Example

Example/Template of a GAP Package and Guidelines for Package Authors

Version 3.4.3

27 March 2013

Werner Nickel
Greg Gamble
Alexander Konovalov

Werner Nickel Email: nickel@mathematik.tu-darmstadt.de
Homepage: http://www.mathematik.tu-darmstadt.de/~nickel

Greg Gamble Email: gregg@math.rwth-aachen.de
Homepage: http://www.math.rwth-aachen.de/~Greg.Gamble

Alexander Konovalov Email: alexk@mcs.st-andrews.ac.uk
Homepage: http://www.cs.st-andrews.ac.uk/~alexk/

Copyright

Acknowledgements

We appreciate very much all past and future comments, suggestions and contributions to this package and its documentation provided by GAP users and developers.

Contents

Chapter 1

The Example Package

This chapter describes the GAP package Example. As its name suggests it is an example of how to create a GAP package. It has little functionality except for being a package.

See Sections 2.1, 2.2 and 2.3 for how to install, compile and load the Example package, or Appendix A for guidelines on how to write a GAP package.

If you are viewing this with on-line help, type:

Example
```
gap> ?Example package
```

to see the functions provided by the Example package.

1.1 The Main Functions

The following functions are available:

1.1.1 ListDirectory

▷ `ListDirectory([dir])` (function)

lists the files in directory *dir* (a string) or the current directory if called with no arguments.

1.1.2 FindFile

▷ `FindFile(directory_name, file_name)` (function)

searches for the file *file_name* in the directory tree rooted at *directory_name* and returns the absolute path names of all occurrences of this file as a list of strings.

1.1.3 LoadedPackages

▷ `LoadedPackages()` (function)

returns a list with the names of the packages that have been loaded so far. All this does is execute

Example
```
gap> RecNames( GAPInfo.PackagesLoaded );
```

You might like to check out some of the other information in the `GAPInfo` record (see (**Reference: GAPInfo**)).

1.1.4 Which

▷ `Which(`*prg*`)` (function)

returns the path of the program executed if `Exec(`*prg*`);` is called, e.g.

```
gap> Which("date");
"/bin/date"
gap> Exec("date");
Fri 28 Jan 2011 16:22:53 GMT
```

1.1.5 WhereIsPkgProgram

▷ `WhereIsPkgProgram(`*prg*`)` (function)

returns a list of paths of programs with name *prg* in the current packages loaded. Try:

```
gap> WhereIsPkgProgram( "hello" );
```

1.1.6 HelloWorld

▷ `HelloWorld()` (function)

executes the C program `hello` provided by the Example package.

1.1.7 FruitCake

▷ `FruitCake` (global variable)

is a record with the bits and pieces needed to make a boiled fruit cake. Its fields satisfy the criteria for `Recipe` (1.1.8).

1.1.8 Recipe

▷ `Recipe(`*cake*`)` (operation)

displays the recipe for cooking *cake*, where *cake* is a record satisfying certain criteria explained here: its recognised fields are `name` (a string giving the type of cake or cooked item), `ovenTemp` (a string), `cookingTime` (a string), `ingredients` (a list of strings each containing an _ which is used to line up the entries and is replaced by a blank), `method` (a list of steps, each of which is a string or list of strings), and `notes` (a list of strings). The global variable `FruitCake` (1.1.7) provides an example of such a string.

Chapter 2

Installing and Loading the Example Package

2.1 Unpacking the Example Package

If the Example package was obtained as a part of the GAP distribution from the “Download” section of the GAP website, you may proceed to Section 2.2. Alternatively, the Example package may be installed using a separate archive, for example, for an update or an installation in a non-default location (see (**Reference: GAP Root Directories**)).

Below we describe the installation procedure for the `.tar.gz` archive format. Installation using other archive formats is performed in a similar way.

To install the Example package, unpack the archive file, which should have a name of form `example-`*`XXX`*`.tar.gz` for some version number *`XXX`*, by typing

```
gzip -dc example-XXX.tar.gz | tar xpv
```

It may be unpacked in one of the following locations:

- in the `pkg` directory of your GAP 4 installation;
- or in a directory named `.gap/pkg` in your home directory (to be added to the GAP root directory unless GAP is started with `-r` option);
- or in a directory named `pkg` in another directory of your choice (e.g. in the directory `mygap` in your home directory).

In the latter case one one must start GAP with the `-l` option, e.g. if your private `pkg` directory is a subdirectory of `mygap` in your home directory you might type:

```
gap -l ";myhomedir/mygap"
```

where *`myhomedir`* is the path to your home directory, which (since GAP 4.3) may be replaced by a tilde (the empty path before the semicolon is filled in by the default path of the GAP 4 home directory).

2.2 Compiling Binaries of the Example Package

After unpacking the archive, go to the newly created `example` directory and call `./configure` to use the default `../..` path to the GAP home directory or `./configure` *`path`* where *`path`* is the path to the GAP home directory, if the package is being installed in a non-default location. So for example if you install the package in the ~/.gap/pkg directory and the GAP home directory is ~/gap4r5 then you have to call

```
./configure ../../../gap4r5/
```

This will fetch the architecture type for which GAP has been compiled last and create a Makefile. Now simply call

```
make
```

to compile the binary and to install it in the appropriate place.

2.3 Loading the Example Package

To use the Example Package you have to request it explicitly. This is done by calling LoadPackage (**Reference: LoadPackage**):

```
gap> LoadPackage("example");
----------------------------------------------------------------
Loading  Example 3.3 (Example/Template of a GAP Package)
by Werner Nickel (http://www.mathematik.tu-darmstadt.de/~nickel),
   Greg Gamble (http://www.math.rwth-aachen.de/~Greg.Gamble), and
   Alexander Konovalov (http://www.cs.st-andrews.ac.uk/~alexk/).
----------------------------------------------------------------
true
```

If GAP cannot find a working binary, the call to LoadPackage will still succeed but a warning is issued informing that the HelloWorld() function will be unavailable.

If you want to load the Example package by default, you can put the LoadPackage command into your gaprc file (see Section (**Reference: The gap.ini and gaprc files**)).

Appendix A

Guidelines for Writing a GAP Package

This appendix 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 (**Reference: GAP Packages**).

There are two basic aspects of creating a GAP package.

First, it is a convenient possibility to load additional functionality into GAP including a smooth integration of the package documentation. Second, a package is a way to make your code available to other GAP users.

Moreover, the GAP Group may provide some help with redistributing your package via the web and ftp site of GAP itself after checking if the package provides some new or improved functionality which looks interesting for other users, if it contains reasonable documentation, and if it seems to work smoothly with the GAP library and other distributed packages. In this case the package can take part in the GAP distribution update mechanism and becomes a *deposited* package.

Furthermore, package authors are encouraged to check if the package would be appropriate for the refereeing process and *submit* it. If the refereeing has been successful, the package becomes an *accepted* package. Check out the GAP Web site `http://www.gap-system.org` for more details.

We start this chapter with a description how the directory structure of a GAP package should be constructed and then add remarks on certain aspects of creating a package, some of these only apply to some packages. Finally, we provide guidelines for the release preparation, wrapping and distribution.

A.1 Structure of a GAP Package

A GAP package should have an alphanumeric name (`package-name`, say); mixed case is fine, but there should be no whitespaces. All files of a GAP package `package-name` must be collected in a single directory `package-dir`, where `package-dir` should be just `package-name` preferably converted to lowercase and optionally followed by the package version (with or without hyphen to separate the version from `package-name`). Let us call this directory the *home directory* of the package.

