# Homework 4 Due April 13th 11:59 pm

Submission rules:

- 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.
- 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.

## Problems

### Problem 1 (10 pts)

Using the following datatype:

```
data Tree a = LeafT a | NodeT (Tree a) (Tree a) deriving (Show, Eq)
```

Define the following function:

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

*balance* takes a **non-empty** list and converts it into a balanced tree. *Hint*: Define a function which splits a list into two halves whose length differs by at most 1.

For example:

```
balance [1,2] ==> NodeT (LeafT 1) (LeafT 2)
balance [1,2,3] ==> NodeT (LeafT 1) (NodeT (LeafT 2) (LeafT 3))
balance [1,2,3,4] ==> NodeT (NodeT (LeafT 1) (LeafT 2)) (NodeT (LeafT 3) (LeafT 4))
```

### Problem 2 (15 pts)

Using the following definition of a binary tree:

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

And the following datatype that represents a traversal of binary tree:

```
data P = GoLeft P | GoRight P | This deriving (Eq, Show)
```

Where *This* represents the entire tree. Now define the following function:

```
allpaths :: T -> [P]
```

Which given a $T$ outputs all possible paths, $P$, from the root of the given tree to each of its subtrees.

For example:

```
allPaths (Node Leaf (Node Leaf Leaf))
```

Evaluates to:

```
[This,GoLeft This,GoRight This,GoRight (GoLeft This),GoRight (GoRight This)]
```

Ordering doesn't matter.

## Problem 3 (15 pts)

Given the following type declaration:

```
data Expr = Val Int | Add Expr Expr deriving (Eq, Show)
```

Define the following function:

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

*folde f g* replaces each *Val* constructor in an expression with the function $f$ and each *Add* constructor by the function $g$.

Then use *folde* to define the following function:

```
eval :: Expr -> Int
```

*eval* evaluations an expression to an integer value.

For example:

```
eval (Add (Val 1) (Val 2)) ==> 3
eval (Add (Add (Val 1) (Val 2)) (Val 3)) ==> 6
```

## Problem 4 (5 pts)

Define the following function:

```
myTakeWhile :: (a -> Bool) -> [a] -> [a]
```

*myTakeWhile* returns the prefix of the list that satisfies the predicate. This function should behave in the same way as the built in function *takeWhile*.

## Problem 5 (5 pts)

Define the following function

```
mySpan :: (a -> Bool) -> [a] -> ([a], [a])
```

*mySpan* returns a pair of lists where the first part is what *myTakeWhile* would return and the second part is the rest of the list. This function should behave in the same way as the built in function *span*.

## Problem 6 (10 pts)

Define the following function

```
combinations3 :: Ord a => [a] -> [[a]]
```

*combinations3* takes a list and returns a list of length 3 lists representing all combinations (order doesn't matter) of length 3 of the input list.

For example:

```
combinations3 [] ==> []
combinations3 "ABCDE" ==> ["ABC","ABD","ABE","ACD","ACE","ADE","BCD","BCE","BDE","CDE"]
```

## Problem 7 (5 pts)

Define the following function using the *and* function and a list comprehension:

```
increasing :: Ord a => [a] -> Bool
```

*increasing* should return True *iff* the list is in increasing sorted order. Note if multiple of the same element appear in sequence this function should still return True.

For example:

```
increasing [] ==> True
increasing [1,2,3] ==> True
increasing [3,2,1] ==> False
increasing [1,2,2,3] ==> True
```

## Problem 8 (20 pts)

Define the following function:

```
combinations :: (Ord a, Integral b) => b -> [a] -> [[a]]
```

*combinations* takes an integral, *k*, and a list of Ords and returns a list of length *k* lists representing all possible combinations of length *k*. *Hint*: Do not use list comprehensions instead start by writing *combinations1* then use *map* and *combinations1* to write *combinations2*. Then use *combinations2* and *map* to write *combinations3*. Now abstract the pattern to write *combinations*.

For example:

```
combinations 3 "ABCDE" ==> ["ABC","ABD","ABE","ACD","ACE","ADE","BCD","BCE","BDE","CDE"]
combinations 2 "ABCDE" ==> ["AB","AC","AD","AE","BC","BD","BE","CD","CE","DE"]
combinations 1 "ABCDE" ==> ["A","B","C","D","E"]
```

## Problem 9 (15 pts)

Complex numbers define addition and subtraction as follows:

*(x + iy) + (u + iv) = (x + u) + (y + v)i*

*(x + iy) * (u + iv) = (xu - yv) + (xv + yu)i*

Use the following datatype to represent complex numbers:

```
data Complex = Complex { real :: Integer, imaginary :: Integer }
```

Now instance this datatype into *Eq*, *Show*, and *Num*.

For *(==)* given two complex numbers *c1* and *c2*, *(==)* should return *True* iff the real part of *c1* == the real part of *c2* and the imaginary part of *c1* == the imaginary part of *c2*.

For *show* you should return a string as above, "(<*real*> + i<*imaginary*>)"

For *(+)* and *(*)* use the definitions given above. You do **not** need to define any of the other *Num* functions.

For example:

```
Complex 1 2 ==> 1+2i
(Complex 1 2) == (Complex 1 2) ==> True
(Complex 1 2) == (Complex 3 4) ==> False
(Complex 1 2) + (Complex 3 4) ==> 4+6i
(Complex 1 2) * (Complex 3 4) ==> -5+10i
```