Haskell – Lab 5

Prepared by Dr. Wooi Ping Cheah

Solution for the Exercises from Chapter 7 – Higher-Order Functions

1. Partial functions

2. map f (filter p xs)

```
ghci>
ghci> func1 f p xs = [f x | x <- xs, p x]
ghci> func1 (+1) even [1,2,3,4,5]
[3,5]
ghci> func2 f p xs = map f (filter p xs)
ghci> func2 (+1) even [1,2,3,4,5]
[3,5]
ghci>
```

3.

map' :: (a -> b) -> [a] -> [b]

map' f ys = foldr (\x xs -> f x : xs) [] ys

filter' :: (a -> Bool) -> [a] -> [a]

filter' p ys = foldr (\x xs -> if p x then x : xs else xs) [] ys

```
ghci>
ghci> map' f ys = foldr (\x xs -> f x:xs) [] ys
ghci> map' (+1) [1,2,3,4,5]
[2,3,4,5,6]
ghci> filter' p ys = foldr (\x xs -> if p x then x:xs else xs) [] ys
ghci> filter' even [1,2,3,4,5]
[2,4]
ghci>
```

Solution for the Exercises from Chapter 8 – Declaring Types and Classes

1.

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

add' :: Nat -> Nat -> Nat
add' Zero n = n
add' (Succ m) n = Succ (add' m n)

mult' :: Nat -> Nat -> Nat
mult' Zero     n = Zero
mult' (Succ m) n = add' n (mult' m n)
```

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

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

```
ghci>
ghci> folde id (+) (*) (Add (Val 1) (Mul (Val 2) (Val 3)))
7
ghci> folde id (+) (*) (Mul (Val 1) (Add (Val 2) (Val 3)))
5
ghci> folde id (+) (*) (Add (Val 2) (Val 3))
5
ghci> folde id (+) (*) (Mul (Val 2) (Val 3))
6
ghci> folde id (+) (*) (Val 2)
2
ghci> folde id (*) (+) (Val 2)
2
ghci>
```

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

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

Exercises from Chapter 10 – Interactive Programming

Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

- Two players take it turn about to remove one or more stars from the end of a single row.

- The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is [5,4,3,2,1].

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\User> cd Desktop\Haskell
PS C:\Users\User\Desktop\Haskell>
PS C:\Users\User\Desktop\Haskell> ghci
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci>
ghci> :load nim.hs
[1 of 2] Compiling Main             ( nim.hs, interpreted )
Ok, one module loaded.
ghci>
ghci> playNim
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *

Player 1 turn:
Which line? 1
How many stars to remove? 5
1:
2: * * * *
3: * * *
4: * *
5: *

Player 2 turn:
Which line? 2
How many stars to remove? 4
1:
2:
3: * * *
4: * *
5: *
```

Windows PowerShell

```
Player 1 turn:
Which line? 3
How many stars to remove? 3
1:
2:
3:
4: * *
5: *

Player 2 turn:
Which line? 4
How many stars to remove? 2
1:
2:
3:
4:
5: *

Player 1 turn:
Which line? 5
How many stars to remove? 1
Player 1 wins!
ghci>
```

You can run the attached **nim.exe** (application) by typing **./nim** directly on the terminal, as follows: `PS C:\Users\User\Desktop\Haskell> ./nim`

You can start writing the program from the code skeleton called **nim_framework.hs**.

The **hangman.hs** program discussed in the lecture is also a good reference.

Exercises
from
Chapter 15 – Lazy Evaluation

1. Define a program

```
fibs :: [Integer]
```

that generates the infinite Fibonacci sequence

```
[0,1,1,2,3,5,8,13,21,34…
```

Using the following simple procedure:

a) The first two numbers are 0 and 1.
b) The next is the sum of the previous two.
c) Return to step b)

```
ghci>
ghci> fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,19
6418,317811,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986,1023341
55,165580141,267914296,433494437,701408733,1134903170,1836311903,2971215073,4807526976,7778742049,12586269025
,20365011074,32951280099,53316291173,86267571272,139583862445,225851433717,365435296162,591286729879,95672202
```

```
fibs :: [Integer]
fibs = 0:fibs2 0 1
  where
    fibs2 :: Integer -> Integer -> [Integer]
    fibs2 a b = b:
```

Recursion

2. Define a function

```
fib :: Integer -> Integer
```

that calculates the nth Fibonnaci number.

```
ghci>
ghci> fib 7
8
ghci> fib 9
21
ghci> fib 10
34
ghci>
```

```
fib :: Integer -> Integer
fib n = fibs !! (                    )
```

nth Fibonnaci number