# Functional Programming

**Home Assignment 2**

Due: 9 Apr 2022 - 23:59

## Instructions

- Please create a source file called ***hw2.hs*** and put all the answers there.
  The file should start with a comment, which contains your **full name** (in English) and **ID**

  ```
  -- Montgomery Burns
  -- 15926535
  ```
- **Important:** Please add the following line after the two comments:

  ```
  module HW2 where
  ```

  This line helps us to test your code.

- When writing a function - write both the **type** and the **body** of the function.
- Be sure to write functions with **exactly the specified name** and **type signature** for each exercise. You may create additional auxiliary/helper functions with whatever names and type signatures you wish.
- Try to write **small functions**, which perform just **a single task**, and then **combine** them to create more complex functions.

## Exercises

1. Write a function **sieve**, which takes an integer n and returns a list of length n of (Int,Bool) values. The Int part of the tuple will be a running index (starting with 1). The Bool part will be True if the index is a prime number.

```
sieve 8 should return
[(1,False),(2,True),(3,True),(4,False),(5,True),(6,False),(7,True),(8,False)]
```

Please note: You should use the Sieve of Eratosthenes to solve this question.

2. As you know, it's impossible to create a list like the following in Haskell:

```
[1, 4, 2, [3,4]]
```

However, we can define a new type to allow us to mimic this structure:

```
data IntList = Single Int | Multi [IntList]  deriving Show
```

2.a

Write a function, sum' which takes a variable of type IntList and returns the sum of all the numbers in it.

```
sum' Multi ([ Single 1, Single 4 , Single 2, Multi [ Single 3, Single 4 ]] should) return 14
```

2.b

Write a function, flatten which takes a variable of type IntList and returns a list of the Int values in it

```
flatten (Multi [ Single 1, Single 4 , Single 2, Multi [ Single 3, Single 4 ]]) should return [1,4,2,3,4]
```

3. Define the following type for binary trees:

```
data BinTree = Empty | Leaf Int |
                Node BinTree Int BinTree
```

3.a Write a function make_balanced_tree, which takes an **ordered** list of Int values and creates a balanced search tree having the same values.

```
t = [1,3,5,6,7]
```

make_balanced_tree t should return

```
Node (Node (Leaf 1) 3 Empty) 5 (Node (Leaf 6) 7 Empty)
```

3.b Write a function add_item which adds an Int value to a balanced search tree. The result should be a new tree that is also balanced.

4. Write a function split_by_either which takes two parameters:

- A list of items of type t
- A function (call it f) that takes a t and returns Either t t

The function should return a pair (tuple of length 2) of lists of type t. The items in these lists will be the items from the original list. The first list will contain the items which are sent by f to Left t. The other list will contain the items sent to Right t.

Example

```
f :: String -> Either String String

f item = if (odd (length item)) then Left item else Right item

split_by_either [ "It" , "was", "a" , "many", "and", "many" , "years", "ago" ] f

should return
(["was","a","and","years","ago"],["It","many","many"])
```

5. Define the type ExprTree as follows:

```
type BinOp = Float -> Float -> Maybe Float

data ExprTree =  ExprValue Float | ExprNode ExprTree BinOp
ExprTree
```

It defines a tree that represents an algebraic expression.

Write a function eval, which takes an ExprTree and evaluates its value. The function first evaluates the left and right sub-trees and then applies the operation.

Example:

```
addOp :: Float -> Float -> Maybe Float

addOp a b = Just (a+b)

divOp :: Float -> Float -> Maybe Float

divOp a b = if ( b == 0 ) then Nothing else Just (a/b)


sample_tree1 = ExprNode ( ExprValue 3  ) divOp
     ( ExprNode ( ExprValue 2 ) addOp ( ExprValue (4) ) )


eval sample_tree1 should return (Just 0.5)
```