Lecture 2

Warming up

We will start with the haskell interpreter `ghci`. To start the interpreter type the following on the command line. You might see something like this.

```
$ ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

The `Prelude>` is the prompt and the interpreter is expecting you to type in stuff. What you can type is any valid haskell expression. You can for example use ghci like a calculator.

```
Prelude> 42
42
Prelude> 2 + 4
6
Prelude> sqrt 2
1.4142135623730951
```

The line 5 `sqrt 2` is a *function application*, the function sqrt is applied on 2. In haskell, applying a function `f` on an argument `x` is done by placing `f` on the left of `x`. Function application associates to the left, i.e. `f x y z` means `((f x) y)z`.

2.1 Types

Although it did not appear to be the case in this limited interaction, Haskell is a strongly typed language unlike python or scheme. All values in haskell have a type. However we can almost always skip types. This is because the compiler/interpreter infers the correct type of the expression for us. You can find the type of an expression prefixing it with ':type' at the prompt. For example

```
Prelude> :type 'a'
'a' :: Char
```

In haskell the a type of a value is asserted by using `::` (two colons). The interpreter just told us that `'a'` is an expression of type `Char`.

Some basic types in Haskell are given in the table below

2.2 Lists and Strings.

A list is one of the most important data structure in Haskell. A list is denoted by enclosing its components inside a square bracket.

```
Prelude> :type ['a','b']
['a','b'] :: [Char]
Prelude> :type "abcde"
"abcde" :: [Char]
```

A string in Haskell is just a list of characters. The syntax of a string is exactly as in say C with escapes etc. For example `"Hello world"` is a string.

Unlike in other languages like python or scheme, a list in Haskell can have only one type of element, i.e. `[1, 'a']` is *not* a valid expression in haskell. A list of type `t` is denoted by `[t]` in haskell. Example the type `String` and `[Char]` are same and denote strings in Haskell.

2.3 Functions.

In haskell functions are first class citizens. They are like any other values. Functions can take functions as arguments and return functions as values. A function type is denoted using the arrow notation, i.e. A function that takes an integer and returns a character is denoted by `Integer -> Char`.

```
Prelude> :type length
length :: [a] -> Int
```

Notice that the interpreter tells us that length is a function that takes the list of `a` and returns an `Int` (its length). The type `a` in this context is a *type variable*, i.e. as far as length is concerned it does not care what is the type of its list, it returns the length of it. In Haskell one can write such generic functions. This feature is called polymorphism. The compiler/interpreter will appropriately infer the type depending on its arguments

```
Prelude> length [1,2,3]
3
Prelude> length "Hello"
5
```

Polymorphism is one of the greatest strengths of Haskell. We will see more of this in time to come.

Let us now define a simple haskell function.

```
fac 0 = 1
fac n = n * fac (n - 1)
```

Save this in a file, say `fac.hs` and load it in the interpreter

```
$ ghci fac.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 1] Compiling Main             ( fac.hs, interpreted )
Ok, modules loaded: Main.
Main> fac 0
1
Main> fac 4
24
```

A haskell function is defined by giving a set of equations. A function equation looks like

```
f pat_1_1 pat_1_2 ... pat_1_n  = expr_1
f pat_2_1 pat_2_2 ... pat_2_n  = expr_2
          ...
f pat_m_1 pat_2_2 ... pat_m_n  = expr_m
```

The formal parameters can be *patterns*. We will define what patterns in detail latter on but a constant like `0` or a variable like `n` is a pattern. When a function is evaluated, its actual arguments are *matched* with each of the equations in turn. We cannot explain what matching means in full detail here because we have not explained patterns yet. However, for constant patterns like `0` or a variable it is easy. An actual argument can match a constant if and only if the argument evaluates to that constant. On the other hand a variable pattern matches any argument. If a function `f` is defined using n equations, then while evaluating the function on a set of arguments each equation is tried out in order. The first equation of `f` whose formal parameters match the actual argument is used for evaluation.

To illustrate we consider the function `fac` that we defined. The expression `fac 0` evaluates to 1 because the actual parameter matches with the formal parameter 0 in the first equation. On the other hand, in the expression `fac 2` the actual argument `2` cannot match the formal argument 0 in the first equation of `fac`. Therefore the next equation is tried namely `fac n = n * fac (n-1)`. Here the formal parameter `n` matches 2 (recall a variable pattern can match any argument). Therefore, `fac 2` evaluates to `2 * fac (2 - 1)` which by recursion evaluates to 2.

We give two alternate defintions of the factorial function, the first using *guards* and the next using `if then else`.

```
fac1 n | n == 0      = 1
       | otherwise   = n * fac1 (n -1)

fac2 n = if n == 0 then 1 else n * fac2 (n -1)
```

To summarise

1. Haskell functions are polymorphic
2. They can be recursive.
3. One can use pattern matching in their definition.

Haskell type	What they are
Bool	Boolean type
Int	Fixed precision integer
Char	Character
Integer	Multi-precision integer