HASKELL PROGRAMMING: GETTING STARTED

LENNART JANSSON AND BRANDON AZAD

1. Installing Haskell

Before you start coding in Haskell, you need to be able to compile and run code on your own computer, or this class will not be very interesting :)

The recommended course of action for getting Haskell up and running on your computer is to install the Haskell Platform (`http://www.haskell.org/platform/`), which includes:

(1) GHC, the most popular and powerful compiler for Haskell
(2) GHCi, an interactive shell (REPL) to test code easily using GHC
(3) Cabal, a package manager for Haskell libraries

and many other more advanced things that we will probably not touch. It's very easy to install on Windows and Mac OSX with OS-specific installers, and all popular Linux distros have some form of the Haskell Platform in their package managers. If you are having trouble, we can help you get things running, this is the plan at our first meeting.

2. The Textbook

For this SoFo the only "textbook" we will be using is *Learn You a Haskell For Great Good!* by Miran Lipovaca, which is freely available online at `http://learnyouahaskell.com/`, it's good stuff and great for beginners. We may occasionally also have recommended supplemental readings from other websites.

3. Running Haskell code

The way we will be running Haskell code, at least at the beginning of the SoFo, is to write the code in a file, save it with a `.hs` extension, then load it up in GHCi. To start GHCi, one simply types `ghci` into a terminal, and this will start the GHCi REPL shell. Then, enter `:l <name of file>` into GHCi.

For instance, to load up the file `foo.hs` and run the function `hello` in that file, we enter `ghci` into terminal, followed by entering `:l foo.hs`, then `hello`, into GHCi, which will then print the result of running function `hello`.

4. Basic introduction to syntax

It's highly recommended to read Chapters 1 and 2 of *Learn You a Haskell* to get a more complete picture of the basics. This is just intended as a quickstart guide for people who are very eager.

Functions in Haskell are different than functions in other language because they are *pure*. This means functions can't do anything that modifies the outside world, and really can only talk to the rest of the world through the value they return. This is called functional purity, and it ensures that if you call a function in Haskell several times with the same parameters, it will return the same thing every time.

Of course, this is not strictly true since there has to be *some* way to do IO. This is handled elegantly in Haskell using the mathematical abstraction of monads; we will not discuss this for several weeks.

4.1. **Function call.** To call functions, give the name of the function, followed by the parameters to give to the function, all separated by spaces (no commas or parentheses necessary). For instance, to call the function `pythag` with parameters `3` and `4`, we write:

```
pythag 3 4
```

4.2. **Operators.** Operators in Haskell are actually treated and defined just like functions, the only difference being that they are most commonly called *infix*, rather than prefix like most functions. That way, you can say `1 + 2` like in most languages, instead of `+ 1 2` like in Lisp dialects. (You would probably agree that `1 + 2` is easier to read!)

However, any function and any operator can be moved either infix or prefix. If you put parentheses around an operator, it becomes a prefix function. So, you could say `(+) 1 2` if you really wanted to, though there's not much reason to do so now. Later we'll talk about higher-order function constructs, and to refer to operators as functions in themselves they will need to have the parentheses.

Similarly, functions with alphabetic names are usually used prefix, but using backticks (`‘`, the other character on the tilde key) a function can be used infix. For example, we can say `mod 5 3` which gives the result `2`, but some people think it looks nicer to write it as `5 ‘mod‘ 2`.

4.3. **Function definition.** Function definition is prettier and contains less noise than function definition in most other languages. Here the definition of the function `pythag`:

```
pythag x y = sqrt (x^2 + y^2)
```

That's all there is to it, for now. The name of the function, not preceded by any keywords, then followed by the names of all the parameters we want for the function separated by spaces. The definition is just a Haskell expression that uses the parameters. Here's a function that uses an `if` statement.

```
evenOrOdd n = if even n
                then "even"
                else "odd"
```

There is some more advanced function syntax that we will encounter next week that make more complex functions easier to write. These include `let` bindings, `where` bindings, pattern matching, and guards.

Make sure you understand that in Haskell the equals sign (`=`) is *definition*, not *assignment*. In Haskell we can't have variables that we can assign values to like in an imperative language. Once we've defined the function `pythag`, we can't redefine it later, and similarly, we can reference the function `pythag` above where it's actually defined in the file. Everything behaves just like the way we define things in mathematics.

5. Tips for the first Problem Set

On the first problem set you'll have a bunch of simple functions to implement, which you can then run by loading the file in GHCi. You won't need any advanced features of Haskell, and all the functions are simple to define in terms of built-in functions in Haskell, as a first exercise in thinking functionally. You should read Chapter 2 of *Learn You a Haskell*, and using GHCi to play around, get a good feel for what all of the following functions do:

Arithmetic functions (`+`, `-`, `*`, `/`, `div`, `mod`), boolean functions (`&&`, `||`), comparison functions (`==`, `/=`, `<`, `>`), `min`, `max`, `++`, `:`, `!!`, `head`, `tail`, `last`, `init`, `length`, `null`, `reverse`, `take`, `drop`, `sum`, `product`, `minimum`, `maximum`, `elem`, `cycle`, `repeat`, `replicate`, `fst`, `snd`, and `zip`. You will also need to get familiar with ranges and list comprehensions.