Cabal User Guide: Developing Cabal packages

Quickstart

Lets assume we have created a project directory and already have a Haskell module or two.

Every project needs a name, we’ll call this example “proglet”.

```
$ cd proglet/
$ ls
```

```
Proglet.hs
```

It is assumed that (apart from external dependencies) all the files that make up a package live under a common project root directory. This simple example has all the project files in one directory, but most packages will use one or more subdirectories.

To turn this into a Cabal package we need two extra files in the project's root directory:

- `proglet.cabal`: containing package metadata and build information.
- `Setup.hs`: usually containing a few standardized lines of code, but can be customized if necessary.

We can create both files manually or we can use `cabal init` to create them for us.

Using "cabal init"

The `cabal init` command is interactive. It asks us a number of questions starting with the package name and version.

```
$ cabal init
Package name [default "proglet"]?
Package version [default "0.1"]?
...
```

It also asks questions about various other bits of package metadata. For a package that you never intend to distribute to others, these fields can be left blank.

One of the important questions is whether the package contains a library or an executable. Libraries are collections of Haskell modules that can be re-used by other Haskell libraries and programs, while executables are standalone programs.

```
What does the package build:
   1) Library
   2) Executable
Your choice?
```

For the moment these are the only choices. For more complex packages (e.g. a library and multiple executables or test suites) the `.cabal` file can be edited afterwards.

Finally, `cabal init` creates the initial `proglet.cabal` and `Setup.hs` files, and depending on your choice of license, a `LICENSE` file as well.

```
Generating LICENSE...
```

```
Generating Setup.hs...
Generating proglet.cabal...

You may want to edit the .cabal file and add a Description field.
```

As this stage the `proglet.cabal` is not quite complete and before you are able to build the package you will need to edit the file and add some build information about the library or executable.

Editing the .cabal file

Load up the `.cabal` file in a text editor. The first part of the `.cabal` file has the package metadata and towards the end of the file you will find the `executable` or `library` section.

You will see that the fields that have yet to be filled in are commented out. Cabal files use "`--`" Haskell-style comment syntax. (Note that comments are only allowed on lines on their own. Trailing comments on other lines are not allowed because they could be confused with program options.)

If you selected earlier to create a library package then your `.cabal` file will have a section that looks like this:

```
library
  exposed-modules:     Proglet
  -- other-modules:
  -- build-depends:
```

Alternatively, if you selected an executable then there will be a section like:

```
executable proglet
  -- main-is:
  -- other-modules:
  -- build-depends:
```

The build information fields listed (but commented out) are just the few most important and common fields. There are many others that are covered later in this chapter.

Most of the build information fields are the same between libraries and executables. The difference is that libraries have a number of "exposed" modules that make up the public interface of the library, while executables have a file containing a `Main` module.

The name of a library always matches the name of the package, so it is not specified in the library section. Executables often follow the name of the package too, but this is not required and the name is given explicitly.

Modules included in the package

For a library, `cabal init` looks in the project directory for files that look like Haskell modules and adds all the modules to the `exposed-modules` field. For modules that do not form part of your package's public interface, you can move those modules to the `other-modules` field. Either way, all modules in the library need to be listed.

For an executable, `cabal init` does not try to guess which file contains your program's `Main` module. You will need to fill in the `main-is` field with the file name of your program's `Main` module (including `.hs` or `.lhs` extension). Other modules included in the executable should be listed in the `other-modules` field.

Modules imported from other packages

While your library or executable may include a number of modules, it almost certainly also imports a number of external modules from the standard libraries or other pre-packaged libraries. (These other libraries are of course just Cabal packages that contain a library.)

You have to list all of the library packages that your library or executable imports modules from. Or to put it another way: you have to list all the other packages that your package depends on.

For example, suppose the example `Proglet` module imports the module `Data.Map`. The `Data.Map` module comes from the `containers` package, so we must list it:

```
library
  exposed-modules:     Proglet
  other-modules:
  build-depends:       containers, base == 4.*
```

In addition, almost every package also depends on the `base` library package because it exports the standard `Prelude` module plus other basic modules like `Data.List`.

You will notice that we have listed `base == 4.*`. This gives a constraint on the version of the base package that our package will work with. The most common kinds of constraints are:

- `pkgname >= n`
- `pkgname >= n && < m`
- `pkgname == n.*`

The last is just shorthand, for example `base == 4.*` means exactly the same thing as `base >= 4 && < 5`.

Building the package

For simple packages that's it! We can now try configuring and building the

package:

```
cabal configure
cabal build
```

Assuming those two steps worked then you can also install the package:

```
cabal install
```

For libraries this makes them available for use in GHCi or to be used by other packages. For executables it installs the program so that you can run it (though you may first need to adjust your system's `$PATH`).

Next steps

What we have covered so far should be enough for very simple packages that you use on your own system.

The next few sections cover more details needed for more complex packages and details needed for distributing packages to other people.

The previous chapter covers building and installing packages – your own packages or ones developed by other people.

Package concepts

Before diving into the details of writing packages it helps to understand a bit about packages in the Haskell world and the particular approach that Cabal takes.

The point of packages

Packages are a mechanism for organising and distributing code. Packages are particularly suited for "programming in the large", that is building big systems by using and re-using code written by different people at different times.

People organise code into packages based on functionality and dependencies. Social factors are also important: most packages have a single author, or a relatively small team of authors.

Packages are also used for distribution: the idea is that a package can be created in one place and be moved to a different computer and be usable in that different environment. There are a surprising number of details that have to be got right for this to work, and a good package system helps to simply this process and make it reliable.

Packages come in two main flavours: libraries of reusable code, and complete programs. Libraries present a code interface, an API, while programs can be run directly. In the Haskell world, library packages expose a set of Haskell modules as their public interface. Cabal packages can contain a library or executables or both.

Some programming languages have packages as a builtin language concept. For example in Java, a package provides a local namespace for types and other definitions. In the Haskell world, packages are not a part of the language itself. Haskell programs consist of a number of modules, and packages just provide a way to partition the modules into sets of related functionality. Thus the choice of module names in Haskell is still important, even when using packages.

Package names and versions

All packages have a name, e.g. “HUnit”. Package names are assumed to be unique. Cabal package names can use letters, numbers and hyphens, but not spaces. The namespace for Cabal packages is flat, not hierarchical.

Packages also have a version, e.g “1.1”. This matches the typical way in which packages are developed. Strictly speaking, each version of a package is independent, but usually they are very similar. Cabal package versions follow the conventional numeric style, consisting of a sequence of digits such as “1.0.1” or “2.0”. There are a range of common conventions for “versioning” packages, that is giving some meaning to the version number in terms of changes in the package. Section [TODO] has some tips on package versioning.

The combination of package name and version is called the *package ID* and is written with a hyphen to separate the name and version, e.g. “HUnit-1.1”.

For Cabal packages, the combination of the package name and version *uniquely* identifies each package. Or to put it another way: two packages with the same name and version are considered to *be* the same.

Strictly speaking, the package ID only identifies each Cabal *source* package; the same Cabal source package can be configured and built in different ways. There is a separate installed package ID that uniquely identifies each installed package instance. Most of the time however, users need not be aware of this detail.

Kinds of package: Cabal vs GHC vs system

It can be slightly confusing at first because there are various different notions of package floating around. Fortunately the details are not very complicated.

Cabal packages

Cabal packages are really source packages. That is they contain Haskell (and sometimes C) source code.

Cabal packages can be compiled to produce GHC packages. They can also be translated into operating system packages.

