CS 240 Homework 7

Alex Vondrak

Due: March 4, 2012

In this homework, you'll be adding a feature to the RPN calculator from Homework 5: variables. This moves the RPN calculator one step closer to being a full-fledged programming language interpreter, so if you're me, you'll find this at least mildly interesting. Specifically, we need to add two new operators: `set` and `get`.

The `set` command should work with arbitrary *tokens*—any ol' whitespace-separated string we've read in. For instance, the code `10 x set` should set the variable `x` to the value `10`. Then, `x get` should push `10` to the stack, as that's the value associated with `x`. If at a future point we say `20 x set`, `x get` should then push `20` (since we've clobbered the old value of `x`).

However, this requires a bit of an overhaul of the Homework 5 code. Not only will integers get pushed to the operand stack, but now tokens representing variables have to be pushed, too. For instance, this means that we no longer have the `Unknown operator` error: we just push arbitrary symbols onto the stack, in case we call `set` or `get` on them.

To this end, I've provided the skeleton code for this project at `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw7.java`.

First, notice that we import `java.util.Stack`. We'll be using Java's stack object instead of rolling our own implementation, since (a) we've already done that before and (b) it's generally preferable to use existing libraries.

Next, perhaps the most important class we now have is called `Token`. Instead of pushing just `Integer`s to the operand stack, we must push `Token`s, which will store their respective string representations. If we suspect that a particular `Token` on the stack is an integer, we can call the `intValue()` method to try to get an actual number out of the string; this will throw a `NumberFormatException` in the event that the `Token` does not actually represent an integer. If, however, we want to use the `Token` as a variable, we can just use its `stringValue()` method to get the variable's name. Since every `Token` has a string representation, this should never raise an error. A side-effect of this is that *numbers* can be used as variables! For example, `10 5 set 5` should just leave `5` on the stack, whereas `10 5 set 5 get` should leave `10`.

Following that is the mostly-empty `AssociationList<K, V>` class, which will be a linked list (you'll need to put a **`private`** node class inside of this one). However, notice that we take *two* generic type parameters: one for the *key* and one for the *value*. In an `AssociationList`, we don't just store single elements, but rather an *association* between pairs of values—a mapping, if you will. Thus, we can use this list to keep track of our variable/integer pairs: using Java `String`s as keys, we associate each key with an `Integer` value. This requires a change to the node structure used in the class, because instead of tracking a single `E data`, we'll need to track two pieces of data: `K key` and `V value`. Then, the specialized `set` and `get` methods of the `AssociationList<K, V>` class will be used to establish/overwrite new mappings and retrieve the value associated with a particular key, respectively.

For example, suppose we have an `AssociationList<String, Integer>` instance called `environment`. `environment.set("x", 5)` will associate the key `"x"` with the value `5`. Then `environment.get("x")` should return `5`. If we then do `environment.set("x", 10)`, `environment.get("x")` should return `10` (as this is a more recent mapping).

What remains is the `RPNCalculator` class, which has been overhauled from my Homework 5 version to facilitate these new `set` and `get` commands. You must implement the following methods:

- **`private void`** `binaryOp(String expr)`, which executes a standard math operator (`"+"`, `"-"`, `"*"`, or `"/"`) by using the `mathOp` method provided, or the `set` operator. If there aren't enough elements on the stack to perform the binary operator, we have to print the error message `"Not␣enough␣operands␣"` like before; this is provided by the `underflow` method in the `RPNCalculator` class. If you try to execute a mathematical operator on two arguments, but at least one of the arguments does *not* represent an integer, you should print the error message `"Can't␣perform␣math␣operator␣on␣non-integers␣"`. E.g., `1 x +` is illegal; you probably meant to say `1 x get +`.

- **`private void`** `unaryOp(String expr)`, which basically just executes `get` (the only unary operator we have at the moment, though it's easy to imagine adding more, like a negation command). Again, if there aren't enough arguments, we issue the error message here.

- **`private void`** `set()`, which executes the logic of the `set` command, and should be called from `binaryOp`. While anything can be a variable, we can only use integers as values. So, if the second-from-the-top `Token` of the stack can't be parsed as an integer, we must print the error message `"'set'␣must␣be␣given␣an␣integer␣"`.

- **`private void`** `get()`, which executes the logic of the `get` command, and should be called from `unaryOp`. This pops the top element of the stack, and looks up its value in **`this`**`.environment`. If there is no such value, you should print out the error message `"Unbound␣variable␣'x'␣"`, where `x` is the name of the variable you tried to look up. If there is a value, push it onto the stack.

Input Format

Input is of the same format as Homework 5.

Output Format

See the problem description for specific error messages. The remaining output is generally dictated by the provided code and Java's implementation of `toString` for its `Stack` library. See the Sample Output and the provided code.

Input Sample

Test against the ACID input available at `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw7.in`

Output Sample

Test against the ACID output available at `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw7.out`

Extra Credit

In the `AssociationList<K, V>` class, the `get` and `set` methods are rather interdependent. How we implement one affects the time and space complexity of the other.

For extra credit, upload a plain-text file `hw7.ec` to your ZFS share along with `hw7.java`. In this new file, describe at least 3 different ways of implementing `set`, what their respective running time complexities are, how much space they use to store nodes in the linked list, and how they respectively impact the corresponding implementation & running time of `get`.

Use prose to explain this! You may have accompanying code examples if you want, but if you just slap down a half-hearted couple of lines of comments, I don't know that you really understand the topic at hand. (Of course, also avoid being overly verbose.)