COM2108: Functional Programming
Review, week 3

Emma Norling

List Comprehensions

```
[x^2 | x <- [1..5]]
```

- “Generate the values x squared such that x is drawn from the list of values 1 to 5”

```
[1,4,9,16,25]
```

Multiple Generators

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

- “Generate values of (x,y) such that x is drawn from the list 1 to 3 and y is drawn from the list x to 3”

```
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Adding filters

```
pythagorean :: Int -> [(Int, Int, Int)]
pythagorean n = [ (x, y, z) |
                  x <- [2 .. n],
                  y <- [x+1 .. n],
                  z <- [y+1 .. n],
                  x*x + y*y == z*z ]
```

The General Form

$$[\ e\ |\ q_1, \ldots, q_n\ ]$$

Each qualifier (q_n) can take one of two forms:

- A generator, of the form
  `x <- list`
  where the list itself can be an expression

- a **guard** which is a Boolean expression, to filter things

List Comprehensions and Strings

- There is nothing special! Strings are just lists of characters.

General approach to recursion

1. Start with the type definition
2. Next, what will the stopping condition (base case) be?
3. Finally, what is the recursive case?

If you’re working on lists…

- Base case is usually the empty list
- But sometimes you also have something for a singleton list
- General case does something with the first item in the list

Example from the exercises

```
replace :: String -> String -> String -> String

replace orig new [] = []

replace orig new (x:xs)
    | orig == prefix = new ++ replace orig new rest
    | otherwise      = x : replace orig new xs
    where (prefix,rest) = splitAt (length orig) (x:xs)
```

Now to the quiz…

Accumulator recursion

- Sometimes, when you write a recursive function, the easiest way to do it is to use an *accumulator*
  - A new function parameter is introduced to accumulate the answer
  - An alternative way of counting the occurrences of an item in a list:

```
count item list
   = count' item list 0
     where
     count' _ [] n = n
     count' item (x:xs) n
        | item == x = count' item xs (n+1)
        | otherwise = count' item xs n
```