GHC packages

This is GHC's view on packages. GHC only cares about library packages, not executables. Library packages have to be registered with GHC for them to be available in GHCi or to be used when compiling other programs or packages.

The low-level tool `ghc-pkg` is used to register GHC packages and to get information on what packages are currently registered.

You never need to make GHC packages manually. When you build and install a Cabal package containing a library then it gets registered with GHC automatically.

Haskell implementations other than GHC have essentially the same concept of registered packages. For the most part, Cabal hides the slight differences.

Operating system packages

On operating systems like Linux and Mac OS X, the system has a specific notion of a package and there are tools for installing and managing packages.

The Cabal package format is designed to allow Cabal packages to be translated, mostly-automatically, into operating system packages. They are usually translated 1:1, that is a single Cabal package becomes a single system package.

It is also possible to make Windows installers from Cabal packages, though this is typically done for a program together with all of its library dependencies, rather than packaging each library separately.

Unit of distribution

The Cabal package is the unit of distribution. What this means is that each Cabal package can be distributed on its own in source or binary form. Of course there may dependencies between packages, but there is usually a degree of flexibility in which versions of packages can work together so distributing them independently makes sense.

It is perhaps easiest to see what being ``the unit of distribution'' means by

contrast to an alternative approach. Many projects are made up of several interdependent packages and during development these might all be kept under one common directory tree and be built and tested together. When it comes to distribution however, rather than distributing them all together in a single tarball, it is required that they each be distributed independently in their own tarballs.

Cabal's approach is to say that if you can specify a dependency on a package then that package should be able to be distributed independently. Or to put it the other way round, if you want to distribute it as a single unit, then it should be a single package.

Explicit dependencies and automatic package management

Cabal takes the approach that all packages dependencies are specified explicitly and specified in a declarative way. The point is to enable automatic package management. This means tools like `cabal` can resolve dependencies and install a package plus all of its dependencies automatically. Alternatively, it is possible to mechanically (or mostly mechanically) translate Cabal packages into system packages and let the system package manager install dependencies automatically.

It is important to track dependencies accurately so that packages can reliably be moved from one system to another system and still be able to build it there. Cabal is therefore relatively strict about specifying dependencies. For example Cabal's default build system will not even let code build if it tries to import a module from a package that isn't listed in the `.cabal` file, even if that package is actually installed. This helps to ensure that there are no "untracked dependencies" that could cause the code to fail to build on some other system.

The explicit dependency approach is in contrast to the traditional "./configure" approach where instead of specifying dependencies declaratively, the `./configure` script checks if the dependencies are present on the system. Some manual work is required to transform a `./configure` based package into a Linux distribution package (or similar). This conversion work is usually done by people other than the package author(s). The practical effect of this is that only the most popular packages will benefit from automatic package management. Instead, Cabal forces the original author to specify the dependencies but the advantage is that every package can benefit from automatic package management.

The "./configure" approach tends to encourage packages that adapt themselves to the environment in which they are built, for example by disabling optional features so that they can continue to work when a particular dependency is not available. This approach makes sense in a world where installing additional dependencies is a tiresome manual process and so minimising dependencies is important. The automatic package management view is that packages should just declare what they need and the package manager will take responsibility

for ensuring that all the dependencies are installed.

Sometimes of course optional features and optional dependencies do make sense. Cabal packages can have optional features and varying dependencies. These conditional dependencies are still specified in a declarative way however and remain compatible with automatic package management. The need to remain compatible with automatic package management means that Cabal's conditional dependencies system is a bit less flexible than with the "./configure" approach.

Portability

One of the purposes of Cabal is to make it easier to build packages on different platforms (operating systems and CPU architectures), with different compiler versions and indeed even with different Haskell implementations. (Yes, there are Haskell implementations other than GHC!)

Cabal provides abstractions of features present in different Haskell implementations and wherever possible it is best to take advantage of these to increase portability. Where necessary however it is possible to use specific features of specific implementations.

For example a package author can list in the package's `.cabal` what language extensions the code uses. This allows Cabal to figure out if the language extension is supported by the Haskell implementation that the user picks. Additionally, certain language extensions such as Template Haskell require special handling from the build system and by listing the extension it provides the build system with enough information to do the right thing.

Another similar example is linking with foreign libraries. Rather than specifying GHC flags directly, the package author can list the libraries that are needed and the build system will take care of using the right flags for the compiler. Additionally this makes it easier for tools to discover what system C libraries a package needs, which is useful for tracking dependencies on system libraries (e.g. when translating into Linux distribution packages).

In fact both of these examples fall into the category of explicitly specifying dependencies. Not all dependencies are other Cabal packages. Foreign libraries are clearly another kind of dependency. It's also possible to think of language extensions as dependencies: the package depends on a Haskell implementation that supports all those extensions.

Where compiler-specific options are needed however, there is an "escape hatch" available. The developer can specify implementation-specific options and more generally there is a configuration mechanism to customise many aspects of how a package is built depending on the Haskell implementation, the operating system, computer architecture and user-specified configuration flags.

Developing packages

The Cabal package is the unit of distribution. When installed, its purpose is to make available:

- One or more Haskell programs.
- At most one library, exposing a number of Haskell modules.

However having both a library and executables in a package does not work very well; if the executables depend on the library, they must explicitly list all the modules they directly or indirectly import from that library. Fortunately, starting with Cabal 1.8.0.4, executables can also declare the package that they are in as a dependency, and Cabal will treat them as if they were in another package that depended on the library.

Internally, the package may consist of much more than a bunch of Haskell modules: it may also have C source code and header files, source code meant for preprocessing, documentation, test cases, auxiliary tools etc.

A package is identified by a globally-unique *package name*, which consists of one or more alphanumeric words separated by hyphens. To avoid ambiguity, each of these words should contain at least one letter. Chaos will result if two distinct packages with the same name are installed on the same system. A particular version of the package is distinguished by a *version number*, consisting of a sequence of one or more integers separated by dots. These can be combined to form a single text string called the *package ID*, using a hyphen to separate the name from the version, e.g. "`HUnit-1.1`".

Note: Packages are not part of the Haskell language; they simply populate the hierarchical space of module names. In GHC 6.6 and later a program may contain multiple modules with the same name if they come from separate packages; in all other current Haskell systems packages may not overlap in the modules they provide, including hidden modules.

Creating a package

Suppose you have a directory hierarchy containing the source files that make up your package. You will need to add two more files to the root directory of the package:

package`.cabal`

> a Unicode UTF-8 text file containing a package description. For details of the syntax of this file, see the section on package descriptions.

`Setup.hs`

a single-module Haskell program to perform various setup tasks (with the interface described in the section on building and installing packages. This module should import only modules that will be present in all Haskell implementations, including modules of the Cabal library. The content of this file is determined by the `build-type` setting in the `.cabal` file. In most cases it will be trivial, calling on the Cabal library to do most of the work.

Once you have these, you can create a source bundle of this directory for distribution. Building of the package is discussed in the section on building and installing packages.

