HW06

Homework 6 (Due 11:59pm Tuesday, October 27, 2020)

Submit via SVN

Preliminaries

This homework can be done using the Functional language level of DrJava, which autogenerates the constructors, accessors, equals, and toString() methods on the assumption that the class narrowed to its fields constitutes a free algebraic type.

Composite Design Pattern for List

The following is an object-oriented formulation of lists of integers.

- `IntList` is an abstract list of `int`.
- `EmptyIntList` is an `IntList`
- `ConsIntList(first, rest)`, where `first` is an `int` and `rest` is an `IntList`, is an `IntList`

The above can be implemented in functional Java (as supported by the DrJava functional language level as follows.

```
/** Abstract list structure.  IntList := EmptyIntList + ConsIntList(int, IntList) */
abstract class IntList { }

/** Concrete empty list structure containing nothing. */
class EmptyIntList extends IntList { }

/** Concrete non-empty list structure containing an int, called first, and an IntList
called rest. */
class ConsIntList extends IntList {
    int first;
    IntList rest;
}
```

The above implementation is an example of what is called the **Composite Design Pattern**. The composite design pattern is a special case of the union pattern where one or more of the variants for the union type T contains fields of root type T. In this pattern, the union is called a composite. Here the union type is `IntList` and the variant `ConsIntList` is said to be a composite because it includes a field of type `IntList`.

The composite pattern also prescribes a coding pattern for the its methods: when a variant is called to perform an operation, it traverses its fields of root type and calls on them to perform the same operation. It allows a client to treat an instance of type T and it embedded instances uniformly using polymorphism.

This coding pattern is called the interpreter design pattern: it interprets the abstract behavior of a class in each of its concrete subclasses. The composite pattern refers to the the structure of the composite type hierarchy, while the interpreter pattern refers to how the behavior of the variants of the type are defined uniformly via polymorphism.

Interpreter Design Pattern for List

The interpreter design pattern applied to the above composite list structure prescribes a coding pattern for list operations that is analogous to Racket function template. It entails declaring an abstract method for each list operation in the abstract list class, `IntList`, and defining

corresponding concrete methods in the concrete list subclasses: the empty list class, `EmptyIntList`, and the non-empty class, `ConsIntList`. The concrete method for `EmptyIntList` corresponds to the base case in the Racket function template while the concrete method in `ConstIntList` corresponds to the recursive case by calling the same method on its rest.

The following is the coding template for the interpreter design pattern for `IntList` and its subclasses.

```
abstract class IntList {
    abstract returnType methodName(parameter_list);
}

class EmptyIntList extends IntList {
    returnType methodName(parameter_list) {
        // base case code
    }
}

class ConsIntList extends IntList {
    int first;
    IntList rest;
    returnType methodName(parameter_list) {
        // ... first ...
        // ... rest.methodName(parameter_list) ...
    }
}
```

Problems

Apply the interpreter design pattern to `IntList` and its subclasses given above to write all of the following methods as augmentations (added code) of the `IntList` class. Also write a JUnit test class, `IntListTest` to test all of your new methods in the `IntList` class. We strongly recommend that you write Template Instantiations for all of these new methods as an intermediate step in developing your code **BUT DO NOT submit** these Template Instantiations (or corresponding Templates) as part of your code documentation. The structure of your program implicitly provides this information. Confine your documentation to writing contracts (purpose statements in HTDP terminology) using javadoc notation (opening the purpose statement (preceding the corresponding definition)with /** and closing it with */.

- (10 pts.) `boolean contains(int key)` : returns true if key is in the list, false otherwise.
- (10 pts.) `int length()` : computes the length of the list.
- (10 pts.) `int sum()` : computes the sum of the elements in the list.
- (10 pts.) `double average()` : computes the average of the elements in the list; returns 0 if the list is empty.
  Hint: you can *cast* an `int` to `double` by using the prefix operator `(double)`.
- (10 pts.) `IntList notGreaterThan(int bound)` : returns a list of elements in this list that are less or equal to `bound` .
- (10 pts.) `IntList remove(int key)` : returns a list of all elements in this list that are not equal to `key` .
- (10 pts.) `IntList subst(int oldN, int newN)` : returns a list of all elements in this list with `oldN` replaced by `newN` .
- (30 pts.) `IntList merge(IntList other)` merges this list with the input list other, assuming that this list and other are sorted in ascending order. Note that the lists need not have the same length.
  Hint: add a method `mergeHelp(ConsIntList other)` that does all of the work if one list is non-empty (a `ConsIntList`). Only `mergeHelp` is recursive. Use dynamic dispatch on the list that may be empty. Recall that `a.merge(b)` is equivalent to `b.merge(a)` . This approach is the Java analog of the extra credit option in HTDP Problem 17.6.1 in HW 3.