Haskell – Lab 4

Prepared by Dr. Wooi Ping Cheah

Solution for the Exercises from Chapter 5 – List Comprehensions

1.

```
pyths :: Int -> [(Int,Int,Int)]
pyths n = [(x,y,z) | x <- r, y <- r, z <- r, x^2 + y^2 == z^2]
   where r = [1..n]

pyths2 :: Int -> [(Int,Int,Int)]
pyths2 n = [(x,y,z) | x <- r, y <- r, z <- r, x <= y, x^2 + y^2 == z^2]
   where r = [1..n]

pyths2' :: Int -> [(Int,Int,Int)]
pyths2' n = [(x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n], x <= y, x^2 + y^2 == z^2]
```

2.

```
factor :: Int -> [Int]
factor n = [x | x <- [1..n], n `mod` x == 0]

perfects :: Int -> [Int]
perfects n = [x | x <- [1..n], x == sum [y | y <- [1..(x-1)], x `mod` y == 0]]

perfects' :: Int -> [Int]
perfects' n = [x | x <- [1..n], x == sum (factor x) - x]
```

3.

```
scalar :: [Int] -> [Int] -> Int
scalar xs ys = sum [x*y | (x,y) <- zip xs ys]

scalar' :: Num a => [a] -> [a] -> a
scalar' xs ys = sum [x*y | (x,y) <- zip xs ys]
```

Solution for the Exercises from Chapter 6 – Recursive Functions

1.

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss

replicate :: Int -> a -> [a]
replicate 0 x = []
replicate n x = x : (replicate (n-1) x)

(!!) :: [a] -> Int -> a
(x:xs) !! 0 = x
(x:xs) !! n = xs !! (n-1)
```

```
elem :: Eq a => a -> [a] -> Bool
elem n xs | null xs      = False
          | n == head xs = True
          | otherwise    = elem n (tail xs)

elem' :: Eq a => a -> [a] -> Bool
elem' x [] = False
elem' x (y:xs) | x == y = True
               | otherwise = elem x xs
```

2.

```
merge :: Ord a => [a] -> [a] -> [a]
merge []     ys     = ys
merge xs      []    = xs
merge (x:xs) (y:ys)
  | x < y     = x : (merge xs (y:ys))
  | otherwise = y : (merge (x:xs) ys)
```

3.

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge (msort (take n xs)) (msort (drop n xs))
   where n = (length xs) `div` 2
```

Exercises
from
Chapter 7 – Higher-Order Functions

(1) What are higher-order functions that return functions as results better known as?

Hints:

You can find the answer to this question from the following videoclip:

Haskell for Imperative Programmers #7 - Partial Function Application & Currying

https://www.youtube.com/watch?v=m12c99qgHBU

(2) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

Hints:

This is a simple one line program

(3) Redefine map f and filter p using foldr.

Define map f using foldr

```
map :: (a -> b) -> [a] -> [b]
map f ys = foldr (  ) [] ys
```

This is a lambda function that accumulates f x, where x is the head of the list ys

Define filter p using foldr

```
filter :: (a -> Bool) -> [a] -> [b]
filter p ys = foldr (\  ) [] ys
```

This is a lambda function that accumulates x if p x is true, where x is the head of the list ys

Exercises
from
Chapter 8 – Declaring Types and Classes

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

Hints:

You are given the type declaration for Nat, and function definition for add':

```
data Nat = Zero | Succ Nat
        deriving Show

add' Zero n = n
add' (Succ m) n = Succ (add' m n)
```

Write a function that multiplies two natural numbers.

You can call this function as mult', which will use the add' function, and is quite similar in pattern to add' function.

```
ghci> mult' (Succ(Succ Zero)) (Succ(Succ(Succ Zero)))
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
```

(2) Define a suitable function folde for expressions, and give a few examples of its use.

Hints:

- You may like to write a **folde** function for the **eval** and **size** functions on Page 19 of Chapter 8.
- **eval** – a function to evaluate an arithmetic expression.
- **size** – a function to compute the size of an arithmetic expression (i.e. the number integers in the expression).
- The application of the **folde** function to find the size of an arithmetic expression, and to evaluate it is shown below:

```
ghci>
ghci> folde (\_ -> 1) (+) (+) (Add (Val 1) (Mul (Val 2) (Val 3)))
3
ghci>
ghci> folde id (+) (*) (Add (Val 1) (Mul (Val 2) (Val 3)))
7
```

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr

folde :: (Int -> a) -> (a -> a -> a) -> (a -> a -> a) -> Expr -> a

folde id _ _ (Val x) = id x

folde id add mul (Add x y) = add (folde id add mul x) (
```

```
)

folde id add mul (Mul x y) = mul (
```

```
) (folde id add mul y)
```

References
Functional Programming & Haskell

foldr explained

Definition for foldr with a simple example

https://www.youtube.com/watch?v=Dmd6Q2i7gdw

Foldr

Application of foldr with simple examples

https://www.youtube.com/watch?v=cyGtltWKWQg