To use the package with GAP, the directory `package-dir` must be a subdirectory of a `pkg` directory in (one of) the GAP root directories (see (**Reference: GAP Root Directories**)). For example, if GAP is installed in `/usr/local/gap4` then the files of the package `MyPack` may be placed in the directory `/usr/local/gap4/pkg/mypack`. The directory `package-dir` preferably should have the following structure (below, a trailing `/` distinguishes directories from ordinary files):

```
Example
package-dir/
  doc/
  lib/
  src/
  tst/
  README
  CHANGES
  configure
  Makefile.in
  PackageInfo.g
  init.g
  read.g
```

There are three file names with a special meaning in the home directory of a package: `PackageInfo.g` and `init.g` which must be present, and `read.g` which is optional. We now describe the above files and directories:

`README`
: This should contain "how to get it" (from the GAP `ftp`- and web-sites) instructions, as well as installation instructions and names of the package authors and their email addresses. The installation instructions should be repeated in the package's documentation, which should be in the `doc` directory (see A.2). Authors' names and addresses should be repeated both in the package's documentation and in the `PackageInfo.g` (see below).

`CHANGES`
: For further versions of the package, it will be also useful to have a `CHANGES` file that records the main changes between versions of the package.

`configure`, `Makefile.in`
: These files are only necessary if the package has a non-GAP component, e.g. some C code (the files of which should be in the `src` directory). The `configure` and `Makefile.in` files of the Example package provide prototypes. The `configure` file typically takes a path *`path`* to the GAP root directory as argument and uses the value assigned to `GAParch` in the file `sysinfo.gap`, created when GAP was compiled to determine the compilation architecture, inserts this in place of the string `@GAPARCH@` in `Makefile.in` and creates a file `Makefile`. When `make` is run (which, of course, reads the constructed `Makefile`), a directory `bin` (if necessary) and subdirectories of `bin` with the path equal to the string assigned to `GAParch` in the file `sysinfo.gap` should be created; any binaries constructed by compiling the code in `src` should end up in this subdirectory of `bin`.

`PackageInfo.g`
: Since GAP 4.4, a GAP package *must* have a `PackageInfo.g` file which contains meta-information about the package (package name, version, author(s), relations to other packages, homepage, download archives, banner, ...). This information is used by the package loading mechanism and also for the distribution of a package to other users. The Example package's `PackageInfo.g` file is well-commented and should be used as a prototype (see also A.5 for further details).

`init.g`, `read.g`

A GAP package *must* have a file `init.g`. As of GAP 4.4, the typical `init.g` and `read.g` files should normally consist entirely of `ReadPackage` (**Reference: ReadPackage**) commands (and possibly also `Read` (**Reference: Read**) commands) for reading further files of the package. If the “declaration” and “implementation” parts of the package are separated (and this is recommended), there should be a `read.g` file. The “declaration” part of a package consists of function and variable *name* declarations and these go in files with `.gd` extensions; these files are read in via `ReadPackage` commands in the `init.g` file. The “implementation” part of a package consists of the actual definitions of the functions and variables whose names were declared in the “declaration” part, and these go in files with `.gi` extensions; these files are read in via `ReadPackage` commands in the `read.g` file. The reason for following the above dichotomy is that the `read.g` file is read *after* the `init.g` file, thus enabling the possibility of a function’s implementation to refer to another function whose name is known but is not actually defined yet (see A.8 below for more details).

The GAP code (whether or not it is split into “declaration” and “implementation” parts) should go in the package’s `lib` directory (see below).

`doc`

This directory should contain the package’s documentation. It is strongly recommended to use an XML-based documentation format supported by the GAP package GAPDoc (see (**GAPDoc: Introduction and Example**)) which is used for the GAP documentation. An alternative is to use the TEX-based system, formerly used for the GAP documentation in GAP 4.4 and earlier releases. This system is described in the document “The gapmacro.tex Manual Format” (the file `gap4r5/doc/gapmacrodoc.pdf` included in the tools archive as described in Section A.2) and is still used by some of the GAP packages whose authors are encouraged to switch at some point to the GAPDoc-based documenation. Please spend some time reading the documentation for whichever system you decide to use for writing your package’s documentation. The Example package’s documentation is written in the XML format supported by the GAPDoc package. If you intend to use this format, please use the Example package’s `doc` directory as a prototype, which as you will observe contains the master file `main.xml`, further `.xml` files for manual chapters (included in the manual via `Include` directives in the master file) and the GAP input file `../makedocrel.g` which generates the manuals. Generally, one should also provide a `manual.bib` BibTEX database file or an `xml` file in the BibXMLext format (see (**GAPDoc: The BibXMLext Format**)). With `gapmacro.tex`, it is also possible to use a `manual.bbl` file.

`lib`

This is the preferred place for the GAP code, i.e. the `.g`, `.gd` and `.gi` files (other than `PackageInfo.g`, `init.g` and `read.g`). For some packages, the directory `gap` has been used instead of `lib`; `lib` has the slight advantage that it is the default subdirectory of a package directory searched for by the `DirectoriesPackageLibrary` (**Reference: DirectoriesPackageLibrary**) command.

`src`

If the package has non-GAP code, e.g. C code, then this “source” code should go in the `src` directory. If there are `.h` “include” files you may prefer to put these all together in a separate `include` directory. There is one further rule for the location of kernel library modules or external programs which is explained in A.10.1 below.

`tst`

If the package has test files, then they should go in the `tst` directory. For a deposited package, a test file with a basic test of the package (for example, to check that it works as expected and/or that the manual examples are correct) may be specified in the `PackageInfo.g` to be included in the GAP standard test suite. More specific and time consuming tests are not supposed to be a part of the GAP standard test suite but may be placed in the `tst` directory with further instructions on how to run them. See Section A.14 about the requirements to the test files formats and further recommendations.

All other files can be organized as you like. But we suggest that you have a look at existing packages and use a similar scheme, for example, put examples in the `examples` subdirectory, data libraries in extra subdirectories, and so on.

Sometimes there may be a need to include an empty directory in the package distribution (for example, as a place to store some data that may appear at runtime). In this case package authors are advised to put in this directory a short `README` file describing its purpose to ensure that such directory will be included in the redistribution.

Concerning the GAP code in packages, it is recommended to use only documented GAP functions, see (**Reference: Undocumented Variables**). In particular if you want to make your package available to other GAP users it is advisable to avoid using “obsolescent” variables (see (**Reference: Replaced and Removed Command Names**)). For that, you can set the `ReadObsolete` component in your `gap.ini` file to `false`, see (**Reference: The gap.ini and gaprc files**).

A.2 Writing Documentation and Tools Needed

If you intend to make your package available to other users it is essential to include documentation explaining how to install and use your programs.

Concerning the installation you should produce a file `README` which gives a short description of the purpose of the package and contains proper instructions how to install your package. Again, check out some existing packages to get an idea how this could look like.

Concerning the documentation of the use of the package there are currently two recognised ways of producing GAP package documentation.

First, there is a recommended XML-based documentation format that is defined in and can be used with the GAPDoc package (see (**GAPDoc: Introduction and Example**)).

Second, there is a method which requires the documentation to be written in $\TeX$ according to the format described in the document “The gapmacro.tex Manual Format”.

In principle it is also possible to use some completely different documentation format. In that case you need to study the Chapter (**Reference: Interface to the GAP Help System**) to learn how to make your documentation available to the GAP help system. There should be at least a text version of your documenation provided for use in the terminal running GAP and some nicely printable version in `.pdf` format. Many GAP users like to browse the documentation in HTML format via their Web browser. As a package author, you are not obliged to provide an HTML version of your package manual, but if you will either use the GAPDoc package or follow the guidelines in the document “The gapmacro.tex Manual Format”, (the file `gap4r5/doc/gapmacrodoc.pdf` included in the tools archive as described in this Section below), you should have no trouble in producing one. Moreover, using the GAPDoc package, it is also possible to produce HTML version of the documentation supporting MathJax (`http://www.mathjax.org/`) for the high quality rendering of mathematical symbols while viewing it online. For example, if you are viewing the HTML version of the manual, compare how this formula will look with MathJax turned on/off:

$$[\chi,\psi] = \left(\sum_{g \in G} \chi(g)\psi(g^{-1}) \right) / |G|.$$

The manual of the Example package is written in the GAPDoc format, and commands needed to build it are contained in the file `makedocrel.g` (you don't need to re-build the manual since it is already included in the package).

Whenever you use the GAPDoc or `gapmacro.tex` TeX-based system, you need to have certain TeX tools installed: to produce manuals in the `.pdf` format, you need `pdflatex` if the GAPDoc is used, and either `pdftex` or `gs` together with `ps2pdf` if your package uses `gapmacro.tex`. Note that using `gs` and `ps2pdf` in lieu of `pdftex` or `pdflatex` is a poor substitute unless your `gs` is at least version 6.*xx* for some *xx*. In addition, the `gapmacro.tex` documentation system requires some more tools described below. If you intend to use the GAPDoc package for the documenation of your package, you may skip the rest of this section and proceed to the next one to see a minimalistic example of a GAP package.

Otherwise, to produce the `.pdf` manual formats, the following GAP tools (supplied as a part of the GAP distribution in the archive `tools.tar.gz` in the in GAP's `etc` directory and installed using the script `install-tools.sh` in the same directory) are needed: `gapmacro.tex` (the macros file that dictates the style and mark-up for the traditional TeX-based system of GAP documentation), `manualindex` (an `awk` script that adjusts the TeX-produced index entries and calls `makeindex` to process them), and `mrabbrev.bib` (usually supplied with your TeX tools but nevertheless a copy of `mrabbrev.bib` should be located in GAP's main `doc` directory. To find it on your system, try: `kpsewhich mrabbrev.bib` or, if that doesn't work and you can't otherwise find it check out a CTAN site, e.g. search for it at: `http://www.dante.de/cgi-bin/ctan-index`.

If your manual cross-refers to GAPDoc- or `gapmacro.tex`-produced manuals, then `manual.lab` for each such other manual is needed. For packages using GAPDoc-manuals since GAP 4.3, this is done by starting GAP and running

```
 gap> GapDocManualLab( "package" );
```

for each such *package* whose manual is cross-referred to (this includes the "main" manuals, e.g. those in the `doc/ref` and `doc/tut` directories). For packages using `gapmacro.tex`-produced manuals, `manual.lab` is generated by running `tex manual` for each package whose manual is cross-referred to. In most cases, packages from the GAP distribution will already have these files since they will be created as a part of building their manuals (see e.g. the last command of the `example/makedocrel.g` script) and included in their distribution, so you will probably not need to create `manual.lab` files yourself.

To produce an HTML version of the manual one needs the Perl 5 program `convert.pl` which is included in the tools archive `tools.tar.gz`. This archive is supplied as a part of the GAP distribution in the GAP's `etc` directory and should be installed using the script `install-tools.sh` in the same directory.

Finally, to ensure the mathematical formulae are rendered as well as they can be in the HTML version, one should also have the program `tth` (TeX to HTML converter); `convert.pl` calls `tth` to translate mathmode formulae to HTML (if it's available). The `tth` program is easy to compile and can be obtained from `http://hutchinson.belmont.ma.us/tth/tth-noncom/download.html`.

A.3 An Example of a GAP Package

We illustrate the creation of a GAP package by an example of a basic package.

Create the following directories in your home area: `.gap`, `.gap/pkg` and `.gap/pkg/test`. Then inside the directory `.gap/pkg/test` create an empty file `init.g`, and a file `PackageInfo.g` with the following contents:

Example
```
SetPackageInfo( rec(
  PackageName := "test",
  Version := "1.0",
  PackageDoc := rec(
      BookName  := "test",
      SixFile   := "doc/manual.six",
      Autoload  := true ),
  Dependencies := rec(
      GAP       := "4.5",
      NeededOtherPackages := [ ["GAPDoc", "1.3"] ],
      SuggestedOtherPackages := [ ] ),
  AvailabilityTest := ReturnTrue ) );
```

This file declares the GAP package with name "test" in version 1.0. The package documentation consists of one autoloaded book; the `SixFile` component is needed by the GAP help system. Package dependencies require at least GAP 4.5 and GAPDoc package at version at least 1.3, and these conditions will be checked when the package will be loaded (see A.13). Since there are no requirements that have to be tested, `AvailabilityTest` just uses `ReturnTrue` (**Reference: ReturnTrue**).

Now start GAP (without using the `-r` option) and the `.gap` directory will be added to the GAP root directory to allow GAP to find the packages installed there (see (**Reference: GAP Root Directories**)).

Example
```
gap> LoadPackage("test");
true
```

This GAP package is too simple to be useful, but we have succeeded in loading it via `LoadPackage` (**Reference: LoadPackage**), satisfying all specified dependencies.

A.4 File Structure

Package files may follow the style used for the GAP library. Every file in the GAP library starts with a header that lists the filename, copyright, a short description of the file contents and the original authors of this file, and ends with a comment line `#E`. Indentation in functions and the use of decorative spaces in the code are left to the decision of the authors of each file. Global (i.e. re-used elsewhere) comments usually are indented by two hash marks and two blanks, in particular, every declaration or method or function installation which is not only of local scope is separated by a header.

Historically, when the GAP main manuals were based on the TeX macros described in the document "The gapmacro.tex Manual Format" (the file `gap4r5/doc/gapmacrodoc.pdf` included in the tools archive as described in Section A.2) such headers were used for the manuals and have the type

Example
```
#############################################################################
##
```

```
#X  ExampleFunction(<A>,<B>)
##
##  This function does great things.
```

where “X” was one of the letters: F, A, P, O, C, R or V indicating whether the object declared will be a function, attribute, property, operation, category, representation or variable, respectively. Additionally M was used in .gi files for method installations. Then a sample usage of the function was given, followed by a comment which described the identifier. This description was automatically be extracted to build the manual source, if there is a \Declaration line in some .msk file together with an appropriate configuration file.

Nowadays, facilities to distribute a document over several files to allow the documentation for parts of some code to be stored in the same file as the code itself are provided by the GAPDoc package (see (**GAPDoc: Distributing a Document into Several Files**)). The same approach is demonstrated by the Example package. E.g. example/doc/example.xml has the statement <#Include Label="ListDirectory"> and example/lib/files.gd contains

Example
```
###############################################################################
##
#F  ListDirectory([<dir>])  . . . . . . . . . . list the files in a directory
##
##  <#GAPDoc Label="ListDirectory">
##  <ManSection>
##  <Func Name="ListDirectory" Arg="[dir]"/>
##
##  <Description>
##  lists the files in directory <A>dir</A> (a string)
##  or the current directory if called with no arguments.
##  </Description>
##  </ManSection>
##  <#/GAPDoc>
DeclareGlobalFunction( "ListDirectory" );
```

This is all put together in the file example/makedocrel.g which builds the package documentation, calling MakeGAPDocDoc (**GAPDoc: MakeGAPDocDoc**) with locations of library files containing parts of the documentation.

A.5 The PackageInfo.g File

As a first step the example in A.3 shows the information needed for the package loading mechanism to load a simple package. However, if your package depends on the functionality of other packages, the component Dependencies given in the PackageInfo.g file becomes important (see A.7 below), and more entries become relevant if you want to distribute your package: they contain lists of authors and/or maintainers including contact information, URLs of the package archives and README files, status information, text for a package overview Web page, and so on.

We suggest to create a PackageInfo.g file for your package by copying the one in the Example package, distributed with GAP, and then adjusting it for your package. Within GAP you can look at this template file for a list and explanation of all recognized entries by

Example
```
Pager(StringFile(Filename(DirectoriesLibrary(),
                          "../pkg/example/PackageInfo.g")));
```

Once you have created the `PackageInfo.g` file for your package, you can test its validity with the function `ValidatePackageInfo` (**Reference: ValidatePackageInfo**).

A.6 Functions and Variables and Choices of Their Names

In writing the GAP code for your package you need to be a little careful on just how you define your functions and variables.

Firstly, in general one should avoid defining functions and variables via assignment statements in the way you would interactively, e.g.

Example
```
gap> Squared := x -> x^2;;
gap> Cubed := function(x) return x^3; end;;
```

The reason for this is that such functions and variables are *easily overwritten* and what's more you are not warned about it when it happens.

