HW 20 **Prof. Caldwell**
Due: 15 November 2011 **COSC 3015**

We have been discussing the type classes Functor, Applicative and Monoid. You can read about Foldable in Chapter 11 of LYAHFGG.

In this assignment you must instantiate a tree datatype as an instance of the Functor type class and as an instance of the Applicative type class.

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

Mapping over a tree is straightforward so the instantiation of Tree as an instance of Functor is easy. Your implementation of Tree as an instance of the Applicative type class behave more like the *ZipList* instantiation as an instance of Applicative than the standard list instantiation does. In other words, the Applicative operator $\langle * \rangle$ (which for trees will be of type *Tree (a → b) → Tree a → Tree b*) should apply nodes pairwise. If t_1 and t_2 are trees, $t_1 \langle * \rangle t_2$ should apply the function in the root node of t_1 to the value stored in the root node of t_2 and the function stored in the root of the left subtree of t_1 should be applied to the value stored in the left subtree of t_2 and so on. Like the *zip* function, if either tree runs out of values (gets to a Leaf), the resulting value should just be *Leaf*. Here are some small example behaviors.

```
*Tree⟩ (Node Leaf (+7) Leaf) ⟨*⟩ Leaf
Leaf
*Tree⟩ Leaf ⟨*⟩ (Node Leaf 1 Leaf)
Leaf
*Tree⟩ (Node Leaf (+7) Leaf) ⟨*⟩ (Node Leaf 7 Leaf)
Node Leaf 14 Leaf
*Tree⟩ (Node Leaf (+7) Leaf) ⟨*⟩ (Node Leaf 7 (Node Leaf 12 Leaf))
Node Leaf 14 Leaf
*Tree⟩ (Node Leaf (+7) Leaf) ⟨*⟩ (Node (Node Leaf 12 Leaf) 7 Leaf)
Node Leaf 14 Leaf
*Tree⟩
```

As noted in the discussion of *ZipList* as an instance of Applicative (read LYAHFGG) this means *pure :: a → Tree a* must result in an infinite tree. So *pure x* is an infinite tree of nodes containing the value x. HINT: Lipovača used *repeat* in his definition of *pure* for *ZipList* but you could have also done it by a recursive call to *pure* in its own defintion.