Mathematics 308—Fall 1996

Basic PostScript

PostScript is a programming language which is **stack-based** and **interpreted**. It is primarily used for graphics and text-processing, but in fact it is a general programming language, somewhat reminiscent of the one used to program HP calculators, It can be used as a simple calculator without trying to draw anything at all. There are several interpreters available for it. Some are included in various program packages from Adobe, the company that introduced PostScript. These run on all platforms and look beautiful, but they cost money. There is also one in the public domain called ghostscript, which is available on a variety of machines, including PC's running MSWindows and recent Macs. Once installed, ghostscript is called by typing **gs**.

1. Reverse Polish notation

Calculations in PostScript take place according to the conventions of **reverse Polish notation** or **RPN**, which means that

- *In* PostScript *operators are applied to the numbers just preceding them.*

For example, the simple expression $3+4$ becomes in RPN $3\ 4\ +$. This notation has the advantage that it eliminates the need for parentheses. Here are some more simple examples:

Expression	RPN form
$3+12$	$3\ \ 12\ \ +$
$3-12$	$3\ \ 12\ \ -$
$3*12$	$3\ \ 12\ \ *$
$3/12$	$3\ \ 12\ \ /$
$(8+5)*7$	$8\ \ 5\ \ +\ \ 7\ \ *$
$7*(8+5)$	$7\ \ 8\ \ 5\ \ +\ \ *$
$3+7*(5+8)$	$3\ \ 7\ \ 5\ \ 8\ \ +\ \ *\ \ +$

One curious fact is that all forms of calculation probably take place implicitly in this fashion, and that RPN therefore has some claim to being more natural than the algebra we use more commonly. For example, when we see a complicated expression like $(3+7*(5+8))$ we presumably first scan it from left to right until we have seen how the grouping by parentheses takes place, and then do the calculation, here perhaps in a kind of backwards pass. In effect, we probably construct in our heads something like the RPN version of the expression before we evaluate it. At any rate, computers generally do exactly this when they evaluate arithmetic expressions. The advantage of using RPN directly is that there expressions can be evaluated on a strict left-to-right scan, with no looking ahead necessary to understand grouping—the calculations are **local**, involving only neighbouring terms.

2. The stack

The **stack** is where the data in an RPN expression is held until all calculations are done. It is an array whose length grows and shrinks as calculations proceed. It grows at the end called the **top**. For example, in evaluating

the expression $3 + 7 * (5 + 8)$ here is a trace of how the stack looks:

3
$3\ 7$
$3\ 7\ 5$
$3\ 7\ 5\ 8$
$3\ 7\ 5\ 8\ +$
$3\ 7\ 13$
$3\ 7\ 13\ *$
$3\ 91$
$3\ 91\ +$
94

- *All calculations take place on the stack, and on the whole calculations affect only the numbers at the top.*

A number is put on the stack by entering it. Two or more numbers must be separated by spaces (including carriage returns). An operation is performed by entering its name, and this will usually immediately replace the numbers affected by the result of the operation. At any point in a PostScript calculation we can see what is on the stack by entering the command **`stack`**, which displays everything on the stack from top to bottom, and does not change any of the items there.

For example, suppose you want to calculate the length of the vector $(9, 7)$, which is

$$\sqrt{9 * 9 + 7 * 7}\ .$$

The following is a record of a ghostscript session that does this, with some extra stuff showing how things are proceeding.

```
GS>9 9 mul
GS<1>stack
81
GS<1>7 7 mul
GS<2>stack
49
81
GS<2>add
GS<1>stack
130
GS<1>sqrt
GS<1>stack
11.4018
GS<1>
```

The program ghostscript has a prompt for input that looks like this:

```
GS<2>
```

which means that there are two items on the stack. In PostScript, operations + etc. are words **`add`** etc. In this example, we first put two copies of the number 9 on the stack and multiply them together; the copies of 9 disappear and the number 81 then sits on the stack. We then do the same for the number 7, and then both 49 and 81 are on the stack. We then add the two numbers on top of the stack by entering the command **`add`**, which replaces the top two numbers by their sum. By entering **`sqrt`** we replace this in turn by the square root of the number on the top of the stack.

It is not necessary to display results as we go. Here is a more succinct run that does the same thing.

```
GS>9 9 mul 7 7 mul add sqrt
GS<1>
```

3. Beginning the PostScript dictionary

As the course proceeds, we shall build up a dictionary of PostScript commands. Here are some of the ones we have used so far, and a few more.