To protect a function or variable against overwriting there is the command `BindGlobal` (**Reference: BindGlobal**), or alternatively (and equivalently) you may define a global function via a `DeclareGlobalFunction` (**Reference: DeclareGlobalFunction**) and `InstallGlobalFunction` (**Reference: InstallGlobalFunction**) pair or a global variable via a `DeclareGlobalVariable` (**Reference: DeclareGlobalVariable**) and `InstallValue` (**Reference: InstallValue**) pair. There are also operations and their methods, and related objects like attributes and filters which also have `Declare...` and `Install...` pairs.

Secondly, it's a good idea to reduce the chance of accidental overwriting by choosing names for your functions and variables that begin with a string that identifies it with the package, e.g. some of the undocumented functions in the Example package begin with `Eg`. This is especially important in cases where you actually want the user to be able to change the value of a function or variable defined by your package, for which you have used direct assignments (for which the user will receive no warning if she accidentally overwrites them). It's also important for functions and variables defined via `BindGlobal`, `DeclareGlobalFunction/InstallGlobalFunction` and `DeclareGlobalVariable/InstallValue`, in order to avoid name clashes that may occur with (extensions of) the GAP library and other packages.

Additionally, since GAP 4.5 a package may place global variables into a local namespace as explained in (**Reference: Namespaces for GAP packages**) in order to avoid name clashes and preserve compatibility. This new feature allows you to define in your package global variables with the identifier ending with the `@` symbol, e.g. `xYz@`. Such variables may be used in your package code safely, as they may be accessed from outside the package only by their full name, i.e. `xYz@YourPackageName`. This helps to prevent clashes between different packages or between a package and the GAP library because of the same variable names.

On the other hand, operations and their methods (defined via `DeclareOperation` (**Reference: DeclareOperation**), `InstallMethod` (**Reference: InstallMethod**) etc. pairs) and their relatives do not need this consideration, as they avoid name clashes by allowing for more than one "method" for the same-named object.

To demonstrate the definition of a function via a `DeclareOperation/InstallMethod` pair, the method `Recipe` (1.1.8) was included in the Example package; `Recipe( FruitCake );` gives a “method” for making a fruit cake (forgive the pun).

Thirdly, functions or variables with `Set`*`XXX`* or `Has`*`XXX`* names (even if they are defined as operations) should be avoided as these may clash with objects associated with attributes or properties (attributes and properties *`XXX`* declared via the `DeclareAttribute` and `DeclareProperty` commands have associated with them testers of form `Has`*`XXX`* and setters of form `Set`*`XXX`*).

Fourthly, it is a good idea to have some convention for internal functions and variables (i.e. the functions and variables you don’t intend for the user to use). For example, they might be entirely CAPITALISED.

Additionally, there is a recommended naming convention that the GAP core system and GAP packages should not use global variables starting in the lowercase. This allows to reserve variables with names starting in lowecase to the GAP user so they will never clash with the system. It is extremely important to avoid using for package global variables very short names started in lowercase. For example, such names like `cs`, `exp`, `ngens`, `pc`, `pow` which are perfectly fine for local variables, should never be used for globals. Additionally, the package must not have writable global variables with very short names even if they are starting in uppercase, for example, `C1` or `ORB`, since they also could be easily overwritten by the user.

It is a good practice to follow naming conventions used in GAP as explained in (**Reference: Naming Conventions**) and (**Tutorial: Changing the Structure**), which might help users to memorize or even guess names of functions provided by the package.

Finally, note the advantage of using `DeclareGlobalFunction/InstallGlobalFunction`, `DeclareGlobalVariable/InstallValue`, etc. pairs (rather than `BindGlobal`) to define functions and variables, which allow the package author to organise her function- and variable- definitions in any order without worrying about any interdependence. The `Declare...` statements should go in files with `.gd` extensions and be loaded by `ReadPackage` statements in the package `init.g` file, and the `Install...` definitions should go in files with `.gi` extensions and be loaded by `ReadPackage` statements in the package `read.g` file; this ensures that the `.gi` files are read *after* the `.gd` files. All other package code should go in `.g` files (other than the `init.g` and `read.g` files themselves) and be loaded via `ReadPackage` statements in the `init.g` file.

In conclusion, here is some practical advice on how to check which variables are used by the package.

Firstly, there is a function `ShowPackageVariables` (**Reference: ShowPackageVariables**). If the package *`pkgname`* is available but not yet loaded then `DisplayPackageVariables( pkgname )` prints a list of global variables that become bound and of methods that become installed when the package is loaded (for that, the package will be actually loaded, so `ShowPackageVariables` can be called only once for the same package in the same GAP session.) The second optional argument *`version`* may specify a particular package version to be loaded. An error message will be printed if (the given version of) the package is not available or already loaded.

Info lines for undocumented variables will be marked with an asterisk `*`. Note that the GAP help system is case insensitive, so it is difficult to document identifiers that differ only by case.

The following entries are omitted from the list: default setter methods for attributes and properties that are declared in the package, and `Set`*`attr`* and `Has`*`attr`* type variables where *`attr`* is an attribute or property.

For example, for this package it currently produces the following output:

Example
```
gap> ShowPackageVariables("example");
```

```
---------------------------------------------------------------
Loading  Example 3.3 (Example/Template of a GAP Package)
by Werner Nickel (http://www.mathematik.tu-darmstadt.de/~nickel),
   Greg Gamble (http://www.math.rwth-aachen.de/~Greg.Gamble), and
   Alexander Konovalov (http://www.cs.st-andrews.ac.uk/~alexk/).
---------------------------------------------------------------
new global functions:
  EgSeparatedString( str, c )*
  FindFile( dir, file )
  HelloWorld(  )
  ListDirectory( arg )
  LoadedPackages(  )
  WhereIsPkgProgram( prg )
  Which( prg )

new global variables:
  FruitCake

new operations:
  Recipe( arg )

new methods:
  Recipe( cake )
```

Another trick is to start GAP with `-r -A` options, immediately load your package and then call `NamesUserGVars` (**Reference: NamesUserGVars**) which returns a list of the global variable names created since the library was read, to which a value is currently bound. For example, for the Example it produces

```
Example
gap> NamesUserGVars();
[ "EgSeparatedString", "FindFile", "FruitCake", "HelloWorld", "ListDirectory",
  "LoadedPackages", "Recipe", "WhereIsPkgProgram", "Which" ]
```

but for packages with dependencies it will also contain variables created by other packages. Nevertheless, it may be a useful check to search for unwanted variables appearing after package loading. A potentially dangerous situation which should be avoided is when the package uses some simply named temporary variables at the loading stage. Such "phantom" variables may then remain unnoticed and, as a result, there will be no warnings if the user occasionally uses the same name as a local variable name in a function. Even more dangerous is the case when the user variable with the same already exists before the package is loaded so it will be silently overwritten.

A.7 Package Dependencies (Requesting one GAP Package from within Another)

It is possible for one GAP package `A`, say, to require another package `B`. For that, one simply adds the name and the (least) version number of the package `B` to the `NeededOtherPackages` component of the `Dependencies` component of the `PackageInfo.g` file of the package `A`. In this situation, loading the package `A` forces that also the package `B` is loaded, and that `A` cannot be loaded if `B` is not available.

If `B` is not essential for `A` but should be loaded if it is available (for example because `B` provides some improvements of the main system that are useful for `A`) then the name and the (least) version number of `B` should be added to the `SuggestedOtherPackages` component of the `Dependencies` component of the `PackageInfo.g` file of `A`. In this situation, loading `A` forces an attempt to load also `B`, but `A` is loaded even if `B` is not available.

Also the component `Dependencies.OtherPackagesLoadedInAdvance` in `PackageInfo.g` is supported, which describes needed packages that shall be loaded before the current package is loaded. See A.8 for details about this and more generally about the order in which the files of the packages in question are read.

All package dependencies must be documented explicitly in the `PackageInfo.g` file. It is important to properly identify package dependencies and make the right decision whether the other package should be "needed" or "suggested". For example, declaring package as "needed" when "suggested" might be sufficient may prevent loading of packages under Windows for no good reason.

It is not appropriate to explicitly call `LoadPackage` (**Reference: LoadPackage**) *when the package is loaded*, since this may distort the order of package loading and result in warning messages. It is recommended to turn such dependencies into needed or suggested packages. For example, a package can be designed in such a way that it can be loaded with restricted functionality if another package (or standalone program) is missing, and in this case the missing package (or binary) is *suggested*. Alternatively, if the package author decides that loading the package in this situation makes no sense, then the missing component is *needed*.

On the other hand, if `LoadPackage` (**Reference: LoadPackage**) is called inside functions of the package then there is no such problem, provided that these functions are called only after the package has been loaded, so it is not necessary to specify the other package as suggested. The same applies to test files and manual examples, which may be simply extended by calls to `LoadPackage` (**Reference: LoadPackage**).

It may happen that a package B that is listed as a suggested package of package A is actually needed by A. If no explicit `LoadPackage` (**Reference: LoadPackage**) calls for B occur in A at loading time, this can now be detected using the new possibility to load a package without loading its suggested packages using the global option `OnlyNeeded` which can be used to (recursively) suppress loading the suggested packages of the package in question. Using this option, one can check whether errors or warnings appear when B is not available (note that this option should be used only for such checks to simulate the situation when package B is not available; it is not supposed to be used in an actual GAP session when package B will be loaded later, since this may cause problems). In case of any errors or warnings, their consequence can then be either turning B into a needed package or (since apparently B was not intended to become a needed package) changing the code accordingly. Only if package A calls `LoadPackage` (**Reference: LoadPackage**) for B at loading time (see above) then package B needs to be *deinstalled* (i.e. removed) to test loading of A without B.

Finally, if the package manual is in the GAPDoc format, then GAPDoc should still be listed as *needed* for this package (not as *suggested*), even though GAPDoc is now a needed package for GAP itself.

A.8 Declaration and Implementation Part of a Package

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 GAP library, and it is resolved there by separating declarations (which define global variables such as filters and operations) and implementations (which install global functions and methods) in different files. Any implementation file may use global variables 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 home directory of the package, this file is read only *after* all the `init.g` files of all (implicitly) required GAP packages are read. Thus one can separate declaration and implementation for a GAP package in the same way as is done for the GAP library, by creating a file `read.g`, restricting the `ReadPackage` (**Reference: ReadPackage**) statements in `init.g` to only read those files of the package that provide declarations, and to read the implementation files from `read.g`.

Examples:

Suppose that there are two packages `A` and `B`, each with files `init.g` and `read.g`.

- If package `A` suggests or needs package `B` and package `B` does not need or suggest any other package then first `init.g` of `B` is read, then `read.g` of `B`, then `init.g` of `A`, then `read.g` of `A`.
- If package `A` suggests or needs package `B` and package `B` (or a package that is suggested or needed by `B`) suggests or needs package `A` then first the files `init.g` of `A` and `B` are read (in an unspecified order) and then the files `read.g` of `A` and `B` (in the same order).

In general, when GAP is asked to load a package then first the dependencies between this packages and its needed and suggested packages are inspected (recursively), and a list of package sets is computed such that no cyclic dependencies occur between different package sets and such that no package in any of the package sets needs any package in later package sets. Then GAP runs through the package sets and reads for each set first all `init.g` files and then all `read.g` files of the packages in the set. (There is one exception from this rule: Whenever packages are autoloaded before the implementation part of the GAP library is read, only the `init.g` files of the packages are read; as soon as the GAP library has been read, the `read.g` files of these packages are also read, and afterwards the above rule holds.)

It can happen that some code of a package depends on the availability of suggested packages, i.e., different initializations are performed depending on whether a suggested package will eventually be loaded or not. One can test this condition with the function `IsPackageMarkedForLoading` (**Reference: IsPackageMarkedForLoading**). In particular, one should *not* call (and use the value returned by this call) the function `LoadPackage` (**Reference: LoadPackage**) inside package code that is read during package loading. Note that for debugging purposes loading suggested packages may have been deliberately disabled via the global option `OnlyNeeded`.

Note that the separation of the GAP code of packages into declaration part and implementation part does in general *not* allow one to actually *call* functions from a package when the implementation part is read. For example, in the case of a "cyclic dependency" as in the second example above, suppose that `B` provides a new function `f` or a new global record `r`, say, which are declared in the declaration part of `B`. Then the code in the implementation part of `A` may contain calls to the functions defined in the declaration part of `B`. However, the implementation part of `A` may be read *before* the implementation part of `B`. So one can in general not assume that during the loading of `A`, the function `f` can be called, or that one can access components of the record `r`.

If one wants to call the function `f` or to access components of the record `r` in the code of the package `A` then the problem is that it may be not possible to determine a cyclic dependency between `A` and `B` from the packages `A` and `B` alone. A safe solution is then to add the name of `B` to the component

Dependencies.OtherPackagesLoadedInAdvance of the PackageInfo.g file of A. The effect is that package B is completely loaded before the file read.g of A is read, provided that there is no cyclic dependency between A and B, and that package A is regarded as not available in the case that such a cyclic dependency between A and B exists.

A special case where Dependencies.OtherPackagesLoadedInAdvance can be useful is that a package wants to force the complete GAP library to be read before the file read.g of the package A is read. In this situation, the "package name" "gap" should be added to this component in the PackageInfo.g file of A.

In the case of cyclic dependencies, one solution for the above problem might be to delay those computations (typically initializations) in package A that require package B to be loaded until all required packages are completely loaded. This can be done by moving the declaration and implementation of the variables that are created in the initialization into a separate file and to declare these variables in the init.g file of the package, via a call to DeclareAutoreadableVariables (**Reference: DeclareAutoreadableVariables**) (see also A.9).

A.9 Autoreadable Variables

Package files containing method installations must be read when the package is loaded. For package files *not* containing method installations (this applies, for example, to many data files) another mechanism allows one to delay reading such files until the data are actually accessed. See **Reference: DeclareAutoreadableVariables** for further details.

A.10 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 package's 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 cover 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 recommend the provision of 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 A.7) being the actual GAP package.

A.10.1 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:

