Homework 4 Due April 19th 11:59 pm

Submission rules:

- All text answers must be given in Haskell comment underneath the problem header.
- You must submit a single .hs file with the following name: firstName-lastName-hw4.hs. Failure to do so will result in -10 points.
- You will lose 10 points if you put a module statement at the top of the file.
- You will lose 10 points for any import statements you have in your file and will automatically miss any problems you used an imported function on.
- If your file doesn't compile you will lose 10 points and miss any problems that were causing the compilation errors.
- This means that any function which is causing compiler errors should be commented out. There will be no partial credit.
- You must use the skeleton file provided and must not alter any type signature. If you alter a type signature you will automatically miss that problem.
- You will lose 10 points if you include a *main* function in your file.

Problems

Problem 1 (Exercise 8.1) (5 pts)

Given the following definitions:

```
data Nat = Zero | Succ Nat
    deriving (Eq, Show)

nat2Int :: Nat -> Int
nat2Int Zero     = 0
nat2Int (Succ n) = 1 + nat2Int n

int2Nat :: Int -> Nat
int2Nat 0 = Zero
int2Nat n = Succ (int2Nat (n - 1))

add :: Nat -> Nat -> Nat
add Zero n2      = n2
add (Succ n1) n2 = Succ (add n1 n2)
```

In a similar manner to the function *add*, define a recursive multiplication function *mult* for the recursive type of natural numbers:

```
mult :: Nat -> Nat -> Nat
```

Problem 2 (Exercise 8.3) (5 pts)

Consider the following type of binary trees:

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

Let us say that a tree is balanced if the number of leaves in the left and right subtree of every node differ by at most one, with leaves themselves being trivially balanced. Define a function *balanced* that decides if a binary tree is balanced or not. Hint, first define a function that calculates the number of leaves in a tree.

```
balanced :: Tree a -> Bool
```

Problem 3 (Exercise 8.4) (5 pts)

Define a function *balance* that converts a non-empty list of integers into a balanced tree. Hint, first define a function that splits a list into two halves that differ by at most 1.

```
balance :: [a] -> Tree a
```

Problem 4 (Exercise 8.5) (10 pts)

Given the data declaration:

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

define a higher-order function *folde* that replaces each *Val* constructor in an expression with the function *f :: Int -> a*, and each *Add* constructor by the function *g :: a -> a -> a*, in the given *Expr*. For example:

```
ghci> folde id (+) (Add (Val 1) (Add (Val 2) (Val 3)))
6
ghci> folde (+1) (*) (Add (Add (Val 1) (Val 2)) (Val 3))
24
ghci> folde (\x -> [x]) (++) (Add (Val 1) (Add (Val 2) (Val 3)))
[1,2,3]
ghci> folde (\x -> (1, x)) (\(l1, s1) (l2, s2) -> (l1 + l2, s1 + s2)) (Add (Val 1) (Add (Val
(3,6)

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

Problem 5 (Exercise 8.6) (5 pts)

Using *folde*, define a function *eval* that evaluates an expression to an integer value, and a function *size*, that calculates the number of values in an expression. For example:

```
ghci> eval (Add (Add (Val 1) (Val 2)) (Val 3))
6
ghci> size (Add (Add (Val 1) (Val 2)) (Val 3))
3

eval :: Expr -> Int

size :: Expr -> Int
```

Problem 6 (10 pts)

Addition and multiplication for complex numbers is defined as follows:

$(x + yi) + (u + vi) = (x + u) + (y + v)i$

$(x + yi) * (u + vi) = (xu - yv) + (xv + yu)i$

A *complex integer*, is a complex number with integer real and imaginary parts. Define a data type for complex integers called *ComplexInteger* with selector function *real* and *imaginary* which return the real and imaginary parts respectively. Then give minimum instance declarations for *Eq*, *Show*, and *Num* typeclasses. For *Num* you only need to define *(+)* and *(*)*. For example:

```
ghci> real (ComplexInteger 1 2)
1
ghci> imaginary (ComplexInteger 1 2)
2
ghci> (ComplexInteger 1 2) == (ComplexInteger 3 4)
False
ghci> ComplexInteger 1 2
1+2i
ghci> (ComplexInteger 1 2) * (ComplexInteger 3 4)
-5+10i
```

Problem 7 (5 pts)

Define a function, *chopN*, which takes a list and splits into a list of list where each inner is list exactly length *n*. If if it cannot be divided evenly then the last inner list should be dropped. You can assume *n* will be >= 1. Bonus 10 points if you define this using *foldr*. For example:

