CS224 – Lab 1

Purpose: We will look at programming language features that are common in functional languages and in Haskell in particular. All functional languages allow functions to return other functions as values, and allow functions to be passed in as parameters. Additionally, we can use partial application of the parameters of a function, which results in a new function.
Haskell also uses lazy rather than eager evaluation of expressions, and we will see that this gives us the benefit of "infinite" data structures.

Knowledge: This lab will help you become familiar with the following content knowledge:

- How to use partial application due to currying
- How to write higher-order functions
- How to use list comprehensions
- How to use lazy evaluation to write infinite data structures

Task: Follow the steps in this lab carefully to complete the assignments. Copy the lab1 folder from my account by using the command:

```
cp -r ~jillz/cs224/lab1 .
```

Note: The period at the end of the command indicates that the destination for the copy command is the current directory.

Take a look at the function `plus` in the file `Example1.hs` file. We will now consider why the type for a function like `plus` listed below is `Int -> Int -> Int` rather than something like `(Int,Int) ->Int`.

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Well the expression `plus 3 4` is really equivalent to `((plus 3) 4)`. This means that the result of `(plus 3)` is applied to the argument 4. It must be the case the value `(plus 3)` is a function!
indeed, we could define a new function to be the result of `plus 3` as follows:

```
plusThree :: Int -> Int
plusThree = plus 3
```

Verifiy that you get the result you expect from the expression `plusThree 4`.

This method of applying functions to one argument at a time is called currying (after Haskell B. Curry). Curried functions can be applied to one argument only, giving another function. Sometimes these new functions can be useful in their own right. Consider the following function:

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

The function `twice` takes as arguments a function and an integer and applies the function twice to the integer argument. We could use the function resulting in using only the first argument to get the following new functions:

```
add2 = twice (+1)

quad = twice square
```

What would be the result of the expressions `add2 3` and `quad 2` ? Try them out.

The function `twice` is an example of a higher-order function. Higher-order functions take functions as parameters and can also return functions. You should already be familiar with the higher-order functions `map` and `filter`. The function `map` takes a function and a list and applies the function to every item in the list. The function `filter` takes a predicate and a list and returns a list of all the items that satisfy the predicate.

We can use `filter` to define the function `quickSort` to sort a list. The idea behind quickSort is to pick an element x in the list and divide the list into three parts, all the items less than x, all the items equal to x, and all the items greater than x. We recursively sort the first and third parts and concatenate them all together.

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x : xs) = (quickSort less) ++ (x : equal) ++ (quickSort more) where
         less = filter (< x) xs
         equal = filter (== x) xs
         more = filter (> x) xs
```

We can define

```
dictionary = ["I", "have", "a", "thing", "for", "Haskell"]
```

What happens when we sort this list with quickSort? The reason is that the typical ordering of characters specifies that upper case characters are "less than" lower case characters.

To make things more flexible we will redefine quickSort to take an extra argument which will be a comparison function that compares two values ans returns values LT, EQ, and GT (representing "less than", "equal" and "greater than").

```
quickSort' :: (a -> a -> Ordering) -> [a] -> [a]
quickSort' _ [] = []
quickSort' c (x : xs) = (quickSort' c less) ++ (x : equal) ++ (quickSort' c more) where
        less = filter (\y-> y `c` x == LT) xs
        equal = filter (\y -> y `c` x == EQ) xs
        more = filter (\y -> y `c` x == GT) xs
```

There is a standard Haskell function `compare` which is the usual comparison function. Try it out with

```
quickSort' compare dictionary
```

Now we can write other comparison functions. For example,

```
descending x y = compare y x
```

We can use partial application due to currying to define a new function:

```
sortDescending :: Ord a => [a] -> [a]
sortDescending = quickSort' descending
```

Try it out.

Assignment 1:
Write a comparison function `insensitive` which compares Strings but is case-insensitive. You will want to use `map` and the function `toLower`.

Criteria for Success: Your function should work as follows:

```
 > quickSort' insensitive dictionary

["a","for","Haskell","have","I","thing"]
```

Another very useful higher-order function which is already in the standard library is `zipWith` that takes a function and two lists as parameters and then joins the two lists by applying the function.

For example,

```
  > zipWith (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]

 > zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
```

Assignment 2:
I have hidden `zipWith` inside the Example1 module. Define this function yourself.

```
zipWith :: (a -> b-> c) -> [a] -> [b] -> [c]
```

Criteria for Success: Your function should behave identically as the standard library function as shown in the previous examples.

Haskell contains a shorthand for filtering a list called a list comprehension. This shorthand looks very much like set notation. For example, consider the following which computes the list of even numbers from 0 to 100 by using a filter.

```
s1 = filter (\x-> x `mod` 2 == 0) [0..100]
```

We can perform the exact same computation this way:

```
s2 = [x | x<-[0..100], x `mod` 2 == 0]
```

Assignment 3:
Write a function factors n which computes a list of all the factors of n.
Hint: The factors of n are all between 1 and n `div` 2 and of course evenly divide n.

```
factors :: Integer -> [Integer]
```

Criteria for Success: The factors of 12 should give [1,2,3,4,6] and the factors of 15 are [1,3,5].

We will experiment a bit with lazy evaluation. First consider the function `f` defined in Example2. This function has an argument x which is never used in the body of the function. Try evaluating:

```
f (1/0)
```

If the language was strict (uses eager evaluation) what result would you expect?

Lazy evaluation gives us the benefit of allowing "infinite" data structures. Look at the definition of `intsFrom`. This recursion is infinite, yet the following expression which takes the first 10 items from this list will halt, due to lazy evaluation. Only the portion of the list that is used will be computed. Try it out.

```
take 10 (intsFrom 1)
```

Look at the definitions of `fibs` and `primes` and see if you can figure out how they work. They both give an infinite list!

Assignment 4:
Write the definition of the infinite list that computes all the powers of 2.
I did this function using multiplication and `map` since each successive value in the list is two times the value of the previous one.

Criteria for Success: Your function should take no parameters and should return the infinite list. You can test your result by using the `take` function.

Assignment 5:
Using the data type Tree, write a function infTree which creates an infinite tree where all the nodes contain the value given by the parameter.

```
infTree :: a -> Tree a
```

Criteria for Success: Wait for the next assignment to test out your tree.

Assignment 6:
Write the function `takeTree` which returns n levels of a Tree. Then use this function to print out a portion of an infTree.

```
takeTree :: Integer -> Tree a -> Tree a
```

Criteria for Success: You print out the appropriate number of levels of your `infTree`.

Submit your files in Canvas for grading.