One of the purposes of Cabal is to make it easier to build a package with different Haskell implementations. So it provides abstractions of features present in different Haskell implementations and wherever possible it is best to take advantage of these to increase portability. Where necessary however it is possible to use specific features of specific implementations. For example one of the pieces of information a package author can put in the package's `.cabal` file is what language extensions the code uses. This is far preferable to specifying flags for a specific compiler as it allows Cabal to pick the right flags for the Haskell implementation that the user picks. It also allows Cabal to figure out if the language extension is even supported by the Haskell implementation that the user picks. Where compiler-specific options are needed however, there is an "escape hatch" available. The developer can specify implementation-specific options and more generally there is a configuration mechanism to customise many aspects of how a package is built depending on the Haskell implementation, the Operating system, computer architecture and user-specified configuration flags.

```
name:     Foo
version:  1.0

library
  build-depends:   base
  exposed-modules: Foo
  extensions:      ForeignFunctionInterface
  ghc-options:     -Wall
  if os(windows)
    build-depends: Win32
```

Example: A package containing a simple library

The HUnit package contains a file `HUnit.cabal` containing:

```
name:           HUnit
version:        1.1.1
synopsis:       A unit testing framework for Haskell
homepage:       http://hunit.sourceforge.net/
category:       Testing
author:         Dean Herington
license:        BSD3
```

```
license-file:   LICENSE
cabal-version:  >= 1.10
build-type:     Simple

library
  build-depends:      base >= 2 && < 4
  exposed-modules:    Test.HUnit.Base, Test.HUnit.Lang,
                      Test.HUnit.Terminal, Test.HUnit.Text, Test.HUnit
  default-extensions: CPP
```

and the following `Setup.hs`:

```
import Distribution.Simple
main = defaultMain
```

Example: A package containing executable programs

```
name:           TestPackage
version:        0.0
synopsis:       Small package with two programs
author:         Angela Author
license:        BSD3
build-type:     Simple
cabal-version:  >= 1.2

executable program1
  build-depends:  HUnit
  main-is:        Main.hs
  hs-source-dirs: prog1

executable program2
  main-is:        Main.hs
  build-depends:  HUnit
  hs-source-dirs: prog2
  other-modules:  Utils
```

with `Setup.hs` the same as above.

Example: A package containing a library and executable programs

```
name:            TestPackage
version:         0.0
synopsis:        Package with library and two programs
license:         BSD3
author:          Angela Author
build-type:      Simple
cabal-version:   >= 1.2

library
  build-depends:   HUnit
  exposed-modules: A, B, C

executable program1
  main-is:         Main.hs
```

```
  hs-source-dirs:  prog1
  other-modules:   A, B

executable program2
  main-is:         Main.hs
  hs-source-dirs:  prog2
  other-modules:   A, C, Utils
```

with `Setup.hs` the same as above. Note that any library modules required (directly or indirectly) by an executable must be listed again.

The trivial setup script used in these examples uses the *simple build infrastructure* provided by the Cabal library (see Distribution.Simple). The simplicity lies in its interface rather that its implementation. It automatically handles preprocessing with standard preprocessors, and builds packages for all the Haskell implementations.

The simple build infrastructure can also handle packages where building is governed by system-dependent parameters, if you specify a little more (see the section on system-dependent parameters). A few packages require more elaborate solutions.

Package descriptions

The package description file must have a name ending in ".`cabal`". It must be a Unicode text file encoded using valid UTF-8. There must be exactly one such file in the directory. The first part of the name is usually the package name, and some of the tools that operate on Cabal packages require this.

In the package description file, lines whose first non-whitespace characters are "`--`" are treated as comments and ignored.

This file should contain of a number global property descriptions and several sections.

- The global properties describe the package as a whole, such as name, license, author, etc.
- Optionally, a number of *configuration flags* can be declared. These can be used to enable or disable certain features of a package. (see the section on configurations).
- The (optional) library section specifies the library properties and relevant build information.
- Following is an arbitrary number of executable sections which describe an executable program and relevant build information.

Each section consists of a number of property descriptions in the form of

field/value pairs, with a syntax roughly like mail message headers.

- Case is not significant in field names, but is significant in field values.
- To continue a field value, indent the next line relative to the field name.
- Field names may be indented, but all field values in the same section must use the same indentation.
- Tabs are *not* allowed as indentation characters due to a missing standard interpretation of tab width.
- To get a blank line in a field value, use an indented "`.`"

The syntax of the value depends on the field. Field types include:

token, *filename*, *directory*

> Either a sequence of one or more non-space non-comma characters, or a quoted string in Haskell 98 lexical syntax. Unless otherwise stated, relative filenames and directories are interpreted from the package root directory.

freeform, *URL*, *address*

> An arbitrary, uninterpreted string.

identifier

> A letter followed by zero or more alphanumerics or underscores.

compiler

> A compiler flavor (one of: `GHC`, `JHC`, `UHC` or `LHC`) followed by a version range. For example, `GHC ==6.10.3`, or `LHC >=0.6 && <0.8`.

Modules and preprocessors

Haskell module names listed in the `exposed-modules` and `other-modules` fields may correspond to Haskell source files, i.e. with names ending in "`.hs`" or "`.lhs`", or to inputs for various Haskell preprocessors. The simple build infrastructure understands the extensions:

- `.gc` (greencard)
- `.chs` (c2hs)
- `.hsc` (`hsc2hs`)
- `.y` and `.ly` (happy)
- `.x` (alex)
- `.cpphs` (cpphs)

When building, Cabal will automatically run the appropriate preprocessor and compile the Haskell module it produces.

Some fields take lists of values, which are optionally separated by commas, except for the `build-depends` field, where the commas are mandatory.

Some fields are marked as required. All others are optional, and unless otherwise specified have empty default values.

Package properties

These fields may occur in the first top-level properties section and describe the package as a whole:

`name:` *package-name* (required)

The unique name of the package, without the version number.

`version:` *numbers* (required)

The package version number, usually consisting of a sequence of natural numbers separated by dots.

`cabal-version:` >= *x.y*

The version of the Cabal specification that this package description uses. The Cabal specification does slowly evolve, introducing new features and occasionally changing the meaning of existing features. By specifying which version of the spec you are using it enables programs which process the package description to know what syntax to expect and what each part means.

For historical reasons this is always expressed using >= version range syntax. No other kinds of version range make sense, in particular upper bounds do not make sense. In future this field will specify just a version number, rather than a version range.

The version number you specify will affect both compatibility and behaviour. Most tools (including the Cabal library and cabal program) understand a range of versions of the Cabal specification. Older tools will of course only work with older versions of the Cabal specification. Most of the time, tools that are too old will recognise this fact and produce a suitable error message.

As for behaviour, new versions of the Cabal spec can change the meaning of existing syntax. This means if you want to take advantage of the new meaning or behaviour then you must specify the newer Cabal version. Tools are expected to use the meaning and behaviour appropriate to the

version given in the package description.

In particular, the syntax of package descriptions changed significantly with Cabal version 1.2 and the `cabal-version` field is now required. Files written in the old syntax are still recognized, so if you require compatibility with very old Cabal versions then you may write your package description file using the old syntax. Please consult the user's guide of an older Cabal version for a description of that syntax.

`build-type:` *identifier*

The type of build used by this package. Build types are the constructors of the BuildType type, defaulting to `Custom`.

If the build type is anything other than `Custom`, then the `Setup.hs` file *must* be exactly the standardized content discussed below. This is because in these cases, `cabal` will ignore the `Setup.hs` file completely, whereas other methods of package management, such as `runhaskell Setup.hs [CMD]`, still rely on the `Setup.hs` file.

For build type `Simple`, the contents of `Setup.hs` must be:

```
import Distribution.Simple
main = defaultMain
```

For build type `Configure` (see the section on system-dependent parameters below), the contents of `Setup.hs` must be:

```
import Distribution.Simple
main = defaultMainWithHooks autoconfUserHooks
```

For build type `Make` (see the section on more complex packages below), the contents of `Setup.hs` must be:

```
import Distribution.Make
main = defaultMain
```

For build type `Custom`, the file `Setup.hs` can be customized, and will be used both by `cabal` and other tools.

For most packages, the build type `Simple` is sufficient.

