1 Prelab

Read pages 362 through the end of Chapter 7 in Main and Savitch.

2 Objective and Introduction

Gain experience with postfix expressions, their meaning, and how they specify a computation in the context of Adobe Postscript. From Adobe’s Postscript Language Reference, 3rd ed., Postscript is:

- A general-purpose programming language with powerful build-in graphics primitives.
- A page description language that includes programming features.
- An interactive system for controlling raster output devices (printers and displays).
- An application- and device-independent interchange format for page descriptions.

Every Postscript file is a computer program written in the Postscript language. Every Postscript printer or viewer application (software such as Ghostscript) has an interpreter built into it that runs Postscript programs. The Postscript language is designed for driving printers efficiently and precisely rather than (like C++) enabling humans to write easy-to-understand code. However, like C++, it is a complete programming language, which means we can write programs in Postscript that are equivalent to programs written in any other programming language. (Two programs, possibly in different languages, are equivalent means that they specify the same computations, roughly. The precise meaning of a “complete programming language” is developed in CSI409 at Albany: Every program in the complete language is equivalent to some program in the language of Turing Machines, when we neglect memory space limitations.)

Software (such as a Postscript printer driver), not people, generally produce Postscript program files. However, most Postscript interpreters (software or built into printers) have a “executive” feature with which a person can operate the Postscript interpreter interactively. You type some Postscript commands and the executive makes the interpreter run them right away. It is much like using a calculator.

In this lab, you will type the “==” operation to command the interpreter to print a value back to the terminal so you can see it. You will also use the “stroke” command to make the interpreter draw lines on a separate graphics output window.

Software that outputs Postscript program files (to make a printer print a document) typically copies pre-written Postscript procedures into the output followed by
data and commands for printing the document’s content. The original copies of these procedures were written by human software developers who have to be skilled in using the Postscript language. A few other people program in Postscript directly to create unusual graphics. Sometimes system administrators have to troubleshoot or configure Postscript printers, possibly by sending them Postscript commands. The “executive” feature is useful to everyone who tests, debugs and experiments with Postscript code.

In this lab exercise, we will barely touch upon the capabilities of Postscript. We emphasize the stack oriented postfix syntax nature of the language, and essentially ignore its very sophisticated and powerful Adobe imaging model.

3 Impressing Yourself

On ITSUNIX in the lab or any X-window system session, start the Ghostscript “Postscript-compatible” interpreter with the command:

```
gs
```

At the prompt, type the Postscript code:

```
(/home1/c/a/acsi310/Lab6/lab6.ps) run
```

(and press enter). Follow the instructions! Now, to see what program this code loaded and ran, make Ghostscript exit with the quit command and run (from the shell):

```
less /home1/c/a/acsi310/Lab6/lab6.ps
```

(Type letter q to make less quit.)

4 Basic Postscript

Start the gs interpreter again, as above. Observe the “GS>” prompt.

1. Type any decimal number, with a decimal point or not, and press enter. Observe that the prompt changes to “GS<1>”. Type another number. What happens?

   The number within the <..> tells you how many items there are currently in the operand stack.

2. The pstack command prints the contents of this stack, from top to bottom. Try it! Type a few more numbers and try pstack after each one.

3. The => command pops (removes) the top entry from the operand stack and prints it. Try it over and over until you think the stack is empty. Then try it again to see what happens. Then try pstack again.

4. Push a few numbers on the stack. Then, try out some of the arithmetic operations: add, sub, mul, div, neg, sqrt. Observe the whole stack after each command. Do this until you fully understand what operations these commands make the interpreter do.

   Try some arithmetic operations like add when the stack is empty or only has one entry. What happens?
5. Use Ghostscript to evaluate postfix expressions from page 372 of Main and Savitch. Make it print the final value with the == command. Notice that the value in the stack which was computed by the arithmetic operators is popped away when it is printed by ==.

6. Get some numbers into the stack. Give the dup command. Observe the stack with pstack. What does dup do?

7. Get an empty stack with the Postscript clear operation. Now, use Ghostscript to evaluate the arithmetic expression from pages 344 and on of Main and Savitch:

\[ (((12 + 9)/3) + 7.2) * ((6 - 4)/8)) \]

To do this, you will have to “parse” this infix expression to figure out which operation is first, etc. A similar problem and solution is traced on the pages following page 344.

8. After you figured out a Postscript expression that evaluates the above infix expression and then prints the final value of 3.55, type it into a file named lab6ans.ps and save the file.

Make Ghostscript read and execute your lab6ans.ps file with the command:

```
/lab6.ps run
```

to verify that it calculates and prints the correct arithmetic result of 3.55

9. Postscript procedures generally

   - use arguments that must be put in the operand stack before the procedure is called.
   - They remove their arguments after using or copying the arguments, and:
   - They put any return values on top of the stack for use by future operations or procedures

Here is an example of a procedure that shows how it is defined. The example uses one argument, it multiplies it by 3, prints that product, and then adds 1 and returns the sum. It’s admittedly artificial but illustrates basic Postscript techniques:

```
/FunFun { 3 mul dup == 1 add } def
```

(the def operation makes the name /FunFun become the name of a procedure.)

