Handout 9
CSCI 136: Spring 2017
05 April

Lab 6

Due 10 April

P.S. It's Just a Stack

1 Short Answers

Include answers to the following problems in the comments at the top of your lab submission. The questions ask you describe how you would modify linear data structures to complete a task, but you do not need to write any code.

Problem 10.3
Problem 10.4
Problem 10.5

2 Lab Program

This week we will implement a small portion of the stack-based language Postscript, a language designed to describe graphical images. When you create a Postscript file or print to a PostScript printer, you actually create a file that contains a program written in this language. A printer or viewer interprets that program to render the image described by the file. The lab is described in Section 10.5 of *Bailey*.

Read the assignment and prepare a design for the program before lab so that you can start working right away. You do not have to hand in your design, but by planning in advance, you will be able to maximize your productivity (and maybe even finish lab before you leave for break).

3 Notes

1. The starter files and javadoc documentation are on the handouts page. Save these files to your lab directory and familiarize yourself with the files *before* starting to work. In addition to starter Java files, we have included several sample postscript programs that will be useful for testing your code.
2. Make use of the functionality of the classes you are given. Be careful not to spend time developing code that is already there!
3. Name your interpreter class `Interpreter`. You should only need to modify the `Interpreter` class, and nothing else. Your program should read commands from standard input. You can also redirect input from a file by using a command like `java Interpreter < sample.ps`.
4. Make your `main` method very short. All it should do is create an `Interpreter` object and tell it to start parsing the postscript program presented at the command line. Create a method `interpret` that takes a single parameter of type `Reader` and processes the PostScript tokens returned by that `Reader`.
5. Develop your `interpret` method incrementally. Get your simple push, pop, and pstack operations working, then move on to the arithmetic operators, and finally the definition and usage of symbols. **Decompose the program into small, manageable helper methods as you go.**
6. Your program should report errors when it encounters invalid input, but these should contain meaningful error messages. You can use `Assert.condition()` and `Assert.fail()` for this.

7. Implementing the basic operations– `pstack`, `add`, `sub`, `mul`, `div`, `dup`, `exch`, `eq`, `ne`, `def`, `pop`, `quit`, and `ptable`– will allow you to earn 18 out of 20 points. You can earn the last two points by implementing the extensions outlined in thought questions 3, 4, and 5 from the book. In particular, you should implement *procedure* definitions and calls, and the `if` instruction. These extensions may require a little thought, but ought to be straight-forward to implement if you have designed your interpreter engine well.

4 Submitting your work

Create a lab directory in the standard way, and turn in your well-documented `Interpreter.java` file before your lab section's designated deadline. Include your short-answer questions as comments in the top of this file.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a your name and a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.