Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 5
PostScript

Objective. To build an application that makes use of stacks.

Discussion. This week we will implement a small version of the programming language, PostScript.[1] PostScript, you may know, is the programming language used within most printers. When you design a poster and print it to a printer, you're really writing a program that is executed by the printer. The output of the program is the image of your poster.

PostScript is an example of a *stack based language*. There are other notable example of stack-based languages, including `forth`, a language once commonly used by astronomers to program telescopes. In addition, if you have an older HP calculator, it is likely to use a stack-based input mechanism to perform calculations.

We will implement a small portion of the PostScript language — the portion that performs simple math. To see how PostScript works, you can run `gs`, a PostScript interpreter, called "ghostscript". To begin running PostScript, type

```
gs -DNODISPLAY
```

To exit the PostScript interpreter, type `quit`. To get a flavor of the programming language, you might try the following example PostScript programs:

- Let's print `Hello, world.` Strings are surrounded by parentheses:

  ```
  (Hello, world.) =
  ```

  The print command (`=`) simply prints the last value seen.

- The following program computes $1 + 1$ and then prints the contents of the stack:

  ```
  1 1 add pstack
  ```

  Every item you type in is a *token*. Tokens can be numbers, booleans, strings, or symbols. Here, we've typed in two number tokens, followed by two symbol tokens. Each number is pushed on an internal stack of values. When the `add` token is encountered, it causes PostScript to pop off two values and add them together. The result of 2 is then pushed back on the stack. (Other mathematical operations include `sub`, `mul`, and `div`.) The `pstack` command prints the entire stack (currently, just the single value 2), without changing it.

  You will notice the prompt at this point is `GS<1>`, which indicates the stack has size 1. If we now type `pop`, it will pop the remaining value from the stack. The prompt returns to the empty stack indicator, `GS>`.

- The following program computes $1 + 3 * 4$ and prints the result, 13:

[1] The PostScript language was designed by Adobe Systems, Inc.

```
1 3 4 mul add =
```

The computation is different than the following program, that computes 16:

```
1 3 add 4 mul =
```

In this case the addition is performed first.

- You can duplicate values—the following squares the number 10:

```
10 dup mul =
```

And the following exchanges two values, computing $1 - 3$:

```
3 1 exch sub =
```

- There are comparison operations that compute logical values:

```
1 2 eq pstack
```

leaves a `false` on the stack, while

```
1 1 eq pstack
```

yields a value of `true`. Notice we really do have a stack: the last `pstack` prints two values, with `true`, the *last value in*, on top. We could use `=` to pop and print these values: `true` is *first out*.

- To define a new symbolic value, we push on a *quoted symbol*—a symbol preceded by a slash. We then push on the *value* to be associated with the symbol, followed by the operator `def`:

```
/radius 9 def
```

Notice that there is no result left on the stack. The symbol is stored in a *dictionary* or *symbol table*. Once we define a symbol, we can use it, *unquoted*, in computations:

```
3.14159 radius dup mul mul =
```

Computes and prints the area of a circle with radius 9.

- Curly braces delimit a list of tokens—a *procedure*—to be interpreted later. Here, we define a general procedure, `area`, that consumes the radius from the stack and pushes on the corresponding circle area:

```
/sqr { dup mul } def
/area { sqr pi mul } def
/pi 3.14159 def
radius area =
```

Notice how, in `area`, we make use of `pi` before we define it. (Not always the best programming practice, but sometimes necessary.) PostScript doesn't attempt to understand the meaning of tokens of the procedures until we *interpret* them later. The actual interpretation happens when we fetch the procedure from the symbol table as part of a real computation.

- Finally, PostScript has two commands that support checking conditions: `if` and `ifelse`. The `if` command takes a boolean condition and a procedure. The procedure is interpreted only if the boolean condition is `true`:

```
false { (Truth will set me free.) = } if
2 dup mul 2 dup add eq { (fact: 2*2 is 2+2) = } if
```

  The first `if` does nothing. The second `if` prints a fact.

  The `ifelse` command pops *three* things off the stack: a condition and *two* procedures. The first procedure is interpreted if the condition is `true`, otherwise the second procedure is interpreted. Here's some interesting code:

```
/spaceX {
   dup 0 eq { (Blast Off!) = pop } { (Tick!) = 1 sub spaceX } ifelse
} def
10 spaceX
```

We've only begun to explore computation in this fun little language. PostScript has many interesting drawing operators as well, but they're beyond the scope of this lab.

This Week's Tasks. We would like you to write an *interpreter*, a program that simulates the behavior of a small subset of PostScript. Here are the steps that we would like you to complete:

1. Clone your `lab5` repository:

```
cd ~/cs136
git clone https://evolene.cs.williams.edu/cs136-labs/22xyz3/lab5.git
```

   replacing `22xyz3` with your CS username. This will create a subdirectory, `lab5` in your `cs136` directory. Look around.