`license:` *identifier* (default: `AllRightsReserved`)

The type of license under which this package is distributed. License names are the constants of the License type.

`license-file:` *filename* or `license-files:` *filename list*

The name of a file(s) containing the precise copyright license for this package. The license file(s) will be installed with the package.

If you have multiple license files then use the `license-files` field instead of (or in addition to) the `license-file` field.

`copyright:` *freeform*

The content of a copyright notice, typically the name of the holder of the copyright on the package and the year(s) from which copyright is claimed. For example: `Copyright: (c) 2006-2007 Joe Bloggs`

`author:` *freeform*

The original author of the package.

Remember that `.cabal` files are Unicode, using the UTF-8 encoding.

`maintainer:` *address*

The current maintainer or maintainers of the package. This is an e-mail address to which users should send bug reports, feature requests and patches.

`stability:` *freeform*

The stability level of the package, e.g. `alpha`, `experimental`, `provisional`, `stable`.

`homepage:` *URL*

The package homepage.

`bug-reports:` *URL*

The URL where users should direct bug reports. This would normally be either:

- A `mailto:` URL, e.g. for a person or a mailing list.
- An `http:` (or `https:`) URL for an online bug tracking system.

For example Cabal itself uses a web-based bug tracking system

```
bug-reports: http://hackage.haskell.org/trac/hackage/
```

`package-url:` *URL*

The location of a source bundle for the package. The distribution should be a Cabal package.

`synopsis:` *freeform*

A very short description of the package, for use in a table of packages. This is your headline, so keep it short (one line) but as informative as possible. Save space by not including the package name or saying it's written in Haskell.

`description:` *freeform*

Description of the package. This may be several paragraphs, and should be aimed at a Haskell programmer who has never heard of your package before.

For library packages, this field is used as prologue text by `setup haddock`, and thus may contain the same markup as haddock documentation comments.

`category:` *freeform*

A classification category for future use by the package catalogue Hackage. These categories have not yet been specified, but the upper levels of the module hierarchy make a good start.

`tested-with:` *compiler list*

A list of compilers and versions against which the package has been tested (or at least built).

`data-files:` *filename list*

A list of files to be installed for run-time use by the package. This is useful for packages that use a large amount of static data, such as tables of values or code templates. Cabal provides a way to find these files at run-time.

A limited form of `*` wildcards in file names, for example `data-files: images/*.png` matches all the `.png` files in the `images` directory.

The limitation is that `*` wildcards are only allowed in place of the file name, not in the directory name or file extension. In particular, wildcards do not include directories contents recursively. Furthermore, if a wildcard is used it must be used with an extension, so `data-files: data/*` is not allowed. When matching a wildcard plus extension, a file's full extension must match exactly, so `*.gz` matches `foo.gz` but not `foo.tar.gz`. A wildcard that does not match any files is an error.

The reason for providing only a very limited form of wildcard is to concisely express the common case of a large number of related files of the same file type without making it too easy to accidentally include unwanted files.

`data-dir:` *directory*

The directory where Cabal looks for data files to install, relative to the source directory. By default, Cabal will look in the source directory itself.

`extra-source-files:` *filename list*

A list of additional files to be included in source distributions built with `setup sdist`. As with `data-files` it can use a limited form of `*` wildcards in file names.

`extra-doc-files:` *filename list*

A list of additional files to be included in source distributions, and also copied to the html directory when Haddock documentation is generated. As with `data-files` it can use a limited form of `*` wildcards in file names.

`extra-tmp-files:` *filename list*

A list of additional files or directories to be removed by `setup clean`. These would typically be additional files created by additional hooks, such as the scheme described in the section on system-dependent parameters.

Library

The library section should contain the following fields:

`exposed-modules:` *identifier list* (required if this package contains a library)

A list of modules added by this package.

`exposed:` *boolean* (default: `True`)

Some Haskell compilers (notably GHC) support the notion of packages being “exposed” or “hidden” which means the modules they provide can be easily imported without always having to specify which package they come from. However this only works effectively if the modules provided by all exposed packages do not overlap (otherwise a module import would be ambiguous).

Almost all new libraries use hierarchical module names that do not clash, so it is very uncommon to have to use this field. However it may be necessary to set `exposed: False` for some old libraries that use a flat module namespace or where it is known that the exposed modules would clash with other common modules.

`reexported-modules:` *exportlist*

Supported only in GHC 7.10 and later. A list of modules to *reexport* from this package. The syntax of this field is `orig-pkg:Name as NewName` to reexport module `Name` from `orig-pkg` with the new name `NewName`. We also support abbreviated versions of the syntax: if you omit `as NewName`, we'll reexport without renaming; if you omit `orig-pkg`, then we will automatically figure out which package to reexport from, if it's unambiguous.

Reexported modules are useful for compatibility shims when a package has been split into multiple packages, and they have the useful property that if a package provides a module, and another package reexports it under the same name, these are not considered a conflict (as would be the case with a stub module.) They can also be used to resolve name conflicts.

The library section may also contain build information fields (see the section on build information).

Opening an interpreter session

While developing a package, it is often useful to make its code available inside an interpreter session. This can be done with the `repl` command:

```
cabal repl
```

The name comes from the acronym REPL, which stands for "read-eval-print-loop". By default `cabal repl` loads the first component in a package. If the package contains several named components, the name can be given as an argument to `repl`. The name can be also optionally prefixed with the component's type for disambiguation purposes. Example:

```
cabal repl foo
cabal repl exe:foo
cabal repl test:bar
cabal repl bench:baz
```

Freezing dependency versions

If a package is built in several different environments, such as a development environment, a staging environment and a production environment, it may be necessary or desirable to ensure that the same dependency versions are selected in each environment. This can be done with the `freeze` command:

```
cabal freeze
```

The command writes the selected version for all dependencies to the `cabal.config` file. All environments which share this file will use the dependency versions specified in it.

Executables

Executable sections (if present) describe executable programs contained in the package and must have an argument after the section label, which defines the name of the executable. This is a freeform argument but may not contain spaces.

The executable may be described using the following fields, as well as build information fields (see the section on build information).

`main-is`: *filename* (required)
: The name of the `.hs` or `.lhs` file containing the `Main` module. Note that it is the `.hs` filename that must be listed, even if that file is generated using a preprocessor. The source file must be relative to one of the directories listed in `hs-source-dirs`.

Running executables

You can have Cabal build and run your executables by using the `run` command:

```
$ cabal run EXECUTABLE [-- EXECUTABLE_FLAGS]
```

This command will configure, build and run the executable `EXECUTABLE`. The double dash separator is required to distinguish executable flags from `run`'s own flags. If there is only one executable defined in the whole package, the executable's name can be omitted. See the output of `cabal help run` for a list of options you can pass to `cabal run`.

Test suites

Test suite sections (if present) describe package test suites and must have an argument after the section label, which defines the name of the test suite. This is a freeform argument, but may not contain spaces. It should be unique among the names of the package's other test suites, the package's executables, and the package itself. Using test suite sections requires at least Cabal version 1.9.2.

The test suite may be described using the following fields, as well as build information fields (see the section on build information).

`type`: *interface* (required)
: The interface type and version of the test suite. Cabal supports two test suite interfaces, called `exitcode-stdio-1.0` and `detailed-0.9`. Each of these types may require or disallow other fields as described below.

Test suites using the `exitcode-stdio-1.0` interface are executables that indicate test failure with a non-zero exit code when run; they may provide human-readable log information through the standard output and error channels. This

interface is provided primarily for compatibility with existing test suites; it is preferred that new test suites be written for the `detailed-0.9` interface. The `exitcode-stdio-1.0` type requires the `main-is` field.

`main-is:` *filename* (required: `exitcode-stdio-1.0`, disallowed: `detailed-0.9`)
: The name of the `.hs` or `.lhs` file containing the `Main` module. Note that it is the `.hs` filename that must be listed, even if that file is generated using a preprocessor. The source file must be relative to one of the directories listed in `hs-source-dirs`. This field is analogous to the `main-is` field of an executable section.

Test suites using the `detailed-0.9` interface are modules exporting the symbol `tests :: IO [Test]`. The `Test` type is exported by the module `Distribution.TestSuite` provided by Cabal. For more details, see the example below.

The `detailed-0.9` interface allows Cabal and other test agents to inspect a test suite's results case by case, producing detailed human- and machine-readable log files. The `detailed-0.9` interface requires the `test-module` field.

`test-module:` *identifier* (required: `detailed-0.9`, disallowed: `exitcode-stdio-1.0`)
: The module exporting the `tests` symbol.

Example: Package using `exitcode-stdio-1.0` interface

The example package description and executable source file below demonstrate the use of the `exitcode-stdio-1.0` interface. For brevity, the example package does not include a library or any normal executables, but a real package would be required to have at least one library or executable.

foo.cabal:

```
Name:           foo
Version:        1.0
License:        BSD3
Cabal-Version:  >= 1.9.2
Build-Type:     Simple

Test-Suite test-foo
    type:       exitcode-stdio-1.0
    main-is:    test-foo.hs
    build-depends: base
```

test-foo.hs:

```
module Main where

import System.Exit (exitFailure)

main = do
    putStrLn "This test always fails!"
```

```
    exitFailure
```

Example: Package using `detailed-0.9` interface

The example package description and test module source file below demonstrate the use of the `detailed-0.9` interface. For brevity, the example package does note include a library or any normal executables, but a real package would be required to have at least one library or executable. The test module below also develops a simple implementation of the interface set by `Distribution.TestSuite`, but in actual usage the implementation would be provided by the library that provides the testing facility.

bar.cabal:

```
Name:           bar
Version:        1.0
License:        BSD3
Cabal-Version:  >= 1.9.2
Build-Type:     Simple

Test-Suite test-bar
    type:       detailed-0.9
    test-module: Bar
    build-depends: base, Cabal >= 1.9.2
```

Bar.hs:

```
module Bar ( tests ) where

import Distribution.TestSuite

tests :: IO [Test]
tests = return [ Test succeeds, Test fails ]
  where
    succeeds = TestInstance
        { run = return $ Finished Pass
        , name = "succeeds"
        , tags = []
        , options = []
        , setOption = \_ _ -> Right succeeds
        }
    fails = TestInstance
        { run = return $ Finished $ Fail "Always fails!"
        , name = "fails"
        , tags = []
        , options = []
        , setOption = \_ _ -> Right fails
        }
```

Running test suites

You can have Cabal run your test suites using its built-in test runner:

```
$ cabal configure --enable-tests
$ cabal build
$ cabal test
```

See the output of `cabal help test` for a list of options you can pass to `cabal test`.

Benchmarks

Benchmark sections (if present) describe benchmarks contained in the package and must have an argument after the section label, which defines the name of the benchmark. This is a freeform argument, but may not contain spaces. It should be unique among the names of the package's other benchmarks, the package's test suites, the package's executables, and the package itself. Using benchmark sections requires at least Cabal version 1.9.2.

The benchmark may be described using the following fields, as well as build information fields (see the section on build information).

`type`: *interface* (required)
: The interface type and version of the benchmark. At the moment Cabal only support one benchmark interface, called `exitcode-stdio-1.0`.

Benchmarks using the `exitcode-stdio-1.0` interface are executables that indicate failure to run the benchmark with a non-zero exit code when run; they may provide human-readable information through the standard output and error channels.

`main-is`: *filename* (required: `exitcode-stdio-1.0`)
: The name of the `.hs` or `.lhs` file containing the `Main` module. Note that it is the `.hs` filename that must be listed, even if that file is generated using a preprocessor. The source file must be relative to one of the directories listed in `hs-source-dirs`. This field is analogous to the `main-is` field of an executable section.

Example: Package using `exitcode-stdio-1.0` interface

The example package description and executable source file below demonstrate the use of the `exitcode-stdio-1.0` interface. For brevity, the example package does not include a library or any normal executables, but a real package would be required to have at least one library or executable.

foo.cabal:

```
Name:           foo
Version:        1.0
License:        BSD3
Cabal-Version:  >= 1.9.2
Build-Type:     Simple
```

```
Benchmark bench-foo
    type:       exitcode-stdio-1.0
    main-is:    bench-foo.hs
    build-depends: base, time
```

bench-foo.hs:

```
{-# LANGUAGE BangPatterns #-}
module Main where

import Data.Time.Clock

fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = do
    start <- getCurrentTime
    let !r = fib 20
    end <- getCurrentTime
    putStrLn $ "fib 20 took " ++ show (diffUTCTime end start)
```

Running benchmarks

You can have Cabal run your benchmark using its built-in benchmark runner:

```
$ cabal configure --enable-benchmarks
$ cabal build
$ cabal bench
```

See the output of `cabal help bench` for a list of options you can pass to `cabal bench`.

Build information

The following fields may be optionally present in a library or executable section, and give information for the building of the corresponding library or executable. See also the sections on system-dependent parameters and configurations for a way to supply system-dependent values for these fields.

`build-depends:` *package list*

> A list of packages needed to build this one. Each package can be annotated with a version constraint.
>
> Version constraints use the operators `==`, `>=`, `>`, `<`, `<=` and a version number. Multiple constraints can be combined using `&&` or `||`. If no version constraint is specified, any version is assumed to be acceptable. For example:
>
> ```
> library
> ```

```
build-depends:
  base >= 2,
  foo >= 1.2 && < 1.3,
  bar
```

Dependencies like `foo >= 1.2 && < 1.3` turn out to be very common because it is recommended practise for package versions to correspond to API versions. As of Cabal 1.6, there is a special syntax to support this use:

```
build-depends: foo ==1.2.*
```

It is only syntactic sugar. It is exactly equivalent to `foo >= 1.2 && < 1.3`.

With Cabal 1.20 and GHC 7.10, `build-depends` also supports module thinning and renaming, which allows you to selectively decide what modules become visible from a package dependency. For example:

```
build-depends: containers (Data.Set, Data.IntMap as Map)
```

This results in only the modules `Data.Set` and `Map` being visible to the user from containers, hiding all other modules. To add additional names for modules without hiding the others, you can use the `with` keyword:

```
build-depends: containers with (Data.IntMap as Map)
```

Note: Prior to Cabal 1.8, build-depends specified in each section were global to all sections. This was unintentional, but some packages were written to depend on it, so if you need your build-depends to be local to each section, you must specify at least `Cabal-Version: >= 1.8` in your `.cabal` file.

`other-modules:` *identifier list*

A list of modules used by the component but not exposed to users. For a library component, these would be hidden modules of the library. For an executable, these would be auxiliary modules to be linked with the file named in the `main-is` field.

Note: Every module in the package *must* be listed in one of `other-modules`, `exposed-modules` or `main-is` fields.

`hs-source-dirs:` *directory list* (default: “.”)

Root directories for the module hierarchy.

For backwards compatibility, the old variant `hs-source-dir` is also recognized.

`extensions:` *identifier list*

A list of Haskell extensions used by every module. Extension names are the

constructors of the Extension type. These determine corresponding compiler options. In particular, `CPP` specifies that Haskell source files are to be preprocessed with a C preprocessor.

