Example

A GAP4 Package

Version 2.0

by

Werner Nickel

AG 2, Fachbereich Mathematik, TU Darmstadt

Schlossgartenstr. 7, 64289 Darmstadt, Germany

email: nickel@mathematik.tu-darmstadt.de

January 2006

Contents

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 and 2.2 for how to install and load the Example package, or Appendix A for hints on how to write a GAP package.

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

```
gap> ?>
```

to see the functions provided by the Example package.

1.1 The main functions

The following functions are available:

1 ► `ListDirectory([`*dir*`])` F

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

2 ► `FindFile(` *directory_name*`,` *file_name* `)` F

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.

3 ► `LoadedPackages()` F

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

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

You might like to check out some of the other information in the `GAPInfo` record.

4 ► `Which(` *prg* `)` F

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

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

5 ► `WhereIsPkgProgram(` *prg* `)` F

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

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

6 ► `HelloWorld()` F

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

7 ► `FruitCake` V

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

8 ► `Recipe(` *cake* `)` M

displays the recipe for cooking *cake*, where *cake* is a record. The fields of *cake* recognised 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).

2 Installing and Loading the Example Package

2.1 Installing the Example Package

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

```
unzoo -x example-XXX
```

in the `pkg` directory of your version of GAP 4, or in a directory named `pkg` (e.g. in your home directory). (The only essential difference with installing Example in a `pkg` directory different to the GAP 4 home directory is that one must start GAP with the `-l` switch, 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.)

After unpacking the archive, go to the newly created `example` directory and call `./configure` *path* where *path* is the path to the GAP home directory. So for example if you install the package in the main `pkg` directory call

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

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.

2.2 Loading the Example Package

To use the Example Package you have to request it explicitly. This is done by calling

```
gap> LoadPackage("example");
---------------------------------------------------------------
Loading  Example 2.0
by Werner Nickel (http://www.mathematik.tu-darmstadt.de/~nickel)
   Greg Gamble (http://www.math.rwth-aachen.de/~Greg.Gamble)
For help, type: ?Example package
---------------------------------------------------------------
true
```

The `LoadPackage` command is described in Section 75.2.1 in the GAP Reference Manual.

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 3.4 in the GAP Reference Manual).

A Hints for writing a GAP Package

The Example package is intended to be a prototype for a package. Here we describe just what features one should emulate when writing one's own GAP package for popular consumption, and a few pointers as to where to go for more information. Much of what is written here is amplified in the section 4 in the Extending GAP Manual.

A.1 Structure of a GAP Package

This section is intended to amplify the recommendations made in Section 4.1 of the Extending GAP Manual.

A GAP package should have an alphanumeric name (*package-name*, say); mixed case is fine, but there should be no whitespace. The directory *package-dir* containing the files of package *package-name* should be just *package-name* converted to lowercase (the restriction that *package-dir* must contain only lowercase characters may change in the future).

The directory *package-dir* should be a subdirectory of `pkg` and preferably should have the following structure (below, a trailing `/` distinguishes directories from ordinary files):

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

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 and authors' names and addresses should be repeated in the package's documentation (which should be in the `doc` directory).

`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 a subdirectory of `bin` with name 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. The Example package's `PackageInfo.g` file is well-commented and should be used as a prototype.

`init.g`, `read.g`
: A GAP package **must** have a file `init.g` (see Section 4.1 in the Extending GAP Manual). As of GAP 4.4, the typical `init.g` and `read.g` files should normally consist entirely of `ReadPackage` (see 75.3.1 in the GAP 4 Reference Manual) commands (and possibly also `Read` commands). 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. 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. Traditionally, a TEX-based system has been used for GAP documentation, which is thoroughly described in Section 2 of the Extending GAP Manual. There is now an alternative XML-based system provided by the GAP package GAPDoc (see Chapter 1 of the GAPDoc Manual). 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 was written using the traditional TEX-based system. If you plan on using this, please use the Example package's `doc` directory as a prototype, which you will observe contains the following files:

```
manual.tex # master file
chapi.tex # chapter file(s) ... 1 for each chapter
manual.mst # MakeIndex style file
make_doc   # script that generates the manuals
```

Generally, one should also provide a `manual.bib` BibTEX database file (or write one's own bibliography `manual.bbl` file). Generating the various formats of the manuals requires various software tools which are called directly or indirectly by `make_doc` and these are listed in Section A.2. The file `manual.mst` is needed for generating a manual index; it should be a copy of the one provided in the Example package. The only adjustments that a package writer should need to make to `make_doc` is to replace occurrences of the word `Example` with *package-name*.

`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 Example package included), 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` command (see 75.3.4 in the GAP 4 Reference Manual).

`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.