```
ghci> chopN 4 [1..10]
[[1,2,3,4],[5,6,7,8]]
ghci> chopN 8 [1..10]
[[1,2,3,4,5,6,7,8]]
ghci> chopN 1 [1..10]
[[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]]
ghci> chopN 2 [1..10]
[[1,2],[3,4],[5,6],[7,8],[9,10]]
```

```
chopN :: Int -> [a] -> [[a]]
```

Problem 8 (10 pts)

Define a function, *subAlphabet*, which creates a *substitution* ciphertext alphabet which is shifted by specific letters being at the beginning of the alphabet and then the rest of the alphabet, in order, following that. For example if we put "*ZEBRAS*" at the beginning of the alphabet:

```
Plaintext alphabet:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ciphertext alphabet: ZEBRASCDFGHIJKLMNOPQTUVWXY

ghci> subAlphabet 'A' 'Z' "ZEBRAS"
"ZEBRASCDFGHIJKLMNOPQTUVWXY"

ghci> subAlphabet 1 26 [1,4,6,2,9,10,23,17]
[1,4,6,2,9,10,23,17,3,5,7,8,11,12,13,14,15,16,18,19,20,21,22,24,25,26]
```

```
subAlphabet :: (Eq a, Enum a) => a -> a -> [a] -> [a]
```

Problem 9 (20 pts)

Given the following data type:

```
data Polynomial = Constant Int | MoreTerms Int Polynomial
```

For example the following *Polynomial*:

```
p = MoreTerms 3 (MoreTerms 4 (Constant 5))
q = MoreTerms 6 (MoreTerms 10 (MoreTerms 35 (Constant 7)))
```

Represents the polynomial $3 + 4x + 5x^2$.

You must do the following:

- Instance *Polynomial* into the *Show* typeclass. For example

  ```
  ghci> p
  3 + 4x + 5x^2
  ghci> q
  6 + 10x + 35x^2 + 7x^3
  ```

- Instance *Polynomial* into the *Num* typeclass, using usual polynomial addition and multiplication. You do not need to define any of the other functions *Num* asks for.

- Write a function, *evalPoly*, that evaluates a given *Polynomial* at a given integer. For example:

  ```
  ghci> evalPoly p 2
  33
  ```

```
evalPoly :: Polynomial -> Int -> Int
```

Problem 10 (10 pts)

Given the following data type:

```
data Pair a b = Pair a b
```

You must do the following:

- Instance *Pair* into *Eq* where both parts of the *Pair* are *Eq*. Your instance should deem two *Pair*s equal if the first elements are equal and the second elements are equal. Your instance header should look like the following:

  ```
  instance (Eq a, Eq b) => Eq (Pair a b) where
  ```

- Instance *Pair* into *Ord* where both parts of the *Pair* are *Ord*. Your instance should follow *lexicographic* ordering which is to say: if one *Pair* has a greater first element, then it is greater; but if the *Pair*s have the same first element then second element is used to determine which is greater. Your instance header should look like the following:

  ```
  instance (Ord a, Ord b) => Ord (Pair a b) where
  ```

For example:

```
ghci> Pair 1 2 == Pair 1 2
True
ghci> Pair 1 2 == Pair 2 1
False
ghci> Pair 1 2 < Pair 2 1
True
ghci> Pair 1 2 > Pair 1 1
True
ghci> Pair 1 2 > Pair 1 3
False
```

Problem 11 (15 pts)

Given the following function which performs "safe" division on floating-point numbers:

```
safeDivide :: Float -> Float -> Maybe Float
safeDivide x y = if y == 0 then Nothing else Just (x / y)
```

You must do the following:

- Define a function, *safeDivide'*, which extends *safeDivide* to a function with the following type signature:

  ```
  safeDivide' :: Maybe Float -> Maybe Float -> Maybe Float
  ```

- Define a function, *hm*, which computes the *harmonic mean* of a given *non-empty* list of *Float*s *if it exists*. This function has the following type signature:

  ```
  hm :: [Float] -> Maybe Float
  ```

- The *harmonic mean* of the numbers $x_1, x_2, \ldots, x_k$ is defined as follows:

$$\frac{k}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_k}}$$

For example:

```
ghci> hm [2.0, 2.0]
Just 2.0
ghci> hm [1.0, 1.0]
Just 1.0
ghci> hm [0.5, 0.5, 1.0]
Just 0.6
ghci> hm [1.0, -1.0]
Nothing
ghci> hm [1.0, -2.0, -2.0]
Nothing
ghci> hm [1.0, -2.0]
Just 4.0
```