Example
```
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`). As initial templates, you may use installation scripts of the Example package.

A.10.2 Test for the Existence of GAP Package Binaries

If an external binary is essential for the workings of a GAP package, the function stored in the component `AvailabilityTest` of the `PackageInfo.g` file of the package 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` (**Reference: Filename (for a directory and a string)**) for checking for the actual binaries in the path given by `DirectoriesPackagePrograms` (**Reference: DirectoriesPackagePrograms**) for the respective package. For example the example GAP package could use the following function to test whether the binary `hello` has been compiled; it will issue a warning if not, and will only load the package if the binary is indeed available:

Example
```
...
AvailabilityTest := function()
  local path,file;
    # test for existence of the compiled binary
    path:= DirectoriesPackagePrograms( "example" );
    file:= Filename( path, "hello" );
    if file = fail then
      LogPackageLoadingMessage( PACKAGE_WARNING,
          [ "The program ‘hello’ is not compiled,",
            "‘HelloWorld()’ is thus unavailable.",
            "See the installation instructions;",
            "type: ?Installing the Example package" ] );
    fi;
    return file <> fail;
  end,
...
```

However, if you look at the actual `PackageInfo.g` file of the example package, you will see that its `AvailabilityTest` function always returns `true`, and just logs the warning if the binary is not available (which may be later viewed with `DisplayPackageLoadingLog` (**Reference: DisplayPackageLoadingLog**)). This means that the binary is not regarded as essential for this package.

You might also have to cope with the situation that external binaries will only run under UNIX (and not, say, under Windows), or may not compile with some compilers or default compiler options. See (**Reference: Testing for the System Architecture**) for information on how to test for the architecture.

Last but not least: do not print anything in the `AvailabilityTest` function of the package via `Print` or `Info`. Instead one should call `LogPackageLoadingMessage` (**Reference: LogPackageLoadingMessage**) to store a message which may be viewed later with `DisplayPackageLoadingLog` (**Reference: DisplayPackageLoadingLog**) (the latter two functions are introduced in GAP 4.5)

A.10.3 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 with `Process` (**Reference: Process**) or `Exec` (**Reference: Exec**) which then writes the output data to file, and finally read in the standalone's output file.

The second way is interfacing via input-output streams, see Section (**Reference: Input-Output Streams**).

Some GAP packages use kernel modules (see (**Reference: Kernel modules in GAP packages**)) instead of external binaries. A kernel module is implemented in C and follows certain conventions to comply with the GAP kernel interface, which we plan to document later. In the meantime, we advise you to look at existing examples of such packages and get in touch with GAP developers if you plan to develop such a package.

A.11 Having an InfoClass

It is a good idea to declare an `InfoClass` for your package. This gives the package user the opportunity to control the verbosity of output and/or the possibility of receiving debugging information (see (**Reference: Info Functions**)). Below, we give a quick overview of its utility.

An `InfoClass` is defined with a `DeclareInfoClass(` *`InfoPkgname`* `);` statement and may be set to have an initial `InfoLevel` other than the zero default (which means no `Info` statement is to output information) via a `SetInfoLevel(` *`InfoPkgname`*`,` *`level`* `);` statement. An initial `InfoLevel` of 1 is typical.

`Info` statements have the form: `Info(` *`InfoPkgname`*`,` *`level`*`,` *`expr1`*`,` *`expr2`*`, ...);` where the expression list *`expr1`*`,` *`expr2`*`, ...` appears just like it would in a `Print` statement. The only difference is that the expression list is only printed (or even executed) if the `InfoLevel` of *`InfoPkgname`* is at least *`level`*.

A.12 The Banner

Since GAP 4.4, the package banner, if one is desired, should be provided by assigning a string to the `BannerString` field of the record argument of `SetPackageInfo` in the `PackageInfo.g` file.

It is a good idea to have a hook into your package documentation from your banner. The banner of the Example package suggests to the GAP user:

Example
```
  For help, type: ?Example package
```

In order for this to display the introduction of the Example package the index-entry <Index>Example package</Index> was added just before the first paragraph of the introductory section in the file example.xml. The Example package uses GAPDoc (see Section A.2) for documentation (you will need some different scheme to achieve this using the gapmacro.tex system).

A.13 Version Numbers

Version numbers are strings containing nonnegative integers separated by non-numeric characters. They are compared by CompareVersionNumbers (**Reference: CompareVersionNumbers**) which first splits them at non-digit characters and then lexicographically compares the resulting integer lists. Thus version "2-3" is larger than version "2-2-5" but smaller than "4r2p3" or "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.

Loading a specific version of a package (that is, *not* one with a larger version number) can be achieved by prepending = to the desired version number. For example, LoadPackage("example", "=1.0") will load version "1.0" of the package "example", even if version "1.1" is available. As a consequence, version numbers must not start with =, so "=1.0" is not a valid version number.

Package authors should choose a version numbering scheme that admits a new version number even after tiny changes to the package, and ensure that version numbers of successive package versions increase. The automatic update of package archives in the GAP distribution will only work if a package has a new version number.

It is a well-established custom to name package archives like name-version.tar.gz, name-version.tar.bz2 etc., where name is the lower case name, and version is the version (another custom is that the archive then should extract to a directory that has exactly the name name-version).

It is very important that there should not ever be, for a given GAP package, two different archives with the same package version number. If you make changes to your package and place a new archive of the package onto the public server, please ensure that a new archive has a new version number. This should be done even for very minor changes.

For most of the packages it will be inappropriate to re-use the date of the release as a version number. It's much more obvious how big are the changes between versions "4.4.12", "4.5.1" and "4.5.2" than between versions "2008.12.17", "2011.04.15" and "2011.09.14".

Since version information is duplicated in several places throughout the package documentation, for GAPDoc-based manuals you may define the version and the release manual in the comments in PackageInfo.g file close to the place where you specified its Version and Date components, for example

```
Example
##  <#GAPDoc Label="PKGVERSIONDATA">
##  <!ENTITY VERSION "3.3">
##  <!ENTITY RELEASEDATE "11/11/2011">
##  <#/GAPDoc>
```

notify MakeGAPDocDoc (**GAPDoc: MakeGAPDocDoc**) that a part of the document is stored in PackageInfo.g (see example/makedocrel.g), read this data into the header of the main document via <#Include Label="PKGVERSIONDATA"> directive and then use them via &VERSION; and

&RELEASEDATE; entities almost everywhere where you need to refer to them (most commonly, in the title page and installation instructions).

A.14 Testing a GAP package

A.14.1 Tests files for a GAP package

The (optional) `tst` directory of your package may contain as many tests of the package functionality as appears appropriate. These tests should be organised into test files similarly to those in the `tst` directory of the GAP distribution as documented in (**Reference: Test Files**).

For a deposited package, a test file with a basic test of the package (for example, to check that it works as expected and/or that the manual examples are correct) may be specified in the component `TestFile` in the `PackageInfo.g` to be included in the GAP standard test suite. This file can either consist of `ReadTest` (**Reference: ReadTest**) calls (in this case, it is common to call it `testall.g`) or be itself a test file having an extension `.tst` and supposed to be read via `ReadTest` (**Reference: ReadTest**). It is assumed that the latter case occurs if and only if the file contains the substring

`"gap> START_TEST("`

(with exactly one space after the GAP prompt).

For deposited packages, these tests are run by the GAP Group regularly, as a part of the standard GAP test suite. For the efficient testing it is important that the test specified in the `PackageInfo.g` file does not display any output (e.g. no test progress indicators) except reporting discrepancies if such occur and the completion report as in the example below:

Example
```
gap> ReadTest("tst/testall.tst");
Line 50 :
+ Example package: testall.tst
Line 50 :
+ GAP4stones: 10000
true
```

Tests which produce extended output and/or requires substantial runtime are not supposed to be a part of the GAP standard test suite but may be placed in the `tst` directory of the packages with further instructions on how to run them elsewhere.

A.14.2 Testing GAP package loading

To test that your package may be loaded into GAP without any problems and conflicts with other packages, test that it may be loaded in various configurations:

- starting GAP with no packages (except needed for GAP) using `-r -A` options and calling `LoadPackage("your-package-name");`
- starting GAP with no packages (except needed for GAP) using `-r -A` options and calling `LoadPackage("your-package-name" : OnlyNeeded );`
- starting GAP in the default configuration (with no options) and calling `LoadPackage("your-package-name");`
- starting GAP in the default configuration (with no options) and calling `LoadPackage("your-package-name" : OnlyNeeded );`

- finally, together with all other packages using `LoadAllPackages` (A.14.3) (see below) in four possible combinations of starting GAP with/without `-r -A` options and calling `LoadAllPackages` (A.14.3) with/without `Reversed` option.

The test of loading all packages is the most subtle one. Quite often it reveals problems which do not occur in the default configuration but may cause difficulties to the users of specialised packages.

For your convenience, `make testpackagesload` called in the GAP root directory will run all package loading tests listed in this subsection and write their output in its `dev/log` directory.

It will produce four files with test logs, corresponding to the four cases above (the letter `N` in the filename stands for “needed”, `A` stands for “autoloaded”):

- `dev/log/testpackagesload1_...`
- `dev/log/testpackagesloadN1_...`
- `dev/log/testpackagesloadA_...`
- `dev/log/testpackagesloadNA_...`

Each file contains small sections for loading individual packages: among those, you need to find the section related to your package and may ignore all other sections. For example, the section for the Example package looks like