Extensions used only by one module may be specified by placing a `LANGUAGE` pragma in the source file affected, e.g.:

```
{-# LANGUAGE CPP, MultiParamTypeClasses #-}
```

Note: GHC versions prior to 6.6 do not support the `LANGUAGE` pragma.

`build-tools:` *program list*

A list of programs, possibly annotated with versions, needed to build this package, e.g. `c2hs >= 0.15, cpphs`.If no version constraint is specified, any version is assumed to be acceptable.

`buildable:` *boolean* (default: `True`)

Is the component buildable? Like some of the other fields below, this field is more useful with the slightly more elaborate form of the simple build infrastructure described in the section on system-dependent parameters.

`ghc-options:` *token list*

Additional options for GHC. You can often achieve the same effect using the `extensions` field, which is preferred.

Options required only by one module may be specified by placing an `OPTIONS_GHC` pragma in the source file affected.

`ghc-prof-options:` *token list*

Additional options for GHC when the package is built with profiling enabled.

`ghc-shared-options:` *token list*

Additional options for GHC when the package is built as shared library.

`includes:` *filename list*

A list of header files to be included in any compilations via C. This field applies to both header files that are already installed on the system and to those coming with the package to be installed. These files typically contain function prototypes for foreign imports used by the package.

`install-includes:` *filename list*

A list of header files from this package to be installed into `$libdir/includes` when the package is installed. Files listed in `install-includes:` should be found in relative to the top of the source tree or relative to one of the directories listed in `include-dirs`.

`install-includes` is typically used to name header files that contain prototypes for foreign imports used in Haskell code in this package, for which the C implementations are also provided with the package. Note that to include them when compiling the package itself, they need to be listed in the `includes:` field as well.

`include-dirs:` *directory list*

A list of directories to search for header files, when preprocessing with `c2hs`, `hsc2hs`, `cpphs` or the C preprocessor, and also when compiling via C.

`c-sources:` *filename list*

A list of C source files to be compiled and linked with the Haskell files.

`js-sources:` *filename list*

A list of JavaScript source files to be linked with the Haskell files (only for JavaScript targets).

`extra-libraries:` *token list*

A list of extra libraries to link with.

`extra-ghci-libraries:` *token list*

A list of extra libraries to be used instead of ‘extra-libraries’ when the package is loaded with GHCi.

`extra-lib-dirs:` *directory list*

A list of directories to search for libraries.

`cc-options:` *token list*

Command-line arguments to be passed to the C compiler. Since the arguments are compiler-dependent, this field is more useful with the setup described in the section on system-dependent parameters.

`cpp-options:` *token list*

Command-line arguments for pre-processing Haskell code. Applies to haskell source and other pre-processed Haskell source like .hsc .chs. Does not apply to C code, that’s what cc-options is for.

`ld-options:` *token list*

Command-line arguments to be passed to the linker. Since the arguments are compiler-dependent, this field is more useful with the setup described in the section on system-dependent parameters>.

`pkgconfig-depends:` *package list*

A list of pkg-config packages, needed to build this package. They can be annotated with versions, e.g. `gtk+-2.0 >= 2.10, cairo >= 1.0`. If no version constraint is specified, any version is assumed to be acceptable. Cabal uses `pkg-config` to find if the packages are available on the system and to find the extra compilation and linker options needed to use the packages.

If you need to bind to a C library that supports `pkg-config` (use `pkg-config --list-all` to find out if it is supported) then it is much preferable to use this field rather than hard code options into the other fields.

`frameworks:` *token list*

On Darwin/MacOS X, a list of frameworks to link to. See Apple's developer documentation for more details on frameworks. This entry is ignored on all other platforms.

Configurations

Library and executable sections may include conditional blocks, which test for various system parameters and configuration flags. The flags mechanism is rather generic, but most of the time a flag represents certain feature, that can be switched on or off by the package user. Here is an example package description file using configurations:

Example: A package containing a library and executable programs

```
Name: Test1
Version: 0.0.1
Cabal-Version: >= 1.2
License: BSD3
Author:  Jane Doe
Synopsis: Test package to test configurations
Category: Example

Flag Debug
  Description: Enable debug support
  Default:     False

Flag WebFrontend
  Description: Include API for web frontend.
  -- Cabal checks if the configuration is possible, first
  -- with this flag set to True and if not it tries with False
```

```
Library
  Build-Depends:   base
  Exposed-Modules: Testing.Test1
  Extensions:      CPP

  if flag(debug)
    GHC-Options: -DDEBUG
    if !os(windows)
      CC-Options: "-DDEBUG"
    else
      CC-Options: "-DNDEBUG"

  if flag(webfrontend)
    Build-Depends: cgi > 0.42
    Other-Modules: Testing.WebStuff

Executable test1
  Main-is: T1.hs
  Other-Modules: Testing.Test1
  Build-Depends: base

  if flag(debug)
    CC-Options: "-DDEBUG"
    GHC-Options: -DDEBUG
```

Layout

Flags, conditionals, library and executable sections use layout to indicate structure. This is very similar to the Haskell layout rule. Entries in a section have to all be indented to the same level which must be more than the section header. Tabs are not allowed to be used for indentation.

As an alternative to using layout you can also use explicit braces `{}`. In this case the indentation of entries in a section does not matter, though different fields within a block must be on different lines. Here is a bit of the above example again, using braces:

Example: Using explicit braces rather than indentation for layout

```
Name: Test1
Version: 0.0.1
Cabal-Version: >= 1.2
License: BSD3
Author:  Jane Doe
Synopsis: Test package to test configurations
Category: Example

Flag Debug {
  Description: Enable debug support
  Default:     False
}

Library {
```

```
  Build-Depends:   base
  Exposed-Modules: Testing.Test1
  Extensions:      CPP
  if flag(debug) {
    GHC-Options: -DDEBUG
    if !os(windows) {
      CC-Options: "-DDEBUG"
    } else {
      CC-Options: "-DNDEBUG"
    }
  }
}
```

Configuration Flags

A flag section takes the flag name as an argument and may contain the following fields.

`description:` *freeform*

The description of this flag.

`default:` *boolean* (default: `True`)

The default value of this flag.

Note that this value may be [overridden in several ways](installing-packages.html#controlling-flag-assignments“). The rationale for having flags default to True is that users usually want new features as soon as they are available. Flags representing features that are not (yet) recommended for most users (such as experimental features or debugging support) should therefore explicitly override the default to False.

`manual:` *boolean* (default: `False`)

By default, Cabal will first try to satisfy dependencies with the default flag value and then, if that is not possible, with the negated value. However, if the flag is manual, then the default value (which can be overridden by commandline flags) will be used.

Conditional Blocks

Conditional blocks may appear anywhere inside a library or executable section. They have to follow rather strict formatting rules. Conditional blocks must always be of the shape

```
  `if `_condition_
       _property-descriptions-or-conditionals*_
```

or

```
  `if `_condition_
       _property-descriptions-or-conditionals*_
  `else`
       _property-descriptions-or-conditionals*_
```

Note that the `if` and the condition have to be all on the same line.

Conditions

Conditions can be formed using boolean tests and the boolean operators `||` (disjunction / logical "or"), `&&` (conjunction / logical "and"), or `!` (negation / logical "not"). The unary `!` takes highest precedence, `||` takes lowest. Precedence levels may be overridden through the use of parentheses. For example, `os(darwin) && !arch(i386) || os(freebsd)` is equivalent to `(os(darwin) && !(arch(i386))) || os(freebsd)`.

The following tests are currently supported.

