CIS 194: Haskell Programming in the Large

Throughout the semester we have concentrated on programming assignments consisting of just one or two modules. As you work on your final projects you may need to use many modules, install and interact with other packages which themselves contain many modules, and so on. This document is intended to guide you through the basics of this aspect of the Haskell ecosystem.

1 Hackage

Hackage (`http://hackage.haskell.org/`) is the standard, central repository for Haskell packages. It contains a huge range of all sorts of packages.[1] If you need some code to accomplish a particular task, look on Hackage first to see whether a package already exists that fits your need.

2 `cabal`

The `cabal` tool, which comes as part of the Haskell Platform, assists in managing Haskell packages. One of its most useful features is the ability to automatically download and install packages from Hackage. For example, executing

```
cabal install pandoc
```

at a command prompt (*not* a `ghci` prompt) will automatically download and compile the `pandoc` package, *along with all of its recursive dependencies*.

Try typing `cabal help` to see a list of commands that are available.

It is not hard to make your own "cabalized" package which can be uploaded to Hackage—use the `cabal init` command to generate the initial infrastructure, then edit the `.cabal` file which is generated. This is not required for your project but it's a nice way to package it up for other people to try (even if you don't upload it to Hackage).

3 Modules

So far in the course we have used module declarations like

```
module Parser where
```

which suffices for simple, standalone files, but there are a few more things you should know about modules when developing larger projects.

[1] In fact, perhaps it contains *too many*: it can be difficult to find things! An in-progress redesign of the site to allow better organization, searching by tags and/or popularity, *etc.*, may help with this.

3.1 Hierarchical module names

First, module names may be *hierarchical*; that is, they may consist of a sequence of names separated by periods, like this:

```
module Text.Pandoc.Writers.LaTeX where
```

Note that GHC expects to find hierarchically-named modules in a corresponding place in the filesystem. For example, the above module should be in a file `Text/Pandoc/Writers/LaTeX.hs`, that is, a file named `LaTeX.hs` contained within a `Writers` directory which is itself a subdirectory of `Pandoc`, ... and so on. This is a good way to give some organization to projects containing many modules.

3.2 Export and import lists

By default, everything defined in a module is *exported*, that is, made available to any other modules which import it. However, you can choose to explicitly export only certain things in a module with an *export list*. An export list looks like this:

```
module My.Awesome.Module (Baz(..), Bar, mkBar) where

data Baz a = EmptyBaz | Node (Int -> Baz a)

data Bar = I Int | C Char

mkBar :: Int -> Bar
mkBar i = I (blerf i)

blerf :: Int -> Int
blerf = (+1)
```

In this example, the type `Baz` is exported, along with its constructors `EmptyBaz` and `Node` (that's what the `(..)` syntax means—to export only some constructors you can also give an explicit comma-separated list of them in place of the `..`). In contrast, the type `Bar` is exported but *its constructors are not*. So anyone who imports `My.Awesome.Module` will be able to use and refer to things of type `Bar` but they will not be able to directly construct or pattern-match on them. The function `mkBar` is exported, which gives clients a way to indirectly construct values of type `Bar`. Notice that `mkBar` calls `blerf` but `blerf` itself is not exported. So clients of `My.Awesome.Module` can call `mkBar` (and hence indirectly call `blerf`), but they cannot directly use `blerf`.

By the same token, one can have explicit *import* lists, to specify that you only want to import certain things from a module. For example,

```
import Data.List (groupBy)
```

means that you are only going to use the `groupBy` function from `Data.List`. This is useful to help document and keep track of what you are actually using from each module, and also sometimes to help prevent name clashes—see below.

3.3 Dealing with name clashes

A problem arises if several modules export functions or types with the same name. There are several solutions to this problem. For concreteness, let's suppose modules `A` and `B` both export a function named `foo`.

If you only need the `foo` from `A`, and you don't care about the one from `B`, you could give an explicit import list for `B` which doesn't include `foo`:

```
import A
import B (baz, bar)
```

Sometimes this is tedious, however, if you are using *lots* of stuff from `B`. In that case you can also specify a `hiding` clause:

```
import A
import B hiding (foo)
```

This means to import everything from `B` *except for* `foo`.

But what if you want to use both `foo`s? In that case you need to use *qualified* imports. For example,

```
import qualified A
import qualified B
```

Now you can say `A.foo` and `B.foo` to disambiguate which `foo` you want. You can also do something like

```
import qualified Long.Module.Name as L
```

and now you get to refer to things imported from `Long.Module.Name` as `L.foo` instead of `Long.Module.Name.foo`.