CSC324 Fall 2010

Assignment 0 : Review of Recursion on Lists and Trees

General guidelines: public comments are not required; don't submit your testing; assume valid uses — error handling is not required.

(1) The following Java/Python classes model immutable non-empty lists.

```
class List {
  private Object first; private List rest;
  private List(Object first, List rest) { this.first = first; this.rest = rest; }
  /* The following three methods are the only ones you may use from this class.
     To remind you how to use static methods:
       List l = List.cons(3, List.cons(2, List.cons(4, null)));
       System.out.println(List.first(l));
       System.out.println(List.rest(l)); */
  public static List cons(Object first, List rest) {
    return new List(first, rest); }
  // Precondition: l != null.
  public static Object first(List l) { return l.first; }
  public static List rest(List l) { return l.rest; }
  /* However, you may add and use methods to help you test,
      e.g. toString, equals. */ }

class List:
  def __init__(self, first, rest):
    self.__first = first
    self.__rest = rest
  # Precondition: rest is a List or None.
  def cons(first, rest): return List(first, rest)
  # Precondition: l is a List.
  def first(l): return l._List__first
  def rest(l): return l._List__rest
  # Like the Java version, don't access the List instance variables or constructor
  #  directly in your solutions. But, similarly, you may access them for testing,
  #  possibly via added methods such as __str__, __cmp__.
  # To complete the similarity, here is the corresponding usage example:
  #   l = cons(3, cons(2, cons(4, None)))
  #   print first(l)
  #   print rest(l)
```

The empty list is being modelled by `null/None`. Below, "list" means a `List` object or `null/None`.

The following Java interface models functions as objects (a common "Design Pattern").

```
interface Function { Object call(Object o); }
```

For Python, model a function as an instance of a class defining a unary method "`call`".

Now, make a Java class `ListOperations` containing (at least) the following static methods, **OR** make a Python module `ListOperations` (importing `*` from `List`) with (at least) the following (top-level) functions:

(a) `isSubsequence` takes two lists, returning whether the first list's elements occur somewhere in the second list in the same order. Compare elements with `equals` (Java) or `==` (Python). E.g., for (printed here Schemeishly) `()` and `(1 2 3)` return true, for `(1 3)` and `(1 2 3)` also return true. For `(3 2)` and `(1 2 3)` return false, for `(1 1)` and `(1 2 3)` also return false.

(b) `map` takes a list and a function object, returning a new list containing the results of calling the function's method `call` on each of the list's elements.
Do this in tail-recursive accumulating style with a helper.

(c) `consAll` takes an object and a list of lists, returning a new list containing the lists with the object cons'd onto each of them.
E.g., for `5` and `((3 2 4) (4 2) (1 2 3))` return `((5 3 2 4) (5 4 2) (5 1 2 3))`.
Do this by making a `Conser` class with one instance variable, that models functions that `cons` a specific object onto a list. Then `map` a `Conser` instance onto the list.

(d) `subsequences` takes a list, returning a list containing all subsequences of the given list.
E.g., for `(3 2 4)` return `(() (4) (2) (2 4) (3) (3 4) (3 2) (3 2 4))` — although you may generate the elements (lists) of the result list in any order.
Hints: write an `append`, use `consAll`, ignore `isSubsequence`.

(2) The following Java/Python classes model immutable non-empty labelled full binary trees:

```
class Tree {
  public String label; public Tree left ; public Tree right;
  // Precondition: label != null ; left and right both null or both non-null.
  public Tree(String label, Tree left, Tree right) {
    this.label = label;
    this.left = left;
    this.right = right; }}

class Tree:
  # Precondition: label a string ; left and right both None or both Trees.
  def __init__(self, label, left, right):
    self.label = label
    self.left = left
    self.right = right
```

For simplicity, the instance variables are publically accessible; but don't assign to them.

Now, make a Java class `TreeOperations` containing (at least) the following static methods, or a Python module `TreeOperations` (importing `*` from `Tree` and `List`) with (at least) the following (top-level) Python functions:

(a) `treeToNestedLists` takes a `Tree`, returning a one or three element `List` (defined in (1)) containing the label and, if not a leaf, the children represented this way as `List`s, recursively. E.g., consider the following tree (drawn sideways, and children ordered top to bottom):

```
a---b---c
 \   \--d---e
  \-g    \--f
```

The returned `List` is (printed here Pythonishly) `[a, [b, [c], [d, [e], [f]]], [g]]`.

(b) `nestedListsToTree`, the inverse of (a).

Now, for (c-e), *don't* use (a) or (b) to transform the argument.

(c) `treeToString` takes a `Tree`, returning a string in the following format (`Tree` as in (a)): `a(b(c,d(e,f)),g)`.

(d) `printCalls` takes a `Tree` and, thinking of its `treeToString` as nested function call code, prints the calls/arguments in the order they would be called/evaluated in Python/Java/C. Print one function/argument name per line.

(e) `printNestedLists` takes a `List` (as in (a)), printing it in the following indented format (consider the first "`(`" unindented):

```
(a
  (b
    c
    (d
      e
      f))
  g)
```

Each level is indented two more spaces.

Hint: accumulate the indentation via a parameter.