`os(`*name*`)`

> Tests if the current operating system is *name*. The argument is tested against `System.Info.os` on the target system. There is unfortunately some disagreement between Haskell implementations about the standard values of `System.Info.os`. Cabal canonicalises it so that in particular `os(windows)` works on all implementations. If the canonicalised os names match, this test evaluates to true, otherwise false. The match is case-insensitive.

`arch(`*name*`)`

> Tests if the current architecture is *name*. The argument is matched against `System.Info.arch` on the target system. If the arch names match, this test evaluates to true, otherwise false. The match is case-insensitive.

`impl(`*compiler*`)`

> Tests for the configured Haskell implementation. An optional version constraint may be specified (for example `impl(ghc >= 6.6.1)`). If the configured implementation is of the right type and matches the version constraint, then this evaluates to true, otherwise false. The match is case-insensitive.

`flag(`*name*`)`

> Evaluates to the current assignment of the flag of the given name. Flag names are case insensitive. Testing for flags that have not been introduced with a flag section is an error.

`true`

Constant value true.

`false`

Constant value false.

Resolution of Conditions and Flags

If a package descriptions specifies configuration flags the package user can control these in several ways. If the user does not fix the value of a flag, Cabal will try to find a flag assignment in the following way.

- For each flag specified, it will assign its default value, evaluate all conditions with this flag assignment, and check if all dependencies can be satisfied. If this check succeeded, the package will be configured with those flag assignments.
- If dependencies were missing, the last flag (as by the order in which the flags were introduced in the package description) is tried with its alternative value and so on. This continues until either an assignment is found where all dependencies can be satisfied, or all possible flag assignments have been tried.

To put it another way, Cabal does a complete backtracking search to find a satisfiable package configuration. It is only the dependencies specified in the `build-depends` field in conditional blocks that determine if a particular flag assignment is satisfiable (`build-tools` are not considered). The order of the declaration and the default value of the flags determines the search order. Flags overridden on the command line fix the assignment of that flag, so no backtracking will be tried for that flag.

If no suitable flag assignment could be found, the configuration phase will fail and a list of missing dependencies will be printed. Note that this resolution process is exponential in the worst case (i.e., in the case where dependencies cannot be satisfied). There are some optimizations applied internally, but the overall complexity remains unchanged.

Meaning of field values when using conditionals

During the configuration phase, a flag assignment is chosen, all conditionals are evaluated, and the package description is combined into a flat package descriptions. If the same field both inside a conditional and outside then they are combined using the following rules.

- Boolean fields are combined using conjunction (logical “and”).
- List fields are combined by appending the inner items to the outer items,

for example

```
Extensions: CPP
if impl(ghc)
  Extensions: MultiParamTypeClasses
```

when compiled using GHC will be combined to

```
Extensions: CPP, MultiParamTypeClasses
```

Similarly, if two conditional sections appear at the same nesting level, properties specified in the latter will come after properties specified in the former.

- All other fields must not be specified in ambiguous ways. For example

```
Main-is: Main.hs
if flag(useothermain)
  Main-is: OtherMain.hs
```

will lead to an error. Instead use

```
if flag(useothermain)
  Main-is: OtherMain.hs
else
  Main-is: Main.hs
```

Source Repositories

It is often useful to be able to specify a source revision control repository for a package. Cabal lets you specifying this information in a relatively structured form which enables other tools to interpret and make effective use of the information. For example the information should be sufficient for an automatic tool to checkout the sources.

Cabal supports specifying different information for various common source control systems. Obviously not all automated tools will support all source control systems.

Cabal supports specifying repositories for different use cases. By declaring which case we mean automated tools can be more useful. There are currently two kinds defined:

- The `head` kind refers to the latest development branch of the package. This may be used for example to track activity of a project or as an indication to outside developers what sources to get for making new contributions.
- The `this` kind refers to the branch and tag of a repository that contains the

sources for this version or release of a package. For most source control systems this involves specifying a tag, id or hash of some form and perhaps a branch. The purpose is to be able to reconstruct the sources corresponding to a particular package version. This might be used to indicate what sources to get if someone needs to fix a bug in an older branch that is no longer an active head branch.

You can specify one kind or the other or both. As an example here are the repositories for the Cabal library. Note that the `this` kind of repository specifies a tag.

```
source-repository head
  type:     darcs
  location: http://darcs.haskell.org/cabal/

source-repository this
  type:     darcs
  location: http://darcs.haskell.org/cabal-branches/cabal-1.6/
  tag:      1.6.1
```

The exact fields are as follows:

`type:` *token*

The name of the source control system used for this repository. The currently recognised types are:

- `darcs`
- `git`
- `svn`
- `cvs`
- `mercurial` (or alias `hg`)
- `bazaar` (or alias `bzr`)
- `arch`
- `monotone`

This field is required.

`location:` *URL*

The location of the repository. The exact form of this field depends on the repository type. For example:

- for darcs: `http://code.haskell.org/foo/`
- for git: `git://github.com/foo/bar.git`
- for CVS: `anoncvs@cvs.foo.org:/cvs`

This field is required.

`module:` *token*

CVS requires a named module, as each CVS server can host multiple named repositories.

This field is required for the CVS repository type and should not be used otherwise.

`branch:` *token*

Many source control systems support the notion of a branch, as a distinct concept from having repositories in separate locations. For example CVS, SVN and git use branches while for darcs uses different locations for different branches. If you need to specify a branch to identify a your repository then specify it in this field.

This field is optional.

`tag:` *token*

A tag identifies a particular state of a source repository. The tag can be used with a `this` repository kind to identify the state of a repository corresponding to a particular package version or release. The exact form of the tag depends on the repository type.

This field is required for the `this` repository kind.

`subdir:` *directory*

Some projects put the sources for multiple packages under a single source repository. This field lets you specify the relative path from the root of the repository to the top directory for the package, i.e. the directory containing the package's `.cabal` file.

This field is optional. It default to empty which corresponds to the root directory of the repository.

Downloading a package's source

The `cabal get` command allows to access a package's source code - either by unpacking a tarball downloaded from Hackage (the default) or by checking out a working copy from the package's source repository.

```
$ cabal get [FLAGS] PACKAGES
```

The `get` command supports the following options:

`-d --destdir` *PATH*

Where to place the package source, defaults to (a subdirectory of) the

current directory.

`-s --source-repository` *[head|this|...]*

Fork the package's source repository using the appropriate version control system. The optional argument allows to choose a specific repository kind.

Accessing data files from package code

The placement on the target system of files listed in the `data-files` field varies between systems, and in some cases one can even move packages around after installation (see prefix independence). To enable packages to find these files in a portable way, Cabal generates a module called `Paths_`*pkgname* (with any hyphens in *pkgname* replaced by underscores) during building, so that it may be imported by modules of the package. This module defines a function

```
getDataFileName :: FilePath -> IO FilePath
```

If the argument is a filename listed in the `data-files` field, the result is the name of the corresponding file on the system on which the program is running.

Note: If you decide to import the `Paths_`*pkgname* module then it *must* be listed in the `other-modules` field just like any other module in your package.

The `Paths_`*pkgname* module is not platform independent so it does not get included in the source tarballs generated by `sdist`.

Accessing the package version

The aforementioned auto generated `Paths_`*pkgname* module also exports the constant `version :: Version` which is defined as the version of your package as specified in the `version` field.

System-dependent parameters

For some packages, especially those interfacing with C libraries, implementation details and the build procedure depend on the build environment. The `build-type Configure` can be used to handle many such situations. In this case, `Setup.hs` should be:

```
import Distribution.Simple
main = defaultMainWithHooks autoconfUserHooks
```

Most packages, however, would probably do better using the `Simple` build type and configurations.

