GAP

Release 4.4.12

17 December 2008

A Tutorial

The GAP Group

http://www.gap-system.org

Acknowledgement

The following list gives the authors, indicated by A, who designed the code in the first place as well as the current maintainers, indicated by M of the various modules of which GAP is composed.

Since the process of modularization was started only recently, there might be omissions both in scope and in contributors. The compilers of the manual apologize for any such errors and promise to rectify them in future editions.

Kernel
> Frank Celler (A), Steve Linton (AM), Frank Lübeck (AM), Werner Nickel (AM), Martin Schönert (A)

Automorphism groups of finite pc groups
> Bettina Eick (A), Werner Nickel (M)

Binary Relations
> Robert Morse (AM), Andrew Solomon (A)

Characters and Character Degrees of certain solvable groups
> Hans Ulrich Besche (A), Thomas Breuer (AM)

Classes in nonsolvable groups
> Alexander Hulpke (AM)

Classical Groups
>   Thomas Breuer (AM), Frank Celler (A), Stefan Kohl (AM), Frank Lübeck (AM), Heiko Theißen (A)

Congruences of magmas, semigroups and monoids
>   Robert Morse (AM), Andrew Solomon (A)

Cosets and Double Cosets
>   Alexander Hulpke (AM)

Cyclotomics
>   Thomas Breuer (AM)

Dixon-Schneider Algorithm
>   Alexander Hulpke (AM)

Documentation Utilities
>   Frank Celler (A), Heiko Theißen (A), Alexander Hulpke (A), Willem de Graaf (A), Steve Linton (A), Werner Nickel (A), Greg Gamble (AM)

Factor groups
>   Alexander Hulpke (AM)

Finitely presented groups
>   Volkmar Felsch (A), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
>   Isabel Araújo (A), Derek Holt (A), Alexander Hulpke (A), James Mitchell (M), Götz Pfeiffer (A), Andrew Solomon (A)

GAP for MacOS
>   Burkhard Höfling (AM)

Group actions
>   Heiko Theißen (A) and Alexander Hulpke (AM)

Homomorphism search
>   Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
>   Alexander Hulpke (AM)

Identification of Galois groups
>   Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
>   Frank Celler (A), Bettina Eick (A), Werner Nickel (M)

Irreducible Modules over finite fields for finite pc groups
>   Bettina Eick (AM)

Isomorphism testing with random methods
>   Hans Ulrich Besche (AM), Bettina Eick (AM)

Lie algebras
>   Thomas Breuer (A), Craig Struble (A), Juergen Wisliceny (A), Willem A. de Graaf (AM)

Monomiality Questions
>   Thomas Breuer (AM), Erzsébet Horváth (A)

Multiplier and Schur cover
>   Werner Nickel (AM), Alexander Hulpke (AM)

One-Cohomology and Complements
>   Frank Celler (A) and Alexander Hulpke (AM)

Partition Backtrack algorithm
      Heiko Theißen (A), Alexander Hulpke (M)

Permutation group composition series
      Ákos Seress (AM)

Permutation group homomorphisms
      Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Permutation Group Pcgs
      Heiko Theißen (A), Alexander Hulpke (M)

Possible Permutation Characters
      Thomas Breuer (AM), Götz Pfeiffer (A)

Possible Class Fusions, Possible Power Maps Thomas Breuer (AM)

Primitive groups library
      Heiko Theißen (A), Colva Roney-Dougal (AM)

Properties and attributes of finite pc groups
      Frank Celler (A), Bettina Eick (A), Werner Nickel (M)

Random Schreier-Sims
      Ákos Seress (AM)

Rational Functions
      Frank Celler (A) and Alexander Hulpke (AM)

Semigroup relations
      Isabel Araujo (A), Robert F. Morse (AM), Andrew Solomon (A)

Special Pcgs for finite pc groups
      Bettina Eick (AM)

Stabilizer Chains
      Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Strings and Characters
      Martin Schönert (A), Frank Celler (A), Thomas Breuer (A), Frank Lübeck (AM)

Structure Descriptions for Finite Groups
      Stefan Kohl (AM), Markus Püschel(A), Sebastian Egner(A)

Subgroup lattice
      Martin Schönert (A), Alexander Hulpke (AM)

Subgroup lattice for solvable groups
      Alexander Hulpke (AM)

Subgroup presentations
      Volkmar Felsch (A), Werner Nickel (M)

The Help System
      Frank Celler (A), Frank Lübeck (AM)

Tietze transformations
      Volkmar Felsch (A), Werner Nickel (M)

Transformation semigroups
      Isabel Araujo (A), Robert Arthur (A), Robert F. Morse (AM), Andrew Solomon (A)

Transitive groups library
      Alexander Hulpke (AM)

Two-cohomology and extensions of finite pc groups
      Bettina Eick (AM)

Contents

Copyright Notice

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, and in particular to changes between the current and previous versions.

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.

There are four further manuals in addition to this one: the "Reference Manual" containing detailed documentation of the mathematical functionality of GAP; "Extending GAP" containing some tutorial material on various aspects of GAP programming; "Programming in GAP 4" containing detailed documentation of various aspects of the system of interest mainly to programmers; and "New Features for Developers" containing details of some newly introduced features which we may wish to change in a future release and so do not want to include in the main reference manual. Some 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. This preface, however, serves as an introduction to the whole system.

Subsequent sections of this preface explain the structure of the system and the arrangements for the attribution of credit for authorship and maintenance of the system; acknowledge those who have made particular contributions to this and previous releases and outline the changes from earlier versions.

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 the Copyright Notice of the previous page.

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 74 in the Reference Manual, and on our World Wide Web site.

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 is this release, 4.4, 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 Web pages in a section entitled "Release history and Prefaces".

For those readers who have used an earlier version of GAP, an overview of the changes from GAP 4.3, and a brief summary of changes from earlier versions is given in section 1.4 below.

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 in his programming;
   - a set of time-critical basic functions, e.g. "arithmetic", operations for integers, finite fields, permutations and words, as well as natural operations for lists and records;
   - an interpreter for the 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 the user can as easily as the original programmers investigate and vary algorithms of the library and add new ones to it, first for own use and eventually for the benefit of all 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 various formats 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` 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. (For technical reasons the library of tables of marks is still distributed in the main system archive.) 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.

1.2 Authorship and Maintenance

Previous versions of GAP have simply included the increasingly long list of all of the authors of the system with no indication as to who contributed what. With GAP 4.3 we have introduced a new concept: **modules**, to allow us to report the authorship of the system in more detail. A module is a part of GAP which provides identifiable functionality and has reasonably clean interfaces with the rest of the system (usually it consists of separate files). Each module has its own lists of authors and maintainers, which are not necessarily the same. A preliminary list of modules and their attributions appears in this manual. Note that we are still in the process of identifying modules within the system, so large parts of the system do not yet fall into any module. Since also we all collaborate closely in designing, developing and debugging the system, it should not be assumed that the list of modules in this manual represents all of everyone's contribution, or that it lists everyone who made any contribution at all to each module.

All GAP packages are also considered to be modules and have their own authors and maintainers. It should however be noted that some packages provide interfaces between GAP and an external program, a copy of which is included for convenience, and that, in these cases, we do not claim that the module authors or maintainers wrote, or maintain, this external program. Similarly, some modules and packages include large data libraries that may have been computed by many people. We try to make clear in each case what credit is attributable to whom.

We have, for some time, operated a refereeing system for contributed packages, both to ensure the quality of the software we distribute, and to provide recognition for the authors. We now consider this to be a refereeing system for modules, and we would note, in particular that, although it does not use the standard package interface, the library of small groups has been refereed and accepted on exactly the same basis as the accepted packages.

We also include with this distribution a number of packages which have not (yet) gone through our refereeing process. Some may be accepted in the future, in other cases the authors have chosen not to submit them. More information can be found on our World Wide Web site, see section 1.5.

1.3 Acknowledgements

Very many people have worked on, and contributed to, GAP over the years since its inception. On our Web site you will find the prefaces to the previous releases, each of which acknowledges people who have made special contributions to that release. Even so, it is appropriate to mention here Joachim Neubüser whose vision of a free, open and extensible system for computational algebra inspired GAP in the first place, and Martin Schönert, who was the technical architect of GAP 3 and GAP 4.

In the past years GAP development has become a more and more widely distributed operation, and increasingly dependent on hard voluntary work by developers not solely or mainly employed to work on GAP.

Nevertheless, the development process has remained constructive and friendly, even when wrangling over difficult technical decisions, or sensitive questions of attribution and credit and I must express our huge gratitude to everyone involved for this.

The list of modules which appears in this manual now gives a partial idea of the contributions of different people, but we would like to mention some people who have made important contributions to the development process over the last years that do not show up there:

Steve Linton has been leading the GAP group in St Andrews over the last years and continues to be the main kernel maintainer and developer. The group in St Andrews also maintains most of the development infrastructure and helps with user support.

Thomas Breuer continues to develop many areas of the system, and to play a vital role in clarifying our underlying concepts, despite now working in industry.

Frank Lübeck set up a new system for automatic handling of packages and helped with various kernel issues.

Bettina Eick and her research group in Braunschweig have contributed much functionality, mainly in the form of packages.

Max Neunhöffer has brought much fresh insight to bear on the design of crucial parts of the system, and also done a lot of the ensuing work; Stefan Kohl and Volkmar Felsch have both brought enormous persistence to pointing out errors and inconsistencies in code and documentation, improving error messages and generally polishing the system; and very many others have contributed ideas, insight and hard work to produce this release. Senior colleagues, especially Joachim Neubüser, Edmund Robertson, and Charley Wright, continue to provide encouragement support and constructive criticism.

1.4 Changes from Earlier Versions

The main changes between GAP 4.3 and GAP 4.4 are:

Potentially Incompatible Changes

- The mechanism for the loading of Packages has changed to allow easier updates independent of main GAP releases. Packages require a file `PackageInfo.g` now. The new `PackageInfo.g` files are available for all Packages with the new version of GAP.

- `IsSimple` returns false now for the trivial group.

- `PrimeBlocks`: The output format has changed.

- `Division Rings`: These are implemented as `IsRingWithOne` now.

- `DirectSumOfAlgebras`: p-th power maps are compatible with the input now.

- The print order for polynomials has been changed.

These changes are, in some respects, departures from our policy of maintaining upward compatibility of documented functions between releases. In the first case, we felt that the old behavior was sufficiently inconsistent, illogical, and impossible to document that we had no alternative but to change it. In the case of the package interface, the change was necessary to introduce new functionality. The planned and phased removal of a few unnecessary functions or synonyms is needed to avoid becoming buried in "legacy" interfaces, but we remain committed to our policy of maintaining upward compatibility whenever sensibly possible.

- Groebner Bases:

   Buchberger's algorithm to compute Groebner Bases has been implemented in GAP. (A. Hulpke)

- For large scale Groebner Basis computations there also is an interface to the Singular system available in the `Singular` Package. (M. Costantini and W. de Graaf)

- New methods for factorizing polynomials over algebraic extensions of the rationals have been implemented in GAP. (A. Hulpke)

- For more functionality to compute with algebraic number fields there is an interface to the Kant system available in the `Alnuth` Package. (B. Assmann and B. Eick)

- A new functionality to compute the minimal normal subgroups of a finite group, as well as its socle, has been installed. (B. Höfling)

- A fast method for recognizing whether a permutation group is symmetric or alternating is available now (A. Seress)

- A method for computing the Galois group of a rational polynomial is available again. (A. Hulpke)

- The algorithm for `BrauerCharacterValue` has been extended to the case where the splitting field is not supported in GAP. (T. Breuer)

- Brauer tables of direct products can now be constructed from the known Brauer tables of the direct factors. (T. Breuer)

– Basic support for vector spaces of rational functions and of uea elements is available now in GAP. (T. Breuer and W. de Graaf)

– Various new functions for computations with integer matrices are available, such as methods for computing normal forms of integer matrices as well as nullspaces or solutions systems of equations. (W. Nickel and F. Gaehler)

New Packages

The following new Packages have been accepted.

– Alnuth: Algebraic Number Theory and an interface to the Kant system. By B. Assmann and B. Eick

– LAGUNA: Computing with Lie Algebras and Units of Group Algebras. By V. Bovdi, A. Konovalov, R. Rossmanith, C. Schneider.

– NQ: The ANU Nilpotent Quotient Algorithm. By W. Nickel.

– KBMAG: Knuth-Bendix for Monoids and Groups. By D. Holt.

– Polycyclic: Computation with polycyclic groups. By B. Eick and W. Nickel

– QuaGroup: Computing with Quantized Enveloping Algebras. By W. de Graaf.

Performance Enhancements

– The computation of irreducible representations and irreducible characters using the Baum-Clausen algorithm and the implementation of the Dixon-Schneider algorithm have been speeded up.

– The algorithm for `PossibleClassFusions` has been changed: the efficiency is improved and a new criterion is used. The algorithm for `PossibleFusionsCharTableTom` has been speeded up. The method for `PrimeBlocks` has been improved following a suggestion of H. Pahlings.

– New improved methods for normalizer and subgroup conjugation in Sn have been installed and new improved methods for IsNaturalSn/An have been implemented. These improve the available methods when groups of large degrees are given.

– The partition split method used in the permutation backtrack is now in the kernel. Transversal computations in large permutation groups are improved. Homomorphisms from free groups into permutation groups now give substantially shorter words for preimages.

– The membership test in SP and SU has been improved using the invariant forms underlying these groups.

– An improvement for the cyclic extension method for the computation of subgroup lattices has been implemented.

– A better method for MinimalPolynomial for finite field matrices has been implemented.

– The display has changed and the arithmetic of multivariate polynomials has been improved.

– The `LogMod` function now uses Pollard's rho method combined with the Pohlig/Hellmann approach.

– Various functions for sets and lists have been improved following suggestions of L. Teirlinck. These include: Sort, Sortex, SortParallel, SortingPerm, NrArrangements.

– The methods for `StructureConstantsTable` and `GapInputSCTable` have been improved in the case of a known (anti-) symmetry following a suggestion of M. Costantini.

The improvements listed in this Section have been implemented by T. Breuer and A. Hulpke.

New Programming and User Features

– The 2GB limit for workspace size has been removed and version numbers for saved workspaces have been introduced. (S. Linton and B. Höfling)

– The limit on the total number of types created in a session has been removed. (S. Linton)

– There is a new mechanism for loading packages available. Packages need a file PackageInfo.g now. (T. Breuer and F. Lübeck)

Finally, as always, a number of bugs have been fixed. This release thus incorporates the contents of all the bug fixes which were released for GAP 4.3. It also fixes a number of bugs discovered since the last bug fix.

The most important changes between GAP 4.2 and GAP 4.3 were:

– The performance of several routines has been substantially improved.

– The functionality in the areas of finitely presented groups, Schur covers and the calculation of Representations has been extended.

– The data libraries of transitive groups, finite integral matrix groups, character tables and tables of marks have been extended.

– The Windows installation has been simplified for the case where you are installing GAP in its standard location.

– Many bugs have been fixed.

The most important changes between GAP 4.1 and GAP 4.2 were:

– A much extended and improved library of small groups as well as associated IdGroup routines.

– The primitive groups library has been made more independent of the rest of GAP, some errors were corrected.

– New (and often much faster) infrastructure for orbit computation, based on a general "dictionary" abstraction.

– New functionality for dealing with representations of algebras, and in particular for semisimple Lie algebras.

– New functionality for binary relations on arbitrary sets, magmas and semigroups.

– Bidirectional streams, allowing an external process to be started and then controlled "interactively" by GAP

– A prototype implementation of algorithms using general subgroup chains.

– Changes in the behavior of vectors over small finite fields.

– A fifth book "New features for Developers" has been added to the GAP manual.

– Numerous bug fixes and performance improvements

The changes between the final release of GAP 3 (version 3.4.4) and GAP 4 are wide-ranging. The general philosophy of the changes is two-fold. Firstly, many assumptions in the design of GAP 3 revealed its authors' primary interest in group theory, and indeed in finite group theory. Although much of the GAP 4 library is concerned with groups, the basic design now allows extension to other algebraic structures, as witnessed by the inclusion of substantial bodies of algorithms for computation with semigroups and Lie algebras. Secondly, as the scale of the system, and the number of people using and contributing to it has grown, some aspects of the underlying system have proved to be restricting, and these have been improved as part of comprehensive re-engineering of the system. This has included the new method selection system, which underpins the library, and a new, much more flexible, GAP package interface.

Details of these changes can be found in chapter 9 of this manual. It is perhaps worth mentioning a few points here.

Firstly, much remains unchanged, from the perspective of the mathematical user:

– The syntax of that part of the GAP language that most users need for investigating mathematical problems.

– The great majority of function names.

– Data libraries and the access to them.

A number of visible aspects have changed:

– Some function names that need finer specifications now that there are more structures available in GAP.

– The access to information already obtained about a mathematical structure. In GAP 3 such information about a group could be looked up by directly inspecting the group record, whereas in GAP 4 functions must be used to access such information.

Behind the scenes, much has changed:

– A new kernel, with improvements in memory management and in the language interpreter, as well as new features such as saving of workspaces and the possibility of compilation of GAP code into C.

– A new structure to the library, based upon a new type and method selection system, which is able to support a broader range of algebraic computation and to make the structure of the library simpler and more modular.

– New and faster algorithms in many mathematical areas.

– Data structures and algorithms for new mathematical objects, such as algebras and semigroups.

– A new and more flexible structure for the GAP installation and documentation, which means, for example, that a GAP package and its documentation can be installed and be fully usable without any changes to the GAP system.

Very few features of GAP 3 are not yet available in GAP 4.

– Not all of the GAP 3 packages have yet been converted for use with GAP 4 (although several new packages are available only in GAP 4).

– The library of crystallographic groups which was present in GAP 3 is now part of a GAP 4 package `crystcat`.

1.5 Further Information about GAP

Information about GAP is best obtained from the GAP Web pages that you find on:

`http://www.gap-system.org`

and its mirror at:

`http://www.math.rwth-aachen.de/~GAP`

There you will find, amongst other things

– directions to the sites from which you can download the current GAP distribution, any bug-fixes, all accepted 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 Web page for details. In particular we will announce bugfixes 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. Section 73.9 in the Reference Manual tells you what to include in a bug report.

- 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 me to wish you all pleasure and success in using GAP, and to invite your constructive comment and criticism.

Fort Collins, March 2004 Alexander Hulpke

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 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).

```
This ''session protocol'' is indented and printed in typewriter style
(like this paragraph) in this tutorial and should look exactly as it
looks on your text terminal or text window.
```

This is to 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 an overview of its sections at the beginning, and 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.

```
$ 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 3.1 in the reference manual.)

```
gap>
```

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

```
gap> quit;
$
```

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

On some systems (for example the Apple Macintosh) minor changes might be necessary. This is explained in chapter 73 in the reference manual.

In most places **whitespace** characters (i.e. *space*s, *tab*s and *return*s) 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.

You should be able to reproduce the results of the **examples** of GAP sessions in this manual, in the following sense. If you start the GAP session with the two commands

```
gap> SizeScreen( [ 80, ] ); LogTo( "logfile1" );
```

(which are used to set the line length to 80 if this isn't already your default line length and to save a listing of the session on some file), then choose any chapter and rerun its examples in one continuous session and in the given order, the GAP output should look like the output shown in the manual, except for a few lines of output which we have edited a little bit with respect to blanks or line breaks in order to improve the readability. However, when random processes are involved, you may get different results if you extract single examples and run them separately.

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 function:

```
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 GAP root directory (the directory containing bin/, doc/, lib/, etc.).

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 9.7.1 in the Reference Manual.

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*.

```
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 *ctl*-B and *ctl*-F and insert or delete characters anywhere in the line. The line editing commands are fully described in section 6.9 of the reference manual.

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.

```
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.

```
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 *ctl*-P to recover the last line of input. It will be written after the prompt with the cursor in the first position. Type *ctl*-E to take the cursor to the end of the line, then *ctl*-B to move the cursor one character back. The cursor is now on the position of the semicolon. Enter the missing parenthesis by simply typing). Now the line is correct and may be passed to 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 *ctl*-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.

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

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

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

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

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

```
gap> 3^132;
95500495079682523689319070177441401191993513897434312983685384
```

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

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

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

```
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.

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

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

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

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

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

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

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

The 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** '.

```
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.

```
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.

```
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()`.

```
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

```
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.

```
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` to test whether this is the case.

```
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`, for example, can be applied to an integer and will return the factorial of this integer.

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

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

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

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

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

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

Some functions will both change an argument and return an object such as the function `Sortex` that sorts a list and returns the permutation of the list elements that it has performed. You will not understand right now what it means to change an object. We will return to this subject several times in the next sections.

A comfortable way to define a function 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.

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

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

```
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 following 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.

```
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 9.7.1, 6.10.1 for this.

The definition of the GAP syntax can be looked up in Chapter 4. A complete list of command line editing facilities is found in Section 6.9. The break loop is described in Section 6.4.

Operators are explained in more detail in Sections 4.7 and 4.11. You will find more information about boolean values in Chapters 20 and 22. Permutations are described in Chapter 40 and characters in Chapter 26.

Variables and assignments are described in more detail in 4.8 and 4.14. A complete list of keywords is contained in 4.5.

More about functions can be found in 4.10 and 4.15.

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 10 prime numbers.

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

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

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

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

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

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

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

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`.

```
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.

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

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

```
gap> lll[1]:= 2;
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` 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` will return the special object `fail`.

```
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.

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

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

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

Now the tilde in the fourth position of `lll` denotes the object that is currently printed. Note that the result of the last operation is the actual value of the object `lll` on the right hand side of the assignment. 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`. You can learn much more about strings in the reference manual.

```
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 }*.

```
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.

```
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`.

```
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`.)

```
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 example.

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

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

Finally try the following examples and explain the results.

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

Now return to Section 3.1 and find out whether the functions `Add` and `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` 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.

```
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';
Lists 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` is `ShallowCopy`, which produces a new mutable list whose ith entry is the ith 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` 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` 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` which takes the list as its argument and returns a set obtained from this list by ignoring holes and duplicates and by sorting the elements.

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

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

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

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

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

The 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` which takes the set `fruits` as its first argument and an element as its second argument and adds the element to the set if it wasn't already there. Note that the object `fruits` is changed.

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

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

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

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

The arguments of the functions `Intersection` and `Union` could be ordinary lists, while their result is always a set. Note that in the preceding example at least one argument of `Intersection` was not a set. The functions `IntersectSet` and `UniteSet` also form the intersection resp. union of two sets. They will however not return the result but change their first argument to be the result. Try them carefully.

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.

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

This compact printed representation of a fairly long list corresponds to a compact internal representation. The function `IsRange` tests whether an object is a range, the function `ConvertToRangeRep` changes the representation of a list that is in fact a range to this compact internal representation.

```
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.

```
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.

```
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.

```
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.

```
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 4.17 and 4.19 of the reference manual.

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.

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

The function `Product` takes a list as its argument and computes the product of the elements of the list. This is possible whenever a multiplication of the elements of the list is defined. So `Product` executes a loop over all elements of the list.

There are other often used loops available as functions. Guess what the function `Sum` does. The function `List` may take a list and a function as its arguments. It will then apply the function to each element of the list and return the corresponding list of results. A list of cubes is produced as follows with the function `cubed` from Section 4.

```
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` to the last list. But we may as well give the function `cubed` to `Sum` as an additional argument.

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

The primes less than 30 can be retrieved out of the list `primes` from Section 3.1 by the function `Filtered`. This function takes the list `primes` and a property as its arguments and will return the list of those elements of `primes` which have this property. Such a property will be represented by a function that returns a boolean value. In this example the property of being less than 30 can be represented by the function `x -> x < 30` since `x < 30` will evaluate to `true` for values `x` less than 30 and to `false` otherwise.

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

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.

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

Finally we mention the function `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` checks whether the function returns `true` for all elements of the list.

```
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 21 of the reference manual.

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.

```
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.

```
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.

```
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.)

```
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.)

```
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).

```
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`, which computes the order of a group element.

```
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` methods might not be able to prove that the matrix has infinite order, and one gets the following warning.

```
#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.

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

`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.

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

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

The value of a record component is accessible by the record name and the record component name separated by one dot as the record component selector.

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

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

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

We may use the `Display` function to illustrate the hierarchy of the record components.

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

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`, 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.

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

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 21, in particular more about identical lists in Section 21.6. A more detailed description of strings is contained in Chapter 26. Fields are described in Chapter 56, some known fields in GAP are described in Chapters 16, 58, and 57. Row vectors and matrices are described in more detail in Chapters 23 and 24. Vector spaces are described in Chapter 59, further matrix related structures are described in Chapters 42, 60, and 61.

You will find more list operations in Chapter 21.

Records and functions for records are described in detail in Chapter 27.

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.

```
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`.

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

Note the additional newline character `"\n"` in the `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.

```
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 a number.

```
gap> sign:= function(n)
>         if n < 0 then
>             return -1;
>         elif n = 0 then
>             return 0;
>         else
>             return 1;
>         fi;
>     end;
function( n ) ... end
gap> sign(0); sign(-99); sign(11);
0
-1
1
```

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 $f(n)$.)

```
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.

```
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.

```
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.

```
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 5. The `if` statement is described in more detail in Section 4.16. More about Fibonacci numbers is found in Section 17.3.1 and more about partitions in Section 17.2.15.

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`, `Size` or `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:

```
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 on eight points, and assigned it to the identifier `s8`. Now the group 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.

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

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` can compute the commutator subgroup of **two** arbitrary subgroups.) The situation is a bit different for Sylow p-subgroups: The function `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 called `ComputedSylowSubgroups`, but we won't dwell on the details here.

```
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.

```
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 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 have to use the function SetName. 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.

```
gap> SetName( elab, "2^3" ); elab;
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.

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

The factor group is again represented as a permutation group. 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 here because hom is not a bijection, so an element of the range can have several preimages or none at all.

```
gap> ker:= Kernel( hom );
2^3
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(2^3,(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.

```
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 has no non-trivial normal subgroups. 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.)

```
gap> IsSimple( f ); IsomorphismTypeInfoFiniteSimpleGroup( f );
true
rec( series := "L", parameter := [ 2, 7 ],
  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)" )
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`; 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 $L_3(2)$.

```
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`. Something like

```
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.

In the following example we temporarily increase the line length limit from its default value 80 to 82 in order to make the long expression fit into one line.

```
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> SizeScreen([ 82, ]);;
gap> List( ccl, Size );
[ 1, 210, 105, 112, 1680, 1120, 2520, 1260, 1344, 1344, 1344, 3360, 2880, 2880 ]
gap> SizeScreen([ 80, ]);;
```

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

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

We use `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`, used to find the first element in a list which passes some test. See 21.20.20 in the reference manual for more details.

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`.

```
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` and `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 fixpoint (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.

```
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`) to indicate that the action is not the default action on points but an action on sets of elements given as sorted lists. (Section 39.2 of the reference manual 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 are monomorphisms). We construct the composition of two homomorphisms with the `*` operator, reading left-to-right.

```
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([ (1,4,2), (3,6,7), (3,8,5,7,6), (1,4)(7,8) ])
```

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`.

```
gap> sgp := Normalizer( a8, Subgroup(a8,[Representative(class)]) );;
gap> Size( sgp );
360
gap> RepresentativeAction( a8, sgp, preim );
(3,4)(7,8)
```

So far we have seen a few applications of the functions `Action` and `ActionHomomorphism`. But perhaps even more interesting is the fact that the natural homomorphism `hom` constructed above is also an **action homomorphism**; this is also the reason why its image is represented as a permutation group: it is the natural representation for actions. We will now look at this action homomorphism again to find out on what objects it operates. These objects form the so-called **external set** which is associated with every action homomorphism. We will mention external sets only superficially in this tutorial, for details see 39.11 in the reference manual. For the moment, we need only know that the external set is obtained by the function `UnderlyingExternalSet`.

```
gap> t := UnderlyingExternalSet( hom );
<xset:RightTransversal(2^3:L_3(2),Group(
[ (1,5)(2,6)(3,7)(4,8), (1,3)(2,4)(5,7)(6,8), (1,2)(3,4)(5,6)(7,8),
  (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6) ]))>
```

For the natural homomorphism `hom` the external set is a **right transversal** of a subgroup U in `norm`, and action on the right transversal really means action on the cosets of the subgroup U. When executing the function call `NaturalHomomorphismByNormalSubgroup( norm, elab )`, GAP has chosen a subgroup U for which the kernel of this action (i.e., the core of U in `norm`) is the desired normal subgroup `elab`. For the purpose of operating on the cosets, the right transversal `t` contains one representative from each coset of U. Regarded this way, a transversal is simply a list of group elements, and you can make GAP produce this list by `AsList(t)`. (Try it.)

The image of such a representative from `AsList(t)` under right multiplication with an element from `norm` will in general not be in `AsList(t)`, because it will not be among the chosen representatives again. Hence right multiplication is not an action on `AsList(t)`. However, GAP uses a special trick to be discussed below to make this a well-defined action on the cosets represented by the elements of `AsList(t)`. For now, it is important to know that the external set `t` is more than just the right transversal on which the group `norm` operates. Altogether three things are necessary to specify an action: a group G, a set D, and a function $opr: D \times G \to D$. We can access these ingredients with the following functions:

```
gap> ActingDomain(t);  # the group
2^3:L_3(2)
gap> Enumerator(t);
RightTransversal(2^3:L_3(2),Group(
[ (1,5)(2,6)(3,7)(4,8), (1,3)(2,4)(5,7)(6,8), (1,2)(3,4)(5,6)(7,8),
  (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6) ]))
gap> FunctionAction(t);
function( pnt, elm ) ... end
gap> NameFunction( last );
"OnRight"
```

The function which is named `"OnRight"` is also assigned to the identifier `OnRight`, and it means multiplication from the right; this is the usual way to operate on a right transversal. `OnRight( d, g )` is defined as $d * g$.

Observe that the external set `t` and its `Enumerator` are printed the same way, but be aware that an external set also comprises the acting domain and the action function. The `Enumerator` itself, i.e., the right transversal, in turn comprises knowledge about the group `norm` and the subgroup U, and this is what allows the special trick promised above. As far as `Position` is concerned, the `Enumerator` behaves as an (immutable) list and you can ask for the position of an element in it.

```
gap> elm := (1,4)(2,7)(3,6)(5,8);;
gap> Position( Enumerator(t), elm );
fail
gap> PositionCanonical( Enumerator(t), elm );
5
```

The result `fail` means that the element was not found at all in the list: it is not among the chosen representatives. The difference between the functions `Position` and `PositionCanonical` is that the first simply looks whether `elm` is contained among the representatives which together form the right transversal `t`, whereas the second really looks for the position of the coset described by the representative `elm`. In other words, it first replaces `elm` by a canonical representative of the same coset (which must be contained in `Enumerator(t)`) and then looks for its position, hence the name. The function `ActionHomomorphism` (and its relatives) always use `PositionCanonical` when they calculate the images of the generators of the source group (here, `norm`) under the homomorphism (here, `hom`). Therefore they can give a well-defined action on an *enumerator*, even if the action would not be well-defined on `AsList( enumerator )`.

The image of the natural homomorphism is the permutation group `f` that results from the action of `norm` on the right transversal. It can be calculated by either of the following commands. The second of them shows that the external set `t` contains all information that is necessary for `Action` to do its work.

```
gap> Action( norm, Enumerator(t), OnRight ) = f;
true
gap> Action( t ) = f;
true
```

We have specified the action function `OnRight` in this example, but we have seen examples like `Action( norm, elab )` earlier where this third argument was not given. If an action function is omitted, GAP

always assumes `OnPoints` 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).

Summary. In this section we have learned how groups can operate on GAP objects such as integers and group elements. We have used `ActionHomomorphism`, among others, to construct a natural homomorphism, in which case the group operated on the right transversal of a suitable subgroup. This right transversal gave us an example for the use of `PositionCanonical`, which allowed us to specify cosets by giving representatives.

5.3 Subgroups as Stabilizers

Action functions can also be used without constructing external sets. 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.

```
gap> u8 := Stabilizer( a8, 1 );
Group([ (2,3,4), (2,4)(3,5), (2,6,4), (2,4)(5,7), (2,8,6,4)(3,5) ])
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` we first generate a list of all pairs.

```
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` because *list ^ perm* is not defined. So we must tell the functions from the actions package how the group elements operate on the elements of the domain. In our example we can do this by simply passing `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`.

```
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` does that for us. It returns a list of all the orbits. We look at the orbit lengths and representatives for the orbits.

```
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.

```
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` does what we need. It creates a homomorphism defined on `a8` whose image is a new group that operates on `[1..56]` in the same way that `a8` operates on the second orbit.

```
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.

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

Remember that we can leave out the function if we mean `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` to find a block system. The (optional) third argument in the following example tells `Blocks` that we want a block system where 1 and 14 lie in one block.

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

The result is a list of sets, such that `a8_56` operates 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` as third argument.

```
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` as third argument to a function like `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`, 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.

```
gap> blocks := Blocks( a8_56, [1..56], [1,13] );;
gap> u28_56 := Stabilizer( a8_56, [1,13], 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 operates primitively on the 28 blocks.

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

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

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

Other functions are also applicable to subgroups. In the following we show that u336 operates regularly on the 60 triples of [4..8] which contain no element twice. We constuct the list of these 60 triples with the function Orbit (using OnTuples as the natural generalization of OnPairs) and then pass it as action domain to the function 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.

```
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 operates on [1..336] in the same way that a8 operates 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.

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

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

```
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].

```
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.

```
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` simply operates via `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).

```
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 *elm* in any finite group G can be computed as `OrbitLength( ` G, *elm* `)`, the command `Size( ConjugacyClass( ` G, *elm* `) )` is probably more efficient.

```
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` 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`.

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

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)(5,8)(6,7), we must use an element of `a8` that maps (1,2)(3,4)(5,6)(7,8) to (1,3)(2,4)(5,8)(6,7). The function `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` returns `fail`.

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

```
gap> u15b := ClosureGroup( u105, (2,3)(6,7) );; Index( a8, u15b );
15
gap> RepresentativeAction( a8, u15, u15b );
fail
```

`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` and `Permutation`. These are fully described in Chapter 39 in the reference manual.

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.

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

gap> PreImage( hom, (1,2,3) );
Error, <map> must be an inj. and surj. mapping called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;

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`. But since the mapping `hom` is not bijective, we cannot use the function `PreImage` for preimages of elements (they can have several preimages). Instead, we have to use `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` 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 in the following 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` 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` can be used; the `NC` stands for "no check".

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

```
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`, GAP trusts you.**

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

```
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` with left and right sides exchanged, in which case it is of course merely a `GroupGeneralMappingByImages`.

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

Now we have $a \overset{hom}{\longmapsto} b \iff b \overset{rev}{\longmapsto} a$ for $a \in$ `s4` and $b \in$ `s3`. Since every such b has 4 preimages under `hom`, it now has 4 images under `rev`. Just as the 4 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' returns the identity element of a group, see 30.10.2 in the reference manual. 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`).

```
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 38 in the reference manual.

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.

```
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 to calculate in 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:

```
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 `StabilizerO-fExternalSet` in GAP (because conjugacy classes are represented as external sets, see Section 37.10 in the reference manual).

```
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 `Conjugacy-Classes( 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** to construct a monomorphism (such as the **hom** in the previous example) and then proceeds as we have done above.

```
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.

```
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([ (1,2)(3,4), (3,4)(5,6), (1,5,2,6), (1,5,3)(2,6,4) ])
gap> IsomorphismGroups( niceaut, SymmetricGroup( 4 ) );
[ (1,2)(3,4), (3,4)(5,6), (1,5,2,6), (1,5,3)(2,6,4) ] ->
[ (1,4)(2,3), (1,3)(2,4), (1,4,2,3), (1,3,2) ]
```

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**), 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**, because a newly-constructed group will not yet have a **NiceMonomorphism** set.

```
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([ f4, f3, f1*f2^2*f3*f4, f2^2*f3*f4 ])
```

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.

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 37. There are several chapters dealing with groups in specific representations, for example Chapter 41 on permutation groups, 43 on polycyclic (including finite solvable) groups, 42 on matrix groups and 45 on finitely presented groups. Chapter 39 deals with group actions. Group homomorphisms are the subject of Chapter 38.

6 Vector Spaces and Algebras

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

6.1 Vector Spaces

A **vector space** over the field F is an additive group that is closed under scalar multiplication with elements in F. 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.

```
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:

```
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:

```
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.

```
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`. In GAP, the default is to view

fields as vector spaces over their **prime** fields. By the function `AsVectorSpace`, we can view fields as vector spaces over fields other than the prime field.

```
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` in the example above. It extracts the field of definition of a vector space. The function `Dimension` provides the dimension of the space. Here is one more 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` to V. The vectors that form the basis are extracted from the basis by `BasisVectors`.

```
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`. Furthermore, linear combinations of the basis vectors are constructed using `LinearCombination`.

```
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`.

```
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`), connects a vector space with its quotient space.

```
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 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:

```
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*, ...] 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:

```
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`. 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.

```
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` from ordinary matrices, the sum and product of Lie matrices are again Lie matrices.

```
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`. Alternatively, if we do not want to be bothered with the function `LieObject`, we can use the function `LieAlgebra` 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` are not contained in the returned Lie algebra.

```
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 $\{x_1, \ldots, x_n\}$, 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 $\{x_1, \ldots, x_8\}$ and that `S[3][7]` equals `[ [ 2, 4, 6 ], [ 1/2, 2, 2/3 ] ]`. Then in the algebra we have the relation

$$x_3 x_7 = (1/2)x_2 + 2x_4 + (2/3)x_6.$$

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`, 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`. 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.

In the following example we temporarily increase the line length limit from its default value 80 to 82 in order to make the long output expression fit into one line.

```
gap> SizeScreen([ 82, ]);;
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 ]
gap> SizeScreen([ 80, ]);;
```

If we have defined a structure constants table, then we can construct the corresponding algebra by `AlgebraByStructureConstants`.

```
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`; the algebra returned by this function knows that it is a Lie algebra, so GAP need not check the Jacobi identity.

```
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.

```
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.

In the following example we temporarily increase the line length limit from its default value 80 to 81 in order to make the long output expression fit into one line.

```
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 generators>
gap> Dimension( subA );
2
gap> SizeScreen([ 81, ]);;
gap> idA:= Ideal( A, [ m-m^3 ] );
<two-sided ideal in <algebra of dimension 3 over Rationals>, (1 generators)>
gap> SizeScreen([ 80, ]);;
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`). After this we create the factor algebra from the homomorphism by the function `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`.

```
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`. The Lie algebras constructed by this function come with a root system attached.

```
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 ])`.)

```
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.

```
    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 used below computes the unique algebraic conjugate of an element in a quadratic subfield of a cyclotomic field.)

```
    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.

```
    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.

```
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 59. Chapter 60 deals with the functionality for general algebras. Furthermore, concerning special functions for Lie algebras, there is Chapter 61.

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` and `GF`, and they could be passed as arguments to other functions such as `DerivedSubgroup` and `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` and `AsSortedList`. For infinite D, `Enumerator` and `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` and `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`, the situation is different because the union of two groups is in general not a group.

```
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.

```
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.

```
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:

```
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` denotes group generators, `GeneratorsOfMonoid` denotes monoid generators, and so on. The result of `GeneratorsOfVectorSpace` is of course to be understood relative to the field of scalars of the vector space in question.

```
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` and `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.

```
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`.

```
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`.

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

The functions `AsList` and `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 a 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 )`.

```
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 )` 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 12.4. Many other other chapters deal with specific types of domain such as groups, vector spaces or algebras.

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 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 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 is an example of an operation.

```
gap> G := Group( (1,2,3,4,5,6,7,8), (1,2) );;
gap> Size( G ); time > 0;   # the time may of course vary on your machine
40320
true
gap> Size( G ); time;
40320
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:

```
  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`, 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` 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.

```
  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`, 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 2 of "Programming in GAP".

We only present the following rule of thumb here: Each installed method for an attribute, say `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` 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`.

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` with required filter `IsPermGroup`.

```
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

```
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 7.1 of the reference manual 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` 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` 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` to `true` for this group. GAP utilizes this celebrated theorem by having an **immediate method** for `IsSolvableGroup` with required filter `HasSize` which checks parity of the size and either sets `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` $\Rightarrow$ `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 2.6 and 2.7 in "Programming in GAP".

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`, 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` 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 13.1 in the reference manual.

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` 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 $nr - 1$ `TryNextMethod`s) and if *nr* = `"all"`, the sorted list of all applicable methods is returned. For details, see 2.3 in "Programming in GAP".

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

```
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 )`.

```
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: 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`. Also the above-mentioned immediate method for `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.

9

Migrating to GAP 4

This chapter is intended to give users who have experience with GAP 3 some information about what has changed in GAP 4.

In particular, it informs about changed command line options (see 9.1), the new global variable `fail` (see 9.2), some functions that have changed their behaviour (see 9.3) or their names (see 9.4), and some conventions used for variable names (see 9.5).

Then the new concepts of GAP 4 are sketched, first that of mutability or immutability (see 3.3), with the explanation of related changes in functions that copy objects (see 9.7), then the concepts of operations and method selection, which are compared with the use of operations records in GAP 3 (see 9.8, 9.10, and 9.11).

More local changes affect the concepts of notions of generation (see 9.9), of parents (see 9.12), of homomorphisms (see 9.13, 9.14, and 9.15), how elements in finitely presented groups are treated (see 9.16), how information about progress of computations can be obtained (see 9.18), and how one gets information in a `break` loop (see 9.19).

While a "GAP 3 compatibility mode" is provided (see 9.20), its use will disable some of the new features of GAP 4. Also it certainly can only try to provide partial compatibility.

For a detailed explanation of the new features and concepts of GAP 4, see the manual "Programming in GAP".

9.1 Changed Command Line Options

In GAP 4, the `-l` option is used to specify the **root directory** (see 9.2 in the Reference Manual) of the GAP distribution, which is the directory containing the `lib` and `doc` subdirectories. Note that in GAP 3 this option was used to specify the path to the `lib` directory.

The `-h` option of GAP 3 has been removed, the path(s) for the documentation are deduced automatically in GAP 4.

The option `-g` is now used to print information only about full garbage collections. The new option `-g -g` generates information about partial garbage collections too.

9.2 Fail

There is a new global variable

1 ▶ `fail`

in GAP 4. It is intended as a return value of a function for the case that it could not perform its task. For example, `Inverse` returns `fail` if it is called with a singular matrix, and `Position` returns `fail` if the second argument is not contained in the list given as first argument.

GAP 3 handled such situations by either signalling an error, for example if it was asked for the inverse of a singular matrix, or by (mis)using `false` as return value, as in the example `Position`. Note that in the first example, in GAP 3 it was necessary to check the invertibility of a matrix before one could safely ask for its inverse, which meant that roughly the same work was done twice.

9.3 Changed Functionality

Some functions that were already available in GAP 3 behave differently in GAP 4. This section lists them.

1 ▶ `Orbit(` G`,`pnt`)`

The GAP 3 manual promised that pnt would be the first entry of the resulting orbit. This was wrong already there in a few cases, therefore GAP 4 does not promise anything about the ordering of points in an orbit.

2 ▶ `Order(` g `)`

only takes the element g and computes its multiplicative order. Calling `Order` with two arguments is permitted only in the GAP 3 compatibility mode, see 9.20. (Note that it does not make sense anymore to specify a group as first argument w.r.t. which the order of the second argument shall be computed, see 9.16.)

3 ▶ `Position(` $list$`,` obj `)`

If obj is not contained in the list $list$ then `fail` is returned in GAP 4 (see 9.2), whereas `false` was returned in GAP 3.

4 ▶ `PermGroupOps.ElementProperty(` G`,` $prop$`[,` K`] )`

In GAP 3, this function took either two or three arguments, the optional argument K being a subgroup of G that stabilizes $prop$ in the sense that for any element g in G, either all elements or no element in the coset $g * K$ have the property $prop$.

The GAP 4 function `ElementProperty`, however, takes between two and four arguments, and the subgroup K known from GAP 3 has to be entered as the **fourth** argument **not** the third. (The third argument in the GAP 4 function denotes a subgroup U stabilizing $prop$ in the sense that either all elements or no element in right cosets $U * g$ have the property $prop$.)

(This discrepancy was discovered only in March 2002, short before the release of GAP 4.3.)

5 ▶ `Print(` obj`, ... )`

Objects may appear on the screen in a different way, depending on whether they are printed by the read eval print loop or by an explicit call of `Print`. The reason is that the read eval print loop calls the operation `ViewObj` and not `PrintObj`, whereas `Print` calls `PrintObj` for each of its arguments. This permits the installation of methods for printing objects in a short form in the read eval print loop while retaining `Print` to display the object completely. See also Section 6.3 in the Reference Manual.

(`PrintObj` is installed as standard method `ViewObj`, so it is not really necessary to have a `ViewObj` method for an object.)

6 ▶ `PrintTo(` $filename$`,` obj`, ... )`

In GAP 3, `PrintTo` could be (mis)used to "redirect" the text **printed** by a function (that is, **not** only the output of a function) to a file by entering the function call as second argument. This was used mainly in order to avoid many calls of `AppendTo`. In GAP 4, this feature has disappeared. One can use streams (see Chapter 10 in the Reference Manual) instead in order to write files efficiently.

9.4 Changed Variable Names

Some functions have changed their name without changing the functionality. A – probably incomplete – list follows

```
        GAP 3                   GAP 4

        AgGroup                 PcGroup              # (also composita)
        ApplyFunc               CallFuncList
        Backtrace               Where
        CharTable               CharacterTable    # (also composita)
        Denominator             DenominatorRat
        DepthVector             PositionNot
        Elements                AsSSortedList
        IsBijection             IsBijective
        IsFunc                  IsFunction
        IsMat                   IsMatrix
        IsRec                   IsRecord
        IsSet                   IsSSortedList
        LengthWord              Length
        NOfCyc                  Conductor
        Numerator               NumeratorRat
        NormedVector            NormedRowVector
        Operation               Action               # (also composita)
        Order(G,g)              Order(g)
        OrderMat                Order
        OrderPerm               Order
        RandomInvertableMat     RandomInvertibleMat
        RecFields               RecNames
        X                       Indeterminate
```

See Section 9.20 for a way to make the old names available again.

9.5 Naming Conventions

The way functions are named has been unified in GAP 4. This might help to memorize or even guess names of library functions.

If a variable name consists of several words then the first letter of each word is capitalized.

If the first part of the name of a function is a verb then the function may modify its argument(s) but does not return anything, for example `Append` appends the list given as second argument to the list given as first argument. Otherwise the function returns an object without changing the arguments, for example `Concatenation` returns the concatenation of the lists given as arguments.

If the name of a function contains the word `By` then the return value is thought of as built in a certain way from the parts given as arguments. For example, `GroupByGenerators` returns a group built from its group generators, and creating a group as a factor group of a given group by a normal subgroup can be done by taking the image of `NaturalHomomorphismByNormalSubgroup` (see also 9.14). Other examples of "By" functions are `GroupHomomorphismByImages` and `UnivariateLaurentPolynomialByCoefficients`.

If the name of a function contains the word `Of` then the return value is thought of as information deduced from the arguments. Usually such functions are attributes (see 8.1 in this Tutorial and 13.5 in the Reference Manual). Examples are `GeneratorsOfGroup`, which returns a list of generators for the group entered as argument, or `DiagonalOfMat`.

For the setter and tester functions of an attribute *attr* (see 9.8 in this Tutorial and 13.5 in the Reference Manual) the names `Set`*attr* resp. `Has`*attr* are available.

If the name of a function *fun1* ends with `NC` then there is another function *fun2* with the same name except that the `NC` is missing. `NC` stands for "no check". When *fun2* is called then it checks whether its arguments are valid, and if so then it calls *fun1*. The functions `SubgroupNC` and `Subgroup` are a typical example.

The idea is that the possibly time consuming check of the arguments can be omitted if one is sure that they are unnecessary. For example, if an algorithm produces generators of the derived subgroup of a group then it is guaranteed that they lie in the original group; `Subgroup` would check this, and `SubgroupNC` omits the check.

Needless to say, all these rules are not followed slavishly, for example there is one operation `Zero` instead of two operations `ZeroOfElement` and `ZeroOfAdditiveGroup`.

9.6 Immutable Objects

GAP 4 supports "immutable" objects. Such objects cannot be changed, attempting to do so issues an error. Typically attribute values are immutable, and also the results of those binary arithmetic operations where both arguments are immutable, see Section 3.8. For example, `[ 1 .. 100 ] + [ 1 .. 100 ]` is a mutable list and `2 * Immutable( [ 1 .. 100 ] )` is an immutable list, both are equal to the (mutable) list `[ 2, 4 .. 200 ]`.

There is no way to **make** an immutable object mutable, one can only get a mutable copy by `Shallow-Copy`. The other way round, `MakeImmutable` makes a (mutable or immutable) object and all its subobjects immutable; one must be very careful to use `MakeImmutable` only for those objects that are really newly created, for such objects the advantage over `Immutable` is that no copy is made.

More about immutability can be found in Sections 3.3 in this tutorial and 12.6 in the Reference Manual.

9.7 Copy

The function `Copy` of GAP 3 is not supported in GAP 4. This function was used to create a copy *cop* of its argument *obj* with the properties that *cop* and *obj* had no subobjects in common and that if two subobjects of *obj* were identical then also the corresponding subobjects of *cop* were identical.

The possibility of having immutable objects (see 3.3) can and should be used to avoid unnecessary copying. Namely, given an immutable object one needs to copy it only if one wants to get a modified object, and in such a situation usually it is sufficient to use `ShallowCopy`, or at least one knows how deep one must copy in order to do the changes one has in mind.

For example, suppose you have a matrix group, and you want to construct a list of matrices by modifying the group generators. This list of generators is immutable, so you call `ShallowCopy` to get a mutable list that contains the same matrices. If you only want to exchange some of them, or to append some other matrices, this shallow copy is already what you need. So suppose that you are interested in a list of matrices where some rows are also changed. For that, you call `ShallowCopy` for the matrices in question, and you get matrices whose rows can be changed. If you want to change single entries in some rows, `ShallowCopy` must be called to get mutable copies of these rows. Note that in all these situations there is no danger to change, i.e., to destroy the original generators of the matrix group.

If one needs the facility of the `Copy` function of GAP 3 to get a copy with the same structure then one can use the new GAP 4 function `StructuralCopy`. It returns a structural copy that has no **mutable** subobject in common with its argument. So if `StructuralCopy` is called with an immutable object then this object itself is returned, and if `StructuralCopy` is called with a mutable list of immutable objects then a shallow copy of this list is returned.

Note that `ShallowCopy` now is an operation. So if you create your own type of (copyable) objects then you must define what a shallow copy of these objects is, and install an appropriate method.

9.8 Attributes vs. Record Components

In GAP 3, many complex objects were represented via records, for example all domains. Information about these objects was stored in components of these records. For the user, this was usually not relevant, since there were functions for computing information about the objects in question. For example, if one was interested in the size of a group then one could call `Size`.

But since it was guaranteed that the size of a domain D was stored as value of the component `size`, it was allowed to access D`.size` if this component was bound, and a check for this was possible via `IsBound(` D`.size )`.

In GAP 4, only the access via functions is admissible. One reason is the following basic rule.

From the information that a given GAP 4 object is for example a domain, one cannot conclude that this object has a certain representation.

For attributes like `Size`, GAP 4 provides two related functions, the **setter** and the **tester** of the attribute, which can be used to set an attribute value and to check whether the value of an attribute is already stored for an object (see also 13.5 in the Reference Manual). For example, if D is a domain in GAP 4 then `HasSize(` D `)` is `true` if the size of D is already stored, and `false` otherwise. In the latter case, if you know that the size of D is *size* then you may store it by `SetSize(` D*, size* `)`.

Besides the flexibility in the internal representation of objects, storing information only via function calls has also the advantage that GAP 4 is able to draw conclusions automatically. For example, as soon as it is stored that a group is nilpotent, it is also stored that it is solvable, see Chapters 13 in the Reference Manual and 2 in "Programming in GAP" for the details.

As a consequence, you cannot put your favourite information into a domain D by assigning it to a new component like D`.myPrivateInfo`. Instead you can introduce a new attribute and then use its setter, see 13.5 in the Reference Manual.

9.9 Different Notions of Generation

As in GAP 3, a **domain** in GAP 4 is a structured set.

The same set can have different structures, for example a field can be regarded as a ring or as an algebra or vector space over a subfield.

In GAP 3, however, an object representing a ring did not represent a field, and an object representing a field did not represent a ring. One reason for this was that the record component `generators` was used to denote the appropriate generators of the domain. For a ring R, the component R`.generators` was a list of ring generators, and for a field F, F`.generators` was a list of field generators.

GAP 4 cleans this up, see 7.3. It supports many different notions of generation, for example one can ask for magma generators of a group or for generators of a field as an additive group. A subtle but important distinction is that between generators of an algebra and of an algebra-with-one.

So the attributes `GeneratorsOfGroup`, `GeneratorsOfMagma`, `GeneratorsOfRing`, `GeneratorsOfField`, `GeneratorsOfVectorSpace`, and so on, replace the access to the `generators` component.

9.10 Operations Records

Already in GAP 3 there were several functions that were applicable to many different kinds of objects, for example `Size` could be applied to any domain, and the binary infix multiplication `*` could be used to multiply two matrices, an integer with a row vector, or a permutation with a permutation group. This was implemented as follows. Functions like `Size` and `*` tried to find out what situation was described by its arguments, and then it called a more specific function to compute the desired information. These more specific functions, let us call them **methods** as they are also called in GAP 4, were stored in so-called **operations records** of the arguments.

For example, every domain in GAP 3 was represented as a record, and the operations record was stored in the record component `operations`. If `Size` was called for the domain then the method to compute the size of the domain was found as value of the `Size` component of the operations record.

This was fine for functions taking only one argument, and in principle it is possible that for those functions an object stored an optimal method in its operations record. But in the case of more arguments this is not possible. In a multiplication of two objects in GAP 3, one had to choose between the methods stored in the operations records of the arguments, and if for example the method stored for the left operand was called, this method had to handle all possible right operands.

So operations records turned out to be not flexible enough. In GAP 4, operations records are not supported (see 9.20 for a possibility to use your GAP 3 code that utilizes operations records, at least to some extent). A detailed description of the new mechanism to select methods can be found in Chapter 2 in "Programming in GAP".

An important point is that the new mechanism allows GAP to take the relation between arguments into account. So it is possible (and recommended) to install different methods for different relations between the arguments. Note that such methods need not do the extensive argument checking that was necessary in GAP 3, because most of the checks are done already by the method selection mechanism.

9.11 Operations vs. Dispatcher Functions

GAP 3 functions like `Size`, `CommutatorSubgroup`, or `SylowSubgroup` did mainly call an appropriate method (see 9.10) after they had checked their arguments. Such functions were called **dispatchers** in GAP 3. In GAP 4, many dispatchers have been replaced by **operations**, due to the fact that methods are no longer stored in operations records (see 2 in "Programming in GAP" for the details).

Most dispatchers taking only one argument were treated in a special way in GAP 3, they had the additional task of storing computed values and using these values in subsequent calls. For example, the dispatcher `Size` first checked whether the size of the argument was already stored, and if so then this value was returned; otherwise a method was called, the value returned by this method was stored in the argument, and then returned by `Size`.

In GAP 4, computed values of operations that take one argument (these operations are called **attributes**) are also stored, only the mechanism to achieve this has changed, see 13.5 and 13.7 in the Reference Manual.

So the behaviour of `Size` is the same in GAP 3 and GAP 4. But note that in GAP 4, it is not possible to access $D.\mathtt{size}$, see 9.8. As described in 9.10, GAP 4 does not admit "bypassing the dispatcher" by calling for example $D.\mathtt{operations.Size}$. This was done in GAP 3 often for efficiency reasons, but the method selection mechanism of GAP 4 is fast enough to make this unnecessary.

If you had written your own dispatchers and put your own methods into existing operations records then this code will not work in GAP 4. See 3 and 2 in "Programming in GAP" for a description of how to define operations and to install methods.

Finally, some functions in GAP 3 were hidden in operations records, e.g., `PermGroupOps.MovedPoints`. These functions became proper operations in GAP 4.

9.12 Parents and Subgroups

In GAP 3 there was a strict distinction between parent groups and subgroups. The use of the name "parent" (instead of "supergroup") was chosen to indicate that the parent of an object was more than just useful information. In fact the main reason for the introduction of parents was to provide a common roof for example for all groups of polycyclic words that belonged to the same PC-presentation, or for all subgroups of a finitely presented group (see 9.16). A subgroup was never a parent group, and it was possible to create subgroups only of parent groups.

In GAP 4 this common roof is provided already by the concept of **families**, see 13.1 in the Reference Manual. Thus it is no longer compulsory to use parent groups at all. On the other hand, parents **may** be used in GAP 4 to provide information about an object, for example the normalizer of a group in its parent group may be stored as an attribute value. Note that there is no restriction on the supergroup that is set to be the parent, it is possible to create a subgroup of any group, this group then being the parent of the new subgroup. This permits for example chains of subgroups with respective parents, of arbitrary length.

As a consequence, the `Parent` command cannot be used in GAP 4 to test whether the two arguments of `CommutatorSubgroup` fit together, this is now a question that concerns the relation between the families of the groups. So the 2-argument version of `Parent` and the now meaningless function `IsParent` have been abolished.

9.13 Homomorphisms vs. General Mappings

In GAP 3 there had been a confusion between group homomorphisms and general mappings, as `GroupHomomorphismByImages` created only a general mapping that did **not** store whether it was a mapping. This caused expensive, unwanted, and unnecessary tests whether the mapping was in fact a group homomorphism. Moreover, the "official" workaround to set some components of the mapping record was quite unwieldy.

In GAP 4, `GroupHomomorphismByImages` checks whether the desired mapping is indeed a group homomorphism; if so then this property is stored in the returned mapping, otherwise `fail` is returned. If you want to avoid the checks then you can use `GroupHomomorphismByImagesNC`. If you want to check whether a general mapping that respects the group operations is really a group homomorphism, you can construct it via `GroupGeneralMappingByImages` and then call `IsGroupHomomorphism` for it. (Note that `IsGroupHomomorphism` returns `true` if and only if both `IsGroupGeneralMapping` and `IsMapping` do, so one does in fact check `IsMapping` in this case.)

There is **no** function `IsHomomorphism` in GAP 4, since there are several different operations with respect to which a mapping can be a homomorphism.

9.14 Homomorphisms vs. Factor Structures

If F is a factor structure of G, with kernel N, complete information about the connection between F and G is provided by the **natural homomorphism**.

In GAP 3, the "official way" to construct this natural homomorphism was to create first the factor structure F, and then to call `NaturalHomomorphism` with the arguments G and F. For that, the data necessary to compute the homomorphism was stored in F when F was constructed.

In GAP 4, factor structures are not treated in a special way, in particular they do not store information about a homomorphism. Instead, the more natural way is taken to construct the natural homomorphism from G and N by `NaturalHomomorphismByNormalSubgroup` if N is a normal subgroup of the group G, or by `NaturalHomomorphismByIdeal` if N is an ideal in the ring G. The factor F can then be accessed as the image of this homomorphism, and of course G is the preimage and N is the kernel.

Note that GAP 4 does not guarantee anything about the representation of the factor F, it may be a permutation group or a polycyclically presented group or another kind of group. Also note that a natural homomorphism need not be surjective.

A consequence of this change is that GAP 4 does **not** allow you to construct a natural homomorphism from the groups G and F.

The other common type of homomorphism in GAP 3, "operation homomorphisms", have been replaced (just a name change) by **action homomorphisms**, which are handled in a similar fashion. That is, an action homomorphism is constructed from an acting group, an action domain, and a function describing the operation. The permutation group arising by the induced action is then the image of this operation homomorphism.

The GAP 3 function `Operation` is still supported, under the name `Action`, but from the original group and the result of `Action` it is not possible to construct the action homomorphism.

9.15 Isomorphisms vs. Isomorphic Structures

In GAP 3, a different representation of a group could be obtained by calling `AgGroup` to get an isomorphic polycyclically presented group, `PermGroup` to get an isomorphic permutation group, and so on. The returned objects stored an isomorphism in the record component `bijection`.

For the same reason as in 9.14, GAP 4 puts emphasis on the isomorphism, and the isomorphic object in the desired representation can be accessed as its image. So you can call `IsomorphismPcGroup` or `IsomorphismPermGroup` in order to get an isomorphism to a polycyclically presented group or a permutation group, respectively, and then call `Image` to get the isomorphic group.

Note that the image of an action homomorphism with trivial kernel is also an isomorphic permutation group, but an action homomorphism need not be surjective, since it may be easier to define it into the full symmetric group.

Further note that in GAP 3, a usual application of isomorphisms to polycyclically presented groups was to utilize the usually more effective algorithms for solvable groups. However, the new concept of polycyclic generating systems in GAP 4 makes it possible to apply these algorithms to arbitrary solvable groups, independent of the representation. For example, GAP 4 can handle polycyclic generating systems of solvable permutation groups. So in many cases, a change of the representation for efficiency reasons may be not necessary any longer.

In general `IsomorphismFpGroup` will define a presentation on generators chosen by the algorithm. The corresponding elements of the original group can be obtained by the command

```
gens:=List(GeneratorsOfGroup(Image(isofp)),i->PreImagesRepresentative(isofp,i));
```

If a presentation in the given generators is needed, the command `IsomorphismFpGroupByGenerators`(G, *gens*) will produce one.

9.16 Elements of Finitely Presented Groups

Strictly speaking, GAP 3 did not support elements of finitely presented groups. Instead, the "words in abstract generators" of the underlying free groups were (mis)used. This caused problems whenever calculations with elements were involved, the most obvious ones being wrong results of element comparisons. Also functions that should in principle work for any group were not applicable to finitely presented groups. In effect, a finitely presented group had to be treated in a special way in GAP 3.

GAP 4 distinguishes free groups and their elements from finitely presented groups and their elements. Comparing two elements of a finitely presented group will yield either the correct result or no result at all.

Note that in GAP 4, the arithmetic and comparison operations for group elements do not depend on a context provided by a group that contains the elements. In particular, in GAP 4 it is not meaningful to call `Order( `G`, `g` )` for a group G and an element g.

9.17 Polynomials

In GAP 3, polynomials were defined over a field. So a polynomial over `GF(3)` was different from a polynomial over `GF(9)`, even if the coefficients were exactly the same.

GAP 4 defines polynomials only over a characteristic. This makes it possible for example to multiply a polynomial over `GF(3)` with a polynomial over `GF(9)` without the need to convert the former to the larger field.

However it has an effect on the result of `DefaultRing` for polynomials: In GAP 3 the default ring for a polynomial was the polynomial ring of the field over which the polynomial was defined. In GAP 4 no field is associated, so (to avoid having to define the algebraic closure as the only other sensible alternative) the default ring of a polynomial is the `DefaultRing` of its coefficients.

This has an effect on `Factors`: If no ring is given, a polynomial is factorized over its `DefaultRing` and so `Factors(`*poly*`)` might return different results.

To be safe from this problem, if you are not working over prime fields, rather call `Factors(`*pring*`,`*poly*`)` with the appropriate polynomial ring and change your code accordingly.

9.18 The Info Mechanism

Sometimes it is useful to get information about the progress of a calculation. Many GAP functions contain statements to display such information under certain conditions.

In GAP 3, these statements were calls to functions such as `InfoGroup1` or `InfoGroup2`, and if the user assigned `Print` to these variables then this had the effect to switch on the printing of information. `InfoGroup2` was used for more detailed information than `InfoGroup1`. One could switch off the printing again by assigning `Ignore` to the variables, and `Ignore` was also the default value.

GAP 4 uses one function `Info` for the same purpose, which is a function that takes as first argument an **info class** such as `InfoGroup`, as second argument an **info level**, and the print statements as remaining arguments. The level of an info class *class* is set to *level* by calling `SetInfoLevel( `*class*`, `*level*` )`. An `Info` statement is printed only if its second argument is smaller than or equal to the current info level. For example,

```
gap> test:= function( obj )
> Info( InfoGroup, 2, "This is useful, isn't it?" );
> return obj;
> end;;
gap> test( 1 );
1
gap> SetInfoLevel( InfoGroup, 2 );
gap> test( 1 );
#I  This is useful, isn't it?
1
```

As in GAP 3, if an info statement is ignored then its arguments are not evaluated.

9.19 Debugging

If GAP 4 runs into an error or is interrupted, it enters a break loop. The command `Where(` *number* `)`, which replaces `Backtrace` of GAP 3, can be used to display *number* lines of information about the current function call stack.

As in GAP 3, access is only possible to the variables of the current level in the function stack, but in GAP 4 the function `DownEnv`, with a positive or negative integer as argument, permits one to step down or up in the stack.

When interrupting, the first line printed by `Where` actually may be one level higher, as the following example shows

```
gap> OnBreak := function() Where(0); end;; # eliminate back-tracing on
gap>                                        # entry to break loop
gap> test:= function( n )
>    if n > 3 then Error( "!\n" ); fi; test( n+1 ); end;;
gap> test( 1 );
Error, !
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> n;
4
brk> DownEnv();
brk> n;
3
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( 2 );
brk> n;
1
brk> Where();
 called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( -2 );
brk> n;
3
brk> quit;
gap> OnBreak := Where;; # restore OnBreak to its default value
```

For purposes of debugging, it can be helpful sometimes, to see what information is stored within an object. In GAP 3 this was possible using `RecFields` because the objects in question were represented via records. For component objects, GAP 4 permits the same by `NamesOfComponents(` *object* `)`, which will list all components present.

9.20 Compatibility Mode

For users who want to use GAP 3 code with as little changes as possible, a compatibility mode is provided by GAP 4. This mode must be turned on explicitly by the user.

It should be noted that this compatibility mode has not been tested thoroughly.

The compatibility mode can be turned on by loading some of the following files with `ReadLib`. The different files address different aspects of compatibility.

`compat3a.g`

> makes some GAP 3 function names available that were changed in GAP 4, and provides code for some GAP 3 features that were deliberately left out from the GAP 4 library. For example, almost all variable names concerning character theory that are mentioned in the GAP 3 manual, such as `CharTable` and `SubgroupFusions`, are available after `compat3a.g` has been read; the only exceptions are names of operations records.

`compat3b.g`

> implements the availability of "components" of domains; besides components that have no meaning for the rest of the GAP 4 library, such as D`.myInfo`, there are components associated to attributes; for example D`.size` is redirected to the call of the attribute `Size`, `IsBound(` D`.size )` to the call of its tester, and D`.size:=` *val* to the call of its setter. (An important special case is the component `operations`, see below.)

`compat3c.g`

> permits you to implement your own elements represented as records, and using operations records to provide a `Print` method and the basic arithmetic operations. When using operations records, it is probably a good idea to use **immutable** operations records; for example, if the results of arithmetic operations are records with operations records then this avoids to create shallow copies of the operations records in the call to `Immutable` for the results.

The following features are accessible only via starting GAP with the command line option `-O` and may damage some features of GAP 4 permanently for the current session.

With this option, also the files listed above are read automatically.

`compat3d.g`

> provides some GAP 3 functions like `Domain`, simulates the GAP 3 behaviour of `IsString` (to convert a list to string representation if possible), and replaces `fail` by `false`; these changes destroy parts of the functionality of GAP 4.

Some words concerning the simulation of operations records may be necessary.

The operations records of the GAP 3 library, such as `DomainOps` and `GroupOps`, are available **only** for access to their components, whose values are GAP 4 operations; for example, the value of both `DomainOps.Size` and `GroupOps.Size` is the operation `Size`. So it is **not** safely possible to delegate from a `Size` method in another operations record to `DomainOps.Size`. Also it is not possible to change these predefined operations records.

If one wants to install individual methods for a given object *obj* via the mechanism of operations records then one can construct a new operations record with `OperationsRecord`, assign the desired methods to components of this record, and then assign the operations record to *obj*`.operations`. Whenever an operation that is associated with a component *nam* of the operations record is called with *obj* as first argument, the value of *nam* is chosen as the method.

In the case of the binary operations =, <, +, -, *, /, `Comm`, and `LeftQuotient`, this also happens if *obj* is the right-hand argument. As in GAP 3, if both arguments of one of the above binary operations have operations

records containing a function for this operation, then the function in the operations record of the right-hand argument is chosen.

We give a small example how the compatibility mode works.

Suppose we want to deal with new objects that are derived from known field elements by distorting their multiplication. Namely, let a' and b' be the new objects corresponding to the field elements a, b, and define $a' * b' = ab - a - b + 2$.

In GAP 3, this problem was solved by representing each new object by a record that stored the corresponding "old" object and an operations record, where the latter was a record containing the functions applicable to the new object. After the library file `compat3c.g` has been read, we can use this construction of the operations record and of the new objects. Note that operations records must be created with the function `OperationsRecord` (this was also the norm in GAP 3), starting with an empty record would not work. For our intended application, we thus start with the following two lines of code.

```
gap> ReadLib( "compat3c.g" );
gap> MyOps:= OperationsRecord( "MyOps" );;
HasMyOps := NewFilter( "HasMyOps" );
```

In order to make the translation from GAP 3 code to GAP 4 easier, GAP prints the definition of filters associated with operations records and the method installations for operations corresponding to components of the operations records. The output line printed by GAP after the call of `OperationsRecord` is one such case.

Now we add our multiplication function to the operations record, and again GAP 4 prints a translation to GAP 4 code.

```
gap> MyOps.\* := function( a, b )
>        return rec( x:= a.x * b.x - a.x - b.x + 2,
>                     operations := MyOps );
>    end;;
# If the following method installation matches the requirements
# of the operation 'PROD' then 'InstallMethod' should be used.
# It might be useful to replace the rank 'SUM_FLAGS' by '0'.
InstallOtherMethod( PROD,
    "for object with 'MyOps' as first argument",
    true,
    [ HasMyOps, IsObject ], SUM_FLAGS,
    MyOps.\* );

# For binary infix operators, a second method is installed
# for the case that the object with 'MyOps' is the right operand;
# since this case has higher priority in GAP 3, the method is
# installed with higher rank 'SUM_FLAGS + 1'.
InstallOtherMethod( PROD,
    "for object with 'MyOps' as second argument",
    true,
    [ IsObject, HasMyOps ], SUM_FLAGS + 1,
    MyOps.\* );
```

Let us look how this installation works.

```
gap> a:= rec( x:= 3, operations:= MyOps );
rec( x := 3, operations := MyOps )
gap> b:= rec( x:= 5, operations:= MyOps );
rec( x := 5, operations := MyOps )
gap> a * b;
rec( x := 9, operations := MyOps )
```

(In more complicated cases, we might run into problems, but this was already the case in GAP 3. For example, suppose we want to support the multiplication of two operands having different operations records; then it is not clear which of the two multiplication functions is to be chosen, and in GAP 3, the only way out was to change the multiplication functions, in order to make them aware of such situations.)

If we are now interested to translate the code to GAP 4 in the sense that no compatibility mode is needed, we can use what GAP 4 has printed above. (The same example is dealt with in Chapter 6 of "Programming in GAP".)

The objects will no longer be records with `operations` component. Instead of records we may use so-called component objects with record-like access to components, and instead of the `operations` component, we give the objects a type that has the filter `HasMyOps` set.

```
HasMyOps := NewFilter( "HasMyOps" );
MyType := NewType( NewFamily( "MyFamily" ),
                   HasMyOps and IsComponentObjectRep );
```

(More about families and representations in this context can be found in the chapter of "Programming in GAP" mentioned above.)

The next step is to write a function that creates a new object. It may look as follows.

```
MyObject := function( val )
    return Objectify( MyType, rec( x:= val ) );
end;
```

The multiplication function shall return an object with the filter `HasMyOp`, so we change it as follows.

```
gap> MyMult := function( a, b )
>           return MyObject( x:= a!.x * b!.x - a!.x - b!.x + 2 );
>      end;;
```

Note that the component access for these objects works via `!.` instead of `.`; further note that no operations record needs to appear here, the filter takes its role.

Finally, we install the multiplication for at least one argument with the new filter, as had been printed by GAP 4 in the session shown above.

```
InstallOtherMethod( PROD,
    "for object with 'MyOps' as first argument",
    true,
    [ HasMyOps, IsObject ], 0,
    MyMult );

InstallOtherMethod( PROD,
    "for object with 'MyOps' as second argument",
    true,
    [ IsObject, HasMyOps ], 1,
    MyMult );
```

And now the example works (again).

```
gap> a:= MyObject( 3 );
<object>
gap> b:= MyObject( 5 );
<object>
gap> a * b;
<object>
gap> last!.x
9
```

We may install a method to print our objects in a nice way; we could have done this for the operations
record `MyOps` in the compatibility mode, the printed output would look similar to the following.

```
InstallOtherMethod( PRINT_OBJ,
    "for object with 'MyOps' as first argument",
    true,
    [ HasMyOps ], 0,
    function( obj ) Print( "MyObject( ", obj!.x, " )" ); end );
```

Now the example behaves as follows.

```
gap> a;  b;  a * b;
MyObject( 3 )
MyObject( 5 )
MyObject( 9 )
```

Maybe now we want to improve the installation. The multiplication function we want to use is apparently
thought only for the case that **both** operands have the filter `HasMyOps` (and a component `x`). So it is
reasonable to replace the two methods for the multiplication by one method for which both arguments are
required to have the filter.

```
InstallOtherMethod( PROD,
    "for two objects with 'MyOps'",
    true,
    [ HasMyOps, HasMyOps ], 0,
    MyMult );
```

At first sight, the GAP 4 approach seems to be much more complicated. But the last example shows that
in GAP 4, each method can be installed more specifically for the appropriate situation. Moreover, it is for
example possible to install a method for the multiplication of an integer and a `HasMyOps` object; note that
–contrary to the situation in GAP 3– such a method is independent from already existing methods in the
sense that these need not be changed when new functionality is added.

Another example that uses this part of the compatibility mode can be found in the file `tst/compat3.tst`
of the GAP 4 distribution.

Bibliography

[Sim90] Charles C. Sims. Computing the order of a solvable permutation group. *J. Symbolic Comput.*, 9:699–705, 1990.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".