```
Example
%%% Loading example 3.3.3
[  ]
### Loaded example 3.3.3
```

so it has clearly passed the test. If there are any error messages displayed between `Loading ...` and `Loaded ...` lines, they will signal on errors during loading of your package.

Additionally, this test collects information about variables created since the library was read (obtained using `NamesUserGVars` (**Reference: NamesUserGVars**)) with either short names (no more than three characters) or names breaking a recommended naming convention that the GAP core system and GAP packages should not use global variables starting in the lowercase (see Section A.6). Their list will be displayed in the test log (in the example above, Example packages does not create any such variables, so an empty list is displayed). It may be hard to attribute a particular identifier to a package, since it may be created by another package loaded because of dependencies, so when a more detailed and precise report on package variables is needed, it may be obtained using `ShowPackageVariables` (**Reference: ShowPackageVariables**) (also, `make testpackagesvars` called in the GAP root directory produces such reports for each package and writes them to a file `dev/log/testpackagesvars_...`).

Finally, each log file finishes with two large sections for loading all packages in the alphabetical and reverse aplhabetical order (to check more combinations of loading one package after another). We are aiming at releasing only collections of package which do not break `LoadAllPackages` (A.14.3) in any of the four configurations, so if it is broken when you plug in the development version of your package into the released version of GAP, it is likely that your package triggers this error. If you observe that `LoadAllPackages` (A.14.3) is broken and suspect that this is not the fault of your package, please contact the GAP Support.

A.14.3 LoadAllPackages

▷ LoadAllPackages(: *Reversed*) (function)

loads all GAP packages from their list sorted in alphabetical order (needed and suggested packages will be loaded when required). This is a technical function to check packages compatibility, so it should NOT be used to run anything except tests; it is known that GAP performance is slower if all packages are loaded. To introduce some variations of the order in which packages will be loaded for testing purposes, `LoadAllPackages` accepts version `Reversed` to load packages from their list sorted in the reverse alphabetical order.

A.14.4 Testing a GAP package with the GAP standard test suite

The `tst` directory of the GAP installation contains a selection of test files and two scripts, `testinstall.g` and `testall.g` which are a part of the GAP standard test suite.

It is important to check that your package does not break GAP standard tests. To perform a clean test and avoid interfering with other packages, first you must start a new GAP session with the following command (assuming that gap starts GAP in your system):

Example
```
gap -N -A -x 80 -r -m 100m -o 512m
```

After that load your package and run either `testinstall.g` or `testall.g` as demonstrated below.

The quicker test, `testinstall.g`, requires about 512MB of memory and runs for about one minute on an Intel Core 2 Duo / 2.53 GHz machine. It may be started with the command

Example
```
gap> Read( Filename( DirectoriesLibrary( "tst" ), "testinstall.g" ) );
```

You will get a large number of lines with output about the progress of the tests.

Example
```
test file         GAP4stones     time(msec)
-------------------------------------------
testing: .../gap4r5/tst/zlattice.tst
zlattice.tst               0              0
testing: .../gap4r5/tst/gaussian.tst
gaussian.tst               0             10
... further lines deleted ...
```

The more thorough test is `testall.g` which exercises more of GAP's capabilities, containing all test files from `testinstall.g` as a subset. It requires about 512MB of memory, runs for about one hour on an Intel Core 2 Duo / 2.53 GHz machine, and produces an output similar to the testinstall.g test. To run it, also start a new GAP session with `gap -N -A -x 80 -r -m 100m -o 512m` and then call

Example
```
gap> Read( Filename( DirectoriesLibrary( "tst" ), "testall.g" ) );
```

You may repeat the same check loading your package with `OnlyNeeded` option. Remember to perform each subsequent test in a new GAP session.

Also you may perform individual tests from the `tst` directory of the GAP installation loading them with `ReadTest` (**Reference: ReadTest**), for example, the file `bugfix.tst`.

A.15 Access to the GAP Development Version

We are aiming at providing a stable platform for package development and testing with official GAP releases. However, when it may be of benefit to obtain access to the GAP development version, please contact the GAP team by mailing to `support@gap-system.org`.

A.16 Selecting a license for a GAP Package

It is advised to make clear in the documentation of the package the basis on which it is being distributed to users. GAP itself is distributed under the GNU Public License version 2 (version 2 or later). We would encourage you to consider the GPL for your packages, but you might wish to be more restrictive (for instance forbidding redistribution for profit) or less restrictive (allowing your software to be incorporated into commercial software). See "Choosing a License for the Distribution of Your Package" from `http://www.gap-system.org/Packages/Authors/authors.html` for further details.

In the past many GAP packages used the text "We adopt the copyright regulations of GAP as detailed in the copyright notice in the GAP manual" or a similar statement. We now advise to be more explicit and make the exact reference to the GPL license, for example:

package-name is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

A.17 Wrapping up a GAP Package

Currently, the GAP distribution provides archives in four different formats.

`.tar.gz`
: a standard UNIX `tar` archive, compressed with `gzip`

`.tar.bz2`
: a standard UNIX `tar` archive, compressed with `bzip2`

`.zip`
: an archive in `zip` format, where text files should have UNIX style line breaks

`-win.zip`
: an archive in `zip` format, where text files should have DOS/Windows style line breaks

For convenience of possible users it is sensible that you provide an archive of your package in at least one of these formats.

For example, if you wish to supply a `.tar.gz` archive, you may create it with the command

