211hw8_S11

Homework 08 (Due 11:59pm Tuesday, March 22, 2011)

Submit via Owl-Space

Preliminaries

This homework on manipulating Java lists must be done using the Functional language level of DrJava.We are providing skeleton classes for each problem with unimplemented methods for you to write. Each such class includes a method with a name like `listString()` that converts lists to more readable String notation similar to that used in Scheme. You can put all of your classes (except test classes) in the same file. Write a test class for each composite data type (`Word`, `WordList`, `ComparableList`, `ObjectList`, and `ArithExpr`).

Composite Design Pattern for List

1. Write the `arrangements` (permutations) function from HTDP problem 12.4.2 in HW2 as a Java method for a `Word` class provided in the file `Word.dj1`. This file includes definitions of the composite pattern classes `Word` and `WordList`. Decompose the problem in exactly the same form as this solution to problem 12.4.2. We are providing skeletons for the classes `Word` and `WordList` in the file WordList.dj1; use them.
2. Write the `mergesort` function from the last problem in HW4 (using exactly the same top-down approach described in HW4) as a Java method in the composite pattern class `ComparableList` provided in the file ComparableList.dj1.
3. Do Exercise 21.2.3 from HTDP using the Java composite pattern class `ObjectList` provided in the file ObjectList.dj1. This file includes the interface `Predicate`, which is the type of Java function arguments passed to the `filter` method, and abstract method stubs for `filter`, `eliminateExp`, `recall`, and `selection` . Exercise 21.2.3 provides Scheme code for the `filter` function which you should directly translate to the corresponding Java method code in `ObjectList`. Note that your `filter` method should work for arbitrary `ObjectLists`. In coding the methods `eliminateExp`, `recall`, and `selection`, use the Java type `Number` (compared using method `doubleValue` in place of the Scheme `number` type and the Java type `Object` in place of the Scheme `symbol` type. The method `doubleValue()` in `Number` returns the value of this converted to a `double`. You will need to cast the `Object` input of the `test` method to type `Number` when filtering lists of numbers.
4. Do Problem 2 from Homework 5 in Java using the composite hierarchy of classes provided in the file ArithExpr.dj1.

Are You The Last Element In A List?

A difficulty when processing lists is how to tell if one is at the last element of the list. This is useful in a number of situations, such as in the merge sorting process above.

In class, the following shortcut was shown but with the clear caveat that this style of programming is only being done for brevity's sake at this point and that it will NOT be tolerated in the future:

(THE FOLLOWING CODE USES GENERIC NAMES FOR EVERYTHING. DO NOT ROTE COPY AND USE THESE NAMES!)

```
// In a method of a ConsList

if(rest == EmptyList.ONLY) {
   // we are at the last element, so process accordingly
}
else
{
   // we are not at the last element, so process accordingly
}
```

The above code is frowned upon because it is an encapsulation violation of `rest`, amounting to a checking of its type. You may use this style for this assignment, but it is <u>not recommended</u> and you can expect to be marked down for it in the future.

The better solution is to delegate to `rest` and let it continue the processing in a manner consistent with what it is. Remember that the last element of a list is defined by the fact that its `rest` is `empty`, or conversely, the parent of an `empty` list is the last element. This thus requires a helper method:

```
class ConsList implements List {

      ...
      // In a method of the ConsList...
      return rest.helper(this);  // delegate to rest and let it decide what to do.
   }

   /**
    * Helper method.   If this method is called, we know that the parent is NOT the last element!
    * @param parent A reference to the parent list, i.e. the caller.
    */
   Object helper(ConsList parent) {
       // Parent is NOT the last element, process the *parent* accordingly.
   }
}

class EmptyList implements List {
   ...

   /**
    * Helper method.   If this method is called, we know that the parent is IS the last element!
    * @param parent A reference to the parent list, i.e. the caller.
    */
   Object helper(ConsList parent) {
       // Parent IS the last element, process the *parent* accordingly.
   }
}
```

This code is longer, but safer, more robust and more extensible.

But Wait, There's More!

How could you extend the delegation ideas in the above technique to enable you to differentiate between the follow four scenarios involving two lists?

1. One list is empty and
    a. the other list is empty or
    b. the other list is non-empty
2. One list is non-empty and
    a. the other list is empty or
    b. the other list is non-empty