Lab 6: P.S. it's a stack

Due 11:59 pm Monday, Mar 31, 2008

1 Assignment

An interpreter is a program that executes other programs. For example, the Java Virtual Machine that you run using the "java" program is an interpreter that executes .class files, which contain Java byte code.

Your assignment is to implement an interpreter for the subset of the PostScript language described in section 2 of this document. To submit your solution, use the commands:

```
tar –cvf ps.tar *.java
turnin –c 136 ps.tar
```

Your implementation must support:

1. Input from standard input (System.in)
2. The following standard PostScript operators:
   add, sub, mul, div, dup, exch, eq, ne, def, pop, gt, lt, ifelse, if, quit, clear, exec, pstack
3. The non-standard operator "ptable" that prints out the contents of the variable table.
4. Input, output, and processing of boolean values and numeric values.
5. Procedures, which are denoted with braces, {…}, in PostScript.

Your interpreter must be in a file named PS.java, which must have a main method that can be used to execute the interpreter. You will need several other files to complete this assignment.

The Unix command "gs" runs the Ghostscript PostScript interpreter. Since this is just a more complicated version of the program that you're writing, you can use it to test small PostScript programs to see if your implementation is correct. Your program's output should be identical (except for printing <GS> and <#> all over the place). Remember that you have to type "pstack" to actually see the values that you've computed.

You can read more about the PostScript language at the following locations. You are encouraged to use these and other PostScript resources on the Internet and in the library to expand your understanding of postfix notation.

- Chapter 10.5, pages 246—250 of the textbook
- http://en.wikipedia.org/wiki/Reverse_Polish_notation
- http://en.wikipedia.org/wiki/PostScript
- http://www.tailrecursive.org/postscript/operators.html
- http://www.adobe.com/products/postscript/pdfs/PLRM.pdf

Only the Adobe document should be considered authoritative.

This week only, it is acceptable for your program to crash as a result of illegal input on the part of the user. This means that, for example, you can freely cast from Object to Double if the only way that the argument could have not been a Double was through user error.

Lab will begin with a graded exercise intended to help you with the assignment. You will be assigned a teammate at the beginning of lab and will have approximately 30 minutes to work on it. Both lab partners will receive the same grade for that portion of the assignment (don't worry, I'll grade it very leniently--this isn't intended as a test). You are free to use or not use the results of that exercise in your own lab.

2 PostScript Subset Specification

2.1 Evaluation

PostScript maintains a stack of values called the operand stack. Operands are always popped from this when executing an operator or procedure, and the output of an operator or procedure is always pushed back onto this stack. PostScript accepts as input an ordered series of commands and values. As each is entered, it is "**pushed**" onto the interpreter as described by the following pseudo-code:

```
PS.push(value v)
    if v is an identifier:

        if v is a variable that has previously been defined:
                let x = the value of that variable
                exec( x )

        else if v is the name of an operator:
                Apply the operator
    else:
        Push v onto the operand stack


PS.exec(value v)
    if v is a procedure:
        for each value x in v:
                push( x )
    else:
        Push v onto the operand stack
```

The push and exec routines are mutually recursive. This is what allows PostScript procedures to call other procedures. Be careful to distinguish between pushing onto the intepreter and pushing onto its operand stack. Real PostScript performs proper tail recursion, so that it does not allocate memory just for executing a procedure. You do not need to do that! Just transform the above code into Java methods.

2.2 Operators

Because each operator reads its operands off the stack, it always appears to the right of the operands in this description. Boldface type indicates the operator, italics are the type of the operands. *Value* means any of the legal operand types. All operators remove their arguments from the stack before adding new values.

symbol value **def**
Binds the variable whose name is the symbol to the value.

procedure **exec**
Execute the procedure. Note that the procedure must have been entered using { … }; it cannot be the value of a variable, since typing the name of the variable would immediately execute the procedure it references.

number number **add**
Produces the sum of the two numbers.

number number **mul**
Produces the product of the two numbers.

number-b number-a **sub**
Produces a – b.

number-b number-a **div**
Produces a / b.

value **dup**
Places the value on the top of the stack twice.

boolean procedure-then procedure-else **ifelse**
If the boolean is true, executes the *then* procedure, otherwise executes the *else* procedure.

boolean procedure **if**
If the boolean is true, executes the procedure.

value-b value-a **exch**
Makes the top of the value stack into: *a b*

value **pop**
Discards the value.

value value **eq**
If the two values are Object.equals, pushes true onto the top of the stack, otherwise places false on the top of the stack.

value value **ne**
If the two values are not Object.equals, pushes true onto the top of the operand stack, otherwise places false on the top of the stack.

number-b number-a **gt**
Returns true if $a > b$

number-b number-a **lt**
Returns true if $a < b$

clear
Removes all values from the stack.

pstack
Displays the value stack with one value per line, with the next value to be popped at the top and the last value to be popped at the bottom. (Note that this is the opposite order that java.util.Stack's iterator produces values, so if you use that class you must figure out how to iterate in the opposite order.)

quit
Terminates the interpreter.

3 Evaluation

In-lab Exercise	**10**
Correctness	**15**
Readability	**20**
Design	**20**
Efficiency	**10**
Total	**75**

3.1 Extra Credit

You may receive extra credit on this assignment if you extend it beyond the specification (any additional Java code that you write will be considered when grading your code for readability and design, so be careful that you don't lose points by adding sloppy code at the end!) You should not work on extra credit unless you've already completed the lab with time to spare.