2. We've created three classes that you will find useful:

   - `Token` — an object that contains a number, boolean, a string, or symbol. Strings and symbols are both represented by Java `Strings`. String-valued tokens are easily identified since their values begin and end with parens, while symbols appear either with or without the forward-slash quote mark.

```
Token n = new Token(3.141);     // a number token
Token b = new Token(true);      // a boolean token
Token s = new Token("(Hello)"); // a string token
Token q = new Token("/pi");     // a symbol token (with a quote mark)
Token p = new Token("pi");      // a symbol token (without a quote)
```

- `Reader` — an object that allows you to read tokens from an input stream. The typical use of a `Reader` is as follows:

```
Reader r = new Reader(); // read from "standard input"
for (Token t : r) { ...do something with t...}
```

  This syntax is another example of an *iterated* `for` *loop*. It considers values that appear in a stream of `Tokens`. It is also possible to construct a `Reader` so that it focuses on a single `Token`. This usage is helpful for interpreting values we find stored in the symbol table.

- `SymbolTable` — an object that allows you to keep track of `Symbol-Token` associations. Here is an example of how to save and recall the value of π:

```
// In the main method:
SymbolTable table = new SymbolTable();
//...later, in the interpreter (note "pi" is unquoted):
table.add("pi",new Token(3.141592653));
//...even later (note we *interpret* it):
if (table.contains("pi")) {
    Token token = table.get("pi");
    interpret(new Reader(token)); // interpret pi's value
}
```

You should familiarize yourself with these classes before you launch into writing any Java.

3. Most of your work will be in the application, `Interpreter.java`. Notice that we've begun writing a `static interpret` method. This method takes a `Reader` and interprets its tokens. When most tokens are interpreted, they're simply pushed on a global stack. When unquoted symbols—commands—are encountered they typically lead to action. We've implemented the `quit` command as an example. Now, add a global `Stack<Token>` and initialize it in `main`. Push on any `Token` that's *not* an unquoted symbol. Add a new command `pstack`, that calls another static method to print the stack in a readable way, with the top of the stack at the top of your output.

4. Implement the PostScript commands `add`, `pop`, and `=`. Think carefully about how to structure your code neatly and identify any violations of assumptions you make about the stack. These operators will allow you to test basic interpretation of PostScript and verify it seems to act similarly to `gs`. Notice that the `=` command prints a string without its surrounding parentheses.

5. Now write `sub`, `mul`, `div`, `dup`, `exch`. This will allow you to write more complicated math expressions.

6. Implement `eq`, `lt`, and `not`. These compute boolean values. The `eq` command is based on the `equals` method for `Tokens`. For numbers and strings, `lt` can use the `compareTo` method. The `not` command simply inverts the boolean on the top of the stack.

   At this point you've earned about 9 points. If this is satisfactory, you can skip to Step 10 to test and submit your code.

7. We'll now implement `def` and `ptable`. First, make sure you add a global `SymbolTable<String,Token>`, and initialize it in your `main` method. The `def` command adds an association between a quoted symbol and an arbitrary `Token` to the symbol table (see `SymbolTable.java`). Make sure that you remove

the quote from the symbol before you place the definition in the symbol table. The non-standard `ptable` command (which is not available in ghostscript) will allow you to print the symbol table:

```
/pi 3.141 def /e 2.718 def ptable
```

should print

```
e=2.718
pi=3.141
```

At this point you should be able to define symbols, but you can't interpret them...yet.

Notice that braces (`{}`) can be used to make the `Reader` "shrink-wrap" a series of `Tokens` into a *procedure* `Token`. Verify that when your interpreter sees a procedure, it is doing the correct thing. Use `gs` to figure out what this is:

```
{ 1 2 add } pstack
```

8. Now, if you don't recognize an unquoted token as a builtin command (like `add`), make one last check to see if it is a symbol with an associated `Token` in the symbol table. If you find a definition, then think of it as a very small program, and recursively interpret it using a *new* `Reader`.

   This one step will allow you to write procedures, like `area`, that you can execute.

9. Implement `if` and `ifelse`. These remove a condition and one or two procedures from the stack. They optionally interpret those procedures, interpreting a new `Reader` if the condition is appropriate.

   This step will allow you to make decisions and write recursive procedures!

10. Test your interpreter. We've provided a number of little postscript programs in the subdirectory `samples`. You can run these by *redirecting* the contents of one of the files into `gs`:

    ```
    gs -DNODISPLAY <samples/0-pstack.ps
    ```

    or, alternatively, into your interpreter:

    ```
    java Interpreter <samples/0-pstack.ps
    ```

    Verify the output from the two interpreters are consistent.[2] You should feel free to write new tests of your interpreter and place them in the `samples` directory.

    At this point you have the potential to earn the fullest credit.

If you're interested in writing other operators, you can learn more about PostScript from Adobe's *PostScript Language Reference Manual*.[3] If you do, please include examples their use in the `samples` directory, and make sure that any add-ons also work with the `gs` interpreter.

Submitting Your Work. To get credit for this week's lab make sure that you've added, committed, and pushed the files `Interpreter.java` and `honorcode.txt` (containing your signature) as well as any new sample scripts. Recall that `git status` and `git push` will help to verify that everything is up-to-date.

⋆

[2] Be aware: tests containing `ptable` can be helpful in debugging your interpreter, but will not run in PostScript.

[3] `https://www.adobe.com/jp/print/postscript/pdfs/PLRM.pdf`