The interaction with Ghostscript below illustrates how to call this function:

```
GS>pstack
GS>5 FunFun
15
GS<1>pstack
16
GS<1>FunFun
48
```
GS<1>pstack
49
GS<1>

Try this example, and try calling FunFun when the stack is empty, or similar examples of your own creation.

10. Postscript procedures and built-in operations generally remove from the stack the arguments they use.

A useful Postscript operator for copying specified procedure arguments so they can be used multiple times is index. It pops the positive or zero integer n from the top of the stack, and then copies the element now n positions down from the top and pushes it onto the top. So, if you know the element you want is at position n below the top, you can retrieve a copy for use by further operations with the code “n index”

Try this: Put some numbers in the stack, e.g.: 
GS>100 200 300 400 500 600
GS<6>2 index pstack
400
600
500
400
300
200
100
GS<7>

How would the results differ if we did GS<6>2 index == pstack instead?

11. Now let’s do some graphics. The moveto operator pops two numbers from the stack and makes the “current point” be the point with those two numbers as coordinates.

The lineto operator pops two numbers and appends to the “current path” the line segment from the current point to the point with the two popped numbers as coordinates.

The stroke operator doesn’t affect the operand stack. Instead, it “paints” along the current path and then re-initializes the current point and path to “undefined”. stoke or another painting operator is necessary to make graphical output visible.

Figure out and try a combination of one moveto, one lineto and one stroke to paint one line. Use x coordinates in the 0-600 and y coordinates in the 0-790 range. (The default units are printer’s points, about 1/72 inch.)

Then try to paint a zig-zagged shape using multiple lineto operations followed by a stroke.
12. Try erasepage.

13. Create a path with 2 or more lines and give the fill operation instead of stroke to paint it. What is the difference?

14. For one lab checkoff credit: Add Postscript code to your lab6ans.ps file that defines a procedure to draw something interesting and then to call the procedure. Verify in a fresh Ghostscript session that when you run your lab6ans.ps file, the correct numerical result is printed and your interesting graphic figure is visible. Then, submit it to project lab6 for the credit.

FYI: If you have access to a Postscript printer, you can make your printer print the graphics from your .ps file by putting the characters %! in one line at the top and showpage on one line at the bottom.

15. For an additional credit, develop a recursive Postscript procedure to generate a random fractal using the algorithm and design presented on pages 429-435 of Main and Savitch. More operators and techniques sufficient to do this are described below. To get the credit, submit your Postscript file named fractal.ps to draw the fractal when it is run by gs as above. Submit it together with your lab6ans.ps file to project lab6

Important hint: Main and Savitch’s random_fractal function has another important postcondition: After the fractal is displayed, the output device is set so the next future call of display will draw the right endpoint of the next segment just after the previous segment properly.

5 More Postscript Operators

1. To put a random floating point number between 0.0 and 1.0 on top of the stack, use the code:
   rand 16#7FFFFFFF div
   (This pushes a random integer between in the 0 to $2^{31} - 1$ range, then $2^{31} - 1$ and then it divides them.)

2. The Postscript operator currentpoint pushes the x (horizontal) and then the y (vertical) coordinate of the current point onto the operand stack.

   Hint: Design your recursive Fractal procedure so that whenever an activation returns, the current point is the right hand endpoint of the fractal curve that this activation had created. This is the analog of the implicit, omitted postcondition of Main and Savitch’s C++ fractal function.

   Hint: Design your Fractal procedure so that it produces graphic results only by calling the lineto operator. Remember the lineto operator merely appends a line segment to the “current path”, without painting anything. Then, the code with the top level call should do an appropriate moveto first, and, after Fractal returns, do a stroke to actually paint the fractal.
3. The operator le pops two numbers, first \( n_2 \) and then \( n_1 \). If \( n_1 \leq n_2 \), it pushes the boolean value true; otherwise it pushes false. Other analogous relational operators are eq, ne, ge, gt, and lt.

4. Conditionals: Suppose a boolean value (true or false) is on top of the stack. Given Postscript code: { some Postscript code } if the boolean value is popped. If it is true the conditional code is executed. Otherwise it is not.

There is also an ifelse operation:
{ if clause code } { else code } ifelse

Example: Here is a recursive procedure to compute the factorial function and the code to demonstrate its use:

```
/Fact { dup % Stack contains n n (we write the top on the right)
  2 ge % Compare top n to 2; Stack is now: n bool.
  % SEE THE ifelse OPERATOR BELOW. IT will pop the bool.
  { % This code runs with one copy of n on top, with n>2. 
    dup 1 sub % Code for n>2 begins. Stack is now: n n-1 
    Fact % Recurse! Stack now: n (n-1)!
    mul % Compute n*(n-1)! on top of stack.
  }
  % End of the "if" clause
  { % else, n<=1, so,
    pop % pop n;
    1 % Push the 1 we will return when n==0 or n==1
  }
  % End of the "else" clause
  ifelse % ifelse operator
} % End of the Fact procedure body
def % operator to define Fact to be this procedure.
```

4 Fact == % Call Fact on 4, pop and print the result.

Tip: Postscript operators (and numbers) can be typed on any number of lines for clarity. Furthermore, the Postscript comment character % functions just like // in C++ so you can put comments at the end of each Postscript line.