Here are three suggestions for extra credit projects:

1. Implement some interesting recursive functions in PostScript as .ps files, like fibonacci or isPrime. Note that in PostScript all variables are global, so the way to pass arguments to a procedure is to

place them on the stack and have the procedure pop them off itself. See factorial.ps for an example.

2. Implement factorial without using the **def** command. Hint: you're going to need **exec** (or, in an extreme hack, **if**).

3. Produce a .ps file that, when viewed using a PostScript preview program or sent to a PostScript printer using the Unix command "lpr –P*printername filename*", produces an interesting image. Note that you need to insert some comment headers to make legal PostScript for printers; see the cited documents. Your interpreter does not have to support this file!

4. Extend your PostScript interpreter with some of the other commands described in the full specification.

5. Rewrite your PostScript interpreter (as a different file, PS2.java) so that it performs the proper tail call optimization. This is really challenging. You need to avoid using Java recursion and instead process commands (and procedures) using an explicit stack.

4 Advice

This lab is going to give you a lot of experience with iterators, interfaces, inheritance, and design. This will serve you well in upcoming labs that involve a lot of classes. Writing a PostScript interpreter is making a "real" application. In fact, the structure of your program is very similar to the Java Virtual Machine that runs your Java programs, and is a simplified version of what is running on almost every printer you've ever used. You may be surprised to learn that the PostScript interpreter that you're writing is just as powerful as Java, because any computation that could be expressed in Java could also be expressed in PostScript (albeit in a less pleasant way).

I provided the following classes for you:

class Identifier	Represents PostScript commands and variables inside your program
class Symbol	Represents PostScript symbols, which begin with / and are used only when defining new variables
class Procedure	Represents a PostScript procedure, as produced by StreamIterator. Note the iterator() method; you can use the special for-loop iterator syntax with this.
class StreamIterator	Parses the input into Boolean, Double, Symbol, Identifier, and Procedure objects. This is both an Iterator and Iterable, so you can use the special for-loop iterator syntax with this.
factorial.ps	Example of an interesting procedure
inc.ps	Example of a simple procedure
test.ps	Test basic arithmetic operations

Recall that you can use the javadoc136 command to generate HTML documentation for these an your own classes, which will be convenient when referring to them.

Start this lab right away. Of the labs that you've done so far, this was the one that took me the longest to implement, even though it had less code than previous ones. Expect to do a lot of thinking (maybe away from a computer) and relatively little programming. Then, expect to do a lot of debugging if your thinking wasn't exactly right the first time.

Get basic operations like pstack, add, and pushing numbers working correctly first. Then implement def and variables. Work on if, ifelse, and procedures last; they are the trickiest.

My PS.java is 125 lines of code and contains 10 methods, as described below. I have very few comments because I felt that the above PostScript specification and my helper methods made the structure clear. If your PS.java is not between 100 and 250 lines of code you're probably going about things in a bad way and should rethink your plan. Talk to other students, the TAs, and me about your ideas.

You can send a file into the standard input pipe of any program using the < operator on the Unix command line, like this:

```
java –ea PS < test.ps
```

The effect of this command is the same as if you had typed all of the contents of test.ps after running PS. You can send those same programs to gs for debugging as well. I provide several test programs for you, and recommend that you write some of your own. When grading I will use a different set of test programs to automatically verify the correctness of your implementation.

I implemented my interpreter using the following design (note the PS class on the next page). Yours might differ; there are many ways of doing this. I'm only showing you the public methods; there are some more private variables and methods. I've also removed most of the comments because I want you to think through the design yourself. I am showing you *all* of the public methods that I used, though!

```
/**  Interface for operators in postscript, like add, sub, dup, etc. */
public interface Operator {
    /** Applies this operator to the operands currently on the stack. */
    public void apply(PS interpreter);

    /** The name that this operator is bound to, e.g. "add" for the AddOperator.*/
    public Identifier getName();
}
```

```
import java.util.Iterator;
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;

/** Limited PostScript interpreter.

   <p>Morgan McGuire
   <br>morgan@cs.williams.edu */
public class PS {

    /** Used by pushIdentifier() to convert operator names into the actual operators. */
    public Map<Identifier, Operator> operatorTable = new HashMap<Identifier, Operator>();

    public PS() {
        loadOperators();
    }

    private void loadOperators() {
        assert operatorTable.size() == 0 : "Operator table is already initialized";

        Operator[ ] opArray =
            {new AddOperator(), new SubOperator(), new MulOperator(), new DivOperator(),
             new DefOperator(), new DupOperator(), new GtOperator(),  new NeOperator(),
             new EqOperator(),  new LtOperator(),  new IfOperator(),  new ExecOperator(),
             new ClearOperator(), new IfElseOperator(), new ExchOperator(),
             new PopOperator(),   new PTableOperator(), new PStackOperator(),
             new QuitOperator()};

        for (int i = 0; i < opArray.length; ++i) {
            Operator op = opArray[i];
            operatorTable.put(op.getName(), op);
        }
    }

    public int count();

    public Object pop();

    /** Place a value on the operand stack or execute a command. If the value is a variable
        that references a procedure, the procedure will be executed.  If the value is a procedure that
        procedure will be put on the operand stack as a value.*/
    public void push(Object value);

    public void exec(Object value);

    public void def(Identifier var, Object val);

    public void ptable();

    public void pstack();

    /** Read input from stdin and process it. */
    static public void main(String[] arg) {
       // Only four lines of code here…
    }
}
```