`%!`	must appear as the first two characters of every PostScript program
`newpath`	starts a path
`x y moveto`	moves the current point to (x, y)
`x y lineto`	draws a line to (x, y)
`x y rmoveto`	shifts the current point by (x, y)
`x y rlineto`	draws a line with relative coordinates (x, y)
`stroke`	draws the current path
`fill`	fills the current path (which must be closed)
`clip`	restricts the area in which drawing occurs to inside the current path
`closepath`	closes up the path
`r g b setrgbcolor`	sets the current colour in **RGB** format
`g setgray`	sets the current level of gray: 0 is black, 1 is white
`w setlinewidth`	sets the current line width. Default is 1 unit.
`0/1/2 setlinejoin`	sets style of joining lines
`0/1/2 setlinecap`	sets style of capping lines
`sx sy scale`	multiplies the current x and y scales by sx and sy
`x y translate`	translates the origin
`theta rotate`	rotates the current drawing map by θ
`x y add`	puts $x + y$ on the stack
`x y sub`	puts $x - y$ on the stack
`x y mul`	puts xy on the stack
`x y div`	puts x/y on the stack
`x neg`	puts $-x$ on the stack
`y x atan`	puts the polar angle of (x, y) (in degrees)
`x sqrt`	puts $\sqrt{x}$ on the stack
`x sin`	puts $\sin x$ on the stack (x in degrees)
`x cos`	puts $\cos x$ on the stack (x in degrees)
`y x exp`	puts y^x on the stack
`x ln`	puts $\ln x$ on the stack
`x abs`	puts $\lvert x \rvert$ on the stack
`/x ... def`	defines `x` to be whatever is on the stack in place of `...`.
`showpage`	changes page and causes the current page to be printed

4. Remarks about running GhostScript

GhostScript, the PostScript interpreter we shall use in this course, is started by typing

```
gs
```

and when you do that you will get a sequence that looks like this:

```
Initializing...  done.
Ghostscript 2.6.1 (5/28/93)
Copyright (C) 1990-1993 Aladdin Enterprises, Menlo Park, CA.
  All rights reserved.
Ghostscript comes with NO WARRANTY: see the file COPYING for details.
GS>
```

and a window will pop up on the screen. The program is now waiting for you to type in PostScript commands. You can type them in one by one, and they will be interpreted one by one. Or you can read in a file with a command like

```
(test.ps) run
```

which causes **gs** to load your file and run it.

By the way, to avoid confusion later on, it is a very good idea to name all your PostScript files with **.ps** extensions.

If things go wrong in **gs** you will get a lot of garbage on the screen. **It is not easy to understand this garbage!** Usually you will have made a simple error in typing, in which case near the beginning of the garbage you will probably get a message about something being **undefined**. this is perhaps the most common error. One thing to be careful about is distinguishing capital from small letters. Another common error is **stackunderflow** which means that you have forgotton to put something on the stack before a command. For example if you have the sequence

```
4 add
```

you will get stack underflow because **gs** is expecting two numbers to add, not one. Here is a sample:

```
Error:  /stackunderflow in exch
Operand stack:
    /s
Execution stack:
   %interp_exit --nostringval-- --nostringval-- --nostringval-- %loop_continue -
--nostringval-- --nostringval-- false --nostringval-- --nostringval-- --nostringval-
-  false  --nostringval--  --nostringval--  --nostringval--
Dictionary stack:
    511/547  0/20  15/200
Current file position is 138
GS<1>
```

Here **exch** was expecting two arguments on the stack but only got one. You will usually want to start over after error garbage on file input by typing **clear**, which empties the stack. You will probably want to run **gs** with both an editor window and the **gs** window visible at the same time.

The last prompt illustrates something useful about **gs**—the prompt **GS<1>** tells you the depth of the stack. Sometimes this is very useful. For example we have this sequence:

```
GS>2
GS<1>3
GS<2>add
GS<1>=
5
GS>5
GS<1>dup =
5
```

```
GS<1>
```

which illustrates this and also shows you one of the simplest debugging tools you have: typing **=** will show you what is on top of the stack—but it will remove it as well. If you type **dup =** you will make a copy on top which you can throw away.

You can look at the whole stack without changing it by **stack**.

```
GS>1
GS<1>2
GS<2>3
GS<3>4
GS<4>stack
4
3
2
1
GS<4>
```

There is a variation on **=** which is **==**. It behaves a bit differently on complicated objects.

```
GS<4>{add sub mul}
GS<5>dup =
--nostringval--
GS<5>dup ==
{add sub mul}
GS<5>
```

In effect, running **gs** interactively you have a way to try out drawing on the screen interactively. You can save stuff you type in such a fashion, say to to a file, by selecting from the **xterm** window and then pressing the middle mouse in an editor window. Of course this will give you a little extra junk to get rid of in your editor window.

You exit from ghostscript by typing **quit** or the keys **control-C**.

5. The types of data in PostScript

The basic types in PostScript that you will usually work with are **integers** are **real numbers**. Integers range from -32768 to 32767 in size. Thta's not a very large range. Real numbers are in floating point format, and handle up to about 7 digits of accuracy. Overflow is not uncommon, and rounding errors can be a nuisance. Normally integers are converted to real numbers if necessary. There is a difference between an integer and a rounded real number. For example, in loops you must use integers to count repeats. To convert from real number to integer you use a conversion **cvi**. Thus

```
8.7 round cvi
```

leaves the integer 8 on the stack.

Later we shall also see **arrays**, **procedures**, and **strings**. Actually we have already seen one string—in the sequence **(test.ps) run** the file name **(test.ps)** is a string.

6. Subroutines in PostScript

We can draw a square with the sequence

```
newpath
0 0 moveto
a 0 lineto
a a lineto
0 a lineto
0 0 lineto
closepath
stroke
```

but if we want to draw several squares this involves a lot of typing. It is more efficient and flexible if we have a subroutine to do the drawing, so that in the middle of a long program we can just type

```
square
```

and have a square drawn in the program.

A **procedure** in PostScript is any sequence of commands inside curly brackets **{** and **}**. Procedures can be treated just like any other objects in PostScript so that the effect of the sequence

```
/square {
newpath
0 0 moveto
a 0 lineto
a a lineto
0 a lineto
0 0 lineto
closepath
stroke
} def
```

is to let the single word **square** be assigned to the procedure inside the curly brackets. The effect of using the word **square** in a program is thereafter to replace it by the long sequence inside the brackets.

- *Procedures in PostScript amount to defining a single word to stand for a complicated sequence of commands.*

As in other programming languages, you can pass arguments to procedures, but through the very simple method of putting them on the stack before you call the procedure. Thus we can rewrite the procedure above to make it more flexible.

```
/square {
/a exch def
newpath
0 0 moveto
a 0 lineto
a a lineto
0 a lineto
0 0 lineto
closepath
stroke
} def
```

has the effect of drawing a square whose side is the number placed on the stack just before the word **square**. Thus with this definition the sequence

```
3 square
```

will draw a square of side 3 units. The line **/a exch def** will probably seem a bit peculiar, but is very simple to understand if you keep in mind that a procedure just substitutes the commands inside **{**, **}** for the procedure name. Thus the command **3 square** will transform to

```
3 /a exch def ...
```

What is the effect? What we want is

```
/a 3 def
```

but we can't do that directly because this assignment takes place inside the procedure. However, the total effect of **`3 /a exch`** is to put the sequence **`/a 3`** on the stack, since **`exch`** just exchanges the top two elements on the stack. Then the **`def`** makes the assignment we want.

The routine above is still somewhat awkward because there may already be a variable **`/a`** in your program, and you probably wouldn't want to change its value. What you want is what is called a **local variable** for your procedure. This is a bit awkward in PostScript, but perfectly possible. I won't explain all the details of why it works, but what you do is add a few lines to the procedure establishing a **local dictionary** of variable names, like this:

```
/square {
1 dict begin
/a exch def
newpath
0 0 moveto
a 0 lineto
a a lineto
0 a lineto
0 0 lineto
closepath
stroke
end
} def
```

The extra lines you add are

```
1 dict begin
...
end
```

at the beginning and end of your procedure. The variable **`a`** referred to here will be one used only in this procedure. The number **`1`** refers to the fact that you are only going to use one local variable. If you want to use more, say so. It does little harm to be over-generous, so that **`5 dict`** would be OK. This would be useful if you think you might want to add more variables later.

Here is a procedure with two arguments:

```
/rectangle {
2 dict begin
/h exch def
/w exch def
newpath
0 0 moveto
w 0 lineto
w h lineto
0 h lineto
0 0 lineto
closepath
stroke
end
} def
```

It assumes that you call it with two items on the stack, the width and the height in that order. When called, it removes the height and assigns its value to a local variable, then does the same for width. Then it draws the rectangle. Thus **`2 3 rectangle`** draws a rectangle of width 2 units and height 3 units.

A few fine points. The procedures above are perfectly correct, but they is probably not the most useful one you might want to write. For example, if you want to fill a square instead of stroking it you would have to write another procedure to do that, too. Instead:

- *In PostScript it is usually a good idea to use procedures to add paths to ones you have already constructed, instead of drawing and stroking or filling the path all in one procedure.*

Thus it would probably be more useful in the long run if you were to define a procedure

```
/square {
1 dict begin
/a exch def
0 0 moveto
a 0 lineto
a a lineto
0 a lineto
0 0 lineto
end
} def
```

so that

```
newpath
3 square
stroke
```

would draw a square and

```
newpath
3 square
fill
```

would fill it.

