10.5 Laboratory: A Stack-Based Language

Objective. To implement a PostScript-based calculator.

Discussion. In this lab we will investigate a small portion of a stack-based language called PostScript. You will probably recognize that PostScript is a file format often used with printers. In fact, the file you send to your printer is a program that instructs your printer to draw the appropriate output. PostScript is stack-based: integral to the language is an operand stack. Each operation that is executed pops its operands from the stack and pushes on a result. There are other notable examples of stack-based languages, including **forth**, a language commonly used by astronomers to program telescopes. If you have an older Hewlett-Packard calculator, it likely uses a stack-based input mechanism to perform calculations.

We will implement a few of the math operators available in PostScript.

To see how PostScript works, you can run a PostScript simulator. (A good simulator for PostScript is the freely available [**ghostscript**](http://www.gm.org).) If you have a simulator handy, you might try the following example inputs. (To exit a PostScript simulator, type `quit`.)

1. The following program computes $1 + 1$:

   ```postscript
   1 1 add pstack
   ```

   Every item you type in is a token. Tokens include numbers, booleans, or symbols. Here, we’ve typed in two numeric tokens, followed by two symbolic tokens. Each number is pushed on the internal stack of operands. When the `add` token is encountered, it causes PostScript to pop off two values and add them together. The result is pushed back on the stack. (Other mathematical operations include `sub`, `mul`, and `div`.) The `pstack` command causes the entire stack to be printed to the console.

2. Provided the stack contains at least one value, the `pop` operator can be used to remove it. Thus, the following computes 2 and prints nothing:

   ```postscript
   1 1 add pop pstack
   ```

3. The following “program” computes $1 + 3 * 4$:

   ```postscript
   1 3 4 mul add pstack
   ```

   The result computed here, 13, is different than what is computed by the following program:

   ```postscript
   1 3 add 4 mul pstack
   ```

   In the latter case the addition is performed first, computing 16.
4. Some operations simply move values about. You can duplicate values—the following squares the number 10.1:

```
10.1 dup mul pstack pop
```

The `exch` operator to exchange two values, computing $1 - 3$:

```
3 1 exch sub pstack pop
```

5. Comparison operations compute logical values:

```
1 2 eq pstack pop
```

tests for equality of 1 and 2, and leaves `false` on the stack. The program

```
1 1 eq pstack pop
```

yields a value of `true`.

6. Symbols are defined using the `def` operation. To define a symbolic value we specify a “quoted” symbol (preceded by a slash) and the value, all followed by the operator `def`:

```
/pi 3.141592653 def
```

Once we define a symbol, we can use it in computations:

```
/radius 1.6 def
pi radius dup mul mul pstack pop
```

computes and prints the area of a circle with radius 1.6. After the pop, the stack is empty.

**Procedure.** Write a program that simulates the behavior of this small subset of PostScript. To help you accomplish this, we've created three classes that you will find useful:

- `Token`. An immutable (constant) object that contains a double, boolean, or symbol. Different constructors allow you to construct different `Token` values. The class also provides methods to determine the type and value of a token.

- `Reader`. A class that allows you to read `Tokens` from an input stream. The typical use of a reader is as follows:

```java
Reader r = new Reader();
Token t;
while (r.hasNext())
{
    t = (Token)r.nextToken();
    if (t.isSymbol() && // only if symbol:
        t.getString().equals("quit")) break;
    // process token
}
```
This is actually our first use of an Iterator. It always returns an Object of type Token.

- SymbolTable: An object that allows you to keep track of String-Token associations. Here is an example of how to save and recall the value of \( \pi \):

  ```java
  SymbolTable table = new SymbolTable();
  // sometime later:
  table.add("pi", new Token(3.141592653));
  // sometime even later:
  if (table.contains("pi"))
  {
    Token token = table.get("pi");
    System.out.println(token.getNumber());
  }
  ```

You should familiarize yourself with these classes before you launch into writing your interpreter.

To complete your project, you should implement the PostScript commands `pstack`, `add`, `sub`, `mul`, `div`, `dup`, `exch`, `eq`, `ne`, `def`, `pop`, `quit`. Also implement the nonstandard PostScript command `ptable` that prints the symbol table.

**Thought Questions.** Consider the following questions as you complete the lab:

1. If we are performing an `eq` operation, is it necessary to assume that the values on the top of the stack are, say, numbers?

2. The `pstack` operation should print the contents of the operand stack without destroying it. What is the most elegant way of doing this? (There are many choices.)

3. PostScript also has a notion of a procedure. A procedure is a series of Tokens surrounded by braces (e.g., `{ 2 add }`). The `Token` class reads procedures and stores the procedure's Tokens in a List. The `Reader` class has a constructor that takes a List as a parameter and returns a `Reader` that iteratively returns Tokens from its list. Can you augment your PostScript interpreter to handle the definition of functions like `area`, below?

   ```postscript
   /pi 3.141592653 def
   /area { dup mul pi mul } def
   1.6 area
   9 area pstack
   quit
   ```

Such a PostScript program defines a new procedure called `area` that computes \( \pi r^2 \) where \( r \) is the value found on the top of the stack when the procedure is called. The result of running this code would be

```
254.469004893
8.042477191680002
```
4. How might you implement the \texttt{if} operator? The \texttt{if} operator takes a boolean and a token (usually a procedure) and executes the token if the boolean is true. This would allow the definition of the absolute value function (given a less than operator, \texttt{lt}): 

\begin{verbatim}
/abs { dup 0 lt { -1 mul } if } def
3 abs
-3 abs
eq pstack
\end{verbatim}

The result is \texttt{true}.

5. What does the following do?

\begin{verbatim}
/count { dup 1 ne { dup 1 sub count } if } def
10 count pstack
\end{verbatim}

Notes: