAP 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 GAP. 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 GAP, but are very useful for human readers of your program text. GAP treats input from a file in the same way that it treats input from the keyboard. Further details can be found in section `Read` (**Reference: Read**).

A simple calculation with GAP 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.

Example

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

Then GAP echoes the result 22 on the next line and shows with the prompt that it is ready for the next problem. Henceforth, we will no longer print this additional prompt.

If you make a mistake while typing the line, but *before* typing the final RETURN, you can use the DELETE key (or sometimes BACKSPACE key) to delete the last typed character. You can also move the cursor back and forward in the line with CTRL-B and CTRL-F and insert or delete characters anywhere in the line. The line editing commands are fully described in section (**Reference: Line Editing**).

If you did omit the semicolon at the end of the line but have already typed RETURN, then GAP 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 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.

Example

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

So the input can consist of several lines, and GAP prints a partial prompt > in each input line except the first, until the command is completed with a semicolon. (GAP may already evaluate part of the input when RETURN is typed, so for long calculations it might take some time until the partial prompt appears.) Whenever you see the partial prompt and you cannot decide what GAP is still waiting for, then you have to type semicolons until the normal prompt returns. In every situation the exact meaning of the prompt gap> is that the program is waiting for a new problem.

But even if you mistyped the command more seriously, you do not have to type it all again. Suppose you mistyped or forgot the last closing parenthesis. Then your command is syntactically incorrect and GAP will notice it, incapable of computing the desired result.

Example

```
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 GAP is right by claiming that there should be a closing parenthesis before the semicolon. Now you can type CTRL-P to recover the last line of input. It will be written after the prompt with the cursor in the first position. Type CTRL-E to take the cursor to the end of the line, then CTRL-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 GAP by hitting the RETURN 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 GAP for evaluation by pressing RETURN regardless of the position of the cursor in that line. We will no longer mention the RETURN key from now on.

Sometimes a syntax error will cause GAP to enter a *break loop*. This is indicated by the special prompt `brk>`. If another syntax error occurs while GAP is in a break loop, the prompt will change to `brk_02>`, `brk_03>` and so on. You can leave the current break loop and exit to the next outer one by either typing `quit;` or by hitting CTRL-D. Eventually GAP will return to its normal state and show its normal prompt `gap>` again.

2.4 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 GAP, 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 GAP. Of course integers may be divided by each other, possibly resulting in noninteger rational values.

Example

```
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.

The next self-explanatory example demonstrates negative numbers.

Example

```
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 GAP as it can handle numbers of (almost) any size.

Example

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

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

Example

```
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. The whitespace around operators that do not consist of letters, e.g., the operators `*` and `-`, is not necessary.

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

Example

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

Besides these arithmetical operators there are comparison operators in GAP. A comparison results in a *boolean value* which is another kind of constant. The comparison operators `=`, `<>`, `<`, `<=`, `>` and `>=`, test for equality, inequality, less than, less than or equal, greater than and greater than or equal, respectively.

Example

```
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.

Example

```
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 GAP are *permutations*. They are written in cycle notation and they can be multiplied.

Example

```
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.

Example

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

The various other constants that GAP can deal with will be introduced when they are used, for example there are elements of finite fields such as `Z(8)`, and complex roots of unity such as `E(4)`.

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

Example

```
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. A comparison results in a boolean value. Boolean values are combined via logical operators. Moreover you have seen that GAP handles numbers of arbitrary size. Numbers and boolean values are constants. There are other types of constants in GAP like permutations. You are now in a position to use GAP as a simple desktop calculator.

2.5 Variables versus Objects

The constants described in the last section are specified by certain combinations of digits and minus signs (in the case of integers) or digits, commas and parentheses (in the case of permutations). These sequences of characters always have the same meaning to GAP. On the other hand, there are *variables*, specified by a sequence of letters and digits (including at least one letter), and their meaning depends on what has been assigned to them. An *assignment* is done by a GAP command *sequence_of_letters_and_digits* := *meaning*, where the sequence on the left hand side is called the *identifier* of the variable and it serves as its name. The meaning on the right hand side can be a constant like an integer or a permutation, but it can also be almost any other GAP object. From now on, we will use the term *object* to denote something that can be assigned to a variable.

There must be no whitespace between the : and the = in the assignment operator. Also do not confuse the assignment operator with the single equality sign = which in GAP is only used for the test of equality.

Example

```
gap> a:= (9 - 7) * (5 + 6);
22
gap> a;
22
gap> a * (a + 1);
506
gap> a = 10;
false
gap> a:= 10;
10
gap> a * (a + 1);
110
```

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

Example

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

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

This is in fact the whole secret of an assignment. An identifier is bound to an object and from this moment points to that object. 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 an object, which has been given its place in memory by the GAP storage manager. This place may change during a GAP session, but that doesn't bother the identifier. *The identifier points to the object, 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 object may in the same session point to any other object 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 GAP 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 objects they point to. 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 GAP. Many of them you will find in the remaining chapters of this manual, since functions are also referred to via identifiers.

You can get an overview of *all* GAP variables by entering `NamesGVars()`. Many of these are predefined. If you are interested in the variables you have defined yourself in the current GAP session, you can enter `NamesUserGVars()`.

Example

```
gap> NamesUserGVars();
[ "a", "w" ]
```

This seems to be the right place to state the following rule: The name of every global variable in the GAP library starts with a *capital letter*. Thus if you choose only names starting with a small letter for your own variables you will not attempt to overwrite any predefined variable. (Note that most of the predefined variables are read-only, and trying to change their values will result in an error message.)

There are some further interesting variables one of which will be introduced now.

Whenever GAP returns an object by printing it on the next line this object is assigned to the variable `last`. So if you computed

Example

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

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

Example

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

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

In this section you have seen how to assign objects to variables. These objects 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 objects. And you have learned that a double semicolon prevents the result of a statement from being printed.

2.6 Objects vs. Elements

In the last section we mentioned that every object is given a certain place in memory by the GAP storage manager (although that place may change in the course of a GAP session). In this sense, objects at different places in memory are never equal, and if the object pointed to by the variable `a` (to be more precise, the variable with identifier `a`) is equal to the object pointed to by the variable `b`, then we should better say that they are not only equal but *identical*. GAP provides the function `IsIdenticalObj` (**Reference: IsIdenticalObj**) to test whether this is the case.

Example

```
gap> a:= (1,2);; IsIdenticalObj( a, a );
true
gap> b:= (1,2);; IsIdenticalObj( a, b );
false
gap> b:= a;; IsIdenticalObj( a, b );
true
```

As the above example indicates, GAP objects `a` and `b` can be unequal although they are equal from a mathematical point of view, i.e., although we should have `a = b`. It may be that the objects `a` and `b` are stored in different places in memory, or it may be that we have an equivalence relation defined on the set of objects under which `a` and `b` belong to the same equivalence class. For example, if `a` $= x^3$ and `b` $= x^{-5}$ are words in the finitely presented group $\langle x \mid x^2 = 1 \rangle$, we would have `a = b` in that group.

GAP uses the equality operator `=` to denote such a mathematical equality, *not* the identity of objects. Hence we often have `a = b` although `IsIdenticalObj( a, b ) = false`. The operator `=` defines an equivalence relation on the set of all GAP objects, and we call the corresponding equivalence classes *elements*. Phrasing it differently, the same element may be represented by various GAP objects.

Non-trivial examples of elements that are represented by different objects (objects that really look different, not ones that are merely stored in different memory places) will occur only when we will be considering composite objects such as lists or domains.

2.7 About Functions

A program written in the GAP language is called a *function*. Functions are special GAP 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` (**Reference: Factorial**), for example, can be applied to an integer and will return the factorial of this integer.

Example

```
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` (**Reference: Gcd**) which computes the greatest common divisor of two integers.

Example

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

There are other functions that do not return an object but only produce a side effect, for example changing one of their arguments. These functions are sometimes called procedures. The function `Print` (**Reference: Print**) is only called for the side effect of printing something on the screen.

Example

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

In order to be able to compose arbitrary text with `Print` (**Reference: 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 an object such as the function `Sortex` (**Reference: 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 yourself is 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.

Example

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

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

Example

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

More complicated functions, especially functions with more than one argument cannot be defined in this way. You will see how to write your own GAP functions in Section 4.1.

In this section you have seen GAP 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 GAP with the maps-to operator.

2.8 Help

The content of the GAP manuals is also available as on-line help. A GAP session loads a long list of index entries. This typically contains all chapter and section headers, all names of documented functions, operations and so on, as well as some explicit index entries defined in the manuals.

The format of a query is as follows.

`?[book:][?]topic`

A simple example would be to type `?help` at the GAP prompt. If there is a single section with index entry *topic* then this is displayed directly.

If there are several matches you get an overview like in the example below.

Example

```
gap> ?sets
Help: several entries match this topic - type ?2 to get match [2]

[1] Tutorial: Sets
[2] Reference: Sets
[3] Reference: sets
[4] Reference: Sets of Subgroups
[5] Reference: setstabilizer
```

GAP's manuals consist of several *books*, which are indicated before the colon in the list above. A help query can be restricted to one book by using the optional *book*: part. For example `?tut : sets` will display the first of these help sections. More precisely, the parts of the string *book* which are separated by white space are interpreted as beginnings of the first words in the name of the book. Try `?books` to see the list of available books and their names.

The search for a matching *topic* (and optional *book*) is done *case insensitively*. If there is another `?` before the *topic*, then a *substring search* for *topic* is performed on all index entries. Otherwise the parts of *topic* which are separated by white space are considered as *beginnings of the first words* in an index entry.

White space is normalized in the search string (and the index entries).

Examples. All the following queries lead to the chapter of the reference manual which explains the use of GAP's help system in more detail.

Example

```
gap> ?Reference: The Help System
gap> ?  REF : t h s
gap> ?ref:?  help   system
```

The query `??sets` shows all help sections in all books whose index entries contain the substring `sets`.

As mentioned in the example above a complete list of commands for the help system is available in Section `?Ref: The Help System` of the reference manual. In particular there are commands to browse through the help sections, see `?Ref: Browsing through the Sections` and there is a way to influence the way *how* the help sections are displayed, see `?Ref: SetHelpViewer`. For example you can use an external pager program, a Web browser, `dvi`-previewer and/or `pdf`-viewer for reading GAP's online help.

2.9 Further Information introducing the System

For large amounts of input data, it might be advisable to write your input first into a file, and then read this into GAP; see `Read` (**Reference: Read**), `Edit` (**Reference: Edit**) for this.

The definition of the GAP syntax can be looked up in Chapter (**Reference: The Programming Language**). A complete list of command line editing facilities is found in Section (**Reference: Line Editing**). The break loop is described in Section (**Reference: Break Loops**).

Operators are explained in more detail in Sections (**Reference: Expressions**) and (**Reference: Comparisons**). You will find more information about boolean values in Chapters (**Reference: Booleans**) and (**Reference: Boolean Lists**). Permutations are described in Chapter (**Reference: Permutations**) and characters in Chapter (**Reference: Strings and Characters**).

Variables and assignments are described in more detail in (**Reference: Variables**) and (**Reference: Assignments**). A complete list of keywords is contained in (**Reference: Keywords**).

More about functions can be found in (**Reference: Function Calls**) and (**Reference: Procedure Calls**).

Chapter 3

Lists and Records

Modern mathematics, especially algebra, is based on set theory. When sets are represented in a computer, they inadvertently turn into lists. That's why we start our survey of the various objects GAP can handle with a description of lists and their manipulation. GAP regards sets as a special kind of lists, namely as lists without holes or duplicates whose entries are ordered with respect to the precedence relation <.

After the introduction of the basic manipulations with lists in 3.1, some difficulties concerning identity and mutability of lists are discussed in 3.2 and 3.3. Sets, ranges, row vectors, and matrices are introduced as special kinds of lists in 3.4, 3.5, 3.8. Handy list operations are shown in 3.7. Finally we explain how to use records in 3.9.

3.1 Plain 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 ten prime numbers.

Example

```
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.

Example

```
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.

Example

```
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

Example

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

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 one 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`.

Example

```
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`. The next position after the end of a list is not the only position capable of taking a new value. If you know that 71 is the 20th prime, you can 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 now has length 20.

Example

```
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 `[ ]`.

Example

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


gap> lll:= []; 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` (**Reference: 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`. If the element is not contained in the list then `Position` (**Reference: Position**) will return the special object `fail`.

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

In all of the above changes to the list `primes`, the list has been automatically resized. There is no need for you to tell GAP 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.

Example
```
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.

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

A list may even be part of itself.

Example
```
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. 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 special type of list, namely a dense list of *characters*, where *dense* means that the list has no holes. Here, *characters* are special GAP objects representing an element of the character set of the operating system. The input of printable characters is by enclosing them in single quotes `'`. A string literal can either be entered as the list of characters or by writing the characters between doublequotes `"`. Strings are handled specially by `Print` (**Reference: Print**). You can learn much more about strings in the reference manual.

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

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

Example
```
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 whose i-th entry is that element of the original list whose position is the i-th entry of the argument in the curly braces.

3.2 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 GAP. This section applies to all GAP objects that have subobjects, e.g., to lists and to records. After reading the section 3.9 about records 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.

Example

```
gap> numbers := primes;; 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`.

Example

```
gap> numbers[3]:= 4;; 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 also *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.

From this we see that *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 `ShallowCopy` which takes an object as its argument and returns a copy of the argument. (We will first restore the old value of `primes`.)

Example

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

Lists and records can be changed this way because GAP objects of these types have subobjects. To clarify this statement consider the following assignments.

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.

Example

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

Now return to Section 3.1 and find out whether the functions `Add` (**Reference: Add**) and `Append` (**Reference: Append**) change their arguments.

3.3 Immutability

GAP has a mechanism that protects lists against changes like the ones that have bothered us in Section 3.2. The function `Immutable` (**Reference: Immutable**) takes as argument a list and returns an immutable copy of it, i.e., a list which looks exactly like the old one, but has two extra properties: (1) The new list is immutable, i.e., the list itself and its subobjects cannot be changed. (2) In constructing the copy, every part of the list that can be changed has been copied, so that changes to the old list will not affect the new one. In other words, the new list has no mutable subobjects in common with the old list.

Example

```
gap> list := [ 1, 2, "three", [ 4 ] ];; copy := Immutable( list );;
gap> list[3][5] := 'w';; list; copy;
[ 1, 2, "threw", [ 4 ] ]
[ 1, 2, "three", [ 4 ] ]
gap> copy[3][5] := 'w';
List Assignment: <list> must be a mutable list
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' and ignore the assignment to continue
brk> quit;
```

As a consequence of these rules, in the immutable copy of a list which contains an already immutable list as subobject, this immutable subobject need not be copied, because it is unchangeable. Immutable lists are useful in many complex GAP objects, for example as generator lists of groups. By making them immutable, GAP ensures that no generators can be added to the list, removed or exchanged. Such changes would of course lead to serious inconsistencies with other knowledge that may already have been calculated for the group.

A converse function to `Immutable` (**Reference: Immutable**) is `ShallowCopy` (**Reference: ShallowCopy**), which produces a new mutable list whose i-th entry is the i-th entry of the old list. The

single entries are not copied, they are just placed in the new list. If the old list is immutable, and hence the list entries are immutable themselves, the result of `ShallowCopy` (**Reference: ShallowCopy**) is mutable only on the top level.

It should be noted that also other objects than lists can appear in mutable or immutable form. Records (see Section 3.9) provide another example.

3.4 Sets

GAP knows several special kinds of lists. A *set* in GAP is a list that contains no holes (such a list is called *dense*) and whose elements are strictly sorted w.r.t. <; in particular, a set cannot contain duplicates. (More precisely, the elements of a set in GAP are required to lie in the same *family*, but roughly this means that they can be compared using the < operator.)

This provides a natural model for mathematical sets whose elements are given by an explicit enumeration.

GAP also calls a set a *strictly sorted list*, and the function `IsSSortedList` (**Reference: IsSSortedList**) tests whether a given list is a set. It returns a boolean value. For almost any list whose elements are contained in the same family, there exists a corresponding set. This set is constructed by the function `Set` (**Reference: 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.

Example

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

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

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

Example

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

The operator `in` can also 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` (**Reference: 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.

Example

```
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" ]
```

Note that inserting new elements into a set with `AddSet` (**Reference: AddSet**) is usually more expensive than simply adding new elements at the end of a list.

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

Example

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

The arguments of the functions `Intersection` (**Reference: Intersection**) and `Union` (**Reference: Union**) could be ordinary lists, while their result is always a set. Note that in the preceding example at least one argument of `Intersection` (**Reference: Intersection**) was not a set. The functions `IntersectSet` (**Reference: IntersectSet**) and `UniteSet` (**Reference: 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.

3.5 Ranges

A *range* is a finite arithmetic progression of integers. This is another special kind of list. A range is described by the first two values and the last value of the arithmetic progression which are given in the form `[`*first*`,`*second*`..`*last*`]`. In the usual case of an ascending list of consecutive integers the second entry may be omitted.

Example

```
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` (**Reference: IsRange**) tests whether an object is a range, the function `ConvertToRangeRep` (**Reference: ConvertToRangeRep**) changes the representation of a list that is in fact a range to this compact internal representation.

Example

```
gap> a:= [-2,-1,0,1,2,3,4,5];
[ -2, -1, 0, 1, 2, 3, 4, 5 ]
```

```
gap> IsRange( a );
true
gap> ConvertToRangeRep( a );;  a;
[ -2 .. 5 ]
gap> a[1]:= 0;; IsRange( a );
false
```

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.

3.6 For and While 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.

Example

```
gap> pp:= [ (1,3,2,6,8)(4,5,9), (1,6)(2,7,8), (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;
(1,8,4,2,3,6,5,9)
```

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 the for loop has the form

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

In GAP this is merely a special case of the general for loop as defined above where the *list* in the loop body is a range (see 3.5):

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.

Example

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

```
gap> ff;
1307674368000
```

The `while` loop 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.

Example

```
gap> n:= 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 of course also holds 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, and the `if` statement covered in 4.2.

Example

```
gap> primes:= [];;
gap> numbers:= [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.

More about `for` and `while` loops can be found in the sections (**Reference: While**) and (**Reference: For**).

3.7 List Operations

There is a more comfortable way than that given in the previous section to compute the product of a list of numbers or permutations.

Example

```
gap> Product([1..15]);
1307674368000
gap> Product(pp);
(1,8,4,2,3,6,5,9)
```

The function `Product` (**Reference: 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` (**Reference: Product**) executes a loop over all elements of the list.

There are other often used loops available as functions. Guess what the function `Sum` (**Reference: Sum**) does. The function `List` (**Reference: list and non-list difference**) 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 Section 4.

Example

```
gap> cubed:= x -> x^3;;
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` (**Reference: Sum**) to the last list. But we may as well give the function `cubed` to `Sum` (**Reference: Sum**) as an additional argument.

Example

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

The primes less than 30 can be retrieved out of the list `primes` from Section 3.1 by the function `Filtered` (**Reference: 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 represented by the function `x -> x < 30` since `x < 30` will evaluate to `true` for values `x` less than 30 and to `false` otherwise.

Example

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

We have already mentioned 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.

Example

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

Finally we mention the function `ForAll` (**Reference: ForAll**) that checks whether a property holds for all elements of a list. It takes as its arguments a list and a function that returns a boolean value. `ForAll` (**Reference: ForAll**) checks whether the function returns `true` for all elements of the list.

Example

```
gap> list:= [ 1, 2, 3, 4 ];;
gap> ForAll( list, x -> x > 0 );
true
gap> ForAll( list, x -> x in primes );
false
```

You will find more predefined `for` loops in chapter (**Reference: Lists**).

3.8 Vectors and Matrices

This section describes how GAP uses lists to represent row vectors and matrices. A *row vector* is a dense list of elements from a common field. A *matrix* is a dense list of row vectors over a common field and of equal length.

Example

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

Row vectors may be added and multiplied by scalars from their field. Multiplication of row vectors of equal length results in their scalar product.

Example

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

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.

Such arithmetical operations (if the results are again vectors) result in *mutable* vectors except if the operation is binary and both operands are immutable; thus the vectors shown in the examples above are all mutable.

So if you want to produce a mutable list with 100 entries equal to 25, you can simply say `25 + 0 * [ 1 .. 100 ]`. Note that ranges are also vectors (over the rationals), and that `[ 1 .. 100 ]` is mutable.

A matrix is a dense list of row vectors of equal length.

Example

```
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, row vectors and other matrices. (If the row vectors and matrices involved in such a multiplication do not have suitable dimensions then the "missing" entries are treated as zeros, so the results may look unexpectedly in such cases.)

Example

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

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

A vector or matrix of integers can also be multiplied with a finite field scalar and vice versa. Such products result in a matrix over the finite field with the integers mapped into the finite field in the obvious way. Finite field matrices are nicer to read when they are `Display`ed rather than `Print`ed. (Here we write `Z(q)` to denote a primitive root of the finite field with `q` elements.)

Example

```
gap> Display( m * One( GF(5) ) );
 1 4 1
 2 . 4
 1 1 1
gap> Display( m^2 * Z(2) + m * Z(4) );
z = Z(4)
 z^1 z^1 z^2
   1   1 z^2
 z^1 z^1 z^2
```

Submatrices can easily be extracted using the expression *mat*`{`*rows*`}{`*columns*`}`. They can also be assigned to, provided the big matrix is mutable (which it is not if it is the result of an arithmetical operation, see above).

Example

```
gap> sm := m{ [ 1, 2 ] }{ [ 2, 3 ] };
[ [ -1, 1 ], [ 0, -1 ] ]
gap> sm{ [ 1, 2 ] }{ [2] } := [[-2],[0]];;  sm;
[ [ -1, -2 ], [ 0, 0 ] ]
```

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

Matrices appear not only in linear algebra, but also as group elements, provided they are invertible. Here we have the opportunity to meet a group-theoretical function, namely `Order` (**Reference: Order**), which computes the order of a group element.

Example

```
gap> Order( m * One( GF(5) ) );
8
gap> Order( m );
infinity
```

For matrices whose entries are more complex objects, for example rational functions, GAP's `Order` (**Reference: Order**) methods might not be able to prove that the matrix has infinite order, and one gets the following warning.

Example

```
#I  Order: warning, order of <mat> might be infinite
```

In such a case, if the order of the matrix really is infinite, you will have to interrupt GAP by pressing *ctl*-C (followed by *ctl*-D or `quit;` to leave the break loop).

To prove that the order of `m` is infinite, we also could look at the minimal polynomial of `m` over the rationals.

Example

```
gap> f:= MinimalPolynomial( Rationals, m );;  Factors( f );
[ x_1-2, x_1^2+3 ]
```

`Factors` (**Reference: Factors**) returns a list of irreducible factors of the polynomial `f`. The first irreducible factor $X-2$ reveals that 2 is an eigenvalue of `m`, hence its order cannot be finite.

3.9 Plain Records

A record provides another way to build new data structures. Like a list a record contains subobjects. In a record the elements, the so-called *record components*, are not indexed by numbers but by names.

In this section you will see how to define and how to use records. Records are changed by assignments to record components or by unbinding record components.

Initially a record is defined as a comma separated list of assignments to its record components.

Example

```
gap> date:= rec(year:= 1997,
>               month:= "Jul",
>               day:= 14);
rec( day := 14, month := "Jul", year := 1997 )
```

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.

Example

```
gap> date.year;
1997
```

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

Example

```
gap> date.time:= rec(hour:= 19, minute:= 23, second:= 12);
rec( hour := 19, minute := 23, second := 12 )
gap> date;
rec( day := 14, month := "Jul",
  time := rec( hour := 19, minute := 23, second := 12 ), year := 1997 )
```

Records are objects that may be changed. An assignment to a record component changes the original object. The remarks made in Sections 3.2 and 3.3 about identity and mutability of lists are also true for records.

Sometimes it is interesting to know which components of a certain record are bound. This information is available from the function `RecNames` (**Reference: RecNames**), which takes a record as its argument and returns a list of names of all bound components of this record as a list of strings.

Example

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

Now return to Sections 3.2 and 3.3 and find out what these sections mean for records.

3.10 Further Information about Lists

(The following cross-references point to the GAP Reference Manual.)

You will find more about lists, sets, and ranges in Chapter (**Reference: Lists**), in particular more about identical lists in Section (**Reference: Identical Lists**). A more detailed description of strings is contained in Chapter (**Reference: Strings and Characters**). Fields are described in Chapter (**Reference: Fields and Division Rings**), some known fields in GAP are described in Chapters (**Reference: Rational Numbers**), (**Reference: Abelian Number Fields**), and (**Reference: Finite Fields**). Row vectors and matrices are described in more detail in Chapters (**Reference: Row Vectors**) and (**Reference: Matrices**); note that GAP supports also linear algebra for objects which are *not* lists, see Chapter (**Reference: Vector and Matrix Objects**). Vector spaces are described in Chapter (**Reference: Vector Spaces**), further matrix related structures are described in Chapters (**Reference: Matrix Groups**), (**Reference: Algebras**), and (**Reference: Lie Algebras**).

You will find more list operations in Chapter (**Reference: Lists**).

Records and functions for records are described in detail in Chapter (**Reference: Records**).

Chapter 4

Functions

You have already seen how to use functions in the GAP library, i.e., how to apply them to arguments.

In this section you will see how to write functions in the GAP language. You will also see how to use the `if` statement and declare local variables with the `local` statement in the function definition. Loop constructions via `while` and `for` are discussed further, as are recursive functions.

4.1 Writing Functions

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

Example

```
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 GAP 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 GAP function is an expression like an integer, a sum or a list. Therefore it 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` (**Reference: Print**).

Example

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

```
    return;
end
```

Note the additional newline character `"\n"` in the `Print` (**Reference: Print**) statement. It is printed after the object `sayhello` to start a new line. The extra `return` statement is inserted by GAP to simplify the process of executing the function.

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

Example

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

However, this is not a typical example as no value is returned but only a string is printed.

4.2 If Statements

In the following example we define a function `sign` which determines the sign of an integer.

Example

```
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
```

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 $f(n) = f(n-1) + f(n-2)$ for $n \geq 3$. Since functions in GAP may call themselves, a function `fib` that computes Fibonacci numbers can be implemented basically by typing the above equations. (Note however that this is a very inefficient way to compute $f(n)$.)

Example

```
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 inserting the necessary tests into this example.

4.3 Local Variables

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.

Example

```
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.

Example

```
gap> c:= 7;;
gap> gcd(30, 63);
3
gap> c;
7
```

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.

4.4 Recursion

We have already seen recursion in the function `fib` in Section 4.2. Here is another, slightly more complicated example.

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. Note that there is just one partition of 0, namely $[]$. 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 or equal to 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.

Example

```
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 indeed takes 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 GAP. 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` was to 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.

Note that the formal parameters `n` and `m` of `np` are treated like local variables.

(Regardless of the recursive structure of an algorithm it is often cheaper (in terms of computing time) to avoid a recursive implementation if possible (and it is possible in this case), because a function call is not very cheap.)

4.5 Further Information about Functions

The function syntax is described in Section (**Reference: Functions**). The `if` statement is described in more detail in Section (**Reference: If**). More about Fibonacci numbers is found in Section `Fibonacci` (**Reference: Fibonacci**) and more about partitions in Section `Partitions` (**Reference: Partitions**).

Chapter 5

Groups and Homomorphisms

In this chapter we will show some computations with groups. The examples deal mostly with permutation groups, because they are the easiest to input. The functions mentioned here, like `Group` (**Reference: Group**), `Size` (**Reference: Size**) or `SylowSubgroup` (**Reference: SylowSubgroup**), however, are the same for all kinds of groups, although the algorithms which compute the information of course will be different in most cases.

5.1 Permutation groups

Permutation groups are so easy to input because their elements, i.e., permutations, are so easy to type: they are entered and displayed in disjoint cycle notation. So let's construct a permutation group:

Example

```
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 to be the symmetric group S_8 on eight points, and assigned it to the identifier `s8`. Now S_8 contains the alternating group on eight points which can be described in several ways, e.g., as the group of all even permutations in `s8`, or as its derived subgroup. Once we ask GAP to verify that the group is an alternating group acting in its natural permutation representation, the system will display the group accordingly.

Example

```
gap> a8 := DerivedSubgroup( s8 );
Group([ (1,2,3), (2,4,3), (2,4,5), (2,5,6,3,4), (3,7,4), (2,6)
(4,7,8,5) ])
gap> Size( a8 ); IsAbelian( a8 ); IsPerfect( a8 );
20160
false
true
gap> IsNaturalAlternatingGroup(a8);
true
gap> a8;
Alt( [ 1 .. 8 ] )
```

Once information about a group like `s8` or `a8` has been computed, it is stored in the group so that it can simply be looked up when it is required again. This holds for all pieces of information

in the previous example. Namely, `a8` stores its order and that it is nonabelian and perfect, and `s8` stores its derived subgroup `a8`. Had we computed `a8` as `CommutatorSubgroup( s8, s8 )`, however, it would not have been stored, because it would then have been computed as a function of *two* arguments, and hence one could not attribute it to just one of them. (Of course the function `CommutatorSubgroup` (**Reference: CommutatorSubgroup**) can compute the commutator subgroup of *two* arbitrary subgroups.) The situation is a bit different for Sylow p-subgroups: The function `SylowSubgroup` (**Reference: SylowSubgroup**) also requires two arguments, namely a group and a prime p, but the result is stored in the group –namely together with the prime p in a list that can be accessed with `ComputedSylowSubgroups`, but we won't dwell on the details here.

Example

```
gap> syl2 := SylowSubgroup( a8, 2 );; Size( syl2 );
64
gap> Normalizer( a8, syl2 ) = syl2;
true
gap> cent := Centralizer( a8, Centre( syl2 ) );; Size( cent );
192
gap> DerivedSeries( cent );; List( last, Size );
[ 192, 96, 32, 2, 1 ]
```

We have typed double semicolons after some commands to avoid the output of the groups (which would be printed by their generator lists). Nevertheless, the beginner is encouraged to type a single semicolon instead and study the full output. This remark also applies for the rest of this tutorial.

With the next examples, we want to calculate a subgroup of `a8`, then its normalizer and finally determine the structure of the extension. We begin by forming a subgroup generated by three commuting involutions, i.e., a subgroup isomorphic to the additive group of the vector space 2^3.

Example

```
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
```

As usual, GAP prints the group by giving all its generators. This can be annoying, especially if there are many of them or if they are of huge degree. It also makes it difficult to recognize a particular group when there are already several around. Note that although it is no problem for *us* to specify a particular group to GAP, by using well-chosen identifiers such as `a8` and `elab`, it is impossible for GAP to use these identifiers when printing a group for us, because the group does not know which identifier(s) point to it, in fact there can be several. In order to give a name to the group itself (rather than to the identifier), you can use the function `SetName` (**Reference: Name**). We do this with the name `2^3` here which reflects the mathematical properties of the group. From now on, GAP will use this name when printing the group for us, but we still cannot use this name to specify the group to GAP, because the name does not know to which group it was assigned (after all, you could assign the same name to several groups). When talking to the computer, you must always use identifiers.

Example

```
gap> SetName( elab, "<group of type 2^3>" ); elab;
<group of type 2^3>
gap> norm := Normalizer( a8, elab );; Size( norm );
1344
```

Now that we have the subgroup `norm` of order 1344 and its subgroup `elab`, we want to look at its factor group. But since we also want to find preimages of factor group elements in `norm`, we really want to look at the *natural homomorphism* defined on `norm` with kernel `elab` and whose image is the factor group.

Example

```
gap> hom := NaturalHomomorphismByNormalSubgroup( norm, elab );
[ (2,3)(6,7), (3,4)(7,8), (3,5)(4,6), (5,7)(6,8), (5,6)(7,8),
  (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8), (1,6)(2,5)(3,8)(4,7)
 ] -> [ (1,2)(5,6), (2,3)(6,7), (2,4)(3,5), (4,6)(5,7), (4,5)(6,7),
  (), (), () ]
gap> f := Image( hom );
Group([ (1,2)(5,6), (2,3)(6,7), (2,4)(3,5), (4,6)(5,7), (4,5)(6,7),
  (), (), () ])
gap> Size( f );
168
```

The factor group is again represented as a permutation group (its last three generators are trivial, meaning that the last three generators of the preimage are in the kernel of `hom`). However, the action domain of this factor group has nothing to do with the action domain of `norm`. (It only happens that both are subsets of the natural numbers.) We can now form images and preimages under the natural homomorphism. The set of preimages of an element under `hom` is a coset modulo `elab`. We use the function `PreImages` (**Reference: PreImages**) here because `hom` is not a bijection, so an element of the range can have several preimages or none at all.

Example

```
gap> ker:= Kernel( hom );
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> x := (1,8,3,5,7,6,2);; Image( hom, x );
(1,7,5,6,2,3,4)
gap> coset := PreImages( hom, last );
RightCoset(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) ]),(2,8,6,7,3,4,5))
```

Note that GAP is free to choose any representative for the coset of preimages. Of course the quotient of two representatives lies in the kernel of the homomorphism.

Example

```
gap> rep:= Representative( coset );
(2,8,6,7,3,4,5)
gap> x * rep^-1 in ker;
true
```

The factor group `f` is a simple group, i.e., it is a non-trivial group whose only normal subgroups are its trivial subgroup and itself. GAP can detect this fact, and it can then also find the name by which this simple group is known among group theorists. (Such names are of course not available for non-simple groups.)

Example

```
gap> IsSimple( f ); IsomorphismTypeInfoFiniteSimpleGroup( f );
true
```

```
rec(
  name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,\
7) = U(2,7) ~ A(2,2) = L(3,2)", parameter := [ 2, 7 ], series := "L",
  shortname := "L3(2)" )
gap> SetName( f, "L_3(2)" );
```

We give `f` the name `L_3(2)` because the last part of the name string reveals that it is isomorphic to the simple linear group $L_3(2)$. This group, however, also has a lot of other names. Names that are connected with a = sign are different names for the same matrix group, e.g., `A(2,2)` is the Lie type notation for the classical notation `L(3,2)`. Other pairs of names are connected via `~`, these then specify other classical groups that are isomorphic to that linear group (e.g., the symplectic group `S(2,7)`, whose Lie type notation would be `C(1,7)`).

The group `norm` acts on the eight elements of its normal subgroup `elab` by conjugation, yielding a representation of $L_3(2)$ in `s8` which leaves one point fixed (namely point `1`). The image of this representation can be computed with the function `Action` (**Reference: Action homomorphisms**); it is even contained in the group `norm` and we can show that `norm` is indeed a split extension of the elementary abelian group 2^3 with this image of $L_3(2)$.

Example

```
gap> op := Action( norm, elab );
Group([ (), (), (), (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6),
  (2,3)(6,7) ])
gap> IsSubgroup( a8, op ); IsSubgroup( norm, op );
true
true
gap> IsTrivial( Intersection( elab, op ) );
true
gap> SetName( norm, "2^3:L_3(2)" );
```

By the way, you should not try the operator < instead of the function `IsSubgroup` (**Reference: IsSubgroup**). Something like

Example

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

will not cause an error, but the result does not signify anything about the inclusion of one group in another; < tests which of the two groups is less in some total order. On the other hand, the equality operator = in fact does test the equality of its arguments.

Summary. In this section we have used the elementary group functions to determine the structure of a normalizer. We have assigned names to the involved groups which reflect their mathematical structure and GAP uses these names when printing the groups.

5.2 Actions of Groups

In order to get another representation of `a8`, we consider another action, namely that on the elements of a certain conjugacy class by conjugation.

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

Note the difference between `Order` (**Reference: Order**) (which means the element order), `Size` (**Reference: Size**) (which means the size of the conjugacy class) and `Length` (**Reference: Length**) (which means the length of a list). We choose to let `a8` operate on the class of length 112.

Example
```
gap> class := First( ccl, c -> Size(c) = 112 );;
gap> op := Action( a8, AsList( class ),OnPoints );;
```

We use `AsList` (**Reference: AsList**) here to convert the conjugacy class into a list of its elements whereas we wrote `Action( norm, elab )` directly in the previous section. The reason is that the elementary abelian group `elab` can be quickly enumerated by GAP whereas the standard enumeration method for conjugacy classes is slower than just explicit calculation of the elements. However, GAP is reluctant to construct explicit element lists, because for really large groups this direct method is infeasible.

Note also the function `First` (**Reference: First**), used to find the first element in a list which passes some test.

In this example, we have specified the action function `OnPoints` (**Reference: OnPoints**) in this example, which is defined as `OnPoints( `d`, `g` ) = `d` ^ `g. This "caret" operator denotes conjugation in a group if both arguments d and g are group elements (contained in a common group), but it also denotes the natural action of permutations on positive integers (and exponentiation of integers as well, of course). It is in fact the default action and will be supplied by the system if not given. Another common action is for example `OnRight` (**Reference: OnRight**), which means right multiplication, defined as d `*` g. (Group actions in GAP are always from the right.)

We now have a permutation representation `op` on 112 points, which we test for primitivity. If it is not primitive, we can obtain a minimal block system (i.e., one where the blocks have minimal length) by the function `Blocks` (**Reference: Blocks**).

Example
```
gap> IsPrimitive( op, [ 1 .. 112 ] );
false
gap> blocks := Blocks( op, [ 1 .. 112 ] );;
```

Note that we must specify the domain of the action. You might think that the functions `IsPrimitive` (**Reference: IsPrimitive**) and `Blocks` (**Reference: Blocks**) could use `[ 1 .. 112 ]` as default domain if no domain was given. But this is not so easy, for example would the default domain of `Group( (2,3,4) )` be `[ 1 .. 4 ]` or `[ 2 .. 4 ]`? To avoid confusion, all action functions require that you specify the domain of action. If we had specified `[ 1 .. 113 ]` in the primitivity test above, point 113 would have been a fixed point (and the action would not even have been transitive).

Now `blocks` is a list of blocks (i.e., a list of lists), which we do not print here for the sake of saving paper (try it for yourself). In fact all we want to know is the size of the blocks, or rather how

many there are (the product of these two numbers must of course be 112). Then we can obtain a new permutation group of the corresponding degree by letting op act on these blocks setwise.

Example

```
gap> Length( blocks[1] );  Length( blocks );
2
56
gap> op2 := Action( op, blocks, OnSets );;
gap> IsPrimitive( op2, [ 1 .. 56 ] );
true
```

Note that we give a third argument (the action function `OnSets` (**Reference: OnSets**)) to indicate that the action is not the default action on points but an action on sets of elements given as strictly sorted lists. (Section (**Reference: Basic Actions**) lists all actions that are pre-defined by GAP.)

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 the associated action homomorphisms (which of course in this case are monomorphisms). We construct the composition of two homomorphisms with the * operator, reading left-to-right.

Example

```
gap> ophom := ActionHomomorphism( a8, op );;
gap> ophom2 := ActionHomomorphism( op, op2 );;
gap> composition := ophom * ophom2;;
gap> stab := Stabilizer( op2, 2 );;
gap> preim := PreImages( composition, stab );
Group([ (1,2,4), (6,7,8), (3,6,8), (5,8,6), (1,2)(3,8) ])
```

Alternatively, it is possible to create action homomorphisms immediately (without creating the action first) by giving the same set of arguments to `ActionHomomorphism` (**Reference: ActionHomomorphism**).

Example

```
gap> nophom := ActionHomomorphism( a8, AsList(class) );
<action homomorphism>
gap> IsSurjective(nophom);
false
gap> Image(nophom,(1,2,3));
(2,43,14)(3,44,20)(4,45,26)(5,46,32)(6,47,38)(8,13,48)(9,19,53)(10,25,
58)(11,31,63)(12,37,68)(15,49,73)(16,50,74)(17,51,75)(18,52,76)(21,54,
77)(22,55,78)(23,56,79)(24,57,80)(27,59,81)(28,60,82)(29,61,83)(30,62,
84)(33,64,85)(34,65,86)(35,66,87)(36,67,88)(39,69,89)(40,70,90)(41,71,
91)(42,72,92)
```

In this situation, however (for performance reasons, avoiding computation an image that might never be needed) the homomorphism is defined to be not into the *Image* of the action, but into the *full symmetric group*, i.e. it is not automatically surjective. Surjectivity can be enforced by giving the string `"surjective"` as an extra last argument. The `Image` of the action homomorphism of course is the same group in either case.

Example

```
gap> Size(Range(nophom));
19745068572210740235368203727599248834127786803497533779665629509490 28\
```

```
589697718114408942243550277793665979573382378536382723349196863856218 1\
1850780464277094400000000000000000000000000
gap> Size(Range(ophom));
20160
gap> nophom := ActionHomomorphism( a8, AsList(class),"surjective" );
<action epimorphism>
gap> Size(Range(nophom));
20160
```

Continuing the example, the normalizer of an element in the conjugacy class `class` is a group of order 360, too. In fact, it is a conjugate of the maximal subgroup we had found before, and a conjugating element in `a8` is found by the function `RepresentativeAction` (**Reference: RepresentativeAction**).

Example

```
gap> sgp := Normalizer( a8, Subgroup(a8,[Representative(class)]) );;
gap> Size( sgp );
360
gap> RepresentativeAction( a8, sgp, preim );
(2,4,3)
```

One of the most prominent actions of a group is on the cosets of a subgroup. Naïvely this can be done by constructing the cosets and acting on them by right multiplication.

Example

```
gap> cosets:=RightCosets(a8,norm);;
gap> op:=Action(a8,cosets,OnRight);
Group([ (1,2,3)(4,6,5)(7,8,9)(10,12,11)(13,14,15), (1,2,3)(4,13,9)
(5,7,11)(6,10,15)(8,14,12), (1,11,6)(2,15,4)(3,9,5)(7,13,10)
(8,12,14), (1,10,12,3,13)(2,7,15,14,5)(4,6,8,9,11), (1,12,15)(2,10,5)
(3,11,8)(4,9,13)(6,14,7), (1,4,13,10)(2,15,14,3)(5,12,11,6)(7,9) ])
gap> NrMovedPoints(op);
15
```

A problem with this approach is that creating (and storing) all cosets can be very memory intensive if the subgroup index gets large. Because of this, GAP provides special objects which act like a list of elements, but do not actually store elements but compute them on the go. Such a simulated list is called an *enumerator*. The easiest example of this concept is the `Enumerator` (**Reference: Enumerator**) of a group. While it behaves like a list of elements, it requires far less storage, and is applicable to potentially huge groups for which it would be completely infeasible to write down all elements:

Example

```
gap> enum:=Enumerator(SymmetricGroup(20));
<enumerator of perm group>
gap> Length(enum);
2432902008176640000
gap> enum[123456789012345];
(1,4,15,3,14,11,8,17,6,18,5,7,20,13,10,9,2,12)
gap> Position(enum,(1,2,3,4,5,6,7,8,9,10));
71948729603
```

For the action on cosets the object of interest is the `RightTransversal` (**Reference: RightTransversal**) of a subgroup. Again, it does not write out actual elements and thus can be created even for subgroups of large index.

Example

```
gap> t:=RightTransversal(a8,norm);
RightTransversal(Alt( [ 1 .. 8 ] ),2^3:L_3(2))
gap> t[7];
(4,6,5)
gap> Position(t,(4,6,7,8,5));
8
gap> Position(t,(1,2,3));
fail
```

For the action on cosets there is the added complication that not every group element is in the transversal (as the last example shows) but the action on cosets of a subgroup usually will not preserve a chosen set of coset representatives. Because of this issue, all action functionality actually uses `PositionCanonical` (**Reference: PositionCanonical**) instead of `Position` (**Reference: Position**). In general, for elements contained in a list, `PositionCanonical` (**Reference: PositionCanonical**) returns the same as `Position`. If the element is not contained in the list (and for special lists, such as transversals), `PositionCanonical` returns the list element representing the same objects, e.g. the transversal element representing the same coset.

Example

```
gap> PositionCanonical(t,(1,2,3));
2
gap> t[2];
(6,7,8)
gap> t[2]/(1,2,3);
(1,3,2)(6,7,8)
gap> last in norm;
true
```

Thus, acting on a `RightTransversal` with the `OnRight` action will in fact (in a slight abuse of definitions) produce the action of a group on cosets of a subgroup and is in general the most efficient way of creating this action.

Example

```
gap> Action(a8,RightTransversal(a8,norm),OnRight);
Group([ (1,2,3)(4,6,5)(7,8,9)(10,12,11)(13,14,15), (1,2,3)(4,13,9)
(5,7,11)(6,10,15)(8,14,12), (1,11,6)(2,15,4)(3,9,5)(7,13,10)
(8,12,14), (1,10,12,3,13)(2,7,15,14,5)(4,6,8,9,11), (1,12,15)(2,10,5)
(3,11,8)(4,9,13)(6,14,7), (1,4,13,10)(2,15,14,3)(5,12,11,6)(7,9) ])
```

Summary. In this section we have learned how groups can operate on GAP objects such as integers and group elements. We have used `ActionHomomorphism` (**Reference: ActionHomomorphism**), among others, to construct the corresponding actions and homomorphisms and have seen how transversals can be used to create the action on cosets of a subgroup.

5.3 Subgroups as Stabilizers

Action functions can also be used to construct subgroups. We will try to find several subgroups in `a8` as stabilizers of such actions. 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.

Example

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

This gives us a hint how to find further subgroups. Each subgroup is the stabilizer of a point of an appropriate transitive action (namely the action on the cosets of that subgroup or another action that is equivalent to this action). So the question is how to find other actions. The obvious thing is to operate on pairs of points. So using the function `Tuples` (**Reference: Tuples**) we first generate a list of all pairs.

Example

```
gap> pairs := Tuples( [1..8], 2 );;
```

Now we would like to have `a8` operate on this domain. But we cannot use the default action `OnPoints` (**Reference: OnPoints**) because powering a list by a permutation via the caret operator `^` is not defined. So we must tell the functions from the actions package how the group elements operate on the elements of the domain (here and below, the word "package" refers to the GAP functionality for group actions, not to a GAP package). In our example we can do this by simply passing `OnPairs` (**Reference: OnPairs**) as an optional last argument. All functions from the actions package accept such an optional argument that describes the action. One example is `IsTransitive` (**Reference: IsTransitive**).

Example

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

The action is of course not transitive, since the pairs `[ 1, 1 ]` and `[ 1, 2 ]` cannot lie in the same orbit. So we want to find out what the orbits are. The function `Orbits` (**Reference: Orbits**) does that for us. It returns a list of all the orbits. We look at the orbit lengths and representatives for the orbits.

Example

```
gap> orbs := Orbits( a8, pairs, OnPairs );; Length( orbs );
2
gap> List( orbs, Length );
[ 8, 56 ]
gap> List( orbs, o -> o[1] );
[ [ 1, 1 ], [ 1, 2 ] ]
```

The action of `a8` on the first orbit (this is the one containing `[1,1]`, try `[1,1] in orbs[1]`) is of course equivalent to the original action, so we ignore it and work with the second orbit.

Example

```
gap> u56 := Stabilizer( a8, orbs[2][1], OnPairs );; 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 `ActionHomomorphism` (**Reference: ActionHomomorphism**) does what we need. It creates a homomorphism defined on `a8` whose image is a new group that acts on `[ 1 .. 56 ]` in the same way that `a8` acts on the second orbit.

Example

```
gap> h56 := ActionHomomorphism( a8, orbs[2], OnPairs );;
gap> a8_56 := Image( h56 );;
```

We would now like to know if the subgroup `u56` of index 56 that we found is maximal or not. As we have used already in Section 5.2, a subgroup is maximal if and only if the action on the cosets of this subgroup is primitive.

Example

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

Remember that we can leave out the function if we mean `OnPoints` (**Reference: OnPoints**) but that we have to specify the action domain for all action functions.

We see that `a8_56` is not primitive. This means of course that the action of `a8` on `orb[2]` is not primitive, because those two actions are equivalent. So the stabilizer `u56` is not maximal. Let us try to find its supergroups. We use the function `Blocks` (**Reference: Blocks**) to find a block system. The (optional) third argument in the following example tells `Blocks` (**Reference: Blocks**) that we want a block system where 1 and 3 lie in one block.

Example

```
gap> blocks := Blocks( a8_56, [1..56], [1,3] );;
```

The result is a list of sets, such that `a8_56` acts on those sets. Now we would like the stabilizer of this action on the sets. Because we want to operate on the sets we have to pass `OnSets` (**Reference: OnSets**) as third argument.

Example

```
gap> u8_56 := Stabilizer( a8_56, blocks[1], OnSets );;
gap> Index( a8_56, u8_56 );
8
gap> u8b := PreImages( h56, u8_56 );; Index( a8, u8b );
8
gap> IsConjugate( a8, u8, u8b );
true
```

So we have found a supergroup of `u56` that is conjugate in `a8` to `u8`. This is not surprising, since `u8` is a point stabilizer, and `u56` is a two point stabilizer in the natural action of `a8` on eight points.

Here is a *warning*: If you specify `OnSets` (**Reference: OnSets**) as third argument to a function like `Stabilizer` (**Reference: Stabilizer**), you have to make sure that the point (i.e. the second argument) is indeed a set. Otherwise you will get a puzzling error message or even wrong results! In the above example, the second argument `blocks[1]` came from the function `Blocks` (**Reference: Blocks**), which returns a list of sets, so everything was OK.

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

Example
```
gap> seed:=First(AllBlocks(a8_56),x->Length(x)=2);;
gap> blocks := Blocks( a8_56, [1..56], seed);;
gap> u28_56 := Stabilizer( a8_56, seed, OnSets );;
gap> u28 := PreImages( h56, u28_56 );;
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 acts primitively on the 28 blocks.

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

Stabilizer (**Reference: 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.

Example
```
gap> u336 := Stabilizer( u56, 3 );;
gap> Index( a8, u336 );
336
```

Other functions are also applicable to subgroups. In the following we show that u336 acts regularly on the 60 triples of [4 .. 8] which contain no element twice. We construct the list of these 60 triples with the function Orbit (**Reference: Orbit**) (using OnTuples (**Reference: OnTuples**) as the natural generalization of OnPairs (**Reference: OnPairs**)) and then pass it as action domain to the function IsRegular (**Reference: IsRegular**). The positive result of the regularity test means that this action is equivalent to the actions of u336 on its 60 elements from the right.

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

Just as we did in the case of the action on the pairs above, we now construct a new permutation group that acts on [1 .. 336] in the same way that a8 acts on the cosets of u336. But this time we let a8 operate on a right transversal, just like norm did in the natural homomorphism above.

Example
```
gap> t := RightTransversal( a8, u336 );;
gap> a8_336 := Action( a8, t, OnRight );;
```

To find subgroups above u336 we again look for nontrivial block systems.

Example
```
gap> blocks := Blocks( a8_336, [1..336] );; blocks[1];
[ 1, 43, 85 ]
```

We see that the union of u336 with its 43rd and its 85th coset is a subgroup in a8_336, its index is 112. We can obtain it as the closure of u336 with a representative of the 43rd coset, which can be found as the 43rd element of the transversal t. Note that in the representation a8_336 on 336 points, this subgroup corresponds to the stabilizer of the block [1, 43, 85].

Example
```
gap> u112 := ClosureGroup( u336, t[43] );;
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 7 and 43 into the first block.

Example
```
gap> blocks := Blocks( a8_336, [1..336], [1,7,43] );;
gap> Length( blocks );
56
gap> u56b := ClosureGroup( u112, t[7] );; Index( a8, u56b );
56
gap> IsPrimitive( a8_336, blocks, OnSets );
true
```

We already mentioned in Section 5.2 that there is another standard action of permutations, namely the conjugation. E.g., since no other action is specified in the following example, OrbitLength (**Reference: OrbitLength**) simply acts via OnPoints (**Reference: OnPoints**), and because *perm_1* ^ *perm_2* is defined as the conjugation of *perm_2* on *perm_1*, in fact we compute the length of the conjugacy class of (1,2)(3,4)(5,6)(7,8).

Example
```
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) );; Index( a8, u105 );
105
```

Note that although the length of a conjugacy class of any element g in any finite group G can be computed as OrbitLength(G, g), the command Size(ConjugacyClass(G, g)) is probably more efficient.

Example
```
gap> Size( ConjugacyClass( a8, (1,2)(3,4)(5,6)(7,8) ) );
105
```

Of course the stabilizer u105 is in fact the centralizer of the element (1,2)(3,4)(5,6)(7,8). Stabilizer (**Reference: Stabilizer**) notices that and computes the stabilizer using the centralizer algorithm for permutation groups. In the usual way we now look for the subgroups above u105.

Example
```
gap> orb:=Set(orb);;
gap> blocks := Blocks( a8, orb );; Length( blocks );
15
gap> Set(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 above u105 that has index 15 in a8, we must form the closure of u105 with an element of the coset that corresponds to any other point in the first block. If we choose the point (1,3)(2,4)(5,7)(6,8), 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 function RepresentativeAction (**Reference: RepresentativeAction**) 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 action as an 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, RepresentativeAction (**Reference: RepresentativeAction**) returns fail.

Example

```
gap> rep := RepresentativeAction( a8, (1,2)(3,4)(5,6)(7,8),
>                                     (1,3)(2,4)(5,7)(6,8) );
(2,3)(6,7)
gap> u15 := ClosureGroup( u105, rep );; 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.

Example

```
gap> u15b := ClosureGroup( u105, (2,3)(6,8) );; Index( a8, u15b );
15
gap> RepresentativeAction( a8, u15, u15b );
fail
```

RepresentativeAction (**Reference: RepresentativeAction**) 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.

Summary. In this section we have demonstrated some functions from the actions 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 (**Reference: Cycle**) and Permutation (**Reference: Permutation**). These are fully described in Chapter (**Reference: Group Actions**).

5.4 Group Homomorphisms by Images

We have already seen examples of group homomorphisms in the last sections, namely natural homomorphisms and action homomorphisms. In this section we will show how to construct a group homomorphism $G \to H$ by specifying a generating set for G and the images of these generators in H. We use the function GroupHomomorphismByImages(*G*, *H*, *gens*, *imgs*) where *gens* is a generating set for *G* and *imgs* is a list whose ith entry is the image of *gens*$[i]$ under the homomorphism.

Example

```
gap> s4 := Group((1,2,3,4),(1,2));; s3 := Group((1,2,3),(1,2));;
gap> hom := GroupHomomorphismByImages( s4, s3,
>           GeneratorsOfGroup(s4), [(1,2),(2,3)] );
[ (1,2,3,4), (1,2) ] -> [ (1,2), (2,3) ]
```

```
gap> Kernel( hom );
Group([ (1,4)(2,3), (1,3)(2,4) ])
gap> Image( hom, (1,2,3) );
(1,2,3)
gap> Size( Image( hom, DerivedSubgroup(s4) ) );
3
```

Example

```
gap> PreImage( hom, (1,2,3) );
Error, <map> must be an inj. and surj. mapping called from
<function "PreImage">( <arguments> )
 called from read-eval loop at line 4 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Example

```
gap> PreImagesRepresentative( hom, (1,2,3) );
(1,4,2)
gap> PreImage( hom, TrivialSubgroup(s3) );  # the kernel
Group([ (1,4)(2,3), (1,3)(2,4) ])
```

This homomorphism from S_4 onto S_3 is well known from elementary group theory. Images of elements and subgroups under `hom` can be calculated with the function `Image` (**Reference: Image**). But since the mapping `hom` is not bijective, we cannot use the function `PreImage` (**Reference: PreImage**) for preimages of elements (they can have several preimages). Instead, we have to use `PreImagesRepresentative` (**Reference: PreImagesRepresentative**), which returns one preimage if at least one exists (and would return `fail` if none exists, which cannot occur for our surjective `hom`). On the other hand, we can use `PreImage` (**Reference: PreImage**) for the preimage of a set (which always exists, even if it is empty).

Suppose we mistype the input when trying to construct a homomorphism as below.

Example

```
gap> GroupHomomorphismByImages( s4, s3,
>        GeneratorsOfGroup(s4), [(1,2,3),(2,3)] );
fail
```

There is no such homomorphism, hence `fail` is returned. But note that because of this, `GroupHomomorphismByImages` (**Reference: GroupHomomorphismByImages**) must do some checks, and this was also done for the mapping `hom` above. One can avoid these checks if one is sure that the desired homomorphism really exists. For that, the function `GroupHomomorphismByImagesNC` (**Reference: GroupHomomorphismByImagesNC**) can be used; the `NC` stands for "no check".

But note that horrible things can happen if `GroupHomomorphismByImagesNC` (**Reference: GroupHomomorphismByImagesNC**) is used when the input does not describe a homomorphism.

Example

```
gap> hom2 := GroupHomomorphismByImagesNC( s4, s3,
>            GeneratorsOfGroup(s4), [(1,2,3),(2,3)] );
[ (1,2,3,4), (1,2) ] -> [ (1,2,3), (2,3) ]
gap> Size( Kernel(hom2) );
24
```

In other words, GAP claims that the kernel is the full s4, yet hom2 obviously has some non-trivial images! Clearly there is no such thing as a homomorphism which maps an element of order 4 (namely, (1,2,3,4)) to an element of order 3 (namely, (1,2,3)). *But if you use the command* `GroupHomomorphismByImagesNC` *(**Reference: GroupHomomorphismByImagesNC**), GAP trusts you.*

Example

```
gap> IsGroupHomomorphism( hom2 );
true
```

And then it produces serious nonsense if the thing is not a homomorphism, as seen above!

Besides the safe command `GroupHomomorphismByImages` (**Reference: GroupHomomorphismByImages**), which returns `fail` if the requested homomorphism does not exist, there is the function `GroupGeneralMappingByImages` (**Reference: GroupGeneralMappingByImages**), which returns a general mapping (that is, a possibly multi-valued mapping) that can be tested with `IsGroupHomomorphism` (**Reference: IsGroupHomomorphism**).

Example

```
gap> hom2 := GroupGeneralMappingByImages( s4, s3,
>            GeneratorsOfGroup(s4), [(1,2,3),(2,3)] );;
gap> IsGroupHomomorphism( hom2 );
false
```

But the possibility of testing for being a homomorphism is not the only reason why GAP offers *group general mappings*. Another (more important?) reason is that their existence allows "reversal of arrows" in a homomorphism such as our original hom. By this we mean the `GroupHomomorphismByImages` (**Reference: GroupHomomorphismByImages**) with left and right sides exchanged, in which case it is of course merely a `GroupGeneralMappingByImages` (**Reference: GroupGeneralMappingByImages**).

Example

```
gap> rev := GroupGeneralMappingByImages( s3, s4,
>           [(1,2),(2,3)], GeneratorsOfGroup(s4) );;
```

Now *hom* maps a to b if and only if *rev* maps b to a, for $a \in$ s4 and $b \in$ s3. Since every such b has four preimages under hom, it now has four images under rev. Just as the four preimages form a coset of the kernel $V_4 \leq$s4 of hom, they also form a coset of the *cokernel* $V_4 \leq$s4 of rev. The cokernel itself is the set of all images of `One( s3 )`. (It is a normal subgroup in the group of all images under rev.) The operation One (**Reference: One**) returns the identity element of a group. And this is why GAP wants to perform such a reversal of arrows: it calculates the kernel of a homomorphism like hom as the cokernel of the reversed group general mapping (here rev).

Example

```
gap> CoKernel( rev );
Group([ (1,4)(2,3), (1,3)(2,4) ])
```

The reason why rev is not a homomorphism is that it is not single-valued (because hom was not injective). But there is another critical condition: If we reverse the arrows of a non-surjective homomorphism, we obtain a group general mapping which is not defined everywhere, i.e., which is not total (although it will be single-valued if the original homomorphism is injective). GAP requires that a group homomorphism be both single-valued and total, so you will get `fail` if you say

`GroupHomomorphismByImages( G, H, gens, imgs )` where *gens* does not generate *G* (even if this would give a decent homomorphism on the subgroup generated by *gens*). For a full description, see Chapter (**Reference: Group Homomorphisms**).

The last example of this section shows that the notion of kernel and cokernel naturally extends even to the case where neither `hom2` nor its inverse general mapping (with arrows reversed) is a homomorphism.

Example

```
gap> CoKernel( hom2 );  Kernel( hom2 );
Group([ (2,3), (1,3) ])
Group([ (3,4), (2,3,4), (1,2,4) ])
gap> IsGroupHomomorphism( InverseGeneralMapping( hom2 ) );
false
```

Summary. In this section we have constructed homomorphisms by specifying images for a set of generators. We have seen that by reversing the direction of the mapping, we get group general mappings, which need not be single-valued (unless the mapping was injective) nor total (unless the mapping was surjective).

5.5 Nice Monomorphisms

For some types of groups, the best method for calculations in it is to use instead an isomorphic group in a "better" representation (say, a permutation group). We call an injective homomorphism, that will give such an isomorphic image a "nice monomorphism".

For example in the case of a matrix group we can take the action on the underlying vector space (or a suitable subset) to obtain such a monomorphism:

Example

```
gap> grp:=GL(2,3);;
gap> dom:=GF(3)^2;;
gap> hom := ActionHomomorphism( grp, dom );; IsInjective( hom );
true
gap> p := Image( hom,grp );
Group([ (4,7)(5,8)(6,9), (2,7,6)(3,4,8) ])
```

To demonstrate the technique of nice monomorphisms, we compute the conjugacy classes of the permutation group and lift them back into the matrix group with the monomorphism `hom`. Lifting back a conjugacy class means finding the preimage of the representative and of the centralizer; the latter is called `StabilizerOfExternalSet` (**Reference: StabilizerOfExternalSet**) in GAP (because conjugacy classes are represented as external sets, see Section (**Reference: Conjugacy Classes**)).

Example

```
gap> pcls := ConjugacyClasses( p );; gcls := [  ];;
gap> for pc  in pcls  do
>      gc:=ConjugacyClass(grp,
>                  PreImagesRepresentative(hom,Representative(pc)));
>      SetStabilizerOfExternalSet(gc,PreImage(hom,
>                                      StabilizerOfExternalSet(pc)));
>      Add( gcls, gc );
>    od;
gap> List( gcls, Size );
[ 1, 8, 12, 1, 8, 6, 6, 6 ]
```

All the steps we have made above are automatically performed by GAP if you simply ask for `ConjugacyClasses( grp )`, provided that GAP already knows that `grp` is finite (e.g., because you asked `IsFinite( grp )` before). The reason for this is that a finite matrix group like `grp` is "handled by a nice monomorphism". For such groups, GAP uses the command `NiceMonomorphism` (**Reference: NiceMonomorphism**) to construct a monomorphism (such as the `hom` in the previous example) and then proceeds as we have done above.

Example

```
gap> grp:=GL(2,3);;
gap> IsHandledByNiceMonomorphism( grp );
true
gap> hom := NiceMonomorphism( grp );
<action isomorphism>
gap> p :=Image(hom,grp);
Group([ (4,7)(5,8)(6,9), (2,7,6)(3,4,8) ])
gap> cc := ConjugacyClasses( grp );; ForAll(cc, x-> x in gcls);
true
gap> ForAll(gcls, x->x in cc); # cc and gcls might be ordered differently
true
```

Note that a nice monomorphism might be defined on a larger group than `grp` –so we have to use `Image( hom, grp )` and not only `Image( hom )`.

Nice monomorphisms are not only used for matrix groups, but also for other kinds of groups in which one cannot calculate easily enough. As another example, let us show that the automorphism group of the quaternion group of order 8 is isomorphic to the symmetric group of degree 4 by examining the "nice object" associated with that automorphism group.

Example

```
gap> p:=Group((1,7,6,8)(2,5,3,4), (1,2,6,3)(4,8,5,7));;
gap> aut := AutomorphismGroup( p );; NiceMonomorphism(aut);;
gap> niceaut := NiceObject( aut );
Group([ (1,4,2,3), (1,5,4)(2,6,3), (1,2)(3,4), (3,4)(5,6) ])
gap> IsomorphismGroups( niceaut, SymmetricGroup( 4 ) );
[ (1,4,2,3), (1,5,4)(2,6,3), (1,2)(3,4), (3,4)(5,6) ] ->
[ (1,4,2,3), (1,4,2), (1,2)(3,4), (1,3)(2,4) ]
```

The range of a nice monomorphism is in most cases a permutation group, because nice monomorphisms are mostly action homomorphisms. In some cases, like in our last example, the group is solvable and you might prefer a pc group as nice object. You cannot change the nice monomorphism of the automorphism group (because it is the value of the attribute `NiceMonomorphism` (**Reference: NiceMonomorphism**)), but you can compose it with an isomorphism from the permutation group to a pc group to obtain your personal nicer monomorphism. If you reconstruct the automorphism group, you can even prescribe it this nicer monomorphism as its `NiceMonomorphism` (**Reference: NiceMonomorphism**), because a newly-constructed group will not yet have a `NiceMonomorphism` (**Reference: NiceMonomorphism**) set.

Example

```
gap> nicer := NiceMonomorphism(aut) * IsomorphismPcGroup(niceaut);;
gap> aut2 := GroupByGenerators( GeneratorsOfGroup( aut ) );;
gap> SetIsHandledByNiceMonomorphism( aut2, true );
gap> SetNiceMonomorphism( aut2, nicer );
gap> NiceObject( aut2 );  # a pc group
Group([ f1*f2, f2^2*f3, f4, f3 ])
```

The star * denotes composition of mappings from the left to the right, as we have seen in Section 5.2 above. Reconstructing the automorphism group may of course result in the loss of other information GAP had already gathered, besides the (not-so-)nice monomorphism.

Prescribing a known map as the nice monomorphism of a group is possible only if this map knows that it is injective. In the above example, this is the case because the map is created as the composition of two maps (a nice monomorphism and an isomorphism) which know that they are injective. In general, the injectivity flag of a map `map` gets set if one tests the property by calling `IsInjective( map )`; if one is sure that `map` is injective and wants to avoid the potentially expensive test then one can set the flag by explicitly calling `SetIsInjective( map, true )`.

Summary. In this section we have seen how calculations in groups can be carried out in isomorphic images in nicer groups. We have seen that GAP pursues this technique automatically for certain classes of groups, e.g., for matrix groups that are known to be finite.

5.6 Further Information about Groups and Homomorphisms

Groups and the functions for groups are treated in Chapter (**Reference: Groups**). There are several chapters dealing with groups in specific representations, for example Chapter (**Reference: Permutation Groups**) on permutation groups, (**Reference: Polycyclic Groups**) on polycyclic (including finite solvable) groups, (**Reference: Matrix Groups**) on matrix groups and (**Reference: Finitely Presented Groups**) on finitely presented groups. Chapter (**Reference: Group Actions**) deals with group actions. Group homomorphisms are the subject of Chapter (**Reference: Group Homomorphisms**).

Chapter 6

Vector Spaces and Algebras

This chapter contains an introduction into vector spaces and algebras in GAP.

6.1 Vector Spaces

A *vector space* V over a field F is an (abelian) additive group that is closed under scalar multiplication by elements in F, such that

1. $1v = v$,
2. $a(bv) = (ab)v$,
3. $a(v+w) = av+aw$, and
4. $(a+b)v = av+bv$,

for all $a, b \in F$ and all $v, w \in V$. In GAP, only those domains that are constructed as vector spaces are regarded as vector spaces. In particular, an additive group that does not know about an acting domain of scalars is not regarded as a vector space in GAP.

Probably the most common F-vector spaces in GAP are so-called *row spaces*. They consist of row vectors, that is, lists whose elements lie in F. In the following example we compute the vector space spanned by the row vectors `[ 1, 1, 1 ]` and `[ 1, 0, 2 ]` over the rationals.

Example

```
gap> F:= Rationals;;
gap> V:= VectorSpace( F, [ [ 1, 1, 1 ], [ 1, 0, 2 ] ] );
<vector space over Rationals, with 2 generators>
gap> [ 2, 1, 3 ] in V;
true
```

The full row space F^n is created by commands like:

Example

```
gap> F:= GF( 7 );;
gap> V:= F^3;   # The full row space over F of dimension 3.
( GF(7)^3 )
gap> [ 1, 2, 3 ] * One( F ) in V;
true
```

In the same way we can also create matrix spaces. Here the short notation *field*^[*dim1*,*dim2*] can be used:

Example

```
gap> m1:= [ [ 1, 2 ], [ 3, 4 ] ];; m2:= [ [ 0, 1 ], [ 1, 0 ] ];;
gap> V:= VectorSpace( Rationals, [ m1, m2 ] );
<vector space over Rationals, with 2 generators>
gap> m1+m2 in V;
true
gap> W:= Rationals^[3,2];
( Rationals^[ 3, 2 ] )
gap> [ [ 1, 1 ], [ 2, 2 ], [ 3, 3 ] ] in W;
true
```

A field is naturally a vector space over itself.

Example

```
gap> IsVectorSpace( Rationals );
true
```

If Φ is an algebraic extension of F, then Φ is also a vector space over F (and indeed over any subfield of Φ that contains F). This field F is stored in the attribute `LeftActingDomain` (**Reference: LeftActingDomain**). In GAP, the default is to view fields as vector spaces over their *prime* fields. By the function `AsVectorSpace` (**Reference: AsVectorSpace**), we can view fields as vector spaces over fields other than the prime field.

Example

```
gap> F:= GF( 16 );;
gap> LeftActingDomain( F );
GF(2)
gap> G:= AsVectorSpace( GF( 4 ), F );
AsField( GF(2^2), GF(2^4) )
gap> F = G;
true
gap> LeftActingDomain( G );
GF(2^2)
```

A vector space has three important attributes: its *field* of definition, its *dimension* and a *basis*. We already encountered the function `LeftActingDomain` (**Reference: LeftActingDomain**) in the example above. It extracts the field of definition of a vector space. The function `Dimension` (**Reference: Dimension**) provides the dimension of the vector space.

Example

```
gap> F:= GF( 9 );;
gap> m:= [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, Z(3)^0 ] ];;
gap> V:= VectorSpace( F, m );
<vector space over GF(3^2), with 2 generators>
gap> Dimension( V );
2
gap> W:= AsVectorSpace( GF( 3 ), V );
<vector space over GF(3), with 4 generators>
gap> V = W;
true
```

```
gap> Dimension( W );
4
gap> LeftActingDomain( W );
GF(3)
```

One of the most important attributes is a *basis*. For a given basis B of V, every vector v in V can be expressed uniquely as $v = \sum_{b \in B} c_b b$, with coefficients $c_b \in F$.

In GAP, bases are special lists of vectors. They are used mainly for the computation of coefficients and linear combinations.

Given a vector space V, a basis of V is obtained by simply applying the function `Basis` (**Reference: Basis**) to V. The vectors that form the basis are extracted from the basis by `BasisVectors` (**Reference: BasisVectors**).

Example

```
gap> m1:= [ [ 1, 2 ], [ 3, 4 ] ];; m2:= [ [ 1, 1 ], [ 1, 0 ] ];;
gap> V:= VectorSpace( Rationals, [ m1, m2 ] );
<vector space over Rationals, with 2 generators>
gap> B:= Basis( V );
SemiEchelonBasis( <vector space over Rationals, with
2 generators>, ... )
gap> BasisVectors( Basis( V ) );
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 0, 1 ], [ 2, 4 ] ] ]
```

The coefficients of a vector relative to a given basis are found by the function `Coefficients` (**Reference: Coefficients**). Furthermore, linear combinations of the basis vectors are constructed using `LinearCombination` (**Reference: LinearCombination**).

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2 ], [ 3, 4 ] ] );
<vector space over Rationals, with 2 generators>
gap> B:= Basis( V );
SemiEchelonBasis( <vector space over Rationals, with
2 generators>, ... )
gap> BasisVectors( Basis( V ) );
[ [ 1, 2 ], [ 0, 1 ] ]
gap> Coefficients( B, [ 1, 0 ] );
[ 1, -2 ]
gap> LinearCombination( B, [ 1, -2 ] );
[ 1, 0 ]
```

In the above examples we have seen that GAP often chooses the basis it wants to work with. It is also possible to construct bases with prescribed basis vectors by giving a list of these vectors as second argument to `Basis` (**Reference: Basis**).

Example

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2 ], [ 3, 4 ] ] );;
gap> B:= Basis( V, [ [ 1, 0 ], [ 0, 1 ] ] );
SemiEchelonBasis( <vector space over Rationals, with 2 generators>,
[ [ 1, 0 ], [ 0, 1 ] ] )
gap> Coefficients( B, [ 1, 2 ] );
[ 1, 2 ]
```

We can construct subspaces and quotient spaces of vector spaces. The natural projection map (constructed by `NaturalHomomorphismBySubspace` (**Reference: NaturalHomomorphismBySubspace**)), connects a vector space with its quotient space.

Example

```
gap> V:= Rationals^4;
( Rationals^4 )
gap> W:= Subspace( V, [ [ 1, 2, 3, 4 ], [ 0, 9, 8, 7 ] ] );
<vector space over Rationals, with 2 generators>
gap> VmodW:= V/W;
( Rationals^2 )
gap> h:= NaturalHomomorphismBySubspace( V, W );
<linear mapping by matrix, ( Rationals^4 ) -> ( Rationals^2 )>
gap> Image( h, [ 1, 2, 3, 4 ] );
[ 0, 0 ]
gap> PreImagesRepresentative( h, [ 1, 0 ] );
[ 1, 0, 0, 0 ]
```

6.2 Algebras

If a bilinear multiplication is defined for the elements of a vector space, and if the vector space is closed under this multiplication then it is called an *algebra*. For example, every field is an algebra:

Example

```
gap> f:= GF(8); IsAlgebra( f );
GF(2^3)
true
```

One of the most important classes of algebras are sub-algebras of matrix algebras. On the set of all $n \times n$ matrices over a field F it is possible to define a multiplication in many ways. The most frequent are the ordinary matrix multiplication and the Lie multiplication.

Each matrix constructed as $[row1, row2, \ldots]$ is regarded by GAP as an *ordinary* matrix, its multiplication is the ordinary associative matrix multiplication. The sum and product of two ordinary matrices are again ordinary matrices.

The *full* matrix associative algebra can be created as follows:

Example

```
gap> F:= GF( 9 );;
gap> A:= F^[3,3];
( GF(3^2)^[ 3, 3 ] )
```

An algebra can be constructed from generators using the function `Algebra` (**Reference: Algebra**). It takes as arguments the field of coefficients and a list of generators. Of course the coefficient field and the generators must fit together; if we want to construct an algebra of ordinary matrices, we may take the field generated by the entries of the generating matrices, or a subfield or extension field.

Example

```
gap> m1:= [ [ 1, 1 ], [ 0, 0 ] ];; m2:= [ [ 0, 0 ], [ 0, 1 ] ];;
gap> A:= Algebra( Rationals, [ m1, m2 ] );
<algebra over Rationals, with 2 generators>
```

An interesting class of algebras for which many special algorithms are implemented is the class of *Lie algebras*. They arise for example as algebras of matrices whose product is defined by the Lie bracket $[A,B] = A*B - B*A$, where $*$ denotes the ordinary matrix product.

Since the multiplication of objects in GAP is always assumed to be the operation `*` (resp. the infix operator `*`), and since there is already the "ordinary" matrix product defined for ordinary matrices, as mentioned above, we must use a different construction for matrices that occur as elements of Lie algebras. Such Lie matrices can be constructed by `LieObject` (**Reference: LieObject**) from ordinary matrices, the sum and product of Lie matrices are again Lie matrices.

Example

```
gap> m:= LieObject( [ [ 1, 1 ], [ 1, 1 ] ] );
LieObject( [ [ 1, 1 ], [ 1, 1 ] ] )
gap> m*m;
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
gap> IsOrdinaryMatrix( m1 ); IsOrdinaryMatrix( m );
true
false
gap> IsLieMatrix( m1 ); IsLieMatrix( m );
false
true
```

Given a field `F` and a list `mats` of Lie objects over `F`, we can construct the Lie algebra generated by `mats` using the function `Algebra` (**Reference: Algebra**). Alternatively, if we do not want to be bothered with the function `LieObject` (**Reference: LieObject**), we can use the function `LieAlgebra` (**Reference: LieAlgebra for an associative algebra**) that takes a field and a list of ordinary matrices, and constructs the Lie algebra generated by the corresponding Lie matrices. Note that this means that the ordinary matrices used in the call of `LieAlgebra` (**Reference: LieAlgebra for an associative algebra**) are not contained in the returned Lie algebra.

Example

```
gap> m1:= [ [ 0, 1 ], [ 0, 0 ] ];;
gap> m2:= [ [ 0, 0 ], [ 1, 0 ] ];;
gap> L:= LieAlgebra( Rationals, [ m1, m2 ] );
<Lie algebra over Rationals, with 2 generators>
gap> m1 in L;
false
```

A second way of creating an algebra is by specifying a multiplication table. Let A be a finite dimensional algebra with basis $(x_1, x_2, \ldots, x_n)$, then for $1 \leq i, j \leq n$ the product $x_i x_j$ is a linear combination of basis elements, i.e., there are c_{ij}^k in the ground field such that $x_i x_j = \sum_{k=1}^{n} c_{ij}^k x_k$. It is not difficult to show that the constants c_{ij}^k determine the multiplication completely. Therefore, the c_{ij}^k are called *structure constants*. In GAP we can create a finite dimensional algebra by specifying an array of structure constants.

In GAP such a table of structure constants is represented using lists. The obvious way to do this would be to construct a "three-dimensional" list `T` such that `T[i][j][k]` equals c_{ij}^k. But it often happens that many of these constants vanish. Therefore a more complicated structure is used in order to be able to omit the zeros. A multiplication table of an n-dimensional algebra is an $n \times n$ array `T` such that `T[i][j]` describes the product of the `i`-th and the `j`-th basis element. This product is encoded in the following way. The entry `T[i][j]` is a list of two elements. The first of these is a list of indices k such that c_{ij}^k is nonzero. The second list contains the corresponding constants c_{ij}^k. Suppose, for

example, that S is the table of an algebra with basis $(x_1, x_2, \ldots, x_8)$ and that `S[3][7]` equals `[ [ 2, 4, 6 ], [ 1/2, 2, 2/3 ] ]`. Then in the algebra we have the relation $x_3x_7 = (1/2)x_2 + 2x_4 + (2/3)x_6$. Furthermore, if `S[6][1] = [ [ ], [ ] ]` then the product of the sixth and first basis elements is zero.

Finally two numbers are added to the table. The first number can be 1, -1, or 0. If it is 1, then the table is known to be symmetric, i.e., $c_{ij}^k = c_{ji}^k$. If this number is -1, then the table is known to be antisymmetric (this happens for instance when the algebra is a Lie algebra). The remaining case, 0, occurs in all other cases. The second number that is added is the zero element of the field over which the algebra is defined.

Empty structure constants tables are created by the function `EmptySCTable` (**Reference: EmptySCTable**), which takes a dimension d, a zero element z, and optionally one of the strings `"symmetric"`, `"antisymmetric"`, and returns an empty structure constants table T corresponding to a d-dimensional algebra over a field with zero element z. Structure constants can be entered into the table T using the function `SetEntrySCTable` (**Reference: SetEntrySCTable**). It takes four arguments, namely T, two indices i and j, and a list of the form $[c_{ij}^{k_1}, k_1, c_{ij}^{k_2}, k_2, \ldots]$. In this call to `SetEntrySCTable`, the product of the i-th and the j-th basis vector in any algebra described by T is set to $\sum_l c_{ij}^{k_l} x_{k_l}$. (Note that in the empty table, this product was zero.) If T knows that it is (anti)symmetric, then at the same time also the product of the j-th and the i-th basis vector is set appropriately.

Example

```
gap> T:= EmptySCTable( 2, 0, "symmetric" );
[ [ [ [  ], [  ] ], [ [  ], [  ] ] ],
  [ [ [  ], [  ] ], [ [  ], [  ] ] ], 1, 0 ]
gap> SetEntrySCTable( T, 1, 2, [1/2,1,1/3,2] );  T;
[ [ [ [  ], [  ] ], [ [ 1, 2 ], [ 1/2, 1/3 ] ] ],
  [ [ [ 1, 2 ], [ 1/2, 1/3 ] ], [ [  ], [  ] ] ], 1, 0 ]
```

If we have defined a structure constants table, then we can construct the corresponding algebra by `AlgebraByStructureConstants` (**Reference: AlgebraByStructureConstants**).

Example

```
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 2 over Rationals>
```

If we know that a structure constants table defines a Lie algebra, then we can construct the corresponding Lie algebra by `LieAlgebraByStructureConstants` (**Reference: LieAlgebraByStructureConstants**); the algebra returned by this function knows that it is a Lie algebra, so GAP need not check the Jacobi identity.

Example

```
gap> T:= EmptySCTable( 2, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [2/3,1] );
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 2 over Rationals>
```

In GAP an algebra is naturally a vector space. Hence all the functionality for vector spaces is also available for algebras.

Example

```
gap> F:= GF(2);;
gap> z:= Zero( F );;  o:= One( F );;
```

```
gap> T:= EmptySCTable( 3, z, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ o, 1, o, 3 ] );
gap> SetEntrySCTable( T, 1, 3, [ o, 1 ] );
gap> SetEntrySCTable( T, 2, 3, [ o, 3 ] );
gap> A:= AlgebraByStructureConstants( F, T );
<algebra of dimension 3 over GF(2)>
gap> Dimension( A );
3
gap> LeftActingDomain( A );
GF(2)
gap> Basis( A );
CanonicalBasis( <algebra of dimension 3 over GF(2)> )
```

Subalgebras and ideals of an algebra can be constructed by specifying a set of generators for the subalgebra or ideal. The quotient space of an algebra by an ideal is naturally an algebra itself.

Example

```
gap> m:= [ [ 1, 2, 3 ], [ 0, 1, 6 ], [ 0, 0, 1 ] ];;
gap> A:= Algebra( Rationals, [ m ] );;
gap> subA:= Subalgebra( A, [ m-m^2 ] );
<algebra over Rationals, with 1 generator>
gap> Dimension( subA );
2
gap> idA:= Ideal( A, [ m-m^3 ] );
<two-sided ideal in <algebra of dimension 3 over Rationals>,
  (1 generator)>
gap> Dimension( idA );
2
gap> B:= A/idA;
<algebra of dimension 1 over Rationals>
```

The call `B:= A/idA` creates a new algebra that does not "know" about its connection with `A`. If we want to connect an algebra with its factor via a homomorphism, then we first have to create the homomorphism (`NaturalHomomorphismByIdeal` (**Reference: NaturalHomomorphismByIdeal**)). After this we create the factor algebra from the homomorphism by the function `ImagesSource` (**Reference: ImagesSource**). In the next example we divide an algebra `A` by its radical and lift the central idempotents of the factor to the original algebra `A`.

Example

```
gap> m1:=[[1,0,0],[0,2,0],[0,0,3]];;
gap> m2:=[[0,1,0],[0,0,2],[0,0,0]];;
gap> A:= Algebra( Rationals, [ m1, m2 ] );;
gap> Dimension( A );
6
gap> R:= RadicalOfAlgebra( A );
<algebra of dimension 3 over Rationals>
gap> h:= NaturalHomomorphismByIdeal( A, R );
<linear mapping by matrix, <algebra of dimension
6 over Rationals> -> <algebra of dimension 3 over Rationals>>
gap> AmodR:= ImagesSource( h );
<algebra of dimension 3 over Rationals>
gap> id:= CentralIdempotentsOfAlgebra( AmodR );
```

```
[ v.3, v.2+(-3)*v.3, v.1+(-2)*v.2+(3)*v.3 ]
gap> PreImagesRepresentative( h, id[1] );
[ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 1 ] ]
gap> PreImagesRepresentative( h, id[2] );
[ [ 0, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 0 ] ]
gap> PreImagesRepresentative( h, id[3] );
[ [ 1, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ]
```

Structure constants tables for the simple Lie algebras are present in GAP. They can be constructed using the function `SimpleLieAlgebra` (**Reference: SimpleLieAlgebra**). The Lie algebras constructed by this function come with a root system attached.

Example

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> R:= RootSystem( L );
<root system of rank 2>
gap> PositiveRoots( R );
[ [ 2, -1 ], [ -3, 2 ], [ -1, 1 ], [ 1, 0 ], [ 3, -1 ], [ 0, 1 ] ]
gap> CartanMatrix( R );
[ [ 2, -1 ], [ -3, 2 ] ]
```

Another example of algebras is provided by *quaternion algebras*. We define a quaternion algebra over an extension field of the rationals, namely the field generated by $\sqrt{5}$. (The number `EB(5)` is equal to $1/2(-1+\sqrt{5})$. The field is printed as `NF(5,[ 1, 4 ])`.)

Example

```
gap> b5:= EB(5);
E(5)+E(5)^4
gap> q:= QuaternionAlgebra( FieldByGenerators( [ b5 ] ) );
<algebra-with-one of dimension 4 over NF(5,[ 1, 4 ])>
gap> gens:= GeneratorsOfAlgebra( q );
[ e, i, j, k ]
gap> e:= gens[1];; i:= gens[2];; j:= gens[3];; k:= gens[4];;
gap> IsAssociative( q );
true
gap> IsCommutative( q );
false
gap> i*j; j*i;
k
(-1)*k
gap> One( q );
e
```

If the coefficient field is a real subfield of the complex numbers then the quaternion algebra is in fact a division ring.

Example

```
gap> IsDivisionRing( q );
true
gap> Inverse( e+i+j );
(1/3)*e+(-1/3)*i+(-1/3)*j
```

So GAP knows about this fact. As in any ring, we can look at groups of units. (The function StarCyc (**Reference: StarCyc**) used below computes the unique algebraic conjugate of an element in a quadratic subfield of a cyclotomic field.)

Example

```
gap> c5:= StarCyc( b5 );
E(5)^2+E(5)^3
gap> g1:= 1/2*( b5*e + i - c5*j );
(1/2*E(5)+1/2*E(5)^4)*e+(1/2)*i+(-1/2*E(5)^2-1/2*E(5)^3)*j
gap> Order( g1 );
5
gap> g2:= 1/2*( -c5*e + i + b5*k );
(-1/2*E(5)^2-1/2*E(5)^3)*e+(1/2)*i+(1/2*E(5)+1/2*E(5)^4)*k
gap> Order( g2 );
10
gap> g:=Group( g1, g2 );;
#I  default 'IsGeneratorsOfMagmaWithInverses' method returns 'true' for
  [ (1/2*E(5)+1/2*E(5)^4)*e+(1/2)*i+(-1/2*E(5)^2-1/2*E(5)^3)*j,
  (-1/2*E(5)^2-1/2*E(5)^3)*e+(1/2)*i+(1/2*E(5)+1/2*E(5)^4)*k ]
gap> Size( g );
120
gap> IsPerfect( g );
true
```

Since there is only one perfect group of order 120, up to isomorphism, we see that the group g is isomorphic to $SL_2(5)$. As usual, a permutation representation of the group can be constructed using a suitable action of the group.

Example

```
gap> cos:= RightCosets( g, Subgroup( g, [ g1 ] ) );;
gap> Length( cos );
24
gap> hom:= ActionHomomorphism( g, cos, OnRight );;
gap> im:= Image( hom );
Group([ (2,3,5,9,15)(4,7,12,8,14)(10,17,23,20,24)(11,19,22,16,13),
  (1,2,4,8,3,6,11,20,17,19)(5,10,18,7,13,22,12,21,24,15)(9,16)(14,23) ])
gap> Size( im );
120
```

To get a matrix representation of g or of the whole algebra q, we must specify a basis of the vector space on which the algebra acts, and compute the linear action of elements w.r.t. this basis.

Example

```
gap> bas:= CanonicalBasis( q );;
gap> BasisVectors( bas );
[ e, i, j, k ]
gap> op:= OperationAlgebraHomomorphism( q, bas, OnRight );
<op. hom. AlgebraWithOne( NF(5,[ 1, 4 ]),
[ e, i, j, k ] ) -> matrices of dim. 4>
gap> ImagesRepresentative( op, e );
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ]
gap> ImagesRepresentative( op, i );
[ [ 0, 1, 0, 0 ], [ -1, 0, 0, 0 ], [ 0, 0, 0, -1 ], [ 0, 0, 1, 0 ] ]
```

```
gap> ImagesRepresentative( op, g1 );
[ [ 1/2*E(5)+1/2*E(5)^4, 1/2, -1/2*E(5)^2-1/2*E(5)^3, 0 ],
  [ -1/2, 1/2*E(5)+1/2*E(5)^4, 0, -1/2*E(5)^2-1/2*E(5)^3 ],
  [ 1/2*E(5)^2+1/2*E(5)^3, 0, 1/2*E(5)+1/2*E(5)^4, -1/2 ],
  [ 0, 1/2*E(5)^2+1/2*E(5)^3, 1/2, 1/2*E(5)+1/2*E(5)^4 ] ]
```

6.3 Further Information about Vector Spaces and Algebras

More information about vector spaces can be found in Chapter (**Reference: Vector Spaces**). Chapter (**Reference: Algebras**) deals with the functionality for general algebras. Furthermore, concerning special functions for Lie algebras, there is Chapter (**Reference: Lie Algebras**).

Chapter 7

Domains

Domain is GAP's name for structured sets. We already saw examples of domains in Chapters 5 and 6: the groups `s8` and `a8` in Section 5.1 are domains, likewise the field `f` and the vector space `v` in Section 6.1 are domains. They were constructed by functions such as `Group` (**Reference: Group**) and `GF` (**Reference: GF for field size**), and they could be passed as arguments to other functions such as `DerivedSubgroup` (**Reference: DerivedSubgroup**) and `Dimension` (**Reference: Dimension**).

7.1 Domains as Sets

First of all, a domain D is a set. If D is finite then a list with the elements of this set can be computed with the functions `AsList` (**Reference: AsList**) and `AsSortedList` (**Reference: AsSortedList**). For infinite D, `Enumerator` (**Reference: Enumerator**) and `EnumeratorSorted` (**Reference: EnumeratorSorted**) may work, but it is also possible that one gets an error message.

Domains can be used as arguments of set functions such as `Intersection` (**Reference: Intersection**) and `Union` (**Reference: Union**). GAP tries to return a domain in these cases, moreover it tries to return a domain with as much structure as possible. For example, the intersection of two groups is (either empty or) again a group, and GAP will try to return it as a group. For `Union` (**Reference: Union**), the situation is different because the union of two groups is in general not a group.

Example

```
gap> g:= Group( (1,2), (3,4) );;
gap> h:= Group( (3,4), (5,6) );;
gap> Intersection( g, h );
Group([ (3,4) ])
```

Two domains are regarded as equal w.r.t. the operator "=" if and only if they are equal *as sets*, regardless of the additional structure of the domains.

Example

```
gap> mats:= [ [ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ],
>             [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ];;
gap> Ring( mats ) = VectorSpace( GF(2), mats );
true
```

Additionally, a domain is regarded as equal to the sorted list of its elements.

Example

```
gap> g:= Group( (1,2) );;
gap> l:= AsSortedList( g );
[ (), (1,2) ]
gap> g = l;
true
gap> IsGroup( l ); IsList( g );
false
false
```

7.2 Algebraic Structure

The additional structure of D is constituted by the facts that D is known to be closed under certain operations such as addition or multiplication, and that these operations have additional properties. For example, if D is a group then it is closed under multiplication ($D \times D \to D$, $(g,h) \mapsto g*h$), under taking inverses ($D \to D$, $g \mapsto g^{-1}$) and under taking the identity `g^0` of each element g in D; additionally, the multiplication in D is associative.

The same set of elements can carry different algebraic structures. For example, a semigroup is defined as being closed under an associative multiplication, so each group is also a semigroup. Likewise, a monoid is defined as a semigroup D in which the identity `g^0` is defined for every element g, so each group is a monoid, and each monoid is a semigroup.

Other examples of domains are vector spaces, which are defined as additive groups that are closed under (left) multiplication with elements in a certain domain of scalars. Also conjugacy classes in a group D are domains, they are closed under the conjugation action of D.

7.3 Notions of Generation

We have seen that a domain is closed under certain operations. Usually a domain is constructed as the closure of some elements under these operations. In this situation, we say that the elements *generate* the domain.

For example, a list of matrices of the same shape over a common field can be used to generate an additive group or a vector space over a suitable field; if the matrices are square then we can also use the matrices as generators of a semigroup, a ring, or an algebra. We illustrate some of these possibilities:

Example

```
gap> mats:= [ [ [ 0*Z(2), Z(2)^0 ],
>               [ Z(2)^0, 0*Z(2) ] ],
>             [ [ Z(2)^0, 0*Z(2) ],
>               [ 0*Z(2), Z(2)^0 ] ] ];;
gap> Size( AdditiveMagma( mats ) );
4
gap> Size( VectorSpace( GF(8), mats ) );
64
gap> Size( Algebra( GF(2), mats ) );
4
gap> Size( Group( mats ) );
2
```

Each combination of operations under which a domain could be closed gives a notion of generation. So each group has group generators, and since it is a monoid, one can also ask for monoid generators of a group.

Note that one cannot simply ask for "the generators of a domain", it is always necessary to specify what notion of generation is meant. Access to the different generators is provided by functions with names of the form `GeneratorsOfSomething`. For example, `GeneratorsOfGroup` (**Reference: GeneratorsOfGroup**) denotes group generators, `GeneratorsOfMonoid` (**Reference: GeneratorsOfMonoid**) denotes monoid generators, and so on. The result of `GeneratorsOfVectorSpace` (**Reference: GeneratorsOfVectorSpace**) is of course to be understood relative to the field of scalars of the vector space in question.

Example

```
gap> GeneratorsOfVectorSpace( GF(4)^2 );
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
gap> v:= AsVectorSpace( GF(2), GF(4)^2 );;
gap> GeneratorsOfVectorSpace( v );
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ], [ Z(2^2), 0*Z(2) ],
  [ 0*Z(2), Z(2^2) ] ]
```

7.4 Domain Constructors

A group can be constructed from a list of group generators *gens* by `Group( gens )`, likewise one can construct rings and algebras with the functions `Ring` (**Reference: Ring**) and `Algebra` (**Reference: Algebra**).

Note that it is not always or completely checked that *gens* is in fact a valid list of group generators, for example whether the elements of *gens* can be multiplied or whether they are invertible. This means that GAP trusts you, at least to some extent, that the desired domain `Something( gens )` does exist.

7.5 Forming Closures of Domains

Besides constructing domains from generators, one can also form the closure of a given domain with an element or another domain. There are different notions of closure, one has to specify one according to the desired result and the structure of the given domain. The functions to compute closures have names such as `ClosureSomething`. For example, if *D* is a group and one wants to construct the group generated by *D* and an element *g* then one can use `ClosureGroup( D, g )`.

7.6 Changing the Structure

The same set of elements can have different algebraic structures. For example, it may happen that a monoid M does in fact contain the inverses of all of its elements, and thus M is equal to the group formed by the elements of M.

Example

```
gap> m:= Monoid( mats );;
gap> m = Group( mats );
true
gap> IsGroup( m );
false
```

The last result in the above example may be surprising. But the monoid `m` is not regarded as a group in GAP, and moreover there is no way to turn `m` into a group. Let us formulate this as a rule:

The set of operations under which the domain is closed is fixed in the construction of a domain, and cannot be changed later.

(Contrary to this, a domain *can* acquire knowledge about properties such as whether the multiplication is associative or commutative.)

If one needs a domain with a different structure than the given one, one can construct a new domain with the required structure. The functions that do these constructions have names such as `AsSomething`, they return a domain that has the same elements as the argument in question but the structure `Something`. In the above situation, one can use `AsGroup` (**Reference: AsGroup**).

Example

```
gap> g:= AsGroup( m );;
gap> m = g;
true
gap> IsGroup( g );
true
```

If it is impossible to construct the desired domain, the `AsSomething` functions return `fail`.

Example

```
gap> AsVectorSpace( GF(4), GF(2)^2 );
fail
```

The functions `AsList` (**Reference: AsList**) and `AsSortedList` (**Reference: AsSortedList**) mentioned above do not return domains, but they fit into the general pattern in the sense that they forget all the structure of the argument, including the fact that it is a domain, and return an immutable list with the same elements as the argument has.

7.7 Subdomains

It is possible to construct a domain as a subset of an existing domain. The respective functions have names such as `Subsomething`, they return domains with the structure `Something`. (Note that the second `s` in `Subsomething` is not capitalized.) For example, if one wants to deal with the subgroup of the domain *D* that is generated by the elements in the list *gens*, one can use `Subgroup( D, gens )`. It is not required that *D* is itself a group, only that the group generated by *gens* must be a subset of *D*.

The superset of a domain *S* that was constructed by a `Subsomething` function can be accessed as `Parent( S )`.

Example

```
gap> g:= SymmetricGroup( 5 );;
gap> gens:= [ (1,2), (1,2,3,4) ];;
gap> s:= Subgroup( g, gens );;
gap> h:= Group( gens );;
gap> s = h;
true
gap> Parent( s ) = g;
true
```

Many functions return subdomains of their arguments, for example the result of `SylowSubgroup( G, prime )` is a group with parent group `G`.

If you are sure that the domain `Something( gens )` is contained in the domain `D` then you can also call `SubsomethingNC( D, gens )` instead of `Subsomething( D, gens )`. The `NC` stands for "no check", and the functions whose names end with `NC` omit the check of containment.

7.8 Further Information about Domains

More information about domains can be found in Chapter (**Reference: Domains**). Many other chapters deal with specific types of domain such as groups, vector spaces or algebras.

Chapter 8

Operations and Methods

8.1 Attributes

In the preceding chapters, we have seen how to obtain information about mathematical objects in GAP: We have to pass the object as an argument to a function. For example, if *G* is a group one can call `Size( G )`, and the function will return a value, in our example an integer which is the size of *G*. Computing the size of a group generally requires a substantial amount of work, therefore it seems desirable to store the size somewhere once it has been calculated. You should imagine that GAP stores the size in some place associated with the object *G* when `Size( G )` is executed for the first time, and if this function call is executed again later, the size is simply looked up and returned, without further computation.

This means that the behavior of the function `Size` (**Reference: Size**) has to depend on whether the size for the argument *G* is already known, and if not, that the size must be stored after it has been calculated. These two extra tasks are done by two other functions that accompany `Size( G )`, namely the *tester* `HasSize( G )` and the *setter* `SetSize( G, size )`. The tester returns `true` or `false` according to whether *G* has already stored its size, and the setter puts *size* into a place from where *G* can directly look it up. The function `Size` (**Reference: Size**) itself is called the *getter*, and from the preceding discussion we see that there must really be at least two *methods* for the getter: One method is used when the tester returns `false`; it is the method which first does the real computation and then executes the setter with the computed value. A second method is used when the tester returns `true`; it simply returns the stored value. This second method is also called the *system getter*. GAP functions for which several methods can be available are called *operations*, so `Size` (**Reference: Size**) is an example of an operation.

Example

```
gap> G := Group(List([1..3], i-> Random(SymmetricGroup(53))));;
gap> Size( G ); time; # the time may of course vary on your machine
427488328406002556429801375338939964969034378836681372467200000000000 0
196
gap> Size( G ); time;
427488328406002556429801375338939964969034378836681372467200000000000 0
0
```

The convenient thing for the user is that GAP automatically chooses the right method for the getter, i.e., it calls a real-work getter at most once and the system getter in all subsequent occurrences.

At most once because the value of a function call like `Size( G )` can also be set for `G` before the getter is called at all; for example, one can call the setter directly if one knows the size.

The size of a group is an example of a class of things which in GAP are called *attributes*. Every attribute in GAP is represented by a triple of a getter, a setter and a tester. When a new attribute is declared, all three functions are created together and the getter contains references to the other two. This is necessary because when the getter is called, it must first consult the tester, and perhaps execute the setter in the end. Therefore the getter could be implemented as follows:

Example

```
getter := function( obj )
local   value;
    if tester( obj )  then
        value := system_getter( obj );
    else
        value := real_work_getter( obj );
        setter( obj, value );
    fi;
    return value;
end;
```

The only function which depends on the mathematical nature of the attribute is the real-work getter, and this is of course what the programmer of an attribute has to install. In both cases, the getter returns the same value, which we also call the value of the attribute (properly: the value of the attribute for the object `obj`). By the way: The names for setter and tester of an attribute are always composed from the prefix `Set` resp. `Has` and the name of the getter.

As a (not typical) example, note that the GAP function `Random` (**Reference: Random**), although it takes only one argument, is of course *not* an attribute, because otherwise the first random element of a group would be stored by the setter and returned over and over again by the system getter every time `Random` (**Reference: Random**) is called in the sequel.

There is a general important rule about attributes: *Once the value of an attribute for an object has been set, it cannot be reset, i.e., it cannot be changed any more.* This is achieved by having two methods not only for the getter but also for the setter: If an object already has an attribute value stored, i.e., if the tester returns `true`, the setter simply does nothing.

Example

```
gap> G := SymmetricGroup(8);; Size(G);
40320
gap> SetSize( G, 0 ); Size( G );
40320
```

Summary. In this section we have introduced attributes as triples of getter, setter and tester and we have explained how these three functions work together behind the scene to provide automatic storage and look-up of values that have once been calculated. We have seen that there can be several methods for the same function among which GAP automatically selects an appropriate one.

8.2 Properties and Filters

Certain attributes, like `IsAbelian` (**Reference: IsAbelian**), are boolean-valued. Such attributes are known to GAP as *properties*, because their values are stored in a slightly different way. A property also has a getter, a setter and a tester, but in this case, the getter as well as the tester returns a boolean

value. Therefore GAP stores both values in the same way, namely as bits in a boolean list, thereby treating property getters and all testers (of attributes or properties) uniformly. These boolean-valued functions are called *filters*. You can imagine a filter as a switch which is set either to `true` or to `false`. For every GAP object there is a boolean list which has reserved a bit for every filter GAP knows about. Strictly speaking, there is one bit for every *simple filter*, and these simple filters can be combined with `and` to form other filters (which are then `true` if and only if all the corresponding bits are set to `true`). For example, the filter `IsPermGroup and IsSolvableGroup` is made up from several simple filters.

Since they allow only two values, the bits which represent filters can be compared very quickly, and the scheme by which GAP chooses the method, e.g., for a getter or a setter (as we have seen in the previous section), is mostly based on the examination of filters, not on the examination of other attribute values. Details of this *method selection* are described in chapter (**Reference: Method Selection**).

We only present the following rule of thumb here: Each installed method for an attribute, for example `Size` (**Reference: Size**), has a "required filter", which is made up from certain simple filters which must yield `true` for the argument *`obj`* for this method to be applicable. To execute a call of `Size( obj )`, GAP selects among all applicable methods the one whose required filter combines the most simple filters; the idea behind is that the more an algorithm requires of *`obj`*, the more efficient it is expected to be. For example, if *`obj`* is a permutation group that is not (known to be) solvable, a method with required filter `IsPermGroup and IsSolvableGroup` is not applicable, whereas a method with required filter `IsPermGroup` (**Reference: IsPermGroup**) can be chosen. On the other hand, if *`obj`* was known to be solvable, the method with required filter `IsPermGroup and IsSolvableGroup` would be preferred to the one with required filter `IsPermGroup` (**Reference: IsPermGroup**).

It may happen that a method is applicable for a given argument but cannot compute the desired value. In such cases, the method will execute the statement `TryNextMethod();`, and GAP calls the next applicable method. For example, [Sim90] describes an algorithm to compute the size of a solvable permutation group, which can be used also to decide whether or not a permutation group is solvable. Suppose that the function `size_solvable` implements this algorithm, and that is returns the order of the group if it is solvable and `fail` otherwise. Then we can install the following method for `Size` (**Reference: Size**) with required filter `IsPermGroup` (**Reference: IsPermGroup**).

Example

```
function( G )
local  value;
    value := size_solvable( G );
    if value <> fail  then  return value;
                      else  TryNextMethod();  fi;
end;
```

This method can then be tried on every permutation group (whether known to be solvable or not), and it would include a mandatory solvability test.

If no applicable method (or no next applicable method) is found, GAP stops with an error message of the form

Example

```
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for 'Size' on 1 arguments called from
... lines deleted here ...
```

You would get an error message as above if you asked for `Size( 1 )`. The message simply says that there is no method installed for calculating the size of `1`. Section (**Reference: Recovery from NoMethodFound-Errors**) contains more information on how to deal with these messages.

Summary. In this section we have introduced properties as special attributes, and filters as the general concept behind property getters and attribute testers. The values of the filters of an object govern how the object is treated in the selection of methods for operations.

8.3 Immediate and True Methods

In the example in Section 8.2, we have mentioned that the operation `Size` (**Reference: Size**) has a method for solvable permutation groups that is so far superior to the method for general permutation groups that it seems worthwhile to try it even if nothing is known about solvability of the group of which the `Size` (**Reference: Size**) is to be calculated. There are other examples where certain methods are even "cheaper" to execute. For example, if the size of a group is known it is easy to check whether it is odd, and if so, the Feit-Thompson theorem allows us to set `IsSolvableGroup` (**Reference: IsSolvableGroup**) to `true` for this group. GAP utilizes this celebrated theorem by having an *immediate method* for `IsSolvableGroup` (**Reference: IsSolvableGroup**) with required filter `HasSize` which checks parity of the size and either sets `IsSolvableGroup` (**Reference: IsSolvableGroup**) or does nothing, i.e., calls `TryNextMethod()`. These immediate methods are executed automatically for an object whenever the value of a filter changes, so solvability of a group will automatically be detected when an odd size has been calculated for it (and therefore the value of `HasSize` for that group has changed to `true`).

Some methods are even more immediate, because they do not require any calculation at all: They allow a filter to be set if another filter is also set. In other words, they model a mathematical implication like `IsGroup and IsCyclic` implies `IsSolvableGroup` and such implications can be installed in GAP as *true methods*. To execute true methods, GAP only needs to do some bookkeeping with its filters, therefore true methods are much faster than immediate methods.

How immediate and true methods are installed is described in (**Reference: Immediate Methods**) and (**Reference: Logical Implications**).

8.4 Operations and Method Selection

The method selection is not only used to select methods for attribute getters but also for arbitrary *operations*, which can have more than one argument. In this case, there is a required filter for each argument (which must yield `true` for the corresponding arguments).

Additionally, a method with at least two arguments may require a certain relation between the arguments, which is expressed in terms of the *families* of the arguments. For example, the methods for `ConjugateGroup( grp, elm )` require that `elm` lies in the family of elements from which `grp` is made, i.e., that the family of `elm` equals the "elements family" of `grp`.

For permutation groups, the situation is quite easy: all permutations form one family, `PermutationsFamily` (**Reference: PermutationsFamily**), and each collection of permutations, for example each permutation group, each coset of a permutation group, or each dense list of permutations, lies in `CollectionsFamily( PermutationsFamily )`.

For other kinds of group elements, the situation can be different. Every call of `FreeGroup` (**Reference: FreeGroup**) constructs a new family of free group elements. GAP refuses to compute `One( FreeGroup( 1 ) ) * One( FreeGroup( 1 ) )` because the two operands of the multiplication lie in different families and no method is installed for this case.

For further information on family relations, see (**Reference: Families**).

If you want to know which properties are already known for an object *obj*, or which properties are known to be true, you can use the functions `KnownPropertiesOfObject( obj )` resp. `KnownTruePropertiesOfObject( obj )`. This will print a list of names of properties. These names are also the identifiers of the property getters, by which you can retrieve the value of the properties (and confirm that they are really `true`). Analogously, there is the function `KnownAttributesOfObject` (**Reference: KnownAttributesOfObject**) which lists the names of the known attributes, leaving out the properties.

Since GAP lets you know what it already knows about an object, it is only natural that it also lets you know what methods it considers applicable for a certain method, and in what order it will try them (in case `TryNextMethod()` occurs). `ApplicableMethod( opr, [ arg_1, arg_2, ... ] )` returns the first applicable method for the call *opr*`( arg_1, arg_2, ... )`. More generally, `ApplicableMethod( opr, [ ... ], 0, nr )` returns the *nr*th applicable method (i.e., the one that would be chosen after *nr* − 1 calls of `TryNextMethod`) and if *nr* `= "all"`, the sorted list of all applicable methods is returned. For details, see (**Reference: Applicable Methods and Method Selection**).

If you want to see which methods are chosen for certain operations while GAP code is being executed, you can call the function `TraceMethods` (**Reference: TraceMethods for a list of operations**) with a list of these operations as arguments.

Example

```
gap> TraceMethods( [ Size ] );
gap> g:= Group( (1,2,3), (1,2) );;  Size( g );
#I  Size: for a permutation group
#I  Setter(Size): system setter
#I  Size: system getter
#I  Size: system getter
6
```

The system getter is called once to fetch the freshly computed value for returning to the user. The second call is triggered by an immediate method. To find out by which, we can trace the immediate methods by saying `TraceImmediateMethods( true )`.

Example

```
gap> TraceImmediateMethods( true );
gap> g:= Group( (1,2,3), (1,2) );;
#I  immediate: Size
#I  immediate: IsCyclic
#I  immediate: IsCommutative
#I  immediate: IsTrivial
gap> Size( g );
#I  Size: for a permutation group
#I  immediate: IsNonTrivial
#I  immediate: Size
#I  immediate: IsFreeAbelian
#I  immediate: IsTorsionFree
#I  immediate: IsNonTrivial
#I  immediate: GeneralizedPcgs
#I  Setter(Size): system setter
#I  Size: system getter
#I  immediate: IsPerfectGroup
#I  Size: system getter
```

```
#I  immediate: IsEmpty
6
gap> TraceImmediateMethods( false );
gap> UntraceMethods( [ Size ] );
```

The last two lines switch off tracing again. We now see that the system getter was called by the immediate method for `IsPerfectGroup` (**Reference: IsPerfectGroup**). Also the above-mentioned immediate method for `IsSolvableGroup` (**Reference: IsSolvableGroup**) was not used because the solvability of g was already found out during the size calculation (cf. the example in Section 8.2).

Summary. In this section and the last we have looked some more behind the scenes and seen that GAP automatically executes immediate and true methods to deduce information about objects that is cheaply available. We have seen how this can be supervised by tracing the methods.

References

[Sim90] C. C. Sims. Computing the order of a solvable permutation group. *J. Symbolic Comput.*, 9(5-6):699–705, 1990. Computational group theory, Part 1. 74

Index

ApplicableMethod, 76
arrays, see lists, 20
assignment, 14
AsSomething, 69

break loops, 12

canonical position, 45
ClosureSomething, 69
cokernel, 53
comments, 10
constants, 12

elements, 16
enumerator, 45

family, 25
filters, 73

GeneratorsOfSomething, 68
getter
    of an attribute, 72
group general mapping, 53
    single-valued, 53
    total, 53
GroupHomomorphismByImages vs. GroupGeneralMappingByImages, 53

homomorphism
    action, 45
    natural, 41
    operation, 45

identifier, 14
IsIdenticalObj, 16

kernel, 53
KnownAttributesOfObject, 75
KnownPropertiesOfObject, 75
KnownTruePropertiesOfObject, 75

last, 15
last2, 15
last3, 15
leaving GAP, 9
line editing, 11, 12
lists
    dense, 22
    identical, 23
    plain, 20
    strictly sorted, 25
loading source code from a file, 10
loop
    for, 27
    while, 27

maps-to operator, 17
matrices, 30
methods, 72
    immediate, 75
    selection, 73
    true, 75

objects, 14
    vs. elements, 16
    vs. variables, 14
operations, 75
operators, 12

quit, 9

Read, 10
read evaluate print loop, 10
reading source code from a file, 10

setter
    of an attribute, 72
Something, 69
starting GAP, 9
strings, 22
Subsomething, 70

`SubsomethingNC`, 70

tester
    of an attribute, 72
`TraceMethods`, 76
transversal, 45
`TryNextMethod`, 74

variables, 14
vectors
    row, 30

whitespace, 10