Tutorial 4

James and Aaron

August 5, 2016

1 Assignment Comments

A possible answer for Assignment 1 Question 1 using pattern matching.

```
1 count :: Eq a => a -> [a] -> Int
2 count _ [] = 0
3 count x (y:ys) | x == y    = 1 + count x ys
4                | otherwise = count x ys
```

Note the type is as general as possible for the inputs, thus allowing for it to be used on strings etc.

Two alternative (and less preferred) implementations.

```
1 count x ls = if null ls then 0
2              else if x == (head ls) then 1 + count x (tail ls)
3              else count x (tail ls)
4
5 count x ls | null ls        = 0
6            | x == (head ls) = 1 + count x (tail ls)
7            | otherwise      =  count x (tail ls)
```

1.a Types

If you leave off the signature, you can see what the derived type is. Note that this form also has return type in general terms, which sometimes having a specific return type like Int will be more descriptive of what you are returning.

```
*Main> :type count
count :: (Num t, Eq t1) => t1 -> [t1] -> t
```

Note sometimes Haskell will have a problem working out the intended type for certain expressions. To resolve the issue you can specify the type using the syntax `expr`:: Type .

1.b Additional comments

allPos was both 1 indexed and specified to be ordered in ascending order.

It is useful to include the examples given in the handout in your tests.

2 Evaluation

In the console we can define new functions using let. Here are three difference ways, note last is implicit as that is how the console works. The first way (let funcDecl in expr) is how you would use it normally in a file.

```
*Main> let strings = ["Hello", "how", "are", "you", "my", "mate"] in strings
["Hello","how","are","you","my","mate"]
*Main> let strings = ["Hello", "how", "are", "you", "my", "mate"]
*Main> strings = ["Hello", "how", "are", "you", "my", "mate"]
```

We can use the list returned by strings in the following expressions.

```
*Main> map (\s -> length s > 4) strings
[True,False,False,False,False,False]

*Main> or (map (\s -> length s > 4) strings)
True
```

When eager evaulation is used, the last expression would work by returning the list of strings, mapping each item across to a new list, and then check that list to see whether any elements were true. This seems fairly inefficient, partically in this case where the first item in the list is true.

As Haskell uses lazy evaluation, it works instead by checking the list that will be returned by map whether the first item is true. For that to happen, map than tries mapping the first item in the list of strings using it's function to a value to provide to or. In this case that is where it would stop, having only mapped one string to a boolean, as that value is true. Had the value not been true, it would have now tried to resolve the second item, and so on until it got a true value or reached the end of the list.

Though that is more wordy than my description of how it works for eager, it is in fact more efficient as only the first item of the list is evaluated (by both or and map).

Lazy evaulation also allows maps to operate on infinite lists. For example the following works with lazy evaulation but not eager.

```
*Main> or (map (<5) [1..])
True
```

Note lazy evaulation is not magic, so the following never terminates as to compute the length you must traverse the entire list even when you are checking whether the result is 0.

```
*Main> let l = [1..]
*Main> if length l == 0 then "empty" else "not empty"
"Interrupted.
```

Instead using pattern matching means it only tries to see whether the list is the empty, so does terminate.

```
*Main> if null l then "empty" else "not empty"
"not empty"
```

3 Algebraic Data Types

We can define our own sequence type that has a constructor `Nil` representing an empty sequence and `Cons` for a sequence of a single value followed by another sequence.

```
1 data Seq a = Nil | Cons a (Seq a)
```

We can use those constuctors to pattern match on parameters of that type.

```
1 seqHead (Cons y ys) = y
2
3 scount _ Nil = 0
4 scount x (Cons y ys) | x == y     = 1 + scount x ys
5                      | otherwise = scount x ys
```

Here is an example of using the above functions. Note it is important to wrap subexpressions in brackets when they are not a single term.

```
*Main> scount 3 Nil
0
*Main> scount 1 (Cons 2 (Cons 1 (Cons 1 Nil)))
2
```

Normal lists are an algebraic data type, with constructors `[]` and `:` . All the list functions use pattern matching to work.

```
1 data [a] = [] | a:[a]
2
3 null [] = True
4 null _ = False
```

Note that `[1,2,3,4]` is just syntatic sugar for using the constructors.

```
*Main> [1,2,3,4]
[1,2,3,4]
*Main> 1:2:3:4:[]
[1,2,3,4]
```

We can define a binary tree (which always has a root node, unlike the example on the slides).

```
1 data BinTree a = Leaf a | Node a (BinTree a) (BinTree a)
```

Some functions to operate on it.

```haskell
-- Works out the depth (Root is depth 0)
depth (Leaf _) = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)

-- Create a tree
tree = (Node 2
            (Leaf 1)
            (Node 3
                (Leaf 4)
                (Leaf 5)))
```

Used:

```
*Main> depth (Leaf 1)
0
*Main> depth (Node 2 (Leaf 1) (Node 3 (Leaf 4) (Leaf 5)))
2
*Main> tree
<interactive>:42:1: error:
    * No instance for (Show (BinTree Integer))
        arising from a use of ‘print’
    * In a stmt of an interactive GHCi command: print it
 *Main> tree == tree
<interactive>:47:1: error:
    * No instance for (Eq (BinTree Integer)) arising from a use of ‘==’
    * In the expression: tree == tree
      In an equation for ‘it’: it = tree == tree
```

Note when the console tried to display the tree it failed as there was no way defined to show it. Similarly we could not check whether the tree equaled itself.

We could define instances of type classes (Possibly covered next lecture) to implement functions that resolve those issues, or we could simply tell Haskell to derive those functions for us using the following syntax.

```haskell
data BinTree a = Leaf a | Node a (BinTree a) (BinTree a)
    deriving (Show, Eq)
```

Now when we try to print or check equality it will work.

```
*Main> tree
Node 2 (Leaf 1) (Node 3 (Leaf 4) (Leaf 5))
*Main> tree == tree
True
```