Exercises on Haskell development tools

The goal of this part is to become more familiar with developing Haskell programs and using Haskell tools.

1 Cabal, cabal-install, Hackage

The `Cabal` library provides an infrastructure for Haskell packages. The `cabal-install` packages provides – somewhat confusingly – the binary frontend called `cabal`.

Exercise 1.1. Figure out how to invoke the `cabal` binary on your machine and type `cabal help` to get some general help.

Exercise 1.2. Download the current package list from Hackage by saying `cabal update`. This can take a little while.

Exercise 1.3. Type `cabal list` to get the long, long list of packages that are available on Hackage.

Exercise 1.4. Go to `http://hackage.haskell.org` and then click on the link "Packages" to see the same list of packages in a (slightly) nicer format.

Exercise 1.5. Say `cabal list hlint`. Then find the `hlint` package also in your browser in the Hackage package list. Click on the link. Verify that the version listed by `cabal list hlint` is the latest version currently available.

Exercise 1.6. Click on the "package description" link near the bottom of the page. Look at the package description in detail. Look what kind of information is presented here, and see if you understand all of it.

Further below, you'll find sections labeled "library" and "executable". This package defines both a library for use in other programs, and a binary that can be executed on the command line by the user once the package is installed.

Note how the "build-depends" line in the "library" section lists other packages of particular versions as dependencies.

Note how the "exposed-modules" line lists a single Haskell module of the library that can be used from other packages, and how "other-modules" lists several modules that are private to the library.

Exercise 1.7. Say `cabal unpack hlint` on your shell. This will download the latest version of `hlint` and unpack it for inspection in a directory underneath your current working directory. Change to that directory, and find the `hlint.cabal` file in there again. Verify that it is the same as the one you've looked at before online.

Exercise 1.8. Try to find the Haskell sources in the package. The `.cabal` file specifies where they are in the lines labeled "hs-source-dirs". Figure out how module names are mapped to the directory hierarchy. Find the single exposed module and look at that. You'll see that this module imports another source module and re-exports that. Find this wrapped module and briefly look at that.

Exercise 1.9. In the module you're just looking at, there is a line at the top that is a so-called *pragma*. It looks a bit like a comment, but is a special instruction for the compiler. Browse to the GHC homepage at `http://www.haskell.org/ghc/`, then find your way from there to "The User's Guide". In the documentation, look for pragmas and in particular the kind of pragma you found in the source file.

Exercise 1.10. The GHC User's Guide mentions a ghc invocation in the section you have just read to display a certain set of capabilities of ghc. Type in that command on your shell and browse the long list.

Exercise 1.11. Type `cabal install --dry-run hlint`. (This will only work if you've issued the command `cabal update` before.) This will present you with a list of packages that `cabal` is about to install on your system. Note how there is a certain similarity between this list of packages and the dependencies listed in `hlint`'s `.cabal` file or on the Hackage package for `hlint`.

Exercise 1.12. Now actually install `cabal install hlint`. This will download and install the packages to your system, underneath your home directory – so no special permissions are required. Depending on your exact configuration, this installation may *fail*, and if you repeat the command, you might see an error message claiming that `happy` could not be found. If so (and *only if you got the error message*), type `cabal install happy`.

You can already see that while `cabal` is great at resolving dependencies between libraries, it is not great at resolving dependencies between build tools, and `happy` is a tool, not a library.

Exercise 1.13. The `cabal` command installs binaries into a non-standard location, namely into the directory `.cabal/bin` underneath your home directory (note the initial `.`, which makes the file "hidden" on a Unix system). Change your search path to include this directory. If you've had to install `happy` before, then you should now be able to run `happy` from the command line, and then run `cabal install hlint` to completion.

Then you should be able to run `hlint` as well.

2 Code quality

We are going to run a few tools on code you and others have produced, to get a feel for how the tools behave, and to gain an understanding for stylistic issues.

Exercise 2.1. Go back to the top directory of the unpacked `hlint` *sources* and type `hlint src` on your shell. Note that even the author of `hlint` gets suggestions to improve his own code. Try to understand a few of the suggestions.

Exercise 2.2. Now run `hlint` on a couple of your own source files. Look at the suggestions closely. Try to fix a few of them. Then run `hlint` again. See if you get more.

Exercise 2.3. Call `ghc -Wall` or `ghci -Wall` on a few of your source files in order to see what warnings GHC will produce. Try to understand (and if possible fix) the warnings.

3 Building a Cabal package

The goal is now to create your own Cabal package out of one of your own programs.

Exercise 3.1. Try to create a small, but complete Haskell application, by taking a file with a couple of functions, making sure that there is no module header or alternative a line called

```
module Main where
```

on top. Then add a main function as follows:

```
main :: IO ()
main = print (...)
```

where you replace ... by an interesting function call. Verify that you can compile this file by calling `ghc` (not `ghci`) on the source file. Run the resulting executable and confirm it is producing the correct result.

Exercise 3.2. Create a new directory, named after the package you are going to create (choose a simple name). Place the source file in that directory. Then change to that directory and type `cabal init` on the shell. Follow the dialogue. Ideally, at the end, you will have a `.cabal` file in your directory. Look at that file.

Make sure the file has an "executable" section and a line "main-is" that points to the right file.

Exercise 3.3. Type `cabal configure` followed by `cabal build`. This should build your sources. The binary will not be installed, but will be located at `dist/build/...` underneath the package directory. To install, type `cabal install` (without further arguments). Run the installed command. Make some changes to the source file, then type `cabal install` again, then run the command again.

Exercise 3.4. Split your package into a library and an executable part, i.e., create a separate module with the function and have the main module import that module and just contain the `main` function. Choose a sufficiently unique module name for your library module.

Edit your `.cabal` file to have an extra `library` section. Use the `.cabal` file of `hlint` for inspiration.

Exercise 3.5. Use `cabal install` on your changed package to install both library and binary. Call `ghci` from a different directory and see if you can use `:m` on the `ghci` prompt to make your module available, and if you can run a function therein.

4 Haddock

Haddock is the most commonly used documentation tool for Haskell.

Exercise 4.1. Verify that `haddock` is properly installed on your machine by typing the command `haddock --version` on the command line.

Exercise 4.2. Go to Hackage in your browser and find the `fibonacci` package. In the "Modules" section, click on the link for the "Data.Numbers.Fibonacci" module. The page you're now seeing has been generated by Haddock.

Both `cabal install` and `cabal unpack` the `fibonacci` package. Find the source file for the module and figure out how the comments in the file have been annotated to produce the Haddock markup.

Look at the Haddock manual, to be found via the Haddock homepage at `http://haskell.org/haddock`, for further documentation on the Haddock markup format.

Exercise 4.3. Edit the library file in your own package and add Haddock comments. Then call `cabal haddock` to generate documentation for your package. Find it in the `dist` directory underneath your package directory, and view it in the browser.