Comp 212 - Intermediate Programming EXAM #2 November 14, 2001
Rice University - Instructors: Nguyen & Wong NAME: ______________________

Instructions

1. This exam is conducted under the Rice Honor Code.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax of defining a class with appropriate fields, methods, and inheritance hierarchy. You will not be penalized on trivial syntax errors, such as missing curly braces, missing semi-colons, etc, but do try to write Java code as syntactically correct as possible. We are more interested in your ability to show us that you understand the concepts than your memorization of syntax! If you can't the code, write your thoughts down in prose.
5. Write your code in the most object-oriented way possible, that is, with the fewest number of control statements and no checking of the states and class types of the objects involved.
6. In all of the questions, feel free to write additional helper methods or visitors to get the job done. All helper visitors should be written as anonymous inner classes. You are expected to know the syntax and rules of anonymous inner classes. If you do not know how to write anonymous inner classes, then write named classed instead and get partial credits.
7. Make sure you use the Singleton pattern whenever appropriate. Unless specified otherwise, you do not need to write any code for it. Just write "singleton pattern" as a comment.
8. For each algorithm you are asked to write, 90% of the grade will be for correctness, and 10% will be for efficiency and code clarity.
9. You can reference all visitors from the lectures, labs, and the homeworks without having to write the code for them.
10. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1) 20	2) 20	3) 20	4) 20	5a) 10	5b) 10	TOTAL 100

1. Given two LRStructs that do not share any node, write a visitor called Zipper that "zips" them together like a zipper into one LRStruct. For example, when you zip (5, 3, 9) together with (-7, 0, 8, -15), both lists become (5, -7, 3, 0, 9, 8, -15). Note that the lists may have different lengths. Use anonymous inner classes for all helper visitors. Use proper indentation to make your code readable. Hint: you may find the following visitor useful. 20 pts

```
/**
* Assigns the input list to the host.  host thus "becomes" the input list.
* In the end, both host and the input list share the same head node.
*/
public class Becomes implements IAlgo {
   public final static Becomes Singleton = new Becomes ();
   private Becomes() {
   }
   /**
    * @param input a LRStruct;
    */
   public Object emptyCase(LRStruct host, Object input) {
      return assign(host, (LRStruct)input);
   }
   /**
    * @param input a LRStruct;
    */
   public Object nonEmptyCase(LRStruct host, Object input) {
      return assign(host, (LRStruct)input);
   }

   final private Object assign (LRStruct lhs, LRStruct rhs) {
      lhs.insertFront (null);
      lhs.setRest (rhs);
      lhs.removeFront();
      return null;
   }
}
```

2. Write a LRStruct visitor called SwapEnds to swap the first element and the last element of the host. For example, suppose host = (5, 12, -9, 0) before execution, then after execution, host = (0, 12, -9, 5). Do this with the fewest list traversals as possible. Do not swap the nodes; just swap the contents of the first data node and the last data node. Write all helpers as anonymous inner classes. 20 pts

3. Write a visitor for BiTree called OneItemPerLine to compute a String representation of the host consisting of one data item per line in in-order traversal. The resulting String should not have any new line preceding the data items, nor in between the data items, nor following the data items. Write all helpers as anonymous inner classes. *Hint*: you may need one helper for the left subtree and another helper for the right subtree. 20 pts

4. A BiTree is said to be a leaf if it is not empty and both of its subtrees are empty. Write a BiTree visitor to return Boolean.TRUE if the host is a leaf, Boolean.FALSE otherwise. Do this without checking the subtrees for emptiness. Write all helpers as anonymous inner classes. 20 pts

5. In this problem, we will build the code to compose functions as an abstract function on functions. That is, given two arbitrary functions, f and g, we want to create a single function that could calculate the "composition" of f and g: f(g(x)). We will make a function that will take a function f and return a function that will take a function g and that will return a function that will take x (an arbitrary input) and that will return the composition of f and g. Whew!

a) Recall the interface *ILambda* represents an abstract function that takes an Object as input and produces an Object as output:

```
public interface ILambda {
        public abstract Object apply(Object arg);
}
```

Using anonymous inner classes, write the code to instantiate an (anonymous) *ILambda* object, "a", whose apply() method takes another *ILambda* object, "g", and returns a third *ILambda* object whose apply() method takes an Object, "x", and returns g.apply(x).

That is:
(a.apply(g)).apply(x) returns the same result as g.apply(x); 10 pts

You may NOT change the ILambda interface in any way, including adding additional methods to any classes that implement it.

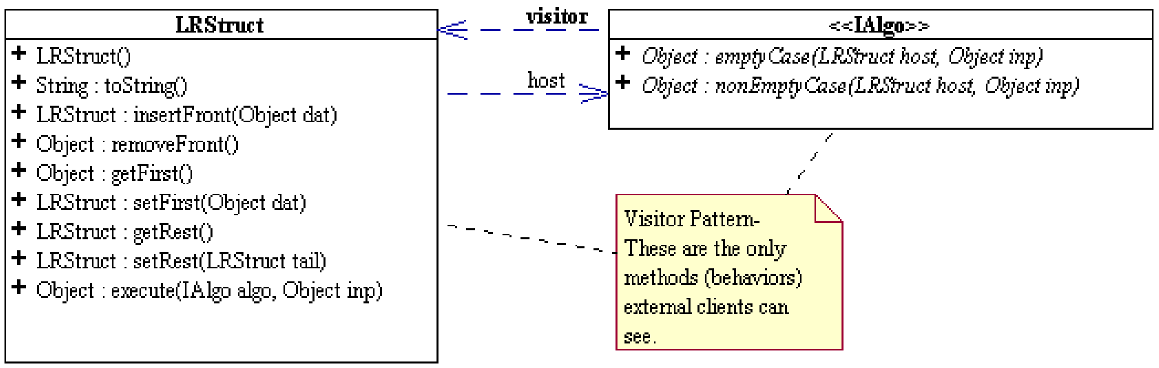
b) Modify your code from previous part so that "a" is now the return value of the apply() method of an (anonymous) instance of an *ILambda* called "compose". The apply() method of compose should take an *ILambda* object "f" as input.