```
tar -cvzf package-name-version.tar.gz package-name
```

The `etc` directory obtained from `tools.tar.gz` (described above in Section A.2) contains a file `packpack` which provides a more versatile packing-up script.

In the past, it was recommended that your GAP package should be packed with the `zoo` archiver. We do not redistribute `.zoo` archives since GAP 4.5, but we still accept package archives in `.zoo` format for backwards compatibility, if no other formats are available.

Because the release of the GAP package is independent of the version of GAP, a GAP package should be wrapped up in separate file that can be installed onto any version of GAP. In this way, a package can be upgraded any time without the need to wait for new GAP releases. To ensure this, the package should be archived from the GAP `pkg` directory, that is all files are archived with the path starting at the package's name.

The archive of a GAP package should contain all files necessary for the package to work. In particular there should be a compiled documentation, which includes the `manual.six`, `manual.toc` and `manual.lab` file in the documentation subdirectory which are created by TEXing the documentation, if you use GAPDoc or the `gapmacro.tex` document formats. (The first two files are needed by the GAP help system, and the `manual.lab` file is needed if the main manuals or another package is referring to your package. Use the command `GAPDocManualLab( packagename );` to create this file for your help books if you use GAPDoc.)

For packages which are redistributed via the GAP Web site, we offer an automatic conversion of any of the formats listed above to all the others. To use this service, you can provide any of the four archive formats or even more than one, however you should adhere to the following rule: text files in `.tar.gz` and `.tar.bz2` archives must have UNIX style line breaks, while text files in `-win.zip` archives must have DOS/Windows line breaks.

The package wrapping tools for the GAP distribution and web pages then will use a sensible list of file extensions to decide if a file is a text file (being conservative, it may miss a few text files). These rules may be prepended by the application of rules from the `PackageInfo.g` file:

- if it has a `.TextFiles` component, then consider the given files as text files before GAP defaults will be applied;
- if it has a `.BinaryFiles` component, then consider given files as binary files before GAP defaults will be applied;
- if it has a `.TextBinaryFilesPatterns` component, then apply it before GAP defaults will be applied;

The `etc` directory obtained from `tools.tar.gz` (described above in Section A.2) contains a file `classifyfiles.py` and two files `patternscolor.txt` and `patternstextbinary.txt` implementing GAP default rules used to classify files in packages. For most of the packages these default rules perfecty detect binary and text files, so there is no need for them to use any of the three optional components. However, `.TextBinaryFilesPatterns`, or `.TextFiles`, or `.BinaryFiles` will become useful if the package has e.g. data files which are recognised as binary files by the default rules, or if the package uses standard extensions in a non-standard way (this is not recommended, of course). If things will go wrong, it is possible that one (or indeed all) created archives have wrong line breaks.

Utility functions available in `gap4r5/lib/lbutil.g`, namely `DosUnixLinebreaks`, `UnixDosLinebreaks`, `MacUnixLinebreaks` may be helpful. They are described in the comments to their source code.

A.18 The WWW Homepage of a Package

If you want to distribute your package you should create a WWW homepage containing some basic information, archives for download, the `README` file with installation instructions, and a copy of the package's `PackageInfo.g` file.

The responsibility for this WWW homepage is with the package authors/maintainers.

If you tell us about your package (say, by mail to `support@gap-system.org`) we may consider either

- adding a link to your package homepage from the GAP website (thus, the package will be an *undeposited contribution*);
- or redistributing the current version of your package as a part of the GAP distribution (this, the package will be *deposited*), also ;

In the latter case we can also provide some services for producing several archive formats from the archive you provide (e.g., you produce a `.tar.gz` version of your archive and we produce also a `.tar.bz2` and a `-win.zip` version from it).

Please also consider submitting your package to the GAP package refereeing process (see `http://www.gap-system.org/Contacts/submit.html` for further information).

A.19 Upgrading the package to work with GAP 4.5

A.19.1 Changes in GAP 4.5 from the packages perspective

Here we list only those changes which may have some implications for the packages.

- Changing the distribution format providing one archive with the core system and all currently redistributed packages.
- The GAP kernel is now compiled by default to use the GMP large integer arithmetic library, speeding up arithmetic by a factor of 4 or more in many cases. This slightly changes the build process, affecting mainly packages with dynamically loaded modules (see `http://www.gap-system.org/Download/` for GAP installation instructions).
- The GAP documentation has been converted to the GAPDoc format and extensively reviewed. Now it has only two books: the Tutorial and the Reference Manual. The two other books, “Extending GAP” and “Programming Tutorial” became parts of the Reference Manual. Packages that refer to parts of the GAP documentation may need to rebuild their manuals to update references.

  Some packages still use the old “gapmacro” format for their manuals, for which support may be discontinued in the future. There is no urgent need to convert such manuals into the GAPDoc format before GAP 4.5 release, but we encourage you very much to consider doing this at some later point.
- The old concept of an autoloaded package has been integrated with the *needed* and *suggested* mechanism that already exists between packages. GAP itself now “needs” certain packages (for instance GAPDoc) and “suggests” others (typically the packages that were autoloaded). The decisions which packages GAP should need or suggest are made by developers based on technical criteria. They can be easily overridden by a user using the new `gap.ini` (see (**Reference: The gap.ini and gaprc files**)). The default file ensures that all previously autoloaded packages are still loaded if present.
- Optional `~/.gap` directory for user’s customisations which may contain e.g. locally installed packages (see (**Reference: GAP Root Directories**)). If package installation instructions explain how to install the package in a non-standard location, they may need an update. This is

intended to replace `.gaprc` files, but those are still supported for backwards compatibility (see (**Reference: The former .gaprc file**)).

- Various improvements in the packages loading mechanism make it more informative, while avoiding confusing the user with warning and error messages for packages they didn't know they were loading. For example, many messages are stored but not displayed using the function `LogPackageLoadingMessage` (**Reference: LogPackageLoadingMessage**) and there is a function `DisplayPackageLoadingLog` (**Reference: DisplayPackageLoadingLog**) to show log messages that occur during package loading. Packages are encouraged to use these mechanisms to report problems in loading (e.g. binaries not compiled), rather than printing messages directly.
- Since GAP 4.5 a package may place global variables into a local namespace as explained in (**Reference: Namespaces for GAP packages**) in order to avoid name clashes and preserve compatibility.
- In GAP 4.5 the internal representation of a record has changed, as well as some of the basic functions to manipulate records. This speeds up considerably the creation of and access to records with many components. Record components are now internally stored in the order in which they were used first, and this means that the internal ordering of components (returned by `RecNames` (**Reference: RecNames**) and so the ordering of records, depends on the GAP session. Therefore, within each session everything is consistent, so one can efficiently remove duplicates with `Set` (**Reference: Set**), sort lists of records, find records with binary search, etc., but a care should be taken not to rely on `RecNames` (**Reference: RecNames**) always returning names of components in the same order.

A.20 Checklists

A.20.1 Package release checklist

The following checklist may be used by package authors, members of the GAP team responsible for package updates, package editors and referees.

- Test that the package:
  - does not break `testinstall.g` and `testall.g` and does not slow them down noticeably (see A.14.4);
  - may be loaded in various configurations (see A.14.2);
  - follows the guidelines of Section A.6 about names of functions and variables;
- Package documentation:
  - is built and included in the package archive together with its source files;
  - states the version, release date and package authors;
  - has the same version, release date and package authors details as stated in the `PackageInfo.g` file;
  - is searchable using the GAP help system;
  - is clear about the license under which the package is distributed;

- `PackageInfo.g` file:
  - has the same version, release date and package authors details as stated in the package manual;
  - has all mandatory components and also optional components where appropriate;
  - in particular, contains hints to distinguish binary and text files in case of non-standard file names and extensions;
  - is validated using `ValidatePackageInfo` (**Reference: ValidatePackageInfo**);
- Package archive(s):
  - have correct permisisons for all files and directories after their unpacking (755 for directories and executables, if any; 644 for other files);
  - contain files with correct line breaks for the given format (see A.17);
  - contain no hidden system files and directories that are not supposed to be included in the package, e.g. `.cvsignore`, `.git` etc.;
- Package availability:
  - not only the package archive(s), but also the `PackageInfo.g` and `README` files are available online;
  - the URL of the `PackageInfo.g` file is validated using the online tool available from `http://www.gap-system.org/Packages/Authors/authors.html`;

A.20.2 Checklist for package upgrade to work with GAP 4.5

The following checklist will help you to check how well your package is ready to work with GAP 4.5.

- Mandatory changes needed for package upgrade to work with GAP 4.5:
  - verify that the package functionality works as required, that examples in the manual are correct and that test files show no discrepancies;
  - if necessary, update manual examples and test files because the order in which record components are printed has changed (but now it will be more consistent and less dependent on how the record was created);
  - check the usage of names of record components in the code: take care not to rely on `RecNames` (**Reference: RecNames**) always returning names of components in the same order (see A.19)

  Revise package dependencies:
  Check the the `PackageInfo.g` has the correct list of needed and suggested packages (see A.7).

  Revise licensing information:
  Check that the package states clearly under which conditions it is distributed (see A.16).

  Rebuild the package documentation:
  whenever your package documentation is GAPDoc or `gapmacro.tex`-based, GAP 4.5 contains new versions of both tools. This will ensure that cross-references from the package manual to the main GAP manuals are correct and that the GAP help system will be

able to navigate to the more precise location in the package manual. This will also improve the layout of the package documentation. In particular, GAPDoc has a better `.css` file and is capable of producing the HTML version with MathJax support to display nicely mathematical formulae.

Note that it's not possible for a package to have an HTML manual which contains correct links to both GAP 4.4 and GAP 4.5 main manuals.

- Optional changes recommended for package upgrade to work with GAP 4.5:
  - When the `AvailabilityTest` component in `PackageInfo.g` differs from `ReturnTrue` (**Reference: ReturnTrue**), use `LogPackageLoadingMessage` (**Reference: LogPackageLoadingMessage**) to store a message which may be viewed later with `DisplayPackageLoadingLog` (**Reference: DisplayPackageLoadingLog**) instead of calling `Print` or `Info` directly (see A.10.2).
  - It is recommended not to call `LoadPackage` (**Reference: LoadPackage**) from a package file while this file is read: instead one should list the package in question in the lists of needed or suggested packages. To verify whether such calls occur in the package first load it and then call `DisplayPackageLoadingLog( PACKAGE_WARNING );`. If there is a genuine need to decide whether some package will be available at runtime, use the function `IsPackageMarkedForLoading` (**Reference: IsPackageMarkedForLoading**) introduced in GAP 4.5.
  - Check if the package still relies on some obsolete variables (see (**Reference: Replaced and Removed Command Names**)) and try to get rid of their usage.

Index