Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

November 01, 2016 Week 6 Lab1

LAB MANUAL: HIGHER ORDER FUNCTIONS

In this tutorial you will learn about

- Higher Order Functions
- Currying
- Partial Application of Functions

Higher Order Functions

Lets have a look at the implementation of the `zip` function we did last time

```
zip' :: [a] -> [b] -> [(a,b)]
zip' [] ys = []
zip' xs [] = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

Lets define a similar function called `zipWith`

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Have a careful look at the highlighted part. What is happening there?

We are passing a function as an argument to `zipWith` function. Have you seen this before anywhere else? This is something unique. Lets have a look at another function

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

`twice` takes a function as argument and applies it to the second argument two times. How can this function be used?

```
Hugs> twice (*2) 3
Hugs> twice reverse [1, 2, 3]
```

Formally speaking, a function that takes a function as an argument or returns a function as a result is called ***higher-order***. In practice, however, because the term **curried** already exists for returning functions as results, the term higher-order is often just used for taking functions as arguments.

Curried Functions / Currying

Recall the lab where we covered the sections. Sections are the operators that are supplied with one argument instead of two. As a result they return a unary function waiting for just one more argument.

```
Hugs> (2+) 3
5
Hugs> (*4) 5
20
```

Please check the type signature of `(+)` and `(2+)` for reconfirmation. In Haskell all functions can be curried. A function that is curried is also called **Partially Applied** function.

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

`multThree` is a function accepting three arguments. Can this function be partially applied or curried? If yes, how?

Exercise

- Write a function `inc` which takes an `Int` as input and increment it by `1`.
- Write a function `incList` which takes a List of `Int` as input and increments every element of that list.
- Write a function `multByTwo` that multiplies every element of a list by 2.
- Can you find a similarity between the logics of `incList` and `multByTwo`?
- How these two functions can be implemented in imperative languages e.g. Java or C++?
- Can you find some pattern between these two functions? Please describe.

Higher Order Function: Map

`map` takes a function and a list and applies that function to every element in the list, producing a new list.

Exercise

- Write the definition of the higher order function map?

  Hint: *Try to extract it from the definition of* `zipWith`
- Can you accomplish the same as `incList` and `multByTwo` using map? Write the commands.

Higher Order Function: Filter

`filter` is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so in our case, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate.

Exercise

- Write the definition of the higher order function `filter`?
- Find some scenarios in which `filter` can be used.
- Re-implement quick sort using `filter`.
- Find the largest number under 100,000 that's divisible by 3829.
- Find the sum of all odd squares that are smaller than 10,000