On the next page is a Postscript program for calculating Fibonacci numbers which you can use as a pattern for the doubly-recursive algorithm coded in Main and Savitch on page 434. It is fully commented so you can pick up some more Postscript programming skills. It shows that literal strings in Postscript are delimited by parentheses (like this) rather than double quotes used in C/C++ "like that". It’s available on ITSUNIX at ~acs310/Lab6/fib.ps and from the course web site.
/DebugFibCall % Prints how Fib is called when called at the beginning of Fib.
{ % n
  (\nCalling Fib\() % n "\nCalling Fib"
  print % n (The string was printed)
  dup % n n (a shorter alternative to 0 index)
  = % n (n was printed)
  (\)...\n) % n "\)...\n"
  print % n (The 2nd string was printed)
} def
/DebugFibRet % Prints what Fib returns when called at the end of Fib.
{ % Fib(n)
  (\nFib returning:) % Fib(n) "\nFib returning:
  print % Fib(n) (The string was printed)
  dup % Fib(n) Fib(n) (a shorter alternative to 0 index)
  = % Fib(n) (The top copy of Fib(n) was printed)
} def
/Fib { % Comments document stack contents: n
  DebugFibCall
dup % n n
  1 % n n 1
  le % n "is n<=1?"
  { % (code to run if Yes.) Stack: n
    pop % Stack:
    1 % 1 (Push fib(0)=fib(1)=1 to return it.)
  } % (End of the "if" clause.)
  { % (code to run if "is n<=1?" is No.)
    Stack: n
    0 index % (alternative to dup) n n
    1 % n n 1
    sub % n n-1
    Fib % (Recursive call) n Fib(n-1)
  1 index % n Fib(n-1) n
  2 sub % n Fib(n-1) n-2
  Fib % n Fib(n-1) Fib(n-2)
  add % n (Fib(n-1)+Fib(n-2)=Fib(n))
  2 1 roll % Fib(n) n
  % This use of roll "rotates" the top 2 stack elements 1 space
  % to the right.
  pop % Fib(n) (which is the value to return)
} % (End of the "else" clause)
ifelse % DONT FORGET TO CODE ifelse AFTER the clauses!
  DebugFibRet
} % End of the Postscript code array for the Fib procedure
def % Operator that defines Fib's value to be this code array.
The rest of the fib.ps file contains code to print messages, call the Fib procedure 
a few times, and print each return value:

\(\text{\textbackslash n}\text{Testing Fib(0):\textbackslash n}\) print  
0 Fib ==  
\(\text{\textbackslash n}\text{Testing Fib(1):\textbackslash n}\) print  
1 Fib ==  
\(\text{\textbackslash n}\text{Testing Fib(5):\textbackslash n}\) print  
5 Fib ==

Here is an outline of the Postscript file to define and call your Fractal function. The paper copy of this assignment also includes the graphic output of drawing one fractal plus results from using Fractal 250 times to draw overlaid fractals in different 
grey-scale colors. Include the \% at the top if you want to try it in a printer.

\%
/Fractal { %leftth rightth width epsilon
1 index 1 index %leftth rightth width epsilon width epsilon
le
{  
%leftth rightth width epsilon
%%%%%%%%%%%%%%%%%%%%%% YOU WRITE!!! %%%%%%%%%%%%%%%%%%%%%%
%(stack is empty)
}
{ %leftth rightth width epsilon
%%%%%%%%%%%%%%%%%%%%%% YOU WRITE!!! %%%%%%%%%%%%%%%%%%%%%%
Fractal
%%%%%%%%%%%%%%%%%%%%%% YOU WRITE!!! %%%%%%%%%%%%%%%%%%%%%%
Fractal
%%%%%%%%%%%%%%%%%%%%%% YOU WRITE!!! %%%%%%%%%%%%%%%%%%%%%%
%(stack is empty)
}
ifelse
} def
0 395 moveto 395 395 612 4 Fractal stroke
%analogous to: random_fractal(395, 395, 612, 4); showpage

Link to downloading the PostScript Cookbook and Tutorial (Blue Book):
http://partners.adobe.com/asn/developer/pssdk/download/samplecode/ps_psbooks
Links to downloadable Technical Documents, including the Postscript Language Reference Manual, 3rd edition. The same page also has marketing and introductory documents.
http://www.adobe.com/products/postscript/resources.html#white
More links which include the Language Reference Manual and other technical
notes:
http://partners.adobe.com/asn/tech/ps/index.jsp
http://partners.adobe.com/asn/tech/ps/technotes.jsp