HW 11 **Prof. Caldwell**
Due: 15 October 2013 **COSC 3015**

We've been writing some code in class for the few meetings. It's your turn.
Here's a type of binary trees.

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

Consider the Functor type class in Haskell.

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Exercise 0.1. Make BTree an instance of the Functor type class by filling in the code for fmap below.

```
instance Functor BTree where
  fmap f Leaf  = ...
  fmap f (Node x lt rt) = ...
```

Data structures like trees can be folded just like lists - *i.e.* you can fold up a tree into a value by combining all the values in some way. Here's an example that sums the values in a tree.

```
sum Leaf = 0
sum (Node x lt rt) = x + (sum lt) + (sum rt)
```

Exercise 0.2. Generalize this function into a fold (like foldr for lists) that works on BTrees.

```
foldBTree :: (a → b → b → b) → b → BTree a → b
foldBTree f e Leaf =  ...
foldBTree f e (Node x lt rt) = ...
```

Here's come code to test you functions - see if you can figure out why they're correct (or even what it means for them to be correct.)

```
mkTree :: [a] → BTree a
balanced rt
  where height Leaf = 0
        height (Node _ lt rt) = 1 + max (height lt) (height rt)
```

Exercise 0.3. Run the test code on the web-page to test your fmap and foldBTree functions.