This illustrates one feature of all good programs in any language. While we are on this topic, I might as well lay out all the basic properties of good programs.

- Correctness

That is to say, the program must do what it is supposed to do.

- Readability

At least if someone else is going to read it. And even if not, you yourself will find it surprisingly hard to read your own programs later on if you are not careful. Readability means as a minimum adequate comments, good line spacing, and suitable indentation.

- Flexibility

You may very well want to extend or modify your program after the first version is written.

- Efficiency

This is not usually a big problem in this course, but occasionally it is something to think about, particularly later on when we want fast drawing to do animation.

- Originality

Of course for most of this course this will not be an issue. But for your projects originality (and imagination) will count a lot.

7. Loops

There are several ways to do loops in PostScript.

Recall that a procedure in PostScript is any sequence of commands and data inside curly brackets **{** and **}**, as we saw in defining procedures.

The simplest loop has the form

```
n { ...  } repeat
```

which simply repeats the procedure inside the brackets n times.

Here is an example. Suppose we have a routine **fillsquare** which fills a square at the origin with side s (using the current grey colouring), where s is to be put on the stack before calling **fillsquare**. Thus

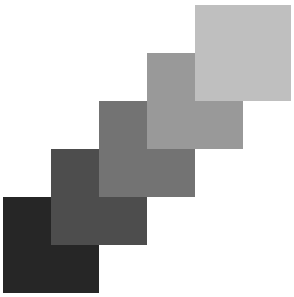
```
10 fillsquare
```

draws a square of side 10, with lower left and upper right corners $(0, 0)$ and $(10, 10)$.

Then the sequence

```
gsave
5 {
currentgray 0.15 add setgray
0.5 fillsquare
0.25 0.25 translate
} repeat
grestore
```

fills 5 unit squares; shifts them up as it goes; lightens the colour; and finally restores the coordinates to what they were before the translations were made.



This picture also illustrates another feature of the way PostScript draws: it is like painting—it covers over what lies underneath. So that, as I have already mentioned in earlier notes, if we want to draw a grey square and then outline it is important to do the filling first and then the outlining or we get the peculiar effect on the left instead of what we probably want on the right. This is true even if the current colour is white—there is no translucent paint in PostScript.

Finally the following sequence draws a square whose first side is from $(0, 0)$ to $(3, 1)$. The state of the stack is noted on the right in comments. This is a useful practice if you have trouble keeping in mind what is on the stack.

```
newpath
0 0 moveto
3 1             % x y
4 {             % x y (different each repeat)
2 copy          % x y x y
rlineto         % x y
neg exch        % -y x
} repeat
pop             % -y
pop             %
closepath
stroke
```

We have here the new commands **rlineto**, **pop**, and **copy**. The first is a relative version of **lineto**, adding its components to the current point to draw the line. The second just removes the top item from the stack without doing anything else. The third has this effect on the stack:

x1 x2 x3 ... xn n copy => x1 x2 x3 ... xn x1 x2 x3 ... xn

In other words, it just provides duplicate copies of the top n items on the stack.

Exercise 7.1. Write a procedure in PostScript that has two parameters r and θ and returns the (x, y) coordinates.

Exercise 7.2. Write procedures in PostScript that evaluate cos and sin for angles expressed in radians.

Exercise 7.3. Write a PostScript program to draw the Pythagorean figure of three squares, with variable sides a, b, and c.

Exercise 7.4. PostScript has a function **atan** but no **acos** or **asin**. Write your own.

Exercise 7.5. Write a PostScript procedure that has parameters x, y, R, N and constructs a path around a polygon centred at (x, y) of radius R and N sides (with one vertex due east).

Exercise 7.6. Write a PostScript procedure with parameters a, b, c and leaves the two roots of the polynomial

$$ax^2 + bx + c = 0$$

on the stack if they are real.

Exercise 7.7. Write a PostScript procedure which fills a clipped page with a grid of lines h apart, and gray-ness equal to g. The numbers g and h are its parameters. Use it to draw such a grid with $h = 0.1$ inch, $g = 0.5$. The clipped boundary of the page should be 0.25 inches. The x and y-axes should be black, and the origin at the centre of the page.

Exercise 7.8. Write a PostScript program which (1) draws a framed page with boundary 0.5 inches in width all around the page; (2) clips drawing to that region; (3) includes a procedure which has arguments the parameters of a line $Ax + By = C$ and a point (x_0, y_0) and draws (a) the line inside the drawing region and (b) a dotted perpendicular segment from the point to the line.