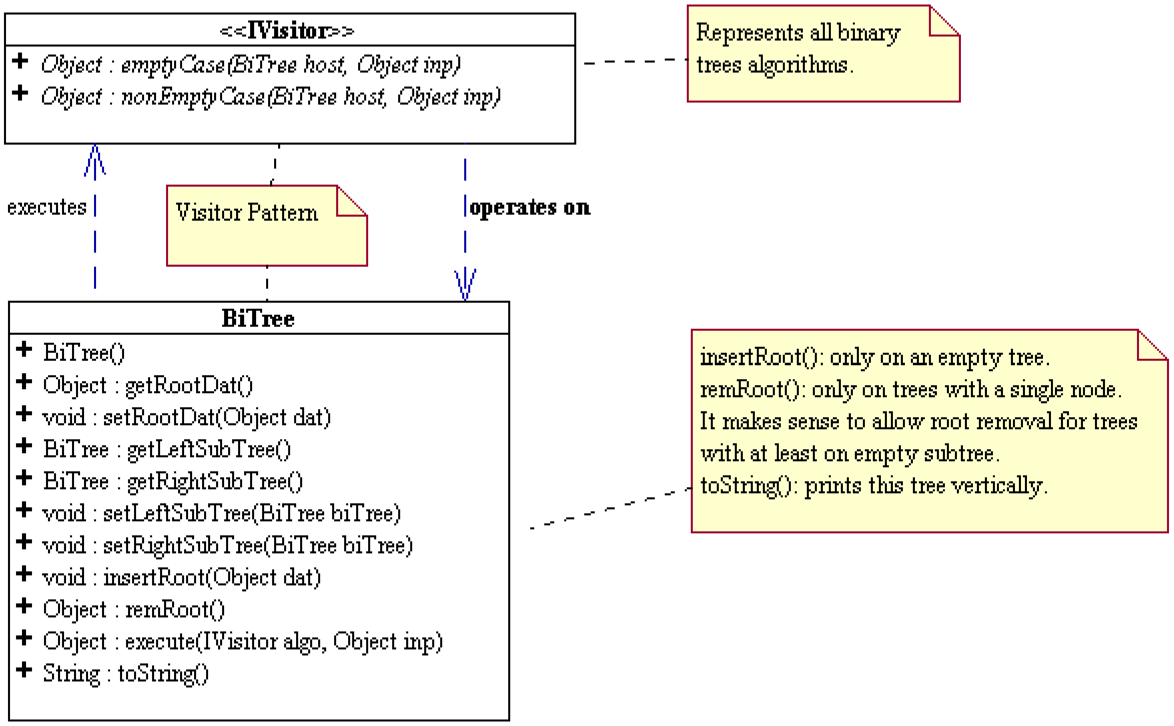
Modify your previous code so that the *ILambda* returned by what was "a" now returns an *ILambda* whose apply() method calculates composition of f and g.

That is
((compose.apply(f)).apply(g)).apply(x) returns the same result as f.apply(g.apply(x))

Write the <u>entire</u> code for your solution below; again you may not change the *ILambda* interface in any way. 10 pts

For your convenience, here are the class diagrams for LRStruct, BiTree, and their respective visitors.


LRStruct
+ LRStruct()
+ String : toString()
+ LRStruct : insertFront(Object dat)
+ Object : removeFront()
+ Object : getFirst()
+ LRStruct : setFirst(Object dat)
+ LRStruct : getRest()
+ LRStruct : setRest(LRStruct tail)
+ Object : execute(IAlgo algo, Object inp)
visitor
host
<<IAlgo>>
+ Object : emptyCase(LRStruct host, Object inp)
+ Object : nonEmptyCase(LRStruct host, Object inp)
Visitor Pattern- These are the only methods (behaviors) external clients can see.


<<IVisitor>>
+ Object : emptyCase(BiTree host, Object inp)
+ Object : nonEmptyCase(BiTree host, Object inp)
Represents all binary trees algorithms.
executes
Visitor Pattern
operates on
BiTree
+ BiTree()
+ Object : getRootDat()
+ void : setRootDat(Object dat)
+ BiTree : getLeftSubTree()
+ BiTree : getRightSubTree()
+ void : setLeftSubTree(BiTree biTree)
+ void : setRightSubTree(BiTree biTree)
+ void : insertRoot(Object dat)
+ Object : remRoot()
+ Object : execute(IVisitor algo, Object inp)
+ String : toString()
insertRoot(): only on an empty tree.
remRoot(): only on trees with a single node.
It makes sense to allow root removal for trees with at least on empty subtree.
toString(): prints this tree vertically.