Homework Assignment #4

Due Friday 11/13

1. We saw an implementation of binary trees in ML (slide 71). Write a similar data structure in Cecil. That is, a tree is either the `Empty` tree constant or a `Node` consisting of an element and two subtrees. (The type of the elements should be unspecified and left dynamically typed, as in Scheme.) More specifically, you should define an abstract `Tree` object with two child objects, a template `Node` object (with the appropriate fields attached and a `new_node` method to create concrete anonymous children of `Node`) and a concrete `Empty` object. Give some examples showing how to construct trees.

2. Implement the `insert` and `member` functions for your tree data structure. You should write a distinct `insert` and `member` method for each of the two kinds of trees, using dynamic dispatching to select the appropriate method based on the kind of tree object receiving the message. You can assume that whatever kind of elements you store in the tree, they can be compared with = and <. Give some examples showing how to use your implementation to do some insertions and membership tests.

3. Implement a preorder traversal operation, named `do`, for your tree data structure. The `do` operation takes a tree and a one-argument closure, and invokes the closure on all elements of the tree, in preorder. The `do` operation does not return any result. Again, `do` should have a distinct implementation for each kind of tree.

You can implement the following method to print out the elements of a tree, using this `do` operation (`print_string` methods are called by the system to print out values returned by top-level expressions):

```
method print_string(tree@Tree) {
   let var s := "tree{";
   let var first := true;
   tree.do(&(elem){
      if(first, {
         first := false;
      }, {
         s := s || ",";
      });
      s := s || elem.print_string;
   });
   s || "}" }
```

4. Write a `sum_tree` function in terms of the `do` operation in the previous question (you do not need to check whether tree elements are numbers). Only one `sum_tree` method should be written, since it can rely on `do` to handle the different kinds of trees.

5. Write a `reduce_tree` function that does not create any explicit intermediate data structure. Use `reduce_tree` to implement `sum_tree` without using any assignments or other side-effects.

6. a. Add a new representation of binary trees to your hierarchy, a `Singleton` leaf holding a value but no subtrees. You should implement `new_singleton`, `insert`, `member`, and `do` methods for your new representation. (All previously written code should continue to work without change.)

b. Is it possible to do this kind of extension in ML without modifying the existing code? How, or why not?