PostScript

What is it?

PostScript is a language used to describe how a picture looks. Some computers describe an image by simply recording all the dots that make it up, but PostScript takes image description one step further. You can ask PostScript for a circle of a certain radius, filled with a certain color, and it will understand what you mean.

Why use it?

A computer can only perform actions that it has all the facts for. If a computer merely manipulates dots, all it knows is where those dots are. If these dots are one-hundredth of an inch on a side (or 100 dots per inch) then when it is printed on a printer that can print at 400dpi (dots per inch), each of the dots in the picture must be made up of *16* printer dots. This is bad since the resulting image is just as coarse as it would be on a 100dpi printer. Result: the image appears jagged instead of smooth even though the printer can print at much higher quality.

PostScript removes this restriction by not describing images as a bunch of dots. Instead, it describes objects—circles, squares and lines, for example. Therefore, although an image may appear at 92dpi on your screen, when you print it, the printer knows enough to print it as smoothly as it possibly can—400dpi.

A Postscript image.

A TIFF image (non-Postscript).

Note: The TIFF is blown up to about 2X it's normal size.

This is why high quality laser printers use the PostScript language—they can print at their maximum resolution. In fact, most typesetters (upwards of 2400dpi!) also use PostScript since it is so powerful.

Why do I need to know all of this?

Normally (whether you realize it or not) when you create a drawing on a computer you use a paint program that lets you draw objects, then translates them into PostScript for you. If you like, however, you can simply create the PostScript directly, and never touch a drawing program again. The rest of this CheatSheet will show you the basics of the language and teach you how to draw pictures using it.

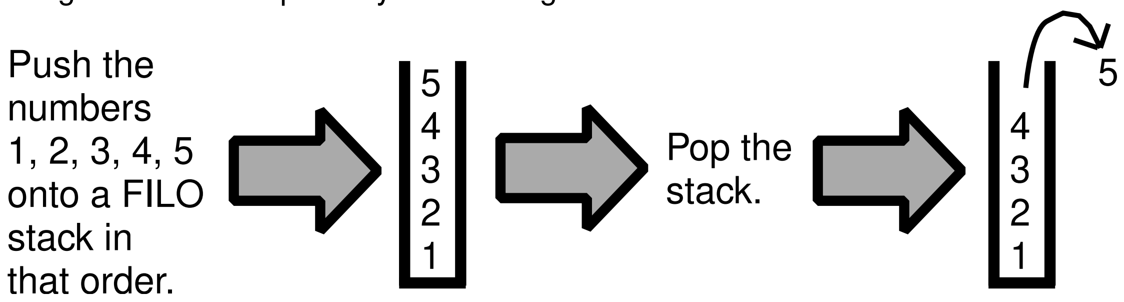
One quick note: PostScript is most commonly used for printing images on paper. This means that if you want to play with it, you normally have to print out the image to see what it looks like. Fortunately, the NeXT has an application that lets you type in a PostScript program and view it directly on the screen. To use it, start **/NeXTdeveloper/Demos/Yap** and wait for a window to appear in the upper right-hand corner. For further information about using Yap, consult the Yap CheatSheet.

The important stuff

PostScript is what is known as a stack-based language. A stack is a method for storing data in an efficient manner. This method shows up again and again in Computer Science in many different applications.

The basic idea of a stack is this: You have a bunch of data that you want the computer to remember, then access later. How do you store this data? You can think of a stack as a large hopper that you can toss the data into. When you want to access the data, you simply reach in and pull out the piece that is on top. Hence, the first piece of data you put into the stack will be the last one to come out. This method is known as FILO (“first in, last out”). Placing a piece of data onto the stack is called a “push”, and pulling it off is called a “pop.”

A diagrammed example may make things a little clearer:



In order to use PostScript, you must always keep stacks in mind. For example, if you want to draw a line from the point (1,5) to (10,15) you would use the following PostScript code:

```
1 5 moveto
10 15 lineto
stroke
```

If we trace this program step-by-step, we get the following:

1. Push the number 1 onto the stack.
2. Push 5 onto the stack.
3. Call the `moveto` routine, which pops one number off the stack for the y-coordinate then another for the x-coordinate, leaving the stack empty. (1,5) is

now the current point.

4. Push 10 onto the stack.
5. Push 15 onto the stack.
6. Call the `lineto` routine, which pops the 15 off the stack for the y-coordinate, and the 10 for the x-coordinate. The stack is empty, and there is an invisible line drawn from (1,5) to (10,15).
7. `stroke` paints this invisible line black.

Easy enough? It should now make sense why the coordinates are given before the commands (known as operators)—it is because the operators must get the data they need off of the stack. *Every PostScript operator that needs data pops it off of the stack, so remember to put it there!*

You are now ready to learn about the different commands you can use. The format of the following chart will show you what must be on the stack when you call each operator. To put something on the stack, just type it—there is no command to place an item on the stack. A fine line lies in the fact that operators are not normally placed on the stack. They operate on data...so they take data *off* of the stack as they need it. Only in specific cases do operators get placed onto the stack.

Since we are dealing with simple graphics here, we are leaving out many of the built-in PostScript operators. To give you an idea of the entire operator set, we have enclosed a *PostScript Reference Sheet* which lists all of the PostScript commands known to mankind. Enjoy!

Generalities

- Remember that PostScript is stack based! Put the data *before* the operator!
- Any path you make will be invisible until you give a `stroke` command.
- Stacks can rapidly become terribly confusing. Always keep track of what you have put onto the stack, and make sure you know what your operators are putting there as well...

The commands

Note: This format is taken from the *PostScript Language Reference Manual* by Adobe Systems. In this book there is a list of ALL PostScript commands in this same general format. The first column is what you need to place on top of the stack before the operator is called. (The element furthest to the right is on top of the stack.) The next column is the operator, and the third column is the stack after the operator. (A "—" means no elements.) The final column is a short description of the operator. "Bool" stands for boolean, either the number 1 (true) or 0 (false). "Proc" stands for procedure, or a block of commands that can be executed. The syntax for creating a block will be covered in Example 4.

• **Drawing operators:**

#1 #2	**moveto**	—	makes the point (#1, #2) current
#1 #2	**lineto**	—	Draws an invisible line from the current point to (#1, #2)
#1 #2	**rlineto**	—	Like **lineto** but current point is treated as (0,0)
—	**stroke**	—	paints an invisible path black
—	**fill**	—	fills an object with black
—	**newpath**	—	makes sure there are no current objects
#1 #2 #3 #4 #5	**arc**	—	makes an invisible arc of radius #3 at point (#1,#2) over the angle from #4 to #5 degrees

• **Stack operators:**

#1	**pop**	—	throws away the top item on the stack
#1 #2	**exch**	#2 #1	reverses the top two elements on the stack
#1	**dup**	#1 #1	puts two of the top element onto the stack

• **Arithmetic operators:**

#1 #2	**mul**	#1*#2	places the product of #1 and #2 on the stack
#1 #2	**add**	#1+#2	places the sum of #1 and #2 on the stack
#1 #2	**div**	#1/#2	places the quotient of #1 and #2 on the stack
#1 #2	**sub**	#1-#2	places the difference of #1 and #2 on the stack
#1	**neg**	-#1	reverses the sