GAP

Release 4.3
05 May 2002

Extending GAP

The GAP Group

http://www.gap-system.org

Acknowledgement

We would like to thank the many people who have made contributions of various kinds to the development of GAP since 1986, in particular:

Isabel M. Araújo, Robert Arthur, Hans Ulrich Besche, Thomas Bischops, Oliver Bonten, Thomas Breuer, Frank Celler, Gene Cooperman, Bettina Eick, Volkmar Felsch, Franz Gähler, Greg Gamble, Willem de Graaf, Burkhard Höfling, Jens Hollmann, Derek Holt, Erzsébet Horváth, Alexander Hulpke, Ansgar Kaup, Susanne Keitemeier, Steve Linton, Frank Lübeck, Bohdan Majewski, Johannes Meier, Thomas Merkwitz, Wolfgang Merkwitz, Jürgen Mnich, Robert F. Morse, Scott Murray, Joachim Neubüser, Max Neunhöffer, Werner Nickel, Alice Niemeyer, Dima Pasechnik, Götz Pfeiffer, Udo Polis, Ferenc Rákóczi, Sarah Rees, Edmund Robertson, Ute Schiffer, Martin Schönert, Ákos Seress, Andrew Solomon, Heiko Theißen, Rob Wainwright, Alex Wegner, Chris Wensley and Charles Wright.

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 (AM)

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

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 (AM), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
Isabel Araújo (AM), Derek Holt (A), Alexander Hulpke (A), Götz Pfeiffer (A), Andrew Solomon (AM)

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

Homomorphism search
Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
Frank Celler (A), Bettina Eick (AM)

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

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

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)

Primitive groups library
Heiko Theißen (A), Alexander Hulpke (M)

Properties and attributes of finite pc groups
Frank Celler (A), Bettina Eick (AM)

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)

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

Subgroup lattice for solvable groups
Alexander Hulpke (AM)

Subgroup presentations
Volkmar Felsch (AM)

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

Tietze transformations
Volkmar Felsch (AM)

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)

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

GAP for MacOS
Burkhard Höfling (AM)

Contents

Copyright Notice

If you obtain GAP please send us a short notice to that effect, e.g., an e-mail message to the address `gap@dcs.st-and.ac.uk`, containing your full name and address. This allows us to keep track of the number of GAP users.

If you publish a mathematical result that was partly obtained using GAP, please cite GAP, just as you would cite another paper that you used (see below for sample citation). Also we would appreciate if you could inform us about such a paper.

Specifically, please refer to

```
[GAP] The GAP Group, GAP --- Groups, Algorithms, and Programming,
        Version 4.3; 2002
 (http://www.gap-system.org)
```

(Should the reference style require full addresses please use: "Centre for Interdisciplinary Research in Computational Algebra, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland; Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany")

If you modify any part of GAP and redistribute it, you must supply a `README` document. This should specify what modifications you made in which files. We do not want to take credit or be blamed for your modifications.

Of course we are interested in all of your modifications. In particular we would like to see bug-fixes, improvements and new functions. So again we would appreciate it if you would inform us about all modifications you make.

1 About: Extending GAP

This is one of four parts of the GAP documentation, the others being the GAP **Tutorial**, a beginner's introduction to GAP, the GAP **Reference Manual**, which contains the official definitions of GAP, and **Programming in** GAP which also provides information for those who want to write their own GAP extensions.

Extending GAP explains how to create files and functions that will work together with mechanisms built in GAP.

This manual is divided into chapters. Each chapter is divided into sections, and within each section, important definitions are numbered. References therefore are triples.

The first chapters of this manual describe how to write documentation, how to interface packages and components, and roughly describes the style used for writing the library. This is followed by chapters that explain advanced programming techniques in GAP. Finally there are chapters (alas, at the moment there is only one due to a lack of manpower) that describe how internal functions work and how to interface ones own code to these internal functions.

Pages are numbered consecutively in each of the four manuals. For manual conventions, see Section 1.1 in the Reference Manual.

2 The gapmacro.tex Manual Format

The current GAP manual books and most of the GAP 4 package documentation is written in a restricted TeX format, using a set of macros defined in the file `GAPPATH/doc/gapmacro.tex`. This chapter describes this format and how to create the final documents (which can be printed or used by GAP's online help) from it.

See 2.5 and 2.7 for details on the restricted set of available TeX commands.

The first sections 2.1 and 2.2 describe the general layout of the files in case you need to write your own package documentation.

If you are planning to write new documentation for a GAP package you can either use the format described in this chapter or use an alternative approach. For example some packages have started to use the GAPDoc package for their documentation, see Chapter 1 in the GAPDoc manual or type

```
gap> ?GAPDoc:chapters
```

in GAP's online help for a table of contents, or (if it is not available in your installation) see:

```
http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/
```

If you want to use yet another document format you must provide certain information to the interface of GAP's online help. This is described in chapter 5.

2.1 The Main File

The main TeX file is called `manual.tex`. This file should contain the following commands:

```
\input ../gapmacro
\Package{package-name}
\BeginningOfBook{name-of-book}
  \UseReferences{book1}
  ...
  \UseReferences{bookn}
  \TitlePage{title}
  \Colophon{text}
  \TableOfContents
  \FrontMatter
    \immediate\write\citeout{\bs bibdata{mybibliography}}
    \Input{file1}
    ...
    \Input{filen}
  \Chapters
    \Input{file1}
    ...
    \Input{filen}
  \Appendices
```

```
    \Input{file1}
    ...
    \Input{filen}
    \Bibliography
    \Index
\EndOfBook
```

Now we describe what these commands do:

`\input` *path*/`gapmacro.tex`
: inputs the GAP "style" and macros file `gapmacro.tex`. If you are writing a GAP package either copy this file or use a relative path. The former method will always work but requires you to keep the file consistent with the system while the latter forces users to change the `manual.tex` file if they are installing a package in a private location. See also Section 9.2 in the Reference Manual.

`\Package{`*package-name*`}`
: defines a macro `\`*package-name* so that when you type `{\`*package-name*`}` (please include the curly braces) the text *package-name* is typeset in the right way for GAP packages, e.g. if you are writing a package MyPackage then you should include the line

```
\Package{MyPackage}
```

in your `manual.tex` file and then in your chapter files use `{\MyPackage}` when you refer to MyPackage by name. There is also the command `\package{`*pkg*`}` when you wish to refer to other GAP packages; don't confuse the two i.e. `\Package{`*package-name*`}` defines a macro `\`*package-name* but produces no text, and `\package{`*pkg*`}` produces *pkg* set in the font that is right for GAP packages.

`\BeginningOfBook{`*name-of-book*`}`
: starts the book *name-of-book*. It is used for cross-references, see 2.4. If you are writing a GAP package use the name of your package here.

`\UseReferences{`*booki*`}`
: If your manual cross-refers to another manual, `\UseReferences` can be used to load the labels of the other books in case cross-references occur. *booki* should be the path of the directory containing the book whose references you want to load. If you are writing a GAP package and you need to reference the main GAP manual, use `\UseReferences` for each book you want to reference. However, as said above this requires changes to the `manual.tex` file if the package is not installed in the standard location.
If your `manual.tex` file lives in `pkg/qwer/doc` and you want to use references to the tutorial use

```
\UseReferences{../../../doc/tut}
```

You may also cross-refer to other package manuals and even GapDoc-produced manuals. Just ensure you get the path to the other manual's directory correct **relative** to the directory in which your manual resides, and if it's a GapDoc-produced manual that you are cross-referring to, use `\UseGapDocReferences` instead of `\UseReferences`.

`\TitlePage`
: produces a page containing the **title**. Please see the example.

`\Colophon`
: `\Colophon` produces a page following the title that can be used for more explicit author information, acknowledgements, dedications or whatsoever.

`\TableOfContents`
: produces a table of contents in double-column format. For short manuals, the double-column format may be inappropriate; in this case, use `\OneColumnTableOfContents` instead.

`\FrontMatter`
: starts the front matter chapters such as a copyright notice or a preface.
The line

`\immediate\write\citeout{\bs bibdata{`*mybibliography*`}}`

: is for users of BibTEX. It will use the file *mybibliography*`.bib` to fetch bibliography information.

`\Chapters`

: starts the chapters of the manual, which are included via `\Input`. `\Input{`*filei*`}` inputs the file *filei*`.tex`, i.e. *filei* should be the name of the file **without** the `.tex` extension. For the chapter format, see Section 2.2.

`\Appendices`

: starts the appendices, i.e. it modifies the `\Chapter` command to use uppercase letters to number chapters.

`\Bibliography`

: produces a bibliography, i.e. it reads and typesets the `manual.bbl` file produced by BibTEX.

`\Index`

: produces an index, i.e. it reads and typesets the `manual.ind` file produced by the external `manualindex` program.

`\EndOfBook`

: Finally `\EndOfBook` closes the book.

Example

Assume you have a GAP package `qwert` with two chapters `Qwert` and `Extending Qwert`, a copyright notice, a preface, no exercises, then your `manual.tex` would basically look like:

```
\input ../../../doc/gapmacro        % The right path from pkg/qwert/doc
\Package{Qwert}                     % Defines macro {\Qwert}
\BeginningOfBook{qwert}
  \TitlePage{
    \centerline{\titlefont Qwert}\medskip          % Package name
    \centerline{\titlefont ---}\medskip
    \centerline{\titlefont A GAP4 Package}\bigskip\bigskip
    \centerline{\secfont Version 1.0}\medskip
    % If the package interfaces with an external program ...
    \centerline{\secfont Based on qwert Standalone Version 3.14}\vfill
    \centerline{\secfont by}\vfill
    \centerline{\secfont Q. Mustermensch}\medskip   % Author
    \centerline{Department of Mathematics}\medskip  % Affiliation
    \centerline{University of Erewhon}\medskip
    \centerline{\secfont email: qmuster@erewhon.uxyz.edu.ut} % Email address
    \vfill
    \centerline{\secfont{\Month} \Year}
  }
  \TableOfContents
  \FrontMatter
    \Input{copyright}
    \Input{preface}
  \Chapters
    \Input{qwert}
    \Input{extend}
  \Appendices
    \Index
\EndOfBook
```

Occasionally there will be the need for additional commands over and above those shown above. The ones described below should be the **only** exceptions.

- There may be other packages that are referred to a lot, so that it's worthwhile to add more `\Package` commands. (There's nothing special about `\Package`, you can use it to define macros for other packages besides the package being documented.)
- Besides the macros `{\Month}` and `{\Year}`, which typeset the current month (as an English word) and the year (all four digits), respectively, there are also `{\Day}` and `{\Today}` which are mainly intended for drafts. `{\Day}` typesets the day of the month as a number and `{\Today}` is equivalent to: `{\Day} {\Month} {\Year}`.
- Sometimes one desires a chapter to be unnumbered in the TeX-produced manuals, e.g. the Tutorial manual has GAP's Copyright Notice as an unnumbered chapter. To achieve this one inputs the file containing the chapter via TeX's `\input` command rather than `\Input`. However, neither the on-line help browser nor the HTML converter "sees" such chapters. Thus if it is desired that the on-line help browser and the HTML manuals should also have such chapters, they must be "input" again via the `\PseudoInput` command (not necessarily in the same manual).
- For chapters that should only appear via the on-line help browser or in the HTML manuals, one may use the `\PseudoInput` command. Any `\PseudoInput` commands should come **after** all `\Input` commands; failure to do this will result in different numbering of `\Input` chapters for TeX-produced and HTML manuals. The syntax of this command is as follows:

`\PseudoInput{`*filename*`}{`*six-entry*`}{`*chaptername*`}`

where *filename* is the name of the file containing the chapter without the `.tex` extension, as for the `\Input` command, *six-entry* is the section-index-entry for the chapter (written to the `manual.six` file) and *chaptername* is the **actual** argument of the `\Chapter` command that appears at the beginning of *filename*`.tex`. The argument *six-entry* enables the on-line text browser to reference the chapter by a name other than *chaptername*. Thus a copyright chapter for the book with name *name-of-book* might have *chaptername* "`Copyright Notice`" but *six-entry* "`Copyright`", which would enable one to access the chapter "`Copyright Notice`" via `?`*name-of-book*`:copyright` via the on-line browser. The HTML converter adds an index entry for both *six-entry* and *chaptername*.

Note

Usage of the commands `\input` and `\PseudoInput` in the way described above will necessitate special treatment of references to such chapters. For such purposes, there is a special variant of the `%display` environment (see 2.11), e.g. a copyright notice appearing via `\input` at the beginning of a TeX-produced manual and appearing in the non-TeX manuals – the on-line help browser or HTML manual – via a `\PseudoInput` command as described above, may be referenced via

```
%display{tex}
See the copyright notice at the beginning of this book.
%display{nontex}
%See "Copyright".
%enddisplay
```

2.2 Chapters and Sections

The contents of each chapter must be in its **own** `.tex` file. The command `\Chapter{`*chaptername*`}` starts a new chapter named *chaptername*; it should constitute the first non-comment (and non-blank) line of the file containing a chapter. A chapter begins with an introduction to the chapter and is followed by sections created with the `\Section{`*secname*`}` command. The strings *chaptername* and *secname* are automatically available as references (see Section 2.4).

There must be **no further commands** on the same line as the `\Chapter` or `\Section` line, and there **must** be an empty line after a `\Chapter` or `\Section` command. This means that `\index` commands referring to the chapter or section can be placed only after this empty line.

Finally, the HTML converter requires that each `\Section` line is preceded by a line starting with at least 16 percentage signs (conventionally, one actually types a full line of percentage signs). The HTML converter stops converting a section whenever it hits such a line; therefore do not add lines starting with 16 or more % signs which are **not** just before a `\Section` command. Failure to include the line of percentage signs before a `\Section` line will cause the converter to crash, due to the discovery of what it sees as two `\Section` commands within the one section.

2.3 Suppressing Indexing and Labelling of a Section and Resolving Label Clashes

Sometimes one does not wish a section to be indexed. To suppress the indexing of a section, simply add the macro `\null` after the `\Section` command, e.g.

`\Section{`*section-name*`}\null`

and then *section-name* will still generate a label (so that you can still refer to it via `Section~"`*section-name*`"`), but *section-name* will not appear in the index.

Occasionally, one has a dedicated section for the description of a single function. If the label generated for the section coincides with the label for a subsection (generated by a `\>` command) a multiply defined label results. In these cases, one would generally rather that the section did not generate a label or an index entry. To suppress the generation of both the label and the index entry of such a section, simply add the macro `\nolabel` immediately after the `\Section` command, e.g. for a section dedicated to the function *func*:

`\Section{`*func*`}\nolabel`

Note: Labels are generated by converting to lowercase and removing whitespace. So coincidences can occur when you might not have expected it. An alternative to index suppression to resolve label clashes is to include a sub-label for the function in the `\>` command (see Section 2.5).

2.4 Labels and References

Each `\Chapter`, `\Section` and `\>` command generates a (short) label *label*, which is extended by *name-of-book* (the argument of `\BeginningOfBook` mentioned earlier in Section 2.1), to create a “long label” *long-label*, and emitted to a file `manual.lab`. The construction of *long-label* is *name-of-book*`:`*label*, where the *label* generated by either of the commands `\Chapter` or `\Section` is just its *chaptername* or *secname* argument. For `\>`, there are a few cases to consider, and we'll consider them in Section 2.5, where we meet the various forms of the `\>` command. To see how to resolve problems with label clashes see Section 2.3.

A reference to a label *any-label* (long or short) is made by enclosing *any-label* in a pair of double quotation marks: `"`*any-label*`"`; it is replaced by the number of the `\Chapter`, `\Section` or `\>` command that generated *any-label* in the first place. Generally, one only needs to make references to long labels when referring to other manuals. For references within the same manual, short labels are sufficient, except when the short label itself contains a colon.

Example

Since the `\BeginningOfBook` command for this manual defines *name-of-book* to be `ext`, the long label for the current section is `ext:Labels and References` and so a reference to this section within this manual might be: `Section "Labels and References"` (which is typeset as: Section 2.4). From another manual, a long label reference is required.

Another example

The first chapter of this manual has the title “About: Extending GAP”, which contains a colon. Hence, to refer to that chapter, one **must** use a long label:

```
Chapter "ext:About: Extending GAP"
```

produces: Chapter 1.

Note

In actual fact long labels are first sanitised by conversion to lower case and removal of superfluous white space (multiple blanks and new lines are converted to a single space). The same sanitisation process is applied to references. Thus,

```
Chapter "ext:about:                                   extending
Gap"
```

also produces: Chapter 1. So, don't worry about references to labels being broken over lines and think of them as being case-insensitive, except that the HTML converter requires that one respects case for the *name-of-book* component of a long label.

2.5 TeX Macros

As the manual pages are also used as on-line help, and are automatically converted to HTML, the use of special TEX commands should be avoided. The following macros can be used to structure the text, the mentioned fonts are used when printing the manual, however the on-line help and HTML are free to use other fonts or even colour. Since, the plain text on-line help, doesn't have special fonts, it leaves in much of the markup, including the left and right quotes that surround something intended to be displayed in typewriter type, the angle brackets that surround something intended to appear in italics, and the dollar-signs enclosing mathematics; you will need to keep that in mind when reading the following section.

‘*text*’

: sets *text* in `typewriter style`. This is typically used to denote GAP keywords such as `for` and `false` or variables that are not arguments to a function, e.g., `‘for’` produces `for`. See also <*text*>. Use `\<` to get a “less than” sign.

‘‘*text*’’

: encloses *text* in double quotes, e.g., `‘‘double-quoted text’’` produces “double-quoted text”. In particular, ‘‘*text*’’ does **not** set ‘*text*’ in typewriter style; use `‘{‘text’}’` to produce `‘text’`. Double quotes are mainly used to mark a phrase which will be defined later or is used in an uncommon way.

`\lq`

: sets a single left quote: ‘. For a phrase *text* that is to be defined later or is used in an uncommon way, please use ‘‘*text*’’ (which encloses *text* in double quotes rather than single quotes).

`\rq`, `\pif`

: each set a single apostrophe (right quote): ’. For the HTML and on-line manuals `\accent19{}` also sets an apostrophe; however the TEX-derived manuals produce an acute-d blankspace (what it in fact is).

`\accent127`

: sets an umlaut, e.g. `\accent127a` produces ä. Do not use the shorthand `\"` (otherwise the HTML converter will not translate it properly).

<*text*>

: sets *text* in italics. This can also be used inside `$...$` and `‘...’`. Use `\<` to get a “less than” sign. `<...>` is used to denote a variable which is an argument of a function; a typical application is the description of a function:

```
\>Group( <gens> ) F
The function ‘Group’ constructs a group generated by <gens>.
```

The `F` at the end of the first line in the above example indicates that `Group` is a function (see the `\>` entry, below).

`*`*text*`*`
: sets *text* in **emphasized style**.

`$a.b$`
: Inside math mode, you can use `.` instead of `\cdot` (a centred multiplication dot). Use `\.` for a full stop inside math mode. For example, `$a.b$` produces $a \cdot b$ while `$a\.b$` produces $a.b$.

`\cite{...}`
: produces a reference to a bibliography entry (the `\cite[...]{...}` option of LaTeX is **not** supported).

`"`*label*`"`
: produces a reference to *label*. Labels are generated by the commands `\Chapter`, `\Section` (see 2.4), and `\>` commands (see below).

`\index{`*index-entry*`}`
: defines an index entry *index-entry*. Besides appearing in the index, *index-entry* is also written to the section index file `manual.six` used by the on-line help. An exclamation mark (`!`), if present, is used to partition *index-entry* into main entry (left part) and subentry (right part) components, in the index. TeX converts *index-entry* to lowercase and sets it in roman type, in the index. The HTML converter respects case and uses the default font, in producing the HTML manual index. *index-entry* must be completely free of special characters and font changing commands; if you need special fonts, characters or commands use one of `\indextt` or `\atindex`.
: Note that for the HTML converter to process indexing commands (`\index`, `\indextt` and `\atindex`) correctly they **must** be on lines of their own. There can be several indexing commands on the same line, but there should be no horizontal whitespace before each indexing command, and if an indexing command needs to be broken over lines place a `%` at the point of the break at the end of the line to mark a "continuation".
: For the HTML converter it works best to put indexing commands all together at the beginning of a paragraph, rather than strewn between lines of a paragraph. However, for the TeX-produced manuals after a maths display one gets a rogue space if you do this (this is a bug); you can work around the bug by putting at least one word of the paragraph followed by your line(s) of indexing commands.
: **Note** also that indexing commands do **not** produce labels for cross-references; they **only** produce entries for the index. Labels are **only** produced by the chapter (`\Chapter`), section (`\Section`) and subsection (`\>`) commands.

`\indextt{`*index-entry*`}`
: is the same as `\index{`*index-entry*`}`, except that *index-entry* is set by TeX in typewriter style, respecting case; the HTML converter sets *index-entry* in the default font. Again, *index-entry* should be completely free of special characters and font changing commands, and `!` may be used for subentries in the same way as for `\index`. Note that a sub-entry component, if present, is **not** set in typewriter style for the TeX-produced manuals; if you want that it is, use `\atindex`.

`\atindex{`*sort-entry*`}{|indexit}`
: is simply a special form of the `\index` command that tells TeX to typeset the page number in italics.

`\atindex{`*sort-entry*`}{@`*index-entry*`}`
: The HTML converter treats this command as if it was `\index{`*index-entry*`}`, except that it strips out any font information and sets it in the default font, but nevertheless respects case. *index-entry* may have `|indexit` at the end which is ignored by the HTML converter.
: The TeX-produced manuals set the index entry as *index-entry* respecting font and case, and list it according to *sort-entry*. If a sub-entry is required then it should be present behind a `!` in **both** the *sort-entry* and *index-entry*; the only difference between the sub-entry in *sort-entry* and that in *index-entry*, is that the *sort-entry* sub-entry should be stripped of mark-up and font changing command. The *index-entry* component is ignored when constructing the `manual.six` files, and is

also ignored by the HTML converter. Anything after an `!` in *sort-entry* is ignored when constructing the `manual.idx` file that is processed by MakeIndex. Macros like `{\GAP}` are allowed in *index-entry*. However, any `‘` that appears in *index-entry* **must** be preceded by `\noexpand`; *sort-entry* must be completely free of special characters and font changing commands.

In general, one should make *sort-entry* the same as *index-entry* modulo fonts and other mark-up, e.g.,

```
\atindex{Fred!Nerk}{@\noexpand‘Fred’!\noexpand‘Nerk’}
```

`{\GAP}`

typesets GAP.

`\package{`*pkg*`}`

typesets *pkg* in the font correct for GAP packages (respecting case). This is intended for cross-referencing other GAP packages. There is also the command `\Package{`*mypkg*`}` command which defines a macro `\`*mypkg* so that when you type `{\`*mypkg*`}` (please include the curly braces) the text *mypkg* is typeset in the right way for GAP packages. The `\Package` command should normally be included in one's `manual.tex` file (see 2.1) and just allows one to type `{\`*mypkg*`}` rather than the longer `\Package{`*mypkg*`}` as one is frequently likely to do when formulating one's own GAP package documentation. So, just to be clear about the difference between `\Package` and `\package`, `\Package{`*mypkg*`}` defines a macro `\`*mypkg* but produces no text, and `\package{`*pkg*`}` produces *pkg* set in the font that is right for GAP packages.

`\>`

produces a subsection. The line following the `\>` entry must either contain another `\>` entry (in which case the further entries are assumed to be variants and do not start a new subsection) or must be empty. The description text will follow this empty line.

There are several forms of the `\>` command. In all forms, a label and index entry are generated; the HTML converter uses the label to form an index entry, respecting case and setting in roman type. If the next non-space character is not a left quote (`‘`) it is assumed that the subsection is for a “function”; we exhibit these forms first.

`\>`*func*

While this form is supported; it is discouraged. If *func* is a 0-argument function, *func* should be followed by an empty pair of brackets (see `\>`*func*`(`*args*`)` below). If *func* is actually a global variable then `\>‘`*global-var*`’ V` should be used instead (see below). In order for this form to be parsed correctly the remainder of the line to the right of *func* must be empty. It generates *func* as both a label and index entry; *func* appears as is, in typewriter type in the TEX-derived manual index.

`\>`*func*`(`*args*`)`

The macro uses the brackets after *func* to parse the arguments *args*. Thus, it is necessary for the function to use brackets and for the arguments to have none. (We use the term “function” loosely here to mean “a GAP command with arguments”; we really mean an object that GAP knows as a: “Function”, “Property”, “Operation”, “Category”, or “Representation” — but not “Variable”, since a “Variable” does not have arguments.) The label and index entry generated consists of the text between the `>` and opening bracket. The index entry is set as is (i.e. without conversion to lowercase) in typewriter type in the TEX-derived manual index. Here is an example of how to use `\>`; the index entry is “`Size`” (in typewriter type, with mixed case preserved).

```
\>Size( <obj> ) A
```

The `A` indicates that `Size` is an “Attribute”. Instead of `A` there can be `F`, `P`, `O`, `C`, or `R` which indicate that a command is a “Function” (probably the most common), “Property”, “Operation”, “Category”, or “Representation”, respectively. For the forms of the `\>` command followed by a left quote, `V` indicating “Variable” (an object without arguments), is also possible. (See Section 1.1 and Chapter 13 in the reference manual).

\>*func*(*args*)!{*sub-entry*}

This is a special form of the previous command, that forms a label *func*!*sub-entry* and an index entry with main entry *func* (set in typewriter type and respecting case) and sub-entry *sub-entry* (set in roman type but also respecting case).

The remaining forms of the command \> expect to be followed by a ‘.

\>‘*command*’{*label*}

works as \> without ‘...’, but will not use bracket matching; it simply displays *command* as a header, which appears in typewriter type. It will use *label* as both the label and index entry, and the index entry is set in roman type. Whenever *label* contains a !, it is used to partition *label* into main entry (left part) and subentry (right part) components, in the index.

```
\>‘<a> + <b>’{addition}
\>‘Size( <obj> )’{size} A
```

In the first of the examples immediately above, the first form of \> cannot be used because no brackets occur. Also, observe that there is no command type (it's not appropriate here); TEX does not need it to correctly parse a \> entry, in general. The second example differs from our previous Size example, in that the index entry will be typeset as “size” (in roman type), rather than “Size”. Also, the index entry is always converted to lowercase, no matter what case or mixed case was used.

\>‘*command*’{*label*}!{*sub-entry*}

is equivalent to: \>‘*command*’{*label*!*sub-entry*}.

\>‘*command*’{*label*}@{*index-entry*}

works as \>‘*command*’{*label*}, except that it uses *label* for sorting the index entry and the index entry itself is printed as *index-entry*. References to the subsection use *label*. (Note that the HTML converter ignores everything after an @ symbol in these commands, essentially treating the command as if it were \>‘*command*’{*label*}. However, the HTML converter also always preserves the case in a label.) Here are two examples.

```
\>‘Size( <obj> )’{size}@{‘Size’} A
\>‘Size( GL( <n>, <q> ) )’{Size!GL( n, q )}@{‘Size’! ‘GL’( \noexpand<n>, \noexpand<q> )} A
```

The first of these examples is equivalent to “\>Size(<obj>)”. For the second example, it was necessary to use ‘ and ’, since the argument contained brackets. Note that \noexpand is needed before < here, but not needed before ‘ in the *index-entry* argument. Otherwise, the rules for sub-entries are the same as for \atindex.

\>‘*global-var*’ V

This is actually a short-hand for: “\>‘*global-var*’{*global-var*}@{‘*global-var*’} V” to save you some typing when creating subsections for global variables, i.e., *global-var* is the label and the index entry appears in typewriter type, with mixed case preserved.

\){\fmark ...}

is like \> except that it produces no label and index entry. It is \fmark that produces the filled in right arrow. Omitting it produces a line in typewriter type.

\){\kernttindent ...}

is useful for producing a line in typewriter type, that you might otherwise have typed between \begintt and \endtt, but where you actually want the TEX macros and variables <...> to be interpreted.

\URL{*url*}

prints the WWW URL *url*. In the HTML version this will be a HREF link.

\Mailto{*email*}

prints the email address *email*. In the HTML version this will be a mailto link.

Note: When a TeX macro is followed by a space, TeX generally swallows up the space; one way, and it is the GAP-preferred way, of preventing the space being swallowed up, is by enclosing the macro in `{...}`. When a macro is often followed by a space, it's a good habit to get into to **always** enclose that macro in `{...}` (the braces do nothing when the macro is not followed by a space, and prevent TeX from swallowing up the space, otherwise). Thus the macro for GAP should **always** be typed `{\GAP}`. Similarly, macros like `\lq`, `\rq` and `\pif` should probably always appear in braces; moreover the word "don't" typeset via "`don{\pif}t`" will actually be interpreted correctly by the on-line browser.

2.6 TeX Macros for Domains

The following macros are required for the following common domains:

`\N` the natural numbers (you should probably indicate whether by your convention $\mathbb{N}$ includes zero or not, when using this);

`\Z` the integers;

`\Q` the rational numbers;

`\R` the real numbers;

`\C` the complex numbers;

`\F` a field; and

`\calR` a general domain e.g. a ring.

2.7 Examples, Lists, and Verbatim

In order to produce a list of items with descriptions use the `\beginitems`, `\enditems` environment, i.e. this is a "description" environment in the parlance of LaTeX and HTML.

For example, the following list describes `base`, `knownBase`, and `reduced`. The different item/description pairs must be separated by blank lines.

```
\beginitems
`base' &
    must be a  list of points ...

`knownBase' &
    If a base for <G> is known in advance ...

`reduced' (default `true') &
    If this is `true' the resulting stabilizer chain will be ...
\enditems
```

This will be printed as

`base`
: must be a list of points ...

`knownBase`
: If a base for G is known in advance ...

`reduced` (default `true`)
: If this is `true` the resulting stabilizer chain will be ...

In order to produce a list in a more compact format, use the `\beginlist`, `\endlist` environment.

An example is the following list.

```
\beginlist
\item{(a)}
    first entry
\item{(b)}
    second entry
\itemitem{--}
    a sub-item of the second entry
\itemitem{--}
    another sub-item of the second entry
\item{(c)}
    third entry
\endlist
```

It is printed as follows.

(a) first entry

(b) second entry

  – a sub-item of the second entry

  – another sub-item of the second entry

(c) third entry

The above example will take advantage of the ordered and unordered list environments in the HTML version, with the addition of slightly more mark-up. First, we present the example again with that additional mark-up, and then we explain how it works.

```
\beginlist%ordered{a}
\item{(a)}
    first entry
\item{(b)}
    second entry
\itemitem{--}%unordered
    a sub-item of the second entry
\itemitem{--}
    another sub-item of the second entry
\item{(c)}
    third entry
\endlist
```

It is printed as follows (of course, you should see no difference in the TeX-produced and on-line versions of this manual).

(a) first entry

(b) second entry

  – a sub-item of the second entry

  – another sub-item of the second entry

(c) third entry

In the HTML version the above example is interpreted as a nested list. The outer list is interpreted as an **ordered** list. The HTML standard provides 5 different types of ordered list, and these mirror the types provided by the `enumerate` LaTeX package. To signify that the outer list was **ordered** the comment

`%ordered` was added after `\beginlist`. If there is no further markup the list is numbered in the default manner, namely with integers. Otherwise, following `%ordered` there should be one of the following:

- `{1}` indicates the list should be numbered with integers (the default obtained when there is nothing following `%ordered`);
- `{a}` indicates the list should be numbered with lowercase letters (`a`, `b`, ...);
- `{A}` indicates the list should be numbered with uppercase letters (`A`, `B`, ...);
- `{i}` indicates the list should be numbered with lowercase roman numerals (`i`, `ii`, ...); and finally
- `{I}` indicates the list should be numbered with uppercase roman numerals (`I`, `II`, ...).

The `\beginlist` of the above example was followed by `%ordered{a}` and so the list is numbered using lowercase letters in the HTML version and using the ordered list environment (rather than the description environment).

Occasionally, it is necessary to break from a list, add some explanatory text and then restart the list, and resume numbering the items from where you left off. To do this follow the comment mark-up already mentioned by an **integer** in curly braces, i.e. if the outer list should actually start at `c` then you would need to have `%ordered{a}{3}` after `\beginlist` because `c` is the 3rd letter of our alphabet. Note that, for an integer-numbered list not starting at 1, you must have the full markup; you cannot omit the `{1}` after `%ordered` in this case.

The inner list of the above example is an **unordered** list (this corresponds to a plain `itemize` environment in LaTeX). To indicate this the first `\itemitem` above was followed by `%unordered`.

Of course, to get an unordered outer list, one would start the list with `\beginlist%ordered`, and to get an ordered inner list numbered with lowercase letters the first `\itemitem` would need to be followed by `%ordered{a}`, i.e. the same syntax is used for the comment added after a `\beginlist` and after the first `\itemitem` in a sequence of `\itemitem`s.

Notes

1. Only lists to a maximum depth of two are supported.
2. You cannot change the type of a sublist halfway through. Only the comment after the first `\itemitem` in a sequence is interpreted.

There are two types of **verbatim** environments. Example GAP sessions are typeset in typewriter style using the `\beginexample`, `\endexample` environment.

```
\beginexample
gap> 1+2;
3
\endexample
```

typesets the example

```
gap> 1+2;
3
```

Examples whose output may vary should use the macro `|unstableoutput`, e.g.

```
\beginexample|unstableoutput
gap> Exec("date");
Sun Oct  7 16:23:45 CEST 2001
\endexample
```

typesets in all manual versions in the same way:

```
gap> Exec("date");
Sun Oct  7 16:23:45 CEST 2001
```

but the `testexample` routine knows to treat the example differently (namely, it ensures there is output but does not insist on it being the same).

Non-GAP examples are typeset in typewriter style using the `\begintt`, `\endtt` environment.

Notes

1. The manual style will automatically indent examples. It also will break examples which become too long to fit on one page. If you want to encourage breaks at specific points in an example, end the example with `\endexample` and immediately start a new example environment with `\beginexample` on the next line.
2. To typeset a pipe symbol `|` in the `\begintt`, `\endtt` environment or `\beginexample`, `\endexample` you need to actually type `||`.

2.8 Tables, Displayed Mathematics and Mathematics Alignments

Tables should normally be set using the `\begintt`, `\endtt` environment. This means that one should enter the appropriate white space so that columns line up. Note that to get a vertical line `|` in the `\begintt`, `\endtt` environment one must actually type `||`. The reason for setting tables this way is so that both the HTML converter and GAP's built-in text browser have no trouble in displaying them correctly.

The HTML converter when used with its `-t` option (which causes it to use TtH to translate mathematics) usually does a reasonable job of converting mathematics displays and mathematics alignments. To help GAP's built-in text browser, however, one should follow a few rules:

– Place the `$$`s that begin and end the mathematics display on lines of their own. (If you don't do this it will be displayed in the same way as ordinary in-line mathematics.)

– Use only the `\matrix{ .. }` environment for mathematics alignments. The `\matrix{` starting the alignment should be on a line on its own, (flush left and no trailing whitespace). The `}` closing the environment should also be on a line of its own. The built-in browser doesn't do anything special to line things up; you must insert the whitespace where it's needed. Any `\hfill` macros you add to help the line things up in the TeX and HTML formats is ignored by the GAP's built-in text browser. The `\matrix{ .. }` environment should be used even when one might like to use TeX's `\cases{ .. }` environment.

The following example shows a typical usage of the `\matrix{ .. }` environment (in particular, it shows how one can use it to avoid using the `\cases{ .. }` environment). Observe, how sufficient whitespace has been added in order that alignment is maintained by GAP's built-in text browser. (Recall that `\right.` which produces nothing is required to match `\left\{`.)

```
From a theorem of Gauss we know that
$$
b_N = \left\{
\matrix{
\frac{1}{2}(-1+\sqrt{N})    &{\rm if} &N \equiv 1  &\pmod 4\cr
\frac{1}{2}(-1+i \sqrt{N})  &{\rm if} &N \equiv -1 &\pmod 4\cr
}
\right.
$$
```

The example produces ...

From a theorem of Gauss we know that

$$b_N = \begin{cases} \frac{1}{2}(-1+\sqrt{N}) & \text{if} \quad N \equiv 1 \pmod 4 \\ \frac{1}{2}(-1+i\sqrt{N}) & \text{if} \quad N \equiv -1 \pmod 4 \end{cases}$$

2.9 Testing the Examples

For purposes of automatically checking the manual, the GAP examples in one chapter (the text between `\beginexample` and `\endexample`) should produce the same output, up to line breaks and whitespace, whenever they are run in the same order immediately after starting GAP (this will ensure that the global random number generator is initialized to the **same** values). For more details, see the last paragraph of 2.1 in the Tutorial.

To permit this automatic running, examples that shall produce error messages should be put between `\begintt` and `\endtt` such that they will not be seen by this automatic test.

The automatic test also requires that examples are not indented in the files; in the printed manual, the lines between `\beginexample` and `\endexample` and the lines between `\begintt` and `\endtt` are automatically indented.

2.10 Usage of the Percent Symbol

The `%` symbol has a number of very specific uses. Take care that you use it correctly. These uses are:

1. A line **beginning** with 16 (or more) `%` symbols marks the **end** of a section, or the **end** of a chapter introduction (which may be empty). Such a line **must** precede **every** `\Section` (see 2.2).
2. A `%` at the beginning of a line tells TeX that the line is a comment and is to be ignored by TeX, **except** in the verbatim environments: `\begintt..\endtt` and `\beginexample..\endexample`. However, `%display` or `%enddisplay` commands have special meaning for the on-line text help browser and for the HTML converter and may temporarily alter the meaning of an initial `%` for these (see 2.11 for details); otherwise the meaning of an initial `%` is the same as for TeX.
3. A `%` at the end of a line marks a "continuation", **except** in the situation mentioned in item 4. A "continuation" may be needed for lines of indexing commands (`\index`, `\indextt` or `\atindex`). Such commands **must** occur on lines of their own (see 2.5), **not** mixed with text, and there must not be any superfluous whitespace (modulo the next statement). Occasionally an indexing command is too long to easily fit on a line; this is where a continuation is desirable; a `%` at the end of such a line indicates that the line is to be joined with the next line after removal of the `%` symbol and any initial whitespace on the next line (this is what TeX does! ... and we mimic this behaviour for both the on-line text help browser and the HTML manuals).

   A "continuation" may also be necessary for subsections, i.e. lines beginning with `\>` or `\)` (again see 2.5); the usage is as for indexing line continuations.
4. A line ending with a `%` that is not an indexing command line or a subsection line that after any initial whitespace is removed matches **exactly** `{%` or `}%`, or begins with `{\` or `\` and is followed by a letter, is ignored by both the on-line browser and the HTML converter. This is intended to screen the on-line browser and HTML converter from TeX commands such as `\obeylines`, `\begingroup`, `\def` etc., without having to resort to using the `%display{tex}..%enddisplay` environment.

Warning. In view of items 3. and 4. above, avoid using a `%` at the end of a line unless you really need it, and it fits into those categories. In particular, do **not** put a `%` at the end of an indexing command line that is immediately followed by a line of text; otherwise, the text line will not appear in the HTML manual or on-line via the text help browser. Similarly, do not put a `%` line at the end of a text line that is immediately followed by an indexing command line; this causes the indexing command line to be ignored by the HTML converter. For the HTML converter it works best to put indexing commands all together at the beginning of a paragraph, rather than strewn between lines of a paragraph. However, for the TeX-produced manuals after a maths display one gets a rogue space if you do this (this is a bug); you can work around the bug by putting at least one word of the paragraph followed by your lines(s) of indexing commands.

2.11 Catering for Plain Text and HTML Formats

As described in 2.5, the use of macros should be restricted to the ones given in the previous sections. By doing so, you should find that the documentation you write will still look ok in GAP's on-line help (plain text format) and in the translated HTML. However, in rare situations one might be forced to use other TEX macros, for example in order to typeset a lattice. In this case you should provide an alternative for the on-line help, and possibly also for the HTML version. This can be done by putting in guiding commands as TEX comments:

```
%display{tex}
TeX version (only used by TeX manual)
%display{html}
%HTML version (only used by HTML manual)
%display{text}
%Text version (only used by the built-in manual browser)
%enddisplay
```

Observe that the lines that should appear only in the TEX-produced manuals do not begin with a `%`. For the HTML (resp. text) version the lines begin with a `%`; each line of a `%display{html}` (resp. `%display{text}`) environment is printed verbatim, after removing the initial `%` symbol. The above example produces:

TeX version (only used by TeX manual)

(Note the above example will vary according to whether you are viewing it as a TEX-produced manual, or as an HTML manual, or via the built-in manual browser — as it should!)

Sometimes one needs a `%display` environment to be not seen by TEX, but still interpreted normally (i.e. not printed verbatim). The following variant of the above provides this capability.

```
%display{tex}
TeX version (only used by TeX manual)
%display{nontex}
%HTML and Text version (interpreted normally, after removing the \% symbol)
%enddisplay
```

The above example produces:

TeX version (only used by TeX manual)

It is permissible to abbreviate any of the above by omitting `%display{tex}`, `%display{html}`, or `%display{text}` if that portion of the environment would be empty.

There are yet two more variants of conditional display. Firstly,

```
%display{nonhtml}
%Text version (interpreted normally by built-in browser, after removing the
%\% symbol)
%enddisplay
```

is normally used to ensure text only appears via the on-line help browser. If there is no initial `%` it also appears in the TEX-produced manuals. The above example produces:

Finally, there is

```
%display{nontext}
%HTML version (interpreted normally by HTML converter, after removing the
%\% symbol)
%enddisplay
```

which excludes text from the on-line help browser. Like the `%display{nonhtml}` environment, if there is no initial `%` it also appears in the TEX-produced manuals. The example produces:

However, the use of these special environments should be avoided as much as possible, since it is much more difficult to maintain such pseudo-duplicated documentation.

2.12 Umlauts

To produce umlauts, use `\accent127` and not the shorthand `\"` (otherwise the HTML converter will not translate it properly).

2.13 Producing a Manual

To produce a manual you will need the following files:

`manual.tex`
: contains the body of the manual (as described in Section 2.1) and an `\Input` command for each chapter/appendix file.

file1`.tex`, *file2*`.tex`, ...
: the chapter/appendix files. There must be one file for each chapter or appendix, and each such file should have a `\Chapter` or `\PreliminaryChapter` command. Alternatively, one can write `.msk` files and use `buildman.pe` to generate the corresponding `.tex` files (see 2.14).

`gapmacro.tex`
: contains the macros for the manual. It must be input by an `\input` statement (**not** and `\Input` statement, which creates a Table of Contents entry) in `manual.tex`. You can either use the version in the `doc` directory of GAP (use a relative path then) or make a copy.

`manual.mst`
: is a "configure" file used by `makeindex` when processing index information in a TeX-generated and `manualindex`-preprocessed `manual.idx` file. It must reside in your manual directory.

`GAPDOCPATH/manualindex`
: is used to call `makeindex`. `GAPDOCPATH` is the path of the `doc` directory of your GAP distribution.

For bibliography information you will need a file `manual.bbl`. If you intend to create it with BibTeX, you will need to indicate the appropriate `.bib` file (as described in section 2.1). Then after running TeX once over the manual, run BibTeX to create the `manual.bbl` file.

Assuming that all necessary files are there (a `manual.lab` file for each *book* argument of a `\UseReferences` command, `mrabbrev.bib` and `manualindex` in the GAP `doc` directory), on a Unix system the following calls will then produce a file `manual.dvi` as well as a file `manual.six` which is used by the GAP help functions. If you are missing some of the needed files and don't have `CVS` access to GAP, just send an email request for them to `gap-trouble@dcs.st-and.ac.uk`.

Go to the directory holding the manual. Call

```
tex manual
```

to produce bibliography information. Unless you provide a `manual.bbl` file which is not produced by BibTeX, call

```
bibtex manual
```

to produce the `manual.bbl` file. Then run TeX twice over the manual to fill all references and produce a stable table of contents:

```
tex manual
tex manual
```

If you have sections which are named like commands, you may get messages about redefined labels. At this point you can ignore these.

Now it is time to produce the index. Call

```
GAPDOCPATH/manualindex manual
```

which preprocesses the `manual.idx` file and then runs `makeindex`. Provided that `manual.mst` exists, this produces a file `manual.ind`. Finally, once again run

```
tex manual
```

to incorporate the index. The manual is ready.

2.14 Using buildman.pe

Rather than write the chapter/appendix `.tex` files directly, one may incorporate one's documentation in comments in one's GAP code. To do it this way, there are four ingredients:

`.gd` files
: GAP files with `.gd` suffixes that have the documentation in comments (actually files with `.g` or `.gi` or any other extension are also possible, but files with extension `.gd` are the default);

`.msk` files
: which are just like the `.tex` files, and must obey all the rules given for `.tex` files previously, but additionally may have `\FileHeader` or `\Declaration` commands at places where text should be inserted from a `.gd` file, and with `{{`*variable*`}}` patterns which are replaced by *replacement* when written to the `.tex` file, if the configuration file *configfile* has a line of form: *variable*=*replacement*;

configfile
: a file which defines `msfiles` (the list of `.msk` files), `gdfiles` (the list of `.gd` files), `LIB` (the directory containing the `.gd` files), `DIR` (the directory in which to put the constructed `.tex` files, one `.tex` file for each `.msk` file), and optionally a line `check` (see below) and *variable*=*replacement* lines; and

`buildman.pe`
: a perl program (in the `etc` directory for those with `CVS` access to GAP), which strips the comments from the `.gd` files according to the `\FileHeader` or `\Declaration` commands in the `.msk` files, translates any `{{`*variable*`}}` patterns defined by the file *configfile* and constructs the `.tex` files.

If you don't have `CVS` access and want to use `buildman.pe`, just email `gap-trouble@dcs.st-and.ac.uk` and ask for it. Please note that there is no obligation for package authors to `buildman.pe`; nor does it attract the same level of support as the rest of GAP; in general, bugs can be expected to be fixed (eventually), but no new features will be added. Also, note that the GAPDoc package provides a similar facility.

The perl program `buildman.pe` is called as follows:

`buildman.pe -f` *configfile*

The form of *configfile*

There is no restriction on how to name *configfile*, but by convention it is of form `config.`*something* or `buildman.config`; *configfile* should contain lines of form:

`msfiles=`*msfile1*`,`*msfile2*`,...,`*msfilem*`;`
`gdfiles=`*gdfile1*`,`*gdfile2*`,...,`*gdfilen*`;`
`LIB=`*gdfile_dir*`;`
`DIR=`*TeX_dir*`;`

Optionally, as mentioned above, one may also have:

```
check;
```

which says to construct a `notfound` file that lists missing expected data, and any number of lines of form

variable=*replacement*

The file *configfile* should obey the following syntactic rules:

- After `msfiles=` there should be a comma-separated and semicolon-terminated list of `.msk` files with the `.msk` extensions removed; `buildman.pe` assumes that the `.msk` files are all in, or at least have path relative to, the directory in which `buildman.pe` is called.
- Similar to the `msfiles` definition, after `gdfiles=` there should be a comma-separated and semicolon-terminated list of "`.gd`" files. If a "`.gd`" file really does have a `.gd` extension, it may be listed without extension; otherwise the extension **must** be included. All the "`.gd`" files must be listed with path relative to the directory defined by `LIB`.
- For both the `msfiles` and `gdfiles` definitions, the lists following the `=` may continue over several lines if necessary, and any whitespace, parentheses (round brackets) or double-quotes characters are ignored.
- The paths after `LIB=` and `DIR=` are assumed relative to the "current directory", i.e. the directory in which `buildman.pe` is executed. For each *msfilei* listed after the `msfiles` keyword, `buildman.pe` constructs from *msfilei*`.msk` a corresponding *msfilei*`.tex` in *TeX_dir*. The `LIB` and `DIR` definitions must be on a single line.
- The terminating `;` is optional on the lines containing the keywords `LIB`, `DIR` or `check`.
- Superfluous characters around any of the keywords `msfiles`, `gdfiles`, `LIB`, `DIR` or `check`, but before the `=` on the lines where `=` is required, are ignored. Whitespace and double-quotes characters are ignored, everywhere.
- The *variable*`=`*replacement* lines (if there are any) should have no other punctuation or whitespace. These lines direct `buildman.pe` to replace any string of form `{{`*variable*`}}` in a `.msk` file with *replacement*.

Special `.msk` file commands

Now we describe the special (non-TEX) commands that direct `buildman.pe` to extract text from "`.gd`" files.

`\FileHeader[`*n*`]{`*gdfile*`}`

This command is replaced by the text following a `#`*n* line (for positive integer *n*) in file *gdfile*`.gd` (or *gdfile* if *gdfile* already contains a suffix). The argument `[`*n*`]` of `\FileHeader` is optional; if it is omitted *n* is taken to be 1. See below for the typical form of a fileheader extracted by the `\FileHeader` command; the comments in the example describe its required format.

`\Declaration{`*func*`}[`*gdfile*`]{`*label*`}!{`*sub-entry*`}@{`*index-entry*`}`

This command is replaced by a `\>` subsection declaration or block of `\>` declarations, and their description extracted from a block in a "`.gd`" file that starts with a line matching `#`*X* *func*, for some letter *X* in `F`, `M`, `A`, `P`, `O`, `C`, `R` or `V`. The line "matches" if there is a `(`, space, or newline after *func*. The argument *func* (in `{..}`) is the only mandatory argument.

If present, `[`*gdfile*`]`, says that *func* is to be found in the file *gdfile*`.gd` (or *gdfile* if *gdfile* already contains a suffix); it is required only if *func* appears in more than one of the "`.gd`" files listed in the file *configfile*. The *gdfile* argument is typically required for distinguishing methods of operations.

The remaining arguments (if present) have exactly the purpose that they have in subsection declarations, i.e. lines of the following forms:

`\>`*func*`!{`*sub-entry*`}`
`\>‘`*command*`’{`*label*`}`
`\>‘`*command*`’!{`*sub-entry*`}`
`\>‘`*command*`’@{`*index-entry*`}`

(see Section 2.5), and are used to build subsection declaration lines of these forms. Note that the *label*, *sub-entry* and *index-entry* arguments, if needed, should follow the `\Declaration` command (and **not** be in the "`.gd`" file `#`*X* *func*`...` lines, where they will be indistinguishable from comments). If in the "`.gd`" file the `#`*X* *func* line is followed by other `#`*Xi* *funci* lines, then each `\>` subsection declaration formed has the same *label*, *sub-entry* and *index-entry* arguments appended.

Corresponding to `\FileHeader[`*n*`]{`*gdfile*`}`, in the "`.gd`" file denoted by *gdfile*, there should be:

```
#n
##  Text for \FileHeader[n]{gdfile}. Each line
##  should have two # characters followed by 2 blank
##  space characters at the left margin. The text
##  can and should include any necessary {\TeX}
##  mark-up and indexing commands.
##
##  A fileheader may consist of any number of paragraphs.
##  It is terminated by a totally empty line (i.e.~a
##  line devoid even of # characters).
##
```

Corresponding to each `\Declaration{`*func*`}...` line of a `.msk` file there should be in one of the "`.gd`" files, a block of form:

```
#X   func( args ) comment
#Y   func2( args2 ) comment2
.
.
#Z   funcn( argsn ) commentn
##
##  description of func, func2, ..., funcn.
##
Declare...( "func" ...);
Declare...( "func2" ...);
.
.
Declare...( "funcn" ...);
```

The above block should comply with the following syntactic rules. Below we use the term "function" in a general sense to mean any one of function (in the strict sense), attribute, category, method, representation, operation, property or variable.

- $X, Y, \ldots, Z \in \{$`A`, `C`, `F`, `M`, `R`, `O`, `P`, `V`$\}$. If the letter is `V` then no parentheses or arguments should follow the "function name" *funci*.
- The letters, X, Y, ..., Z are printed in the manual. If a letter is `A` or `P`, then also the letters `S` and `T` are printed if the setter and the tester are available. If the letter is `A`, then the letter `M` is printed if the attribute is mutable.
- The comments *comment*, *comment2*, ..., *commentn* (by convention starting with spaced dots) which do not appear in the manual, are optional.
- The X, Y, ..., Z "function name" lines must appear on consecutive lines, i.e. not intermingled with text lines.
- After the "function name" lines there should be text lines describing the "functions". As with fileheader text these text lines should contain any TeX mark-up and indexing commands that are necessary, and there should be two blank space characters between the `##` and the text. Lines starting with `#T` (or some other non-`#` character in place of `T`) are ignored.
- It is assumed that for each "function name" *func*, *func2*, ..., *funcn* there is a corresponding GAP declaration (which need not be via a `Declare...` command, e.g. it might be `BindGlobal`) after the `##` text lines (and comment lines), **and** that they appear in the **same** order.

Example

The file `lib/algebra.gd` contains the following declaration:

```
#############################################################################
##
#O  DirectSumOfAlgebras( <A1>, <A2> )
#O  DirectSumOfAlgebras( <list> )
##
##  is the direct sum of the two algebras <A1> and <A2> respectively of the
##  algebras in the list <list>.
##
##  If all involved algebras are associative algebras then the result is also
##  known to be associative.
##  If all involved algebras are Lie algebras then the result is also known
##  to be a Lie algebra.
##
##  All involved algebras must have the same left acting domain.
##
##  The default case is that the result is a structure constants algebra.
##  If all involved algebras are matrix algebras, and either both are Lie
##  algebras or both are associative then the result is again a
##  matrix algebra of the appropriate type.
##
DeclareOperation( "DirectSumOfAlgebras", [ IsDenseList ] );
```

The file `doc/build/algebra.msk` contains the line:

```
\Declaration{DirectSumOfAlgebras}
```

The “config” file `doc/build/config.alg`:

```
@msfiles = ("algebra","algfp","alglie","mgmring");
@gdfiles = ("algebra","alghom","alglie","object","liefam","mgmring","algrep",
            "lierep");
DIR = "../ref";
LIB = "../../lib";
```

specifies `algebra.msk` via the first entry of `msfiles` and `lib/algebra.gd` via the first entry of `gdfiles` and (its directory by) the definition of `LIB`. Observe that there are `@` and `"` symbols, as well as parentheses and whitespace, in the above “config” file; **none** of these is necessary, but they don’t do any harm either. Generally, one calls `buildman.pe` in the same directory that contains the `msfiles` (which is why one doesn’t need to specify the directory containing the `msfiles`) and the “config” file. Since `DIR = "../ref"`, `buildman.pe` constructs `algebra.tex` from `algebra.msk` in directory `doc/ref`. The subsection generated in `algebra.tex` by the above `\Declaration` command starts with the header:

```
\>DirectSumOfAlgebras( <A1>, <A2> ) O
\>DirectSumOfAlgebras( <list> ) O
```

and is followed by its description, i.e. the lines beginning with two hashes and two blanks, but with the hashes and blanks stripped away, so that when it is processed the resulting subsection appears as:

▶ `DirectSumOfAlgebras(` *A1* `,` *A2* `)` O
▶ `DirectSumOfAlgebras(` *list* `)` O

is the direct sum of the two algebras *A1* and *A2* respectively of the algebras in the list *list*.

If all involved algebras are associative algebras then the result is also known to be associative. If all involved algebras are Lie algebras then the result is also known to be a Lie algebra.

All involved algebras must have the same left acting domain.

The default case is that the result is a structure constants algebra. If all involved algebras are matrix algebras, and either both are Lie algebras or both are associative then the result is again a matrix algebra of the appropriate type.

Variable replacement

As mentioned above the "config" file may also contain lines that assign variables, e.g.

```
versionnumber=4.3
versionsuffix=4r3
```

Occurrences of these variables in double curly braces will be replaced by their value. For example the lines

```
When 'unzoo -x' is applied to {\GAP}~{{versionnumber}}'s 'zoo' file
'gap{{versionsuffix}}.zoo' a directory 'gap{{versionsuffix}}' is formed.
```

in a `.msk` file will be replaced by:

```
When 'unzoo -x' is applied to {\GAP}~4.3's 'zoo' file
'gap4r3.zoo' a directory 'gap4r3' is formed.
```

in the corresponding `.tex` file. This feature is very handy for information that changes over time.

Final note

There is a document for version `0.0` of `buildman.pe` that describes features that have either never been used or have since been disabled. Only the features described in this section can be relied upon to have currency.

3 Library Files

This chapter describes some of the conventions used in the GAP library files. These conventions are intended as a help on how to read library files and how to find information in them. So everybody is recommended to follow these conventions, although they do not prescribe a compulsory programming style – GAP itself will not bother with the formatting of files.

Filenames have traditionally GAP adhered to the 8+3 convention (to make it possible to use the same filenames even on a MS-DOS file system) and been in lower case (systems that do not recognize lower case in file names will convert them automatically to upper case). It is no longer so important to adhere to these conventions, but at the very least filenames should adhere to a 16+5 convention, and be distinct even after identifying upper and lower case. Directory names of packages, however, **must** be in lower case (the `RequirePackage` command (see 74.3.1 in the Reference manual) assumes this).

3.1 File Types

The GAP library consists of the following types of files, distinguished by their suffixes:

`.g`
: Files which contain parts of the "inner workings" of GAP. These files usually do not contain mathematical functionality, except for providing links to kernel functions.

`.gd`
: Declaration files. These files contain declarations of all categories, attributes, operations, and global functions. These files also contain the operation definitions in comments.

`.gi`
: Implementation files. These files contain all installations of methods and global functions. Usually declarations of representations are also considered to be part of the implementation and are therefore found in the `.gi` files.
As a rule of thumb, all `.gd` files are read in before the `.gi` files are read. Therefore a `.gi` file usually may use any operation or global function (it has been declared before), and no care has to be taken towards the order in which the `.gi` files are read.

`.co`
: Completion files. They are used only to speed up loading (see 3.5 in the Reference Manual).

3.2 File Structure

Every file starts with a header that lists the filename, copyright, a short description of the file contents and the original authors of this file.

This is followed by a revision entry:

```
Revision.file_suf :=
    "@(#)$Id: libform.tex,v 4.12 2001/12/03 07:35:26 gap Exp $";
```

where `file.suf` is the file name. The revision control system used for the development will automatically append text to the string "`Id: `" which indicates the version number. The reason for these revision entries is to give the possibility to check from within GAP for revision numbers of a file. (Do not mistake these revision numbers for the version number of GAP itself.)

Global comments usually are indented by two hash marks and two blanks. If a section of such a comment is introduced by a line containing a hash mark and a number it will be used for the manual (stripped of the hash marks and leading two blanks; see Section 2.14).

Every declaration or method or function installation which is not only of local scope is introduced by a function header of the following type.

```
#############################################################################
##
#X  ExampleFunction(<A>,<B>)
##
##  This function does nothing.
```

The *X* in the example is one of the letters: `F`, `A`, `P`, `O`, `C`, `R` or `V`, and has the same meaning as at the end of a declaration line in the Reference Manual (see 1.1 in the Reference Manual); it indicates whether the object declared will be a function, attribute, property, operation, category, representation or variable, respectively. Additionally `M` is used in `.gi` files for method installations. The line then gives a sample usage of the function. This is followed by a comment which describes the identifier. This description will automatically be extracted to build the Reference Manual source, if there is a `\Declaration` line in some `.msk` file together with an appropriate configuration file (see Section 2.14).

Indentation in functions and the use of decorative spaces in the code are left to the decision of the authors of each file.

The file ends with an

```
#E
```

comment section that may be used to store formatting descriptions for an editor.

3.3 Finding Implementations in the Library

There is no general browsing tool that would point you to the place in the library where a certain method or global function is installed. However the following remarks might be of help:

You can use `ApplicableMethod` (see 7.2.1 in the reference manual) to get the function which implements a method for specific arguments. Setting its print level higher will also give you the installation string for this method.

To find the occurrence of functions and methods in the library, one can use the `grep` tool under UNIX. To find a function, search for the function name in the `.gd` files (as it is declared only once, only one file will show up), the function installation is likely to occur in the corresponding `.gi` file.

To find a method search for `Method(` (this catches `InstallMethod` and `InstallOtherMethod`) and the installation string or the operation name.

3.4 Undocumented Variables

For several global variables in GAP, no information is available via the help system (see Section 2.7 in the Tutorial, for a quick overview of the help system, or Chapter 2 in the reference manual, for details). There are various reasons for “hiding” a variable from the user; namely, the variable may be regarded as of minor importance (for example, it may be a function called by documented GAP functions that first compute many input parameters for the undocumented function), or it belongs to a part of GAP that is still experimental in the sense that the meaning of the variable has not yet been fixed or even that it is not clear whether the variable will vanish in a more developed version.

As a consequence, it is dangerous to use undocumented variables because they are not guaranteed to exist or to behave the same in future versions of GAP.

Conversely, for **documented** variables, the definitions in the GAP manual can be relied on for future GAP versions (unless they turn out to be erroneous); if the GAP developers find that some piece of minor, but documented functionality is an insurmountable obstacle to important developments, they may make the smallest possible incompatible change to the functionality at the time of a major release. However, in any such case it will be announced clearly in the GAP Forum what has been changed and why.

So on the one hand, the developers of GAP want to keep the freedom of changing undocumented GAP code. On the other hand, users may be interested in using undocumented variables.

In this case, whenever you write GAP code involving undocumented variables, and want to make sure that this code will work in future versions of GAP, you may ask at `gap-trouble@dcs.st-and.ac.uk` for documentation about the variables in question. The GAP developers then decide whether these variables shall be documented or not, and if yes, what the definitions shall be.

In the former case, the new documentation is added to the GAP manual, this means that from then on, this definition is protected against changes.

In the latter case (which may occur for example if the variables in question are still experimental), you may add the current values of these variables to your private code if you want to be sure that nothing will be broken later due to changes in GAP.

4 Writing a GAP Package

This chapter explains the basics of how to write a GAP package so that it interfaces properly to GAP. For the role of GAP packages and the ways to load them, see Chapter 74 in the Reference Manual.

4.1 The Structure of a GAP Package

Every GAP package is in a subdirectory of the directory `pkg` which itself is a subdirectory of a GAP root directory (see Section 9.2 in the Reference Manual). This directory is called the home directory of the GAP package. **The name of this directory (modulo case) is the name of the package.** (The package name may have mixed case; but the directory must be **all** lower case.)

The home directory of the GAP package must contain a file `init.g`. This file contains the necessary GAP code to set up all that the package needs. Section 4.2 explains how such an `init` file must look. If declaration and implementation are separated (see 4.9) or if completion (see 4.10) is used there also will be a second file `read.g` which reads in further files.)

In addition, the home directory can contain other files and subdirectories. It is usually a good idea to follow the general example of the subdirectory structure of the GAP root directory. Although this is not strictly necessary, it makes it easier to look through a GAP package and some administrative functions for GAP packages rely on it.

Typically there are at least two subdirectories called `lib` and `doc`. The first contains files with GAP code and corresponds to the GAP library directory. The second contains the manual and documentation of the package.

If the GAP package has stand-alone programs, it should have a subdirectory called `bin` that contains the executables. This directory is subdivided into several others. Refer to section 4.4 on stand-alone programs for a more detailed explanation. In this case there might also be a directory called `src` containing the source code of the stand-alone programs and instructions for their compilation.

4.2 The Init File of a GAP Package

The initialization file of a GAP package is a file `init.g` in the home directory of the package. This file tells GAP about the version number of the GAP package, about the documentation and about the actual code. It is a proper GAP file (so syntax and comments are standard), but only the following GAP functions may be called (if further functions are called the loading process, which proceeds in two stages, may not work):

The first command in the file will be a call to `DeclarePackage` (see 74.4.1 in the Reference Manual). This indicates the package's version number and permits to test whether necessary external binaries or other GAP packages are installed. If instead `DeclareAutoPackage` (see 74.4.1 in the Reference Manual) is called, the package will be loaded automatically when GAP has started. (To be exact: GAP loads the library, then reads the users `.gaprc` or `gap.rc` file and then loads the GAP packages which are intended for automatic loading and which are not yet loaded via the users `.gaprc` or `gap.rc` file.)

At the moment automatic loading is only available for the packages listed in the file `pkg/ALLPKG`. This is due to the fact that there is no standard C-Function that will list the contents of a subdirectory.

The next command will be a call to `DeclarePackageDocumentation` (see 74.4.2 in the Reference Manual) to indicate the location of the packages documentation to GAP. If instead `DeclarePackageAutoDocumentation` (see 74.4.2 in the Reference Manual) is called, the documentation will be read automatically when GAP starts (however `RequirePackage` still has to be used to load the actual package in case the first command in the file is `DeclarePackage`).

If the package requires other GAP packages (see 4.8) this is followed by calls to `RequirePackage`.

Finally, calls to `ReadPkg` (see 74.4.3 in the Reference Manual) will read files which contain the actual GAP functions the package provides.

To separate declaration and implementation parts, for larger packages it is recommended to only load the declaration part from within `init.g` and to use the separate `read.g` file to load the implementation part. See sections 4.9 and 4.10 for details.

For example this is the `init.g` file of a simple "test" GAP package:

```
# announce the package version
DeclarePackage("test","1.0",ReturnTrue);

# install the documentation
DeclarePackageDocumentation( "test", "doc" );

# read the actual code.
ReadPkg( "test", "gap/code.g");
```

This file installs the GAP package "test" in version 1.0. There are no requirements that have to be tested, so `ReturnTrue` (see 5.3.1 in the Reference Manual) is used as test function. It then installs documentation for the package, which can be found in the `doc` subdirectory. Finally, it reads the file `code.g` in the `gap` subdirectory, which contains the actual code.

If the package author wants the GAP package to display a separate banner the printing of the banner must be issued from within a file which will be read via `ReadPkg` from the `init.g` file. If the package is intended for automatic loading we however recommend not to print any separate banner as the user will get a list of the loaded GAP packages at the end of the initialization process.

A GAP package which has been loaded once cannot be unloaded and also cannot be loaded again. However, as with any file individual files may be "reread".

1 ► `RereadPkg(` *pkg*`,` *file* `)` F
► `RereadPkg(` *pkg-file* `)` F

In the first form, `RereadPkg` rereads the file *file* of the GAP package *pkg*, where *file* is given as a relative path to the directory of *pkg*. In the second form where only one argument *pkg-file* is given, *pkg-file* should be the complete path of a file relative to a `pkg` subdirectory of a GAP root path (see 9.2 in the Reference Manual). Each of *pkg*, *file* and *pkg-file* should be a string.

When writing (and debugging) a GAP package it may be therefore helpful to use `RereadPkg` on single files or to use a separate "reader" file which does not use the `RequirePackage` mechanism.

4.3 An Example of a GAP Package

We illustrate the creation of a GAP package by an example of a basic package. (The following assumes that you are using GAP under UNIX.)

Create the following directories in your home area: `pkg` and `pkg/test`. Inside the directory `test` create the file `init.g` with the lines

```
DeclarePackage("test","1.0",ReturnTrue);
ReadPkg( "test", "banner.g");
```

and create a file `banner.g` containing the single line

```
Print( "#I reading the GAP package ‘‘test’’\n" );
```

The next bit is a bit tricky because you have to find out what the root directories of GAP on your system are. This is done by starting GAP and looking at the variable `GAP_ROOT_PATHS`. This a list of directories which GAP searches upon startup for things like the GAP library.

```
gap> GAP_ROOT_PATHS;
[ "/gap/4.0/" ]
```

Now start GAP with the command

```
gap -l "./;/gap/4.0/"
```

The string between the pair of double quotes gives the components of `GAP_ROOT_PATHS` separated by semicolons. We have added at the beginning the string `./` denoting the current directory. This adds the current directory to the list of GAP root directories. Now you can load your GAP package `test`:

```
gap> RequirePackage("test");
#I reading the GAP package ‘‘test’’
true
```

This GAP package is too simple to be useful, but we have succeeded in loading it via `RequirePackage()`.

4.4 Standalone Programs in a GAP Package

GAP packages that involve stand-alone programs are fundamentally different from GAP packages that consist entirely of GAP code.

This difference is threefold: A user who installs the GAP package must also compile (or install) the packages binaries, the package must check whether the binaries are indeed available and finally the GAP code of the package has to start the external binary and to communicate with it. We will treat these three points in the following sections.

If the package does not solely consist of an interface to an external binary and if the external program called is not just special-purpose code, but a generally available program, chances are high that sooner or later other GAP packages might also require this program.

We therefore strongly suggest to provide a documented GAP function that will call the external binary. We also suggest to create actually two GAP packages; the first providing only the binary and the interface and the second (requiring the first, see 4.8) being the actual GAP package.

4.5 Installation of GAP Package Binaries

The scheme for the installation of package binaries which is described further on is intended to permit the installation on different architectures which share a common file system (and share the architecture independent file).

A GAP package which includes external binaries contains a `bin` subdirectory. This subdirectory in turn contains subdirectories for the different architectures on which the GAP package binaries are installed. The names of these directories must be the same as the names of the architecture dependent subdirectories of the main `bin` directory. Unless you use a tool like `autoconf` yourself, you must obtain the correct name of the binary directory from the main GAP branch. To help with this, the main GAP directory contains a file `sysinfo.gap` which assigns the shell variable `GAParch` to the proper name as determined by GAP's `configure` process. For example on a Linux system, the file `sysinfo.gap` may look like this:

```
GAParch=i586-unknown-linux2.0.31-gcc
```

We suggest that your GAP package contains a file `configure` which is called with the path of the GAP root directory as parameter. This file then will read `sysinfo.gap` and set up everything for compiling under the given architecture (for example creating a `Makefile` from `Makefile.in`.

The standard GAP distribution contains a GAP package "example" whose installation script shows an example way of how to do this.

4.6 Test for the Existence of GAP Package Binaries

If an external binary is essential for the workings of a GAP package, the test function called from `DeclarePackage` should test whether the program has been compiled on the architecture (and inhibit package loading if this is not the case). This is especially important if the package is loaded automatically.

The easiest way to accomplish this is to use `Filename` (see 9.4.1 in the Reference Manual) for checking for the actual binaries in the path given by `DirectoriesPackagePrograms` (see 74.4.7 in the Reference Manual) for the respective package. For example the "example" GAP package uses the following commands to test whether the binary `hello` has been compiled, it issues a warning if not and will only load if it is indeed available.

```
DeclarePackage("example","1.0",
  function()
  local path,file;
    # test for existence of the compiled binary
    path:=DirectoriesPackagePrograms("example");
    file:=Filename(path,"hello");
    if file=fail then
      Info(InfoWarning,1,
        "Package ``example'': The program `hello' is not compiled");
    fi;
    return file<>fail;
  end);
```

You might also have to cope with the situation that external binaries will only run under UNIX (and not, say on a Macintosh). See section 73.15 in the reference manual for information on how to test for the architecture.

4.7 Calling of and Communication with External Binaries

There are two reasons for this: the input data has to be passed on to the stand-alone program and the stand-alone program has to be started from within GAP.

There are two principal ways of doing this.

The first possibility is to write all the data for the stand-alone to one or several files, then start the stand-alone which then writes the output data to file and finally read in the standalone's output file.

The second way is interfacing via iostreams (see Section 10.8 in the Reference Manual). The support for this is in its infancy.

4.8 Requesting one GAP Package from within Another

It is possible for one GAP package to require another. In principle this can be simply done by calling `RequirePackage` from the `init` file of the one package. However, the other package might be essential and we might want to fail loading if the other package is not available. To achieve this, the *tester* function in `DeclarePackage` can call `TestPackageAvailability` (see 74.4.4) and check, whether it returns `fail` or not.

(If the other GAP package is not compulsory, it is also possible to leave out the `TestPackageAvailability` command and only to call `RequirePackage` later in the `init.g` file. In this case the other GAP package will be loaded if available but if this is not the case this will not inhibit loading of the first package. See 4.12.)

Even if we tested the availability of another package this way, we still need to eventually load the other package using `RequirePackage`. This might initially look like having to do the same work twice. The reason of this split between testing availability and package loading is to ensure the possibility of two GAP packages compulsory requesting each other.

For example, the `init.g` file of a GAP package "test", which requires the GAP package "example" in at least version 2.3 would look like:

```
DeclarePackage("test","1.0",
  function()
  local a;
    a:=TestPackageAvailability("example","2.3");
    if a=fail then
      Info(InfoWarning,1,"required GAP package ‘‘example’’ not available");
    fi;
    return a<>fail;
  end
  );

DeclarePackageDocumentation( "test", "doc" );

RequirePackage("example","2.3");

ReadPkg( "test", "gap/read.g");
```

If the GAP package "example" was not available (or only available in an older version), the installation of the package "test" would fail, no code would be read in and no documentation would be installed.

4.9 Declaration and Implementation Part

When GAP packages require each other in a circular way, a "bootstrapping" problem arises of defining functions before they are called. The same problem occurs in the library, it is resolved there by separating declarations (which define categories, operations etc.) and implementations (which install methods) in different files. An implementation file may use declarations defined in any declaration file. GAP initially reads all declaration files (in the library they have a `.gd` suffix) and afterwards reads all implementation files (which have a `.gi` suffix).

Something similar is possible for GAP packages: if a file `read.g` exists in the same place as `init.g`, this `read.g` file is read only **after** all the `init.g` files of all (implicitly) required GAP packages are read. Therefore it is possible to separate declaration and implementation for a GAP package in the same way as done for the library by creating such a file `read.g`, and restricting the `ReadPkg` statements in `init.g` to only load those files of the package which provide declaration and to load the implementation files from `read.g`.

See Section 3.16 in the Programmers' Tutorial which discusses further the commands that should appear in the declaration part (in `init.g`) and the implementation part (in `read.g`) of a package.

4.10 Package Completion

Reading a larger package can take a few moments and will take up user workspace. This might be a nuisance to users, especially if the package is loaded automatically. The same problem of course affects the library, the problem there is solved using completion files (see 3.5 in the reference manual).

Completion files make it possible to read only short function headers initially which are completed to full functions only if the functions are actually called. This section explains how to set up completion for a GAP package:

Completion works for those files which are read (using `ReadPkg`) from the `read.g` file. (This is no real restriction as completion affects only the implementation part.) To create completion files, load the GAP package. Then use the following command.

1 ► `CreateCompletionFilesPkg(`*pkgname*`)`

This will create a new file `read.co` in the GAP package's home directory *pkgname* (so you must have write permissions there). When reading the GAP package this file is used in place of `read.g` if it exists and automatically takes care of completion.

When you change files which are completed, GAP will complain about non-matching CRC files and will not load them. In this case remove the `read.co` file and create it anew.

As a GAP package author you should consider including a completion file with the package.

If you start GAP with the command line option `-D`, it displays information about reading and completion, the command line option `-N` turns completion off (as if all `.co` files were erased). (Section 3.2 in the Reference Manual describes the options `-D` and `-N`.)

4.11 Version Numbers

A version number is a string which contains nonnegative integers separated by non-numeric characters. Examples of valid version numbers are for example:

```
"1.0"   "3.141.59"  "2-7-8.3" "5 release 2 patchlevel 666"
```

Version numbers are interpreted as lists of integers and are compared in that way. Thus version `"2-3"` is larger than version `"2-2-5"` but smaller than `"11.0"`.

It is possible for code to require GAP packages in certain versions. In this case, all versions, whose number is equal or larger than the requested number are acceptable. It is the task of the package author to provide upwards compatibility.

The global variable `VERSION` contains the version number of the version of GAP and also can be checked against (using `CompareVersionNumbers`, see 74.4.8 in the Reference Manual).

4.12 Testing for a GAP Package

There are two ways in which one might want to test for a GAP package.

The first is whether a package is available for loading or has been loaded. This can be done via the function `TestPackageAvailability` (see 74.4.4 in the Reference Manual).

In order to test whether a GAP package *name* has been loaded already, you can check the global variable `LOADED_PACKAGES` for a record component `LOADED_PACKAGES.(`*name*`)`.

4.13 Writing Documentation

There are now two recognised ways of producing GAP package documentation. There is the method that has been used to produce the main manuals for GAP which requires the documentation to be written in TeX according to the format described in chapter 2. There is now also an XML-based documentation method that uses the GAPDoc package (see 1). Documentation written according to the format described in chapter 2 must include a `manual.tex` file to actually run TeX over the manual, the separate TeX files for the manual chapters and the file `manual.six`, produced by running TeX. Whatever system that is used to produce the documentation, there must be a `manual.six` file, since this file is used by the online help to get the index information. If you choose to use a method other than the two methods described above then you need to read chapter 5 to find out how to provide the necessary help handler functions and `manual.six` file. By the way, this does not mean that it is in anyway acceptable to provide documentation in just MSWord `.doc` format. Whatever, system you use to produce documentation, the output format must be one or more of plain text, TeX `dvi`, PostScript, PDF, or HTML, and preferably at least both plain text and TeX `dvi`. It may also be helpful to add a preTeXed `dvi` or `ps` file of the package's manual.

The main directory of the GAP package should also contain a file `INSTALL` or `README` that briefly tells the user what the package does and (if applicable) how to compile and install binaries.

4.14 Wrapping Up a GAP Package

The releases of GAP packages are independent of releases of GAP. Therefore GAP packages should be wrapped up in separate files that can be installed onto any version of GAP. Similarly a GAP package can be upgraded any time without the need to wait for new releases of GAP.

Because it is independent of the version of GAP a GAP package should be archived from the GAP `pkg` directory, that is all files are archived with the path starting the package's name.

The archive of a GAP package should contain all files necessary for the package to work. This includes a `manual.six` file in the documentation subdirectory which is created by TeXing the documentation. If the package provides a substantial amount of code, especially if it is intended to be loaded automatically, create a completion file (see 4.10) and include it with the package.

We use the `zoo` archiver to provide GAP archives and we ask you to use this format for your GAP packages. If `zoo` is not installed on your system, you can get source code for example at

`ftp://ftp.tex.ac.uk/pub/archive/tools/zoo`

or try a search engine like Google

`http://www.google.com`

with the keywords: `zoo archiving software`.

`zoo` by itself archives all files as binary files. This may lead to minor problems when viewing files on DOS or Macintosh systems, which use a different textfile format (CRLF versus LF). To overcome such problems we use the following mechanism for the GAP archive files, which is supported by the `unzoo` we are providing (but is unfortunately not part of the standard `zoo` format):

All files are archived as Unix versions. (If you are using a Windows PC or a Macintosh the easiest way to achieve this is to ftp the files in text mode to a Unix machine and do the archiving there.)

When using `zoo` one can add comments to files. Whenever a file gets a comment `!TEXT!` this file will be considered as text (respectively one can enforce binary mode by a `!BINARY` comment) and `unzoo` will extract it accordingly, adding a CR on non-Unix systems.

If you are using the standard UNIX version of `zoo` the command

```
zoo c archive.zoo filename
```

will prompt you for a comment to add to the file *filename* in the archive `archive.zoo`. You will enter for example

```
!TEXT!
/END
```

Normally, all files of a GAP package except `dvi` files or special binary files are text files.

If you are unsure about the format, you can use the "list" feature of `zoo` on an existing package (for example `XGAP`) to see all files with comments:

```
zoo lc xgap.zoo
```

5 Interface to the GAP Help System

In this chapter we describe which information the help system needs about a manual book and how to tell it this information. The code which implements this interface can be found in `lib/helpbase.gi`.

If you are intending to use a documentation format that is already used by some other help book you probably don't need to know anything from this chapter. However, if you want to create a new format and make it available to GAP then hopefully you will find the necessary information here.

The basic idea of the help system is as follows: One tells GAP a directory which contains a file `manual.six`, see 5.1. When the GAP help is asked something about this book it reads in some basic information from the file `manual.six`: strings like section headers, function names, and index entries to be searched by the online help; information about the available formats of this book like text, html, dvi, and pdf; the actual files containing the documentation, corresponding section numbers, and page numbers: and so on. See 5.2 for a description of the format of the `manual.six` file.

It turns out that there is almost no restriction on the format of the `manual.six` file, except that it must provide a string, say `"myownformat"` which identifies the format of the help book. Then the basic actions on a help book are delegated by the help system to handler functions stored in a record `HELP_BOOK_HANDLER.myownformat`. See 5.3 for information which functions must be provided by the handler and what they are supposed to do. The main work to teach GAP to use a new document format is to write these handler functions and to produce an appropriate `manual.six` file.

5.1 Installing a Help Book

1 ► `HELP_ADD_BOOK(` *short*`,` *long*`,` *dir* `)`

This command tells GAP that in directory *dir* (given as either a string describing the path relative to the GAP root directory `GAP_ROOT_PATHS[1]` or as directory object) contains the basic information about a help book. The string *short* is used as an identifying name for that book by the online help. The string *long* should be a short explanation of the content of the book. Both strings together should easily fit on a line, since they are displayed with `?books`.

It is possible to reinstall a book with different strings *short*, *long*; (for example, documentation of a not-loaded GAP package indicates this in the string *short* and if you later load the package, GAP quietly changes the string *short* as it reinstalls its documentation).

The only condition necessary to make the installation of a book **valid** is that the directory *dir* must contain a file `manual.six`. The next section explains how this file must look.

5.2 The manual.six File

If a `manual.six` file for a help book is not in the format of the `gapmacro.tex` macros, explained in chapter `The gapmacro.tex Manual Format` (see 2), the first non-empty line of `manual.six` must be of the form

`#SIXFORMAT` *myownformat*

where *myownformat* is an identifying string for this format. The reading of the (remainder of the) file is then delegated to the function `HELP_BOOK_HANDLER.`*myownformat*`.ReadSix` which must exist. Thus there are no further regulations for the format of the `manual.six` file, other that what you yourself impose. If such a line is missing then it is assumed that the `manual.six` file complies with the `gapmacro.tex` documentation format which is the `default` format.

The next section explains what the return value of `HELP_BOOK_HANDLER.`*myownformat*`.ReadSix` should look like and which further function should be contained in `HELP_BOOK_HANDLER.`*myownformat*.

5.3 The Help Book Handler

For each document format *myownformat* there must be a record `HELP_BOOK_HANDLER.`*myownformat* of functions with the following names and functionality.

An implementation example of such a set of handler functions is the `default` format, which is the format name used for the `gapmacro.tex` documentation format, and this is contained in the file `lib/helpdef.gi`.

The package GapDoc (see Chapter 1) also defines a format (as it should) which is called: `GapDocGAP` (the case **is** significant).

As you can see by the above two examples, the name for a document format can be anything, but it should be in some way meaningful.

`ReadSix(` *stream* `)`
: For an input text stream *stream* to a `manual.six` file, this must return a record *info* which has at least the following two components: `bookname` which is the short identifying name of the help book, and `entries`. Here *info*`.entries` must be a list with one entry per search string (which can be a section header, function name, index entry, or whatever seems sensible to be searched for matching a help query). A **match** for the GAP help is a pair (*info*, *i*) where *i* refers to an index for the list *info*`.entries` and this corresponds to a certain position in the document. There is one further regulation for the format of the entries of *info*`.entries`. They must be lists and the first element of such a list must be a string which is printed by GAP for example when several matches are found for a query (so it should essentially be the string which is searched for the match, except that it may contain upper and lower case letters or some markup). There may be other components in *info* which are needed by the functions below, but their names and formats are not prescribed. The *stream* argument is typically generated using `InputTextFile` (see 10.5.1), e.g.

```
gap> dirs := DirectoriesPackageLibrary("gapdoc", "doc");;
gap> file := Filename(dirs, "manual.six");;
gap> stream := InputTextFile(file);;
```

`ShowChapters(` *info* `)`
: This must return a text string or list of text lines which contains the chapter headers of the book *info*`.bookname`.

`ShowSection(` *info* `)`
: This must return a text string or list of text lines which contains the section (and chapter) headers of the book *info*`.bookname`.

`SearchMatches(` *info*`,` *topic*`,` *frombegin* `)`
: This function must return a list of indices of *info*`.entries` for entries which match the search string *topic*. If *frombegin* is `true` then those parts of *topic* which are separated by spaces should be considered as the beginnings of words to decide the matching. It *frombegin* is `false`, a substring search should be performed. The string *topic* can be assumed to be already normalized (transformed to lower case, and whitespace normalized). The function must return a list with two entries `[exact, match]` where `exact` is the list of indices for exact matches and `match` a list of indices of the remaining matches.

`MatchPrevChap(` *info*`,` *i* `)`
: This should return the match [*info*, `j`] which points to the beginning of the chapter containing match [*info*, *i*], respectively to the beginning of the previous chapter if [*info*, *i*] is already the beginning of a chapter. (Corresponds to `?<<`.)

`MatchNextChap(` *info*`,` *i* `)`
: Like the previous function except that it should return the match for the beginning of the next chapter. (Corresponds to `?>>`.)

`MatchPrev(` *info*`,` *i* `)`
: This should return the previous section (or appropriate portion of the document). (Corresponds to `?<`.)

`MatchNext(` *info*`,` *i* `)`
: Like the previous function except that it should return the next section (or appropriate portion of the document). (Corresponds to `?>`.)

`HelpData(` *info*`,` *i*`,` *type* `)`
: This returns for match [*info*, *i*] some data whose format depends on the string *type*, or `fail` if these data are not available. The values of *type* which currently must be handled and the corresponding result format are described in the list below.

The `HELP_BOOK_HANDLER.`*myownformat*`.HelpData` function must recognize the following values of the *type* argument.

`"text"`
: This must return a corresponding text string in a format which can be fed into the `Pager`, see 2.4.1.

`"url"`
: If the help book is available in HTML format this must return an URL as a string (Probably a `file://` URL containing a label for the exact start position in that file). Otherwise it returns `fail`.

`"dvi"`
: If the help book is available in dvi-format this must return a record of form `rec( file :=` *filename*`, page :=` *pagenumber* `)`. Otherwise it returns `fail`.

`"pdf"`
: Same as case `"dvi"`, but for the corresponding pdf-file.

`"secnr"`
: This must return a pair like `[[3,3,1], "3.3.1"]` which gives the section number as chapter number, section number, subsection number triple and a corresponding string (a chapter itself is encoded like `[[4,0,0], "4."]`). Useful for cross-referencing between help books.

5.4 Introducing new Viewer for the Online Help

There is a record `HELP_VIEWER_INFO` which contains one component for each help viewer. Such a record contains two components.

The component `.type` refers to one of the *type*s recognized by the `HelpData` handler function explained in the previous section (currently one of `"text"`, `"url"`, `"dvi"`, or `"pdf"`).

The component `.show` is a function which gets as input the result of a corresponding `HelpData` handler call, if it was not `fail`. This function has to perform the actual display of the data. (E.g., by calling a function like `Pager` or by starting up an external viewer program.)

6 Function-Operation-Attribute Triples

GAP is eager to maintain information that it has gathered about an object, possibly by lengthy calculations. The most important mechanism for information maintenance is the automatic storage and look-up that takes place for **attributes**; and this was already mentioned in section 8.1 in the tutorial. In this chapter we will describe further mechanisms that allow storage of results that are not values of attributes.

The idea which is common to all sections is that certain operations, which are not themselves attributes, have an attribute associated with them. To automatically delegate tasks to the attribute, GAP knows, in addition to the **operation** and the **attributes** also a **function**, which is "wrapped around" the other two. This "wrapper function" is called by the user and decides whether to call the operation or the attribute or possibly both. The whole **f**unction-**o**peration-**a**ttribute triple (or **FOA triple**) is set up by a single GAP command which writes the wrapper function and already installs some methods, e.g., for the attribute to fall back on the operation. The idea is then that subsequent methods, which perform the actual computation, are installed only for the operation, whereas the wrapper function remains unaltered, and in general no additional methods for the attribute are required either.

6.1 Key Dependent Operations

There are several functions that require as first argument a domain, e.g., a group, and as second argument something much simpler, e.g., a prime. `SylowSubgroup` is an example. Since its value depends on two arguments, it cannot be an attribute, yet one would like to store Sylow subgroups once they have been computed.

The idea is to provide an attribute of the group, called `ComputedSylowSubgroups`, and to store the groups in this list. The name implies that the value of this attribute may change in the course of a GAP session, whenever a newly-computed Sylow subgroup is put into the list. Therefore, this is a **mutable attribute** (see 3.3 in "Programming in GAP"). The list contains primes in each bound odd position and a corresponding Sylow subgroup in the following even position. More precisely, if p `= ComputedSylowSubgroups(` G `)[` *even* `- 1 ]` then `ComputedSylowSubgroups(` G `)[` *even* `]` holds the value of `SylowSubgroup(` G, p `)`. The pairs are sorted in increasing order of p, in particular at most one Sylow p subgroup of G is stored for each prime p. This attribute value is maintained by the operation `SylowSubgroup`, which calls the operation `SylowSubgroupOp(` G, p `)` to do the real work, if the prime p cannot be found in the list. So methods that do the real work should be installed for `SylowSubgroupOp` and not for `SylowSubgroup`.

The same mechanism works for other functions as well, e.g., for `PCore`, but also for `HallSubgroup`, where the second argument is not a prime but a set of primes.

1 ► `KeyDependentOperation(` *name*, *dom-req*, *key-req*, *key-test* `)`

declares at the same time all three: two operations with names *name* and *name*`Op`, respectively, and an attribute with name and the attribute as described above, with names *name*, *name*`Op`, and `Computed`*name*`s`. *dom-req* and *key-req* specify the required filters for the first and second argument of the operation *name*`Op`, which are needed to create this operation with `NewOperation` (see 3.5.1). *dom-req* is also the required filter for the corresponding attribute `Computed`*name*`s`. The fourth argument *key-test* is in general a function to which the second argument *info* of *name*`(` *D*`,` *info* `)` will be passed. This function can perform tests on *info*, and raise an error if appropriate.

For example, to set up the three objects `SylowSubgroup`, `SylowSubgroupOp`, and `ComputedSylowSubgroups` together, the declaration file "lib/grp.gd" contains the following line of code.

```
KeyDependentOperation( "SylowSubgroup", IsGroup, IsPosInt, "prime" );
```

In this example, *key-test* has the value `"prime"`, which is silently replaced by a function that tests whether its argument is a prime.

```
gap> s4 := Group((1,2,3,4),(1,2));;
gap> SylowSubgroup( s4, 5 );;  ComputedSylowSubgroups( s4 );
[ 5, Group(()) ]
gap> SylowSubgroup( s4, 2 );;  ComputedSylowSubgroups( s4 );
[ 2, Group([(3,4),(1,4)(2,3),(1,3)(2,4)]), 5, Group(()) ]

gap> SylowSubgroup( s4, 6 );
Error, SylowSubgroup: <p> must be a prime called from
<compiled or corrupted call value>  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;
```

Thus the prime test need not be repeated in the methods for the operation `SylowSubgroupOp` (which are installed to do the real work). Note that no methods need be installed for `SylowSubgroup` and `ComputedSylowSubgroups`. If a method is installed with `InstallMethod` for a wrapper operation such as `SylowSubgroup` then a warning is signalled provided the `InfoWarning` level is at least 1. (Use `InstallOtherMethod` in order to suppress the warning.)

6.2 In Parent Attributes

This section describes how you can add new "in parent attributes" (see 30.8 and 30.7 in the Reference Manual). As an example, we describe how `Index` and its related functions are implemented.

There are two operations `Index` and `IndexOp`, and an attribute `IndexInParent`. They are created together as shown below, and after they have been created, methods need be installed only for `IndexOp`. In the creation process, `IndexInParent` already gets one default method installed (in addition to the usual system getter of each attribute, see 13.5 in the Reference Manual), namely `D -> IndexOp( Parent( D ), D )`.

The operation `Index` proceeds as follows.

- If it is called with the two arguments *super* and *sub*, and if `HasParent(` *sub* `)` and `IsIdenticalObj(` *super*`, Parent(` *sub* `) )` are `true`, `IndexInParent` is called with argument *sub*, and the result is returned.
- Otherwise, `IndexOp` is called with the same arguments that `Index` was called with, and the result is returned.

(Note that it is in principle possible to install even `Index` and `IndexOp` methods for a number of arguments different from two, with `InstallOtherMethod`, see 3.3 in "Programming in GAP").

1 ▶ `InParentFOA(` *name*`,` *super-req*`,` *sub-req*`, DeclareAttribute )`
▶ `InParentFOA(` *name*`,` *super-req*`,` *sub-req*`, DeclareProperty )`

declares the operations and the attribute as described above, with names *name*, *name*`Op`, and *name*`InParent`. *super-req* and *sub-req* specify the required filters for the first and second argument of the operation *name*`Op`, which are needed to create this operation with `NewOperation` (see 3.5.1). *sub-req* is also the required filter for the corresponding attribute *name*`InParent`; note that `HasParent` is **not** required for the argument *U* of *name*`InParent`, because even without a parent stored, `Parent(` *U* `)` is legal, meaning *U* itself (see 30.7 in the Reference Manual). The fourth argument is `DeclareProperty` if *name*`InParent` takes only boolean values (for example in the case `IsNormalInParent`), and `DeclareAttribute` otherwise.

For example, to set up the three objects `Index`, `IndexOp`, and `IndexInParent` together, the declaration file "lib/domain.gd" contains the following line of code.

```
InParentFOA( "Index", IsGroup, IsGroup, DeclareAttribute );
```

Note that no methods need be installed for `Index` and `IndexInParent`.

6.3 Operation Functions

Chapter 39 of the Reference Manual and, in particular, the Section 39.1 explain that certain operations such as `Orbits` (see 39.3), besides their usual usage with arguments *G*, *D*, and *opr*, can also be applied to an external set (*G*-set), in which case they can be interpreted as attributes. Moreover, they can also be interpreted as attributes for permutation group, meaning the natural action on the set of its moved points.

The definition of `Orbits` says that a method should be a function with arguments *G*, *D*, *gens*, *oprs*, and *opr*, as in the case of the operation `ExternalSet` when specified via *gens* and *oprs* (see 39.11 in the Reference Manual). All other syntax variants allowed for `Orbits` (e.g., leaving out *gens* and *oprs*) are handled by default methods.

The default methods for `Orbits` support the following behaviour.

1. If the only argument is an external set *xset* and the attribute tester `HasOrbits(` *xset* `)` returns `true`, the stored value of that attribute is returned.
2. If the only argument is an external set *xset* and the attribute value is not known, the default arguments are obtained from the data of *xset*.
3. If *gens* and *oprs* are not specified, *gens* is set to `Pcgs(` *G* `)` if `CanEasilyComputePcgs(` *G* `)` is `true`, and to `GeneratorsOfGroup(` *G* `)` otherwise; *oprs* is set to *gens*.
4. The default value of *opr* is `OnPoints`.
5. In the case of an operation of a permutation group *G* on `MovedPoints(` *G* `)` via `OnPoints`, if the attribute tester `HasOrbits(` *G* `)` returns `true`, the stored attribute value is returned.
6. The operation is called as *result*`:= Orbits(` *G*`,` *D*`,` *gens*`,` *oprs*`,` *opr* `)`.
7. In the case of an external set *xset* or a permutation group *G* in its natural action, the attribute setter is called to store *result*.
8. *result* is returned.

The declaration of operations that match the above pattern is done as follows.

1 ▶ `OrbitsishOperation(` *name*`,` *reqs*`,` *usetype*`, NewAttribute )` F
▶ `OrbitsishOperation(` *name*`,` *reqs*`,` *usetype*`, NewProperty )` F

declares an attribute *op* as described above, with name *name*. The second argument *reqs* specifies the list of required filters for the usual (five-argument) methods that do the real work.

If the third argument *usetype* is `true`, the function call *op*`(` *xset* `)` will — if the value of *op* for *xset* is not yet known — delegate to the five-argument call of *op* with second argument *xset* rather than with *D* (cf. step 6 above). This allows certain methods for *op* to make use of the type of *xset*, in which the types of the external subsets of *xset* and of the external orbits in *xset* are stored. (This is used to avoid repeated calls of `NewType` in functions like `ExternalOrbits(` *xset* `)`, which call `ExternalOrbit(` *xset*`,` *pnt* `)` for several values of *pnt*.)

For property testing functions such as `IsTransitive`, the fourth argument is `NewProperty`, otherwise it is `NewAttribute`.

For example, to set up the operation `Orbits`, the declaration file "lib/oprt.gd" contains the following line of code:

```
OrbitsishOperation( "Orbits", OrbitsishReq, false, NewAttribute );
```

The variable `OrbitsishReq` contains the standard requirements

```
OrbitsishReq := [ IsGroup, IsList,
  IsList,
  IsList,
  IsFunction ];
```

which are usually entered in calls to `OrbitsishOperation`.

A similar mechanism is provided for operations such as `Orbit` that do not have an associated attribute but still need a wrapper function to standardize the arguments for the associated operation.

2 ► `OrbitishFO(` *name*`,` *reqs*`,` *famrel*`,` *usetype* `)` F

declares a wrapper function and its operation, with names *name* and *name*`Op`. The second argument *reqs* specifies the list of required filters for the operation *name*`Op`.

The third argument *famrel* is used to test the family relation between the second and third argument of *name*`(` *G*`,` *D*`,` *elm* `)`. For example, in the call `Orbit(` *G*`,` *D*`,` *pnt* `)`, *pnt* must be an element of *D*, so *famrel* `= IsCollsElms` is appropriate, and in the call `Blocks(` *G*`,` *D*`,` *seed* `)`, *seed* must be a subset of *D*, and the family relation is `IsIdenticalObj`. The fourth argument *usetype* serves the same purpose as in the case of `OrbitsishOperation`.

For example, to setup the function `Orbit` and its operation `OrbitOp`, the declaration file "lib/oprt.gd" contains the following line of code:

```
OrbitishFO( "Orbit", OrbitishReq, IsCollsElms, false );
```

The variable `OrbitishReq` contains the standard requirements

```
OrbitishReq  := [ IsGroup, IsList, IsObject,
  IsList,
  IsList,
  IsFunction ];
```

which are usually entered in calls to `OrbitishFO`.

The relation test via *famrel* is used to provide a uniform construction of the wrapper functions created by `OrbitishFO`, in spite of the different syntax of the specific functions. For example, `Orbit` admits the calls `Orbit(` *G*`,` *D*`,` *pnt*`,` *opr* `)` and `Orbit(` *G*`,` *pnt*`,` *opr* `)`, i.e., the second argument *D* may be omitted; `Blocks` admits the calls `Blocks(` *G*`,` *D*`,` *seed*`,` *opr* `)` and `Blocks(` *G*`,` *D*`,` *opr* `)`, i.e., the third argument may be omitted. The translation to the appropriate call of `OrbitOp` or `BlocksOp`, for either operation with five or six arguments, is handled via *famrel*.

As a consequence, there must not only be methods for `OrbitOp` with the six arguments corresponding to `OrbitishReq`, but also methods for only five arguments (i.e., without *D*). Plenty of examples are contained in the implementation file "lib/oprt.gi".

In order to handle a few special cases (currently `Blocks` and `MaximalBlocks`), also the following form is supported.

`OrbitishFO(` *name*`,` *reqs*`,` *famrel*`,` *attr* `) F`

The functions in question depend upon an argument *seed*, so they cannot be regarded as attributes. However, they are most often called without giving *seed*, meaning "choose any minimal resp. maximal block system". In this case, the result can be stored as the value of the attribute *attr* that was entered as fourth argument of `OrbitishFO`. This attribute is considered by a call `Blocks(` G`,` D`,` *opr* `)` (i.e., without *seed*) in the same way as `Orbits` considers `OrbitsAttr`.

To set this up, the declaration file "lib/oprt.gd" contains the following lines:

```
DeclareAttribute( "BlocksAttr", IsExternalSet );
OrbitishFO( "Blocks",
    [ IsGroup, IsList, IsList,
      IsList,
      IsList,
      IsFunction ], IsIdenticalObj, BlocksAttr );
```

And this extraordinary FOA triple works as follows:

```
gap> s4 := Group((1,2,3,4),(1,2));; Blocks( s4, MovedPoints(s4), [1,2] );
[ [ 1, 2, 3, 4 ] ]
gap> Tester( BlocksAttr )( s4 );
false

gap> Blocks( s4, MovedPoints(s4) );
[ [ 1, 2, 3, 4 ] ]
gap> Tester( BlocksAttr )( s4 );  BlocksAttr( s4 );
true
[ [ 1, 2, 3, 4 ] ]
```

7 Weak Pointers

This chapter describes the use of the kernel feature of **weak pointers**. This feature is intended for use only in GAP internals, and is **not recommended** for use in GAP packages, user code, or at the higher levels of the library.

The GASMAN garbage collector is the part of the kernel that manages memory in the users workspace. It will normally only reclaim the storage used by an object when the object cannot be reached as a subobject of any GAP variable, or from any reference in the kernel. We say that any link to object *a* from object *b* "keeps object *a* alive", as long as *b* is alive. It is occasionally convenient, however to have a link to an object which **does not keep it alive**, and this is a weak pointer. The most common use is in caches, and similar structures, where it is only necessary to remember how to solve problem x as long as some other link to x exists.

The following section 7.1 describes the semantics of the objects that contain weak pointers. Following sections describe the functions available to manipulate them.

7.1 Weak Pointer Objects

A **weak pointer object** is similar to a mutable plain list, except that it does not keep its subobjects alive during a garbage collection. From the GAP viewpoint this means that its entries may become unbound, apparently spontaneously, at any time. Considerable care is therefore needed in programming with such an object.

7.2 WeakPointerObj

1 ▶ `WeakPointerObj(` *list* `)`

`WeakPointerObj` returns a weak pointer object which contains the same subobjects as *list*, that is it returns a **shallow** weak copy of *list*.

```
gap> w := WeakPointerObj( [ 1, , [2,3], fail, rec() ] );
WeakPointerObj( [ 1, , [ 2, 3 ], fail, rec( ) ] )
gap> GASMAN("collect");
gap> w;
WeakPointerObj( [ 1, , , fail ] )
```

Note that `w` has failed to keep its list and record subobjects alive during the garbage collection. Certain subobjects, such as small integers and elements of small finite fields, are not stored in the workspace, and so are not subject to garbage collection, while certain other objects, such as the Boolean values, are always reachable from global variables or the kernel and so are never garbage collected.

Subobjects reachable without going through a weak pointer object do not evaporate, as in:

```
gap> l := [1,2,3];;
gap> w[1] := l;;
gap> w;
WeakPointerObj( [ [ 1, 2, 3 ], , , fail ] )
gap> GASMAN("collect");
gap> w;
eakPointerObj( [ [ 1, 2, 3 ], , , fail] )
```

Note also that the global variables `last`, `last2` and `last3` will keep things alive – this can be confusing when debugging.

7.3 Low Level Access Functions for Weak Pointer Objects

1 ► `SetElmWPObj(`*wp*`,`*pos*`,`*val*`)`
► `UnbindElmWPObj(`*wp*`,`*pos*`)`
► `ElmWPObj(`*wp*`, `*pos*`)`
► `IsBOundElmWPObj(`*wp*`,`*pos*`)`
► `LengthWPObj(`*wp*`)`

The functions `SetElmWPObj(`*wp*`,`*pos*`,`*val*`)` and `UnbindElmWPObj(`*wp*`,`*pos*`)` set and unbind entries in a weak pointer object.

The function `ElmWPObj(`*wp*`, `*pos*`)` returns the element at position *pos* of the weak pointer object *wp*, if there is one, and `fail` otherwise. A return value of `fail` can thus arise either because (a) the value `fail` is stored at position *pos*, or (b) no value is stored at position *pos*. Since `fail` cannot vanish in a garbage collection, these two cases can safely be distinguished by a **subsequent** call to `IsBoundElmWPObj(`*wp*`,`*pos*`)`, which returns `true` if there is currently a value bound at position *pos* of *wp* and `false` otherwise.

Note that it is **not** safe to write: `if IsBoundElmWpObj(w,i) then x:= ElmWPObj(w,i); fi;` and treat `x` as reliably containing a value taken from `w`, as a badly timed garbage collection could leave `x` containing `fail`. Instead use `x := ElmWPObj(w,i); if x <> fail or IsBoundElmWPObj(w,i) then . . .`.

```
gap> w := WeakPointerObj( [ 1, , [2,3], fail, rec() ] );
WeakPointerObj( [ 1, , [ 2, 3 ], fail, rec( ) ] )
gap> SetElmWPObj(w,5,[]);
gap> w;
WeakPointerObj( [ 1, , [ 2, 3 ], fail, [  ] ] )
gap> UnbindElmWPObj(w,1);
gap> w;
WeakPointerObj( [ , , [ 2, 3 ], fail, [  ] ] )
gap> ElmWPObj(w,3);
[ 2, 3 ]
gap> ElmWPObj(w,1);
fail
gap> 2;;3;;4;;GASMAN("collect"); # clear last etc.
gap> ElmWPObj(w,3);
fail
gap> w;
WeakPointerObj( [ , , , fail, [  ] ] )
gap> ElmWPObj(w,4);
fail
gap> IsBoundElmWPObj(w,3);
false
gap> IsBoundElmWPObj(w,4);
true
```

7.4 Accessing Weak Pointer Objects as Lists

Weak pointer objects are members of `ListsFamily` and the categories `IsList` and `IsMutable`. Methods based on the low-level functions in the previous section, are installed for the list access operations, enabling them to be used as lists. However, it is **not recommended** that these be used in programming. They are supplied mainly as a convenience for interactive working, and may not be safe, since functions and methods for lists may assume that after `IsBound(w[i])` returns true, access to `w[i]` is safe.

7.5 Copying Weak Pointer Objects

A `ShallowCopy` method is installed, which makes a new weak pointer object containing the same objects as the original.

It is possible to apply `StructuralCopy` to a weak pointer object, obtaining a new weak pointer object containing copies of the objects in the original. This **may not be safe** if a badly timed garbage collection occurs during copying.

Applying `Immutable` to a weak pointer object produces an immutable plain list containing immutable copies of the objects contained in the weak pointer object. An immutable weak pointer object is a contradiction in terms.

7.6 The GASMAN Interface for Weak Pointer Objects

The key support for weak pointers is in `gasman.c` and `gasman.h`. This document assumes familiarity with the rest of the operation of GASMAN. A kernel type (tnum) of bags which are intended to act as weak pointers to their subobjects must meet three conditions. Firstly, the marking function installed for that tnum must use `MarkBagWeakly` for those subbags, rather than `MARK_BAG`. Secondly, before any access to such a subbag, it must be checked with `IS_WEAK_DEAD_BAG`. If that returns true, then the subbag has evaporated in a recent garbage collection and must not be accessed. Typically the reference to it should be removed. Thirdly, a **sweeping function** must be installed for that tnum which copies the bag, removing all references to dead weakly held subbags.

The files `weakptr.c` and `weakptr.h` use this interface to support weak pointer objects. Other objects with weak behaviour could be implemented in a similar way.

8 Stabilizer Chains (preliminary)

This chapter contains some rather technical complements to the material handled in the chapters 40 and 41 of the reference manual.

8.1 Generalized Conjugation Technique

The command `ConjugateGroup(` G`,` p `)` (see 37.2.5 in the reference manual) for a permutation group G with stabilizer chain equips its result also with a stabilizer chain, namely with the chain of G conjugate by p. Conjugating a stabilizer chain by a permutation p means replacing all the points which appear in the `orbit` components by their images under p and replacing every permutation g which appears in a `labels` or `transversal` component by its conjugate g^p. The conjugate g^p acts on the mapped points exactly as g did on the original points, i.e., $(pnt \cdot p) \cdot g^p = (pnt \cdot g) \cdot p$. Since the entries in the `translabels` components are integers pointing to positions of the `labels` list, the `translabels` lists just have to be permuted by p for the conjugated stabilizer. Then `generators` is reconstructed as `labels{ genlabels }` and `transversal{ orbit }` as `labels{ translabels{ orbit } }`.

This conjugation technique can be generalized. Instead of mapping points and permutations under the same permutation p, it is sometimes desirable (e.g., in the context of permutation group homomorphisms) to map the points with an arbitrary mapping *map* and the permutations with a homomorphism *hom* such that the compatibility of the actions is still valid: $map(pnt) \cdot hom(g) = map(pnt \cdot g)$. (Of course the ordinary conjugation is a special case of this, with $map(pnt) = pnt \cdot p$ and $hom(g) = g^p$.)

In the generalized case, the "conjugated" chain need not be a stabilizer chain for the image of *hom*, since the "preimage" of the stabilizer of $map(b)$ (where b is a base point) need not fix b, but only fixes the preimage $map^{-1}(map(b))$ setwise. Therefore the method can be applied only to one level and the next stabilizer must be computed explicitly. But if *map* is injective, we have $map(b) \cdot hom(g) = map(b) \iff b \cdot g = b$, and if this holds, then $g = w(g_1, \ldots, g_n)$ is a word in the generators $g_1, \ldots, g_n$ of the stabilizer of b and $hom(g) =^* w(hom(g_1), \ldots, hom(g_n))$ is in the "conjugated" stabilizer. If, more generally, *hom* is a right inverse to a homomorphism φ (i.e., $\varphi(hom(g)) = g \, \forall g$), equality $*$ holds modulo $\mathrm{Ker}\,\varphi$; in this case the "conjugated" chain can be made into a real stabilizer chain by extending each level with the generators $\mathrm{Ker}\,\varphi$ and appending a proper stabilizer chain of $\mathrm{Ker}\,\varphi$ at the end. These special cases will occur in the algorithms for permutation group homomorphisms (see 38 in the reference manual).

To "conjugate" the points (i.e., `orbit`) and permutations (i.e., `labels`) of the Schreier tree, a loop is set up over the `orbit` list constructed during the orbit algorithm, and for each vertex b with unique edge $a(l)b$ ending at b, the label l is mapped with *hom* and b with *map*. We assume that the `orbit` list was built w.r.t. a certain ordering of the labels, where $l' < l$ means that every point in the orbit was mapped with l' before it was mapped with l. This shape of the `orbit` list is guaranteed if the Schreier tree is extended only by `AddGeneratorsExtendSchreierTree`, and it is then also guaranteed for the "conjugated" Schreier tree. (The ordering of the labels cannot be read from the Schreier tree, however.)

In the generalized case, it can happen that the edge $a(l)b$ bears a label l whose image is "old", i.e., equal to the image of an earlier label $l' < l$. Because of the compatibility of the actions we then have $map(b) = map(a) \cdot hom(l)^{-1} = map(a) \cdot hom(l')^{-1} = map(al'^{-1})$, so $map(b)$ is already equal to the image of the vertex al'^{-1}. This vertex must have been encountered before $b = al^{-1}$ because $l' < l$. We conclude that the image of a label can be "old" only if the vertex at the end of the corresponding edge has an "old" image, too, but then it need not be "conjugated" at all. A similar remark applies to labels which map under *hom* to the identity.

8.2 The General Backtrack Algorithm with Ordered Partitions

Section 41.11 in the reference manual describes the basic functions for a backtrack search. The purpose of this section is to document how the general backtrack algorithm is implemented in GAP and which parts you have to modify if you want to write your own backtrack routines.

Internal representation of ordered partitions. GAP represents an ordered partition as a record with the following components.

`points`
: a list of all points contained in the partition, such that the points of each cell from lie consecutively,

`cellno`
: a list whose ith entry is the number of the cell which contains the point i,

`firsts`
: a list such that `points[firsts[`j`]]` is the first point in `points` which is in cell j,

`lengths`
: a list of the cell lengths.

Some of the information is redundant, e.g., the `lengths` could also be read off the `firsts` list, but since this need not be increasing, it would require some searching. Similar for `cellno`, which could be replaced by a systematic search of `points`, keeping track of what cell is currently being traversed. With the above components, the mth cell of a partition P is expressed as P`.points [ `P`.firsts[`m`] .. `P`.firsts[`m`] + `P`.lengths[`m`] - 1 ]` . The most important operations, however, to be performed upon P are the splitting of a cell and the reuniting of the two parts. Following the strategy of J. Leon, this is done as follows:

(1) The points which make up the cell that is to be split are sorted so that the ones that remain inside occupy positions `[ `P`.firsts[`m`] .. `*last*` ]` in the list P`.points` (for a suitable value of *last*).

(2) The points at positions `[ `*last*` + 1 .. `P`.firsts[`m`] + `P`.lengths[`m`] - 1 ]` will form the additional cell. For this new cell requires additional entries are added to the lists P`.firsts` (namely, *last*`+1`) and P`.lengths` (namely, P`.firsts[`m`] + `P`.lengths[`m`] - `*last*` - 1`).

(3) The entries of the sublist P`.cellno [ `*last*`+1 .. `P`.firsts[`m`] + P.lengths[`m`]-1 ]` must be set to the number of the new cell.

(4) The entry P`.lengths[`m`]` must be reduced to *last*` - `P`.firsts[`m`] + 1`.

Then reuniting the two cells requires only the reversal of steps 2 to 4 above. The list P`.points` need not be rearranged.

Functions for setting up an R-base. This subsection explains some GAP functions which are local to the library file `lib/stbcbckt.gi` which contains the code for backtracking in permutation groups. They are mentioned here because you might find them helpful when you want to implement you own backtracking function based on the partition concept. An important argument to most of the functions is the R-base $\mathcal{R}$, which you should regard as a black box. We will tell you how to set it up, how to maintain it and where to pass it as argument, but it is not necessary for you to know its internal representation. However, if you insist to learn the whole story: Here are the record components from which an R-base is made up:

`domain`
: the set Ω on which the group G operates

`base`
: the sequence $(a_1, \ldots, a_r)$ of base points

`partition`
: an ordered partition, initially Π_0, this will be refined to $\Pi_1, \ldots, \Pi_r$ during the backtrack algorithm

`where`
: a list such that a_i lies in cell number `where[` i `]` of Π_i

`rfm`
: a list whose ith entry is a list of refinements which take Σ_i to Σ_{i+1}; the structure of a refinement is described below

`chain`
: a (copy of a) stabilizer chain for G (not if G is a symmetric group)

`fix`
: only if G is a symmetric group: a list whose i entry contains `Fixcells(` Π_i `)`

`level`
: initially equal to `chain`, this will be changed to chains for the stabilizers $G_{a_1 \ldots a_i}$ for $i = 1, \ldots, r$ during the backtrack algorithm; if G is a symmetric group, only the number of moved points is stored for each stabilizer

`lev`
: a list whose ith entry remembers the `level` entry for $G_{a_1 \ldots a_{i-1}}$

`level2`, `lev2`
: a similar construction for a second group (used in intersection calculations), `false` otherwise. This second group H activated if the R-base is constructed as `EmptyRBase( [` G`,` H `],` Ω`,` Π_0 `)` (if G `=` H, GAP sets `level2 = true` instead).

`nextLevel`
: this is described below

As our guiding example, we present code for the function `Centralizer` which calculates the centralizer of an element g in the group G. (The real code is more general and has a few more subtleties.)

```
Π_0 := TrivialPartition( Ω );
R := EmptyRBase( G, Ω, Π_0 );
R.nextLevel := function( Π, rbase )
local  fix, p, q, where;
  NextRBasePoint( Π, rbase );
  fix := Fixcells( Π );
  for p in fix do
    q := p ^ g;
    where := IsolatePoint( Π, q );
     if where <> false then
       Add( fix, q );
       ProcessFixpoint( R, q );
       AddRefinement( R, "Centralizer", [ Π.cellno[ p ], q, where ] );
       if Π.lengths[ where ] = 1 then
         p := FixpointCellNo( Π, where );
         ProcessFixpoint( R, p );
         AddRefinement( R, "ProcessFixpoint", [ p, where ] );
       fi;
     fi;
   od;
```

```
22 end;
23 return PartitionBacktrack(
24      G,
25      c -> g ^ c = g,
26      false,
27      R,
28      [ Π0, g ],
29      L, R );
```

The list numbers below refer to the line numbers of the code above.

1. Ω is the set on which G acts and Π_0 is the first member of the decreasing sequence of partitions mentioned in 41.11 in the reference manual. We set $\Pi_0 = (\Omega)$, which is constructed as `TrivialPartition(` Ω `))`, but we could have started with a finer partition, e.g., into unions of g-cycles of the same length.
2. This statement sets up the R-base in the variable $\mathcal{R}$.
3. – 21. These lines define a function $\mathcal{R}$`.nextLevel` which is called whenever an additional member in the sequence $\Pi_0 \geq \Pi_1 \geq \ldots$ of partitions is needed. If Π_i does not yet contain enough base points in one-point cells, GAP will call $\mathcal{R}$`.nextLevel(` Π_i`,` $\mathcal{R}$ `)`, and this function will choose a new base point a_{i+1}, refine Π_i to Π_{i+1} (thereby **changing** the first argument) and store all necessary information in $\mathcal{R}$.

5. This statement selects a new base point a_{i+1}, which is not yet in a one-point cell of Π and still moved by the stabilizer $G_{a_1 \ldots a_i}$ of the earlier base points. If certain points of Ω should are preferred as base point (e.g., because they belong to long cycles of g), a list of points starting with the most wanted ones, can be given as an optional third argument to `NextRBasePoint` (actually, this is done in the real code for `Centralizer`).
6. `Fixcells(` Π `)` returns the list of points in one-point cells of Π (ordered as the cells are ordered in Π).
7. For every point $p \in \mathit{fix}$, if we know the image p `^` g under $c \in C_G(e)$, we also know `(` p `^` g `) ^` c `= (` p `^` c `) ^` g. We therefore want to isolate these extra points in Π.

9. This statement puts point q in a cell of its own, returning in *where* the number of the cell of Π from which q was taken. If q was already the only point in its cell, *where* `= false` instead.

12. This command does the necessary bookkeeping for the extra base point q: It prescribes q as next base in the stabilizer chain for G (needed, e.g., in line 5) and returns `false` if q was already fixed the stabilizer of the earlier base points (and `true` otherwise; this is not used here). Another call to `ProcessFixpoint` like this was implicitly made by the function `NextRBasePoint` to register the chosen base point. By contrast, the point q was not chosen this way, so `ProcessFixpoint` must be called explicitly for q.
13. This statement registers the function which will be used during the backtrack search to perform the corresponding refinements on the "image partition" Σ_i (to yield the refined Σ_{i+1}). After choosing an image b_{i+1} for the base point a_{i+1}, GAP will compute $\Sigma_i \wedge (\{b_{i+1}\}, \Omega - \{b_{i+1}\})$ and store this partition in $\mathcal{I}$`.partition`, where $\mathcal{I}$ is a black box similar to $\mathcal{R}$, but corresponding to the current "image partition" (hence it is an "R-image" in analogy to the R-base). Then GAP will call the function `Refinements.Centralizer(` $\mathcal{R}$`,` $\mathcal{I}$`,` Π`.cellno[` p `],` p`,` *where* `)`, with the then current values of $\mathcal{R}$ and $\mathcal{I}$, but where Π`.cellno[` p `]`, p, *where* still have the values they have at the time of this `AddRefinement` command. This function call will further refine $\mathcal{I}$`.partition` to yield Σ_{i+1} as it is programmed in the function `Refinements.Centralizer`, which is described below. (The global variable `Refinements` is a record which contains all refinement functions for all backtracking procedures.)
14. – 19. If the cell from which q was taken out had only two points, we now have an additional one-point cell. This condition is checked in line 13 and if it is true, this extra fixpoint p is taken (line 15),

processed like q before (line 16) and is then (line 17) passed to another refinement function `Refinements.ProcessFixpoint(` $\mathcal{R}$, $\mathcal{I}$, p, *where* `)`, which is also described below.

23. – 29. This command starts the backtrack search. Its result will be the centralizer as a subgroup of G. Its arguments are

24. the group we want to run through,
25. the property we want to test, as a GAP function,
26. `false` if we are looking for a subgroup, `true` in the case of a representative search (when the result would be one representative),
27. the R-base,
28. a list of data, to be stored in $\mathcal{I}$`.data`, which has in position 1 the first member Σ_0 of the decreasing sequence of "image partitions" mentioned in 41.11 in the reference manual. In the centralizer example, position 2 contains the element that is to be centralized. In the case of a representative search, i.e., a conjugacy test g `^` c `?=` h, we would have h instead of g here, and possibly a Σ_0 different from Π_0 (e.g., a partition into unions of h-cycles of same length).
29. two subgroups $L \leq C_G(g)$ and $R \leq C_G(h)$ known in advance (we have $L = R$ in the centralizer case).

Refinement functions for the backtrack search. The last subsection showed how the refinement process leading from Π_i to Π_{i+1} is coded in the function $\mathcal{R}$`.nextLevel`, this has to be executed once the base point a_{i+1}. The analogous refinement step from Σ_i to Σ_{i+1} must be performed for each choice of an image b_{i+1} for a_{i+1}, and it will depend on the corresponding value of $\Sigma_i \wedge (\{b_{i+1}\}, \Omega - \{b_{i+1}\})$. But before we can continue our centralizer example, we must, for the interested reader, document the record components of the other black box $\mathcal{I}$, as we did above for the R-base black box $\mathcal{R}$. Most of the components change as GAP walks up and down the levels of the search tree.

`data`
: this will be mentioned below

`depth`
: the level i in the search tree of the current node Σ_i

`bimg`
: a list of images of the points in $\mathcal{R}$`.base`

`partition`
: the partition Σ_i of the current node

`level`
: the stabilizer chain $\mathcal{R}$`.lev[` i `]` at the current level

`perm`
: a permutation mapping `Fixcells(` Π_i `)` to `Fixcells(` Σ_i `)` (this implies mapping $(a_1, \ldots, a_i)$ to $(b_1, \ldots, b_i)$)

`level2`, `perm2`
: a similar construction for the second stabilizer chain, `false` otherwise (and `true` if $\mathcal{R}$`.level2 = true`)

As declared in the above code for `Centralizer`, the refinement is performed by the function `Refinement.Centralizer(` $\mathcal{R}$, $\mathcal{I}$, Π`.cellno[` p `]`, p, *where* `)`. The functions in the record `Refinement` always take two additional arguments before the ones specified in the `AddRefinement` call (in line 13 above), namely the R-base $\mathcal{R}$ and the current value $\mathcal{I}$ of the "R-image". In our example, p is a fixpoint of $\Pi = \Pi_i \wedge (\{a_{i+1}\}, \Omega - \{a_{i+1}\})$ such that *where* `=` Π`.cellno[` p `^` g `]`. The `Refinement` functions must return `false` if the refinement is unsuccessful (e.g., because it leads to Σ_{i+1} having different cell sizes from Π_{i+1}) and `true` otherwise. Our particular function looks like this.

```
1 Refinements.Centralizer := function( R, I, cellno, p, where )
2 local  Σ, q;
3   Σ := I.partition;
4   q := FixpointCellNo( Σ, cellno ) ^ I.data[ 2 ];
5   return IsolatePoint( Σ, q ) = where and ProcessFixpoint( I, p, q );
6 end;
```

The list numbers below refer to the line numbers of the code immediately above.

3. The current value of $\Sigma_i \wedge (\{b_{i+1}\}, \Omega - \{b_{i+1}\})$ is always found in $\mathcal{I}$`.partition`.
4. The image of the only point in cell number *cellno* `=` Π_i`.cellno[` p `]` in Σ under g `=` $\mathcal{I}$`.data[ 2 ]` is calculated.
5. The function returns `true` only if the image q has the same cell number in Σ as p had in Π (i.e., *where*) and if q can be prescribed as an image for p under the coset of the stabilizer $G_{a_1 \ldots a_{i+1}} \cdot c$ where $c \in G$ is an (already constructed) element mapping the earlier base points $a_1, \ldots, a_{i+1}$ to the already chosen images $b_1, \ldots, b_{i+1}$. This latter condition is tested by `ProcessFixpoint(` $\mathcal{I}$`,` p`,` q `)` which, if successful, also does the necessary bookkeeping in $\mathcal{I}$. In analogy to the remark about line 12 in the program above, the chosen image b_{i+1} for the base point a_{i+1} has already been processed implicitly by the function `PartitionBacktrack`, and this processing includes the construction of an element $c \in G$ which maps `Fixcells(` Π_i `)` to `Fixcells(` Σ_i `)` and a_{i+1} to b_{i+1}. By contrast, the extra fixpoints p and q in Π_{i+1} and Σ_{i+1} were not chosen automatically, so they require an explicit call of `ProcessFixpoint`, which replaces the element c by some $c' \cdot c$ (with $c' \in G_{a_1 \ldots a_{i+1}}$) which in addition maps p to q, or returns `false` if this is impossible.

You should now be able to guess what `Refinements.ProcessFixpoint(` $\mathcal{R}$`,` $\mathcal{I}$`,` p`,` *where* `)` does: it simply returns `ProcessFixpoint(` $\mathcal{I}$`,` p`, FixpointCellNo(` $\mathcal{I}$`.partition,` *where* `) )`.

