Lecture 3

More on pattern matching.

In the last lecture we, defined the factorial function and illustrated the use of pattern matching. In this chapter we elaborate on it a bit more.

3.1 Pattern matching on lists.

We have already seen the list type. There is an algebraic way of defining a list. A list either an empty list or an element attached to the front of an already constructed list. An empty list in Haskell is expressed as `[]` where as a list whose first element is `x` and the rest is `xs` is denoted by `x:xs`. The notation `[1,2,3]` is just a *syntactic sugar* for `1:(2:(3:[]))` or just `1:2:3:[]` as `:` is a right associative operator. We now illustrate pattern matching on list by giving the example of the `map` function. The `map` function takes a function and a list and applies the function on each element of the list. Here is the complete definition including the type

```
> import Prelude hiding (map, null, curry, uncurry)
>        -- hide Prelude functions defined here
>
> map :: (a -> b) -> [a] -> [b]
> map f []           = []
> map f (x:xs)       = f x : map f xs
```

Since a list can either be empty or of the form `(x:xs)` this is a complete definition. Also notice that pattern matching of variables is done here.

3.2 Literate programming detour.

Before we proceed further, let us clarify the >'s at the beginning of the of the lines. This is the literate programming convention that Haskell supports. Literate programming is a style of programming championed by Knuth, where comments are given more importance than the code. It is not restricted to Haskell alone; in fact TeX and METAFONT were written first written by Knuth in a literate version Pascal language called WEB and later on ported to CWEB, a literate version of the C Programming language. The `ghc` compiler and the `ghci` interpreter supports both the literate and non-literate version of Haskell.

Normally any line is treated as part of the program unless the commented. In literate haskell all lines other than

1. Those that start with a '>' or
2. A block of lines enclosed in a `\begin{code} \end{code}`

are treated as comments. We will use this literate style of programming; we will use only the first form i.e. program lines start with a >. The advantage is that one can download the notes directly and compile and execute them.

3.3 Wild card patterns.

We now illustrate the use of wild card patterns. Consider the function that tests whether a list is empty. This can be defined as follows.

```
> null [] = True
> null _  = False
```

The pattern _ (under score) matches any expression just like a variable pattern. However, unlike a variable pattern where the matched value is bound to the variable, a wild card discards the value. This can be used when we do not care about the value in the RHS of the equation.

3.4 Tuples and pattern matching.

Besides lists Haskell supports the tuple type. Tuple types corresponds to taking set theoretic products. For example the tuple `(1,"Hello")` is an ordered pair consisting of the integer `1` and the string `"Hello"`. Its type is `(Int,String)` or equivalently `(Int,[Char])` as `String` is nothing but `[Char]`. We illustrate the pattern matching of tuples by giving the definition of the standard functions `curry` and `uncurry`.

3.5 A brief detour on currying

In haskell functions are univariate functions unlike other languages. Multi-parameter functions are captured using the process called *currying*. A function taking two arguments `a` and `b` and returning `c` can be seen as a function taking the `a` and returning a function that takes `b` and returning `c`. This kind of function is called a `curried` function. Another way in which we can represent a function taking 2 arguments is to think of the function as taking a tuple. This is its uncurried form. We now define the higher order functions that transforms between these two forms.

```
> curry    :: ((a,b) -> c)  -> a -> b -> c
> uncurry  :: (a -> b -> c) -> (a,b) -> c
>
> curry   f a b   = f (a,b)
> uncurry f (a,b) = f a b
```

The above code clearly illustrates the power of Haskell when it comes to manipulating functions. Use of higher order functions is one of the features that we will find quite a bit of use.

3.6 Summary

In this lecture we saw

1. Pattern matching for lists,
2. Tuples and pattern matching on them,
3. Literate haskell
4. Higher order functions.