8 Abstracting Over the Data Type

Practice Problems

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Finish Lab 8 and include all the work in your portfolio.

Pair Programming Assignment

8.1 Problem

Binary Search

Start with a new project and create two files: **Algorithms.java** and **ExamplesAlgorithms.java**.

A. In the `ExamplesAlgorithms` make examples of sorted `ArrayLists` of `String`s and `Integer`s.

   Of course, there is no constructor that creates an `ArrayList` filled with values. You need to define a method `initData` that adds the values to the initially empty `ArrayList`s one at a time.

B. Next, design two classes that implement the `Comparator` interface in Java Collections — one that compares `String`s by lexicographical ordering, one that compares `Integer`s by their magnitude.

C. Now, design the method `binarySearch` in the class `Algorithms` that consumes the lower index (inclusive), the upper index (exclusive), an `ArrayList` of data of the type `T`, a `Comparator` of the type `T`, and an object of the type `T` and produces the index for this object in the given `ArrayList` or throws a `RuntimeException` if the object is not found.

8.2 Problem

Abstracting Over the Data Type

Download the file *Expressions.java*. It includes the implementation and some sample tests of the classes that represent an arithmetic expression where the values can only be integers, and the only operation allowed is addition.

A. Study the class diagram for this class hierarchy. Extend the example so that the expressions can also include multiplication.

   Hint: Add the class `Times`.

B. Design the method `toString` that produces a `String` representation of this expression with parentheses surrounding every binary expression. Define examples that represent the following expressions and include tests that verify that they have been correctly rendered as `String`s':

```
(2 + (3 + 4))
((3 + 5) * ((2 * 3) + 5))
```

C. We now want to represent relational expressions (that compare two integer values and produce a boolean value). We limit our choices to the *greater than* and *equal to* comparisons. We also want to represent boolean expressions, *and* as well as *or*.

   Change the definitions so that they are parametrized over the type of data you will use.

   The `IExp` interface is parametrized only over the type of value it represents when evaluated.

   The `BinOp` class needs to be parametrized over the type of operands it receives, as well as the type of value it produces.

D. Add the necessary class definitions so you can represent *relational* and *arithmetic* expressions.

   Make sure you have examples for each of them, as well as tests for the `eval` method.

E. Now design two new classes `IntVar` and `BoolVar` that will represent a variable of the appropriate type in the expression and `implements` `IExp`. It needs to keep track of its name, e.g. `x`, or `width`, etc.

It should include a method `substInt` for the class `IntVar` and the method `substBool` for the class `BoolVar` that consumes a `String` and an argument of the appropriate type and produces an instance of a `Value` that represents the given value, provided the given `String` matches the variable name. In all other cases it just returns `this`.

Of course, it has to include the method `eval`. However, this method should `throw` an exception, indicating that an expression with a variable in it cannot be evaluated.

F. Design the method `noVars`, a predicate that verifies that the expression does not contain any variables.

G. Design the methods `substInt` and `substBool` for the entire `IExp` class hierarchy, that produces a new `IExp` in which every occurrence of `Var` that matches the given name is replaced with an instance of the class `Value` with the given value. Throw an exception if there is an attempt to substitute a `boolean` value for the identifier that represents an `int` value as well as if there is an attempt to substitute a `int` value for the identifier that represents an `boolean` value.

8.3 Problem

Abstract Data Type

During the lectures we have defined the interface *DataSet.java* as follows:

```
// to represent a collection of data of the type T
interface DataSet<T>{

  // add the given item to this data set
  void add(T t);

  // EFFECT: remove an item from this data set
  // return the item that has been removed
  // throw a RuntimeException if this data set is empty
  T remove();

  // return the the number of items in this data set
  int size();
}
```

A. Make examples of `ArrayList`s of `String`s that represent playing cards. If you do not wish to use playing cards as examples, you can use any other collection of `String`s. In our choice of a simple representation we have:

```
"Qh"  - for queen of hearts
"10s"  - for 10 of spades
"3d"  - for 3 of diamonds
"Jc"  - for jack of clubs
etc.
```

Again, you will need an `initData` method to fill the sample lists with values.

Use these examples to design tests for the next two classes:

B. Design the class `Stack` that implements the `DataSet` interface using an `ArrayList` to hold the data items and adds and removes the items at the same end.

   This is also known as *LIFO* — last in, first out organization.

C. Design the class `Queue` that implements the `DataSet` interface using an `ArrayList` to hold the data items and adds the data items at one end and removes the items from the other end.

   This is also known as *FIFO* — first in, first out organization.