The `build-type Configure` differs from `Simple` in two ways:

- The package root directory must contain a shell script called `configure`. The configure step will run the script. This `configure` script may be produced by autoconf or may be hand-written. The `configure` script typically discovers information about the system and records it for later steps, e.g. by generating system-dependent header files for inclusion in C source files and preprocessed Haskell source files. (Clearly this won't work for Windows without MSYS or Cygwin: other ideas are needed.)

- If the package root directory contains a file called *package*`.buildinfo` after the configuration step, subsequent steps will read it to obtain additional settings for build information fields,to be merged with the ones given in the `.cabal` file. In particular, this file may be generated by the `configure` script mentioned above, allowing these settings to vary depending on the build environment.

The build information file should have the following structure:

> *buildinfo*
>
> `executable:` *name buildinfo*
>
> `executable:` *name buildinfo* ...

where each *buildinfo* consists of settings of fields listed in the section on build information. The first one (if present) relates to the library, while each of the others relate to the named executable. (The names must match the package description, but you don't have to have entries for all of them.)

Neither of these files is required. If they are absent, this setup script is equivalent to `defaultMain`.

Example: Using autoconf

This example is for people familiar with the autoconf tools.

In the X11 package, the file `configure.ac` contains:

```
AC_INIT([Haskell X11 package], [1.1], [libraries@haskell.org], [X11])

# Safety check: Ensure that we are in the correct source directory.
AC_CONFIG_SRCDIR([X11.cabal])

# Header file to place defines in
AC_CONFIG_HEADERS([include/HsX11Config.h])

# Check for X11 include paths and libraries
AC_PATH_XTRA
AC_TRY_CPP([#include <X11/Xlib.h>],,[no_x=yes])

# Build the package if we found X11 stuff
```

```
if test "$no_x" = yes
then BUILD_PACKAGE_BOOL=False
else BUILD_PACKAGE_BOOL=True
fi
AC_SUBST([BUILD_PACKAGE_BOOL])

AC_CONFIG_FILES([X11.buildinfo])
AC_OUTPUT
```

Then the setup script will run the `configure` script, which checks for the presence of the X11 libraries and substitutes for variables in the file `X11.buildinfo.in`:

```
buildable: @BUILD_PACKAGE_BOOL@
cc-options: @X_CFLAGS@
ld-options: @X_LIBS@
```

This generates a file `X11.buildinfo` supplying the parameters needed by later stages:

```
buildable: True
cc-options:  -I/usr/X11R6/include
ld-options:  -L/usr/X11R6/lib
```

The `configure` script also generates a header file `include/HsX11Config.h` containing C preprocessor defines recording the results of various tests. This file may be included by C source files and preprocessed Haskell source files in the package.

Note: Packages using these features will also need to list additional files such as `configure`, templates for `.buildinfo` files, files named only in `.buildinfo` files, header files and so on in the `extra-source-files` field to ensure that they are included in source distributions. They should also list files and directories generated by `configure` in the `extra-tmp-files` field to ensure that they are removed by `setup clean`.

Quite often the files generated by `configure` need to be listed somewhere in the package description (for example, in the `install-includes` field). However, we usually don't want generated files to be included in the source tarball. The solution is again provided by the `.buildinfo` file. In the above example, the following line should be added to `X11.buildinfo`:

```
install-includes: HsX11Config.h
```

In this way, the generated `HsX11Config.h` file won't be included in the source tarball in addition to `HsX11Config.h.in`, but it will be copied to the right location during the install process. Packages that use custom `Setup.hs` scripts can update the necessary fields programmatically instead of using the `.buildinfo` file.

Conditional compilation

Sometimes you want to write code that works with more than one version of a

dependency. You can specify a range of versions for the dependency in the `build-depends`, but how do you then write the code that can use different versions of the API?

Haskell lets you preprocess your code using the C preprocessor (either the real C preprocessor, or `cpphs`). To enable this, add `extensions: CPP` to your package description. When using CPP, Cabal provides some pre-defined macros to let you test the version of dependent packages; for example, suppose your package works with either version 3 or version 4 of the `base` package, you could select the available version in your Haskell modules like this:

```
#if MIN_VERSION_base(4,0,0)
... code that works with base-4 ...
#else
... code that works with base-3 ...
#endif
```

In general, Cabal supplies a macro `MIN_VERSION_`*`package`*`_(A,B,C)` for each package depended on via `build-depends`. This macro is true if the actual version of the package in use is greater than or equal to `A.B.C` (using the conventional ordering on version numbers, which is lexicographic on the sequence, but numeric on each component, so for example 1.2.0 is greater than 1.0.3).

Since version 1.20, there is also the `MIN_TOOL_VERSION_`*`tool`* family of macros for conditioning on the version of build tools used to build the program (e.g. `hsc2hs`).

Cabal places the definitions of these macros into an automatically-generated header file, which is included when preprocessing Haskell source code by passing options to the C preprocessor.

Cabal also allows to detect when the source code is being used for generating documentation. The `__HADDOCK_VERSION__` macro is defined only when compiling via haddock instead of a normal Haskell compiler. The value of the `__HADDOCK_VERSION__` macro is defined as `A*1000 + B*10 + C`, where `A.B.C` is the Haddock version. This can be useful for working around bugs in Haddock or generating prettier documentation in some special cases.

More complex packages

For packages that don't fit the simple schemes described above, you have a few options:

- By using the `build-type Custom`, you can supply your own `Setup.hs` file, and customize the simple build infrastructure using *hooks*. These allow you to perform additional actions before and after each command is run, and also to specify additional preprocessors. A typical `Setup.hs` may look like this:

  ```
  import Distribution.Simple
  ```

```
main = defaultMainWithHooks simpleUserHooks { postHaddock = posthaddock }

posthaddock args flags desc info = ....
```

See `UserHooks` in Distribution.Simple for the details, but note that this interface is experimental, and likely to change in future releases.

- You could delegate all the work to `make`, though this is unlikely to be very portable. Cabal supports this with the `build-type Make` and a trivial setup library Distribution.Make, which simply parses the command line arguments and invokes `make`. Here `Setup.hs` should look like this:

```
import Distribution.Make
main = defaultMain
```

The root directory of the package should contain a `configure` script, and, after that has run, a `Makefile` with a default target that builds the package, plus targets `install`, `register`, `unregister`, `clean`, `dist` and `docs`. Some options to commands are passed through as follows:

  - The `--with-hc-pkg`, `--prefix`, `--bindir`, `--libdir`, `--datadir`, `--libexecdir` and `--sysconfdir` options to the `configure` command are passed on to the `configure` script. In addition the value of the `--with-compiler` option is passed in a `--with-hc` option and all options specified with `--configure-option=` are passed on.

  - The `--destdir` option to the `copy` command becomes a setting of a `destdir` variable on the invocation of `make copy`. The supplied `Makefile` should provide a `copy` target, which will probably look like this:

```
copy :
        $(MAKE) install prefix=$(destdir)/$(prefix) \
                        bindir=$(destdir)/$(bindir) \
                        libdir=$(destdir)/$(libdir) \
                        datadir=$(destdir)/$(datadir) \
                        libexecdir=$(destdir)/$(libexecdir) \
                        sysconfdir=$(destdir)/$(sysconfdir) \
```

- Finally, with the `build-type Custom`, you can also write your own setup script from scratch. It must conform to the interface described in the section on building and installing packages, and you may use the Cabal library for all or part of the work. One option is to copy the source of `Distribution.Simple`, and alter it for your needs. Good luck.