A.2 Documentation Software Tools Needed

Whether you use the traditional `gapmacro.tex` TeX-based system or GAPDoc you will need to have the various following TeX tools installed:

`tex` (or `latex` for GAPDoc), `bibtex` and `makeindex`
: for generating `.dvi`;

`dvips`
: for generating `.ps`; and

`pdftex` or `gs` and `ps2pdf` (or `pdflatex` for GAPDoc)
: for generating `.pdf`;

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*.

The rest of this section describes the various additional tools needed for the `gapmacro.tex` documentation system.

To produce the `.dvi`, `.ps` and `.pdf` manual formats, the following GAP tools (usually located in GAP's main `doc` directory) are needed (provided by `tools`*XXX*`.zoo` for some version number *XXX* at the GAP `ftp`- or web-sites, or can be obtained by emailing `support@gap-system.org`).

`gapmacro.tex`
: The macros file that dictates the style and mark-up for the traditional TeX-based system of GAP documentation.

`manualindex`
: This is an `awk` script that adjusts the TeX-produced index entries and calls `makeindex` to process them.

`mrabbrev.bib`
: This is 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 other `gapmacro.tex`-produced manuals (and so has `\UseReferences` commands in its `manual.tex`), then a `manual.lab` file (generated by running `tex manual`) for each such other manual is needed (this includes the "main" manuals, e.g. those in the `doc/ref`, `doc/ext` etc. directories).

If your manual cross-refers to GAPDoc-produced manuals (and so has `\UseGapDocReferences` commands in its `manual.tex`), then `manual.lab` files need to be generated for these too. Since GAP 4.3, this is done by starting GAP and running:

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

for each *package* whose manual is cross-referred to.

To produce an HTML version of the manual one needs the Perl 5 program `convert.pl` which is usually located in the subdirectory `etc` of the GAP root directory. The `etc` directory is not part of the usual GAP distribution. The `etc` directory files are obtained from `tools`*XXX*`.zoo` for some version number *XXX* at the GAP `ftp`- or web-sites, or can be obtained by emailing `support@gap-system.org`.

Finally, to ensure the mathematics 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
```

As a package author, you are not obliged to provide an HTML version of your package manual, but if you have kept to the guidelines in Section 2 of the Extending GAP Manual, you should have no trouble in producing one. A prototype of the command to execute is in the file `make_doc`; note that the HTML manual is produced in files with `.htm` extensions in a directory `htm` outside the `doc` directory. The beginning of the file `convert.pl` contains instructions on its usage should you need them.

A.3 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.

```
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` (see 4.9.7 in the GAP Reference Manual), or alternatively (and equivalently) you may define a global function via a `DeclareGlobalFunction` and `InstallGlobalFunction` pair or a global variable via a `DeclareGlobalVariable` and `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 haved 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. On the other hand, operations and their methods (defined via `DeclareOperation`, `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.

The method `Recipe` was included in the Example package to demonstrate the definition of a function via a `DeclareOperation`/`InstallMethod` pair; `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.

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.

A.4 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 7.4 in the GAP Reference Manual). 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.5 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 Example package suggests to the GAP user:

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

In order for this to display the introduction of the Example package an `\atindex` equivalent of the following index-entry:

```
\index{Example package}
```

was added just before the first paragraph of the introductory section in the file `example.tex`. The Example package uses the `gapmacro.tex` system (see Section A.2) for documentation (you will need some different scheme to achieve this using GAPDoc).

A.6 Packing up your GAP Package

In the past, it was recommended that your GAP package should be packed via the `zoo` program, but now any of four different archive formats are accepted (see Section 4.14 in the Extending GAP Manual for the details). The Example package file `make_zoo` provides a template packing-up script that uses `zoo`. The `etc` directory obtained from `tools`*XXX*`.zoo` for some version number *XXX* (this is described above in Section A.2) contains a file `packpack` which provides a more versatile packing-up script.

A.7 New versions of your GAP Package

You will notice that there is a file `VERSION` which contains the current version of the Example package. Such a file is entirely optional. Note that this file is **not** read at all when GAP loads the package. GAP establishes the package version by reading the `PackageInfo.g` file. The current maintainer of the Example package finds it convenient to have a file `VERSION` that is read both by `doc/manual.tex` and `make_zoo`. It is however important that each new version of a package has a new number and that version numbers of successive package versions increase (see 4.13 in the Extending GAP Manual for the details).

It's also useful to have a `CHANGES` file that records the main changes between versions of your package.

A.8 CVS

When your package is ready to be refereed and/or made available as an "accepted" GAP package, it may be of benefit to obtain CVS access to GAP; as a first step towards this you should make a request to the GAP team via an email to `support@gap-system.org`.

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".