Summary. When you write your own backtrack functions using the partition technique, you have to supply an R-base, including a component `nextLevel`, and the functions in the `Refinements` record which you need. Then you can start the backtrack by passing the R-base and the additional data (for the `data` component of the "R-image") to `PartitionBacktrack`.

Functions for meeting ordered partitions. A kind of refinement that occurs in particular in the normalizer calculation involves computing the meet of Π (cf. lines 6ff. above) with an arbitrary other partition Λ, not just with one point. To do this efficiently, GAP uses the following two functions.

1 ▶ `StratMeetPartition(` $\mathcal{R}$`,` Π`,` Λ `[,` g `] )`
▶ `MeetPartitionStrat(` $\mathcal{R}$`,` $\mathcal{I}$`,` Λ' `[,` g' `],` *strat* `)`

Such a `StratMeetPartition` command would typically appear in the function call $\mathcal{R}$`.nextLevel(` Π`,` $\mathcal{R}$ `)` (during the refinement of Π_i to Π_{i+1}). This command replaces Π by $\Pi \wedge \Lambda$ (thereby **changing** the second argument) and returns a "meet strategy" *strat*. This is (for us) a black box which serves two purposes: First, it allows GAP to calculate faster the corresponding meet $\Sigma \wedge \Lambda'$, which must then appear in a `Refinements` function (during the refinement of Σ_i to Σ_{i+1}). It is faster to compute $\Sigma \wedge \Lambda'$ with the "meet strategy" of $\Pi \wedge \Lambda$ because if the refinement of Σ is successful at all, the intersection of a cell from the left hand side of the $\wedge$ sign with a cell from the right hand side must have the same size in both cases (and *strat* records these sizes, so that only non-empty intersections must be calculated for $\Sigma \wedge \Lambda'$). Second, if there is a discrepancy between the behaviour prescribed by *strat* and the behaviour observed when refining Σ, the refinement can immediately be abandoned.

On the other hand, if you only want to meet a partition Π with Λ for a one-time use, without recording a strategy, you can simply type `StratMeetPartition(` Π, Λ `)` as in the following example, which also demonstrates some other partition-related commands.

```
gap> P := Partition( [[1,2],[3,4,5],[6]] );;  Cells( P );
[ [ 1, 2 ], [ 3, 4, 5 ], [ 6 ] ]
gap> Q := OnPartitions( P, (1,3,6) );;  Cells( Q );
[ [ 3, 2 ], [ 6, 4, 5 ], [ 1 ] ]
gap> StratMeetPartition( P, Q );
[  ]  # the ``meet strategy'' was not recorded, ignore this result
gap> Cells( P );
[ [ 1 ], [ 5, 4 ], [ 6 ], [ 2 ], [ 3 ] ]
```

You can even say `StratMeetPartition(` Π, Δ `)` where Δ is simple a subset of Ω, it will then be interpreted as the partition $(\Delta, \Omega - \Delta)$.

GAP makes use of the advantages of a "meet strategy" if the refinement function in `Refinements` contains a `MeetPartitionStrat` command where *strat* is the "meet strategy" calculated by `StratMeetPartition` before. Such a command replaces $\mathcal{I}$`.partition` by its meet with Λ', again changing the argument $\mathcal{I}$. The necessary reversal of these changes when backtracking from a node (and prescribing the next possible image for a base point) is automatically done by the function `PartitionBacktrack`.

In all cases, an additional argument g means that the meet is to be taken not with Λ, but instead with $\Lambda \cdot g^{-1}$, where operation on ordered partitions is meant cellwise (and setwise on each cell). (Analogously for the primed arguments.)

```
gap> P := Partition( [[1,2],[3,4,5],[6]] );;
gap> StratMeetPartition( P, P, (1,6,3) );;  Cells( P );
[ [ 1 ], [ 5, 4 ], [ 6 ], [ 2 ], [ 3 ] ]  # P·(1,3,6) = Q
```

Avoiding multiplication of permutations. In the description of the last subsections, the backtrack algorithm constructs an element $c \in G$ mapping the base points to the prescribed images and finally tests the property in question for that element. During the construction, c is obtained as a product of transversal elements from the stabilizer chain for G, and so multiplications of permutations are required for every c submitted to the test, even if the test fails (i.e., in our centralizer example, if g `^` c `<>` g). Even if the construction of c stops before images for all base points have been chosen, because a refinement was unsuccessful, several multiplications will already have been performed by (explicit or implicit) calls of `ProcessFixpoint`, and, actually, the general backtrack procedure implemented in GAP avoids this.

For this purpose, GAP does not actually multiply the permutations but rather stores all the factors of the product in a list. Specifically, instead of carrying out the multiplication in $c \mapsto c' \cdot c$ mentioned in the comment to line 5 of the above program — where $c' \in G_{a_1 \ldots a_{i+1}}$ is a product of factorized inverse transversal elements, see 41.8 in the reference manual — GAP appends the list of these factorized inverse transversal elements (giving c') to the list of factors already collected for c. Here c' is multiplied from the left and is itself a product of **inverses** of strong generators of G, but GAP simply spares itself all the work of inverting permutations and stores only a "list of inverses", whose product is then $(c' \cdot c)^{-1}$ (which is the new value of c^{-1}). The "list of inverses" is extended this way whenever `ProcessFixpoint` is called to improve c.

The product has to be multiplied out only when the property is finally tested for the element c. But it is often possible to delay the multiplication even further, namely until after the test, so that no multiplication is required in the case of an unsuccessful test. Then the test itself must be carried out with the factorized version of the element c. For this purpose, `PartitionBacktrack` can be passed its second argument (the property in question) in a different way, not as a single GAP function, but as a list like in lines 2–4 of the following alternative excerpt from the code for `Centralizer`.

```
1 return PartitionBacktrack( G,
```

```
2   [ g, g,
3     OnPoints,
4     c -> c!.lftObj = c!.rgtObj ],
5   false, R, [ Π0, g ], L, R );
```

The test for c to have the property in question is of the form *opr*`(` *left*`,` c `) =` *right* where *opr* is an operation function as explained in 39.11 in the reference manual. In other words, c passes the test if and only if it maps a “left object” to a “right object” under a certain operation. In the centralizer example, we have *opr* `= OnPoints` and *left* = *right* = g, but in a conjugacy test, we would have *right* = h.

2. Two first two entries (here g and g) are the values of *left* and *right*.
3. The third entry (here `OnPoints`) is the operation *opr*.
4. The fourth entry is the test to be performed upon the mapped left object *left* and preimage of the right object *opr*`(` *right*`,` c`^-1 )`. Here GAP operates with the inverse of c because this is the product of the permutations stored in the “list of inverses”. The preimage of *right* under c is then calculated by mapping *right* with the factors of c^{-1} one by one, without the need to multiply these factors. This mapping of *right* is automatically done by the `ProcessFixpoint` function whenever c is extended, the current value of *right* is always stored in c`!.rgtObj`. When the test given by the fourth entry is finally performed, the element c has two components c`!.lftObj =` *left* and c`!.rgtObj =` *opr*`(` *right*`,` c`^-1 )`, which must be used to express the desired relation as a function of c. In our centralizer example, we simply have to test whether they are equal.

8.3 Stabilizer Chains for Automorphisms Acting on Enumerators

This section describes a way of representing the automorphism group of a group as permutation group, following [Sim97]. The code however is not yet included in the GAP library.

In this section we present an example in which objects we already know (namely, automorphisms of solvable groups) are equipped with the permutation-like operations `^` and `/` for action on positive integers. To achieve this, we must define a new type of objects which behave like permutations but are represented as automorphisms acting on an enumerator. Our goal is to generalize the Schreier-Sims algorithm for construction of a stabilizer chain to groups of such new automorphisms.

An operation domain for automorphisms. The idea we describe here is due to C. Sims. We consider a group A of automorphisms of a group G, given by generators, and we would like to know its order. Of course we could follow the strategy of the Schreier-Sims algorithm (described in 41.5 in the reference manual) for A acting on G. This would involve a call of `StabChainStrong( EmptyStabChain( [ ], One(` A `) ), GroupGenerators(` A `) )` where `StabChainStrong` is a function as the one described in the pseudo-code below:

```
StabChainStrong := function( S, newgens )
  Extend the Schreier tree of S with newgens.
  for sch in Schreier generators do
    if sch ∉ S.stabilizer then
      StabChainStrong( S.stabilizer, [ sch ] );
    fi;
  od;
end;
```

The membership test $sch \notin S$`.stabilizer` can be performed because the stabilizer chain of S`.stabilizer` is already correct at that moment. We even know a base in advance, namely any generating set for G. Fix such a generating set $(g_1, \ldots, g_d)$ and observe that this base is generally very short compared to the degree $|G|$ of the operation. The problem with the Schreier-Sims algorithm, however, is then that the length of the first basic orbit $g_1 \cdot A$ would already have the magnitude of $|G|$, and the basic orbits at deeper levels would not be much shorter. For the advantage of a short base we pay the high price of long basic orbits, since the product of the (few) basic orbit lengths must equal $|A|$. Such long orbits make the Schreier-Sims algorithm infeasible, so we have to look for a longer base with shorter basic orbits.

Assume that G is solvable and choose a characteristic series with elementary abelian factors. For the sake of simplicity we assume that $N < G$ is an elementary abelian characteristic subgroup with elementary abelian factor group G/N. Since N is characteristic, A also acts as a group of automorphisms on the factor group G/N, but of course not necessarily faithfully. To retain a faithful action, we let A act on the disjoint union G/N with G, and choose as base $(g_1N, \ldots, g_dN, g_1, \ldots, g_d)$. Now the first d basic orbits lie inside G/N and can have length at most $[G : N]$. Since the base points $g_1N, \ldots, g_dN$ form a generating set for G/N, their iterated stabilizer $A^{(d+1)}$ acts trivially on the factor group G/N, i.e., it leaves the cosets g_iN invariant. Accordingly, the next d basic orbits lie inside g_iN (for $i = 1, \ldots, d$) and can have length at most $|N|$.

Generalizing this method to a characteristic series $G = N_0 > N_1 > \ldots > N_l = \{1\}$ of length $l > 2$, we can always find a base of length $l \cdot d$ such that each basic orbit is contained in a coset of a characteristic factor, i.e. in a set of the form g_iN_{j-1}/N_j (where g_i is one of the generators of G and $1 \le j \le l$). In particular, the length of the basic orbits is bounded by the size of the corresponding characteristic factors. To implement a Schreier-Sims algorithm for such a base, we must be able to let automorphisms act on cosets of characteristic factors g_iN_{j-1}/N_j, for varying i and j. We would like to translate each such action into an action on $\{1, \ldots, [N_{j-1} : N_j]\}$, because then we need not enumerate the operation domain

$$G/N_1 \dot\cup G/N_2 \dot\cup \ldots \dot\cup G/N_l$$

as a whole. Enumerating it as a whole would result in basic orbits like `orbit` $\subseteq \{1001, \ldots, 1100\}$ with a `transversal` list whose first 1000 entries would be unbound, but still require 4 bytes of memory each (see 41.8 in the reference manual).

Identifying each coset g_iN_{j-1}/N_j into $\{1, \ldots, [N_{j-1} : N_j]\}$ of course means that we have to change the action of the automorphisms on every level of the stabilizer chain. Such flexibility is not possible with permutations because their effect on positive integers is "hardwired" into them, but we can install new operations for automorphisms.

Enumerators for cosets of characteristic factors. So far we have not used the fact that the characteristic factors are elementary abelian, but we will do so from here on. Our first task is to implement an enumerator (see 28.2.6 and 21.23 in the reference manual) for a coset of a characteristic factor in a solvable group G. We assume that such a coset gN/M is given by

(1) a pcgs for the group G (see 43.2.1 in the reference manual), let n `= Length(` *pcgs* `);`

(2) a range *range* `= [` *start* `..` *stop* `]` indicating that $N = \langle$*pcgs*`{ [` *start* `..` *n* `] }` $\rangle$ and $M = \langle$*pcgs*`{ [` *stop* `+ 1 ..` *n* `] }` $\rangle$, i.e., the cosets of *pcgs*`{` *range* `}` form a base for the vector space N/M;

(3) the representative g.

We first define a new representation for such enumerators and then construct them by simply putting these three pieces of data into a record object. The enumerator should behave as a list of group elements (representing cosets modulo M), consequently, its family will be the family of the *pcgs* itself.

```
IsCosetSolvableFactorEnumeratorRep := NewRepresentation
    ( "isCosetSolvableFactorEnumerator", IsEnumerator,
                                 [ "pcgs", "range", "representative" ] );
```

```
EnumeratorCosetSolvableFactor := function( pcgs, range, g )
    return Objectify( NewKind( FamilyObj( pcgs ),
                   IsCosetSolvableFactorEnumeratorRep ),
                   rec( pcgs := pcgs,
                       range := range,
              representative := g ) );
end;
```

The definition of the operations `Length`, `\[\]` and `Position` is now straightforward. The code has sometimes been abbreviated and is meant “cum grano salis”, e.g., the declaration of the local variables has been left out.

```
InstallMethod( Length, [ IsCosetSolvableFactorEnumeratorRep ],
    enum -> Product( RelativeOrdersPcgs( enum!.pcgs ){ enum!.range } ) );

InstallMethod( \[\], [ IsCosetSolvableFactorEnumeratorRep,
        IsPosRat and IsInt ],
    function( enum, pos )
    elm := ();
    pos := pos - 1;
    for i  in Reversed( enum!.range )  do
        p := RelativeOrderOfPcElement( enum!.pcgs, i );
        elm := enum!.pcgs[ i ] ^ ( pos mod p ) * elm;
        pos := QuoInt( pos, p );
    od;
    return enum!.representative * elm;
end );

InstallMethod( Position, [ IsCosetSolvableFactorEnumeratorRep,
        IsObject, IsZeroCyc ],
    function( enum, elm, zero )
    exp := ExponentsOfPcElement( enum!.pcgs,
                   LeftQuotient( enum!.representative, elm ) );
    pos := 0;
    for i  in enum!.range  do
        pos := pos * RelativeOrderOfPcElement( pcgs, i ) + exp[ i ];
    od;
    return pos + 1;
end );
```

Making automorphisms act on such enumerators. Our next task is to make automorphisms of the solvable group *pcgs*`!.group` act on `[ 1 .. Length(` *enum* `) ]` for such an enumerator *enum*. We achieve this by introducing a new representation of automorphisms on enumerators and by putting the enumerator together with the automorphism into an object which behaves like a permutation. Turning an ordinary automorphism into such a special automorphism requires then the construction of a new object which has the new kind. We provide an operation `PermOnEnumerator(` *model*`,` *aut* `)` which constructs such a new object having the same kind as *model*, but representing the automorphism *aut*. So *aut* can be either an ordinary automorphism or one which already has an enumerator in its kind, but perhaps different from the one we want (i.e. from the one in *model*).

```
IsPermOnEnumerator := NewCategory( "IsPermOnEnumerator",
    IsMultiplicativeElementWithInverse and IsPerm );

IsPermOnEnumeratorDefaultRep := NewRepresentation
    ( "IsPermOnEnumeratorDefaultRep",
      IsPermOnEnumerator and IsAttributeStoringRep,
      [ "perm" ] );

PermOnEnumerator := NewOperation( "PermOnEnumerator",
    [ IsEnumerator, IsObject ] );

InstallMethod( PermOnEnumerator,
        [ IsEnumerator, IsObject ],
    function( enum, a )
    SetFilterObj( a, IsMultiplicativeElementWithInverse );
    a := Objectify( NewKind( PermutationsOnEnumeratorsFamily,
                 IsPermOnEnumeratorDefaultRep ),
                 rec( perm := a ) );
    SetEnumerator( a, enum );
    return a;
end );

InstallMethod( PermOnEnumerator,
        [ IsEnumerator, IsPermOnEnumeratorDefaultRep ],
    function( enum, a )
    a := Objectify( TypeObj( a ), rec( perm := a!.perm ) );
    SetEnumerator( a, enum );
    return a;
end );
```

Next we have to install new methods for the operations which calculate the product of two automorphisms, because this product must again have the right kind. We also have to write a function which uses the enumerators to apply such an automorphism to positive integers.

```
InstallMethod( \*, IsIdenticalObj,
        [ IsPermOnEnumeratorDefaultRep, IsPermOnEnumeratorDefaultRep ],
    function( a, b )
    perm := a!.perm * b!.perm;
    SetIsBijective( perm, true );
    return PermOnEnumerator( Enumerator( a ), perm );
end );

InstallMethod( \^,
        [ IsPosRat and IsInt, IsPermOnEnumeratorDefaultRep ],
    function( p, a )
    return PositionCanonical( Enumerator( a ),
                   Enumerator( a )[ p ] ^ a!.perm );
end );
```

How the corresponding methods for *p* `/` *aut* and *aut* `^` *n* look like is obvious.

Now we can formulate the recursive procedure `StabChainStrong` which extends the stabilizer chain by adding in new generators *newgens*. We content ourselves again with pseudo-code, emphasizing only the lines which set the `EnumeratorDomainPermutation`. We assume that initially S is a stabilizer chain for the trivial subgroup with a level for each pair (*range*, g) characterizing an enumerator (as described above). We also assume that the `identity` element at each level already has the kind corresponding to that level.

```
StabChainStrong := function( S, newgens )
  for i in [ 1 .. Length( newgens ) ] do
    newgens[ i ] := AutomorphismOnEnumerator( S.identity, newgens[ i ] );
  od;
  Extend the Schreier tree of S with newgens.
  for sch in Schreier generators do
    if sch ∉ S.stabilizer then
      StabChainStrong( S.stabilizer, [ sch ] );
    fi;
  od;
end;
```

Bibliography

[Sim97] Charles C. Sims. Computing with subgroups of automorphism groups of finite groups. In Wolfgang Küchlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 40–403. The Association for Computing Machinery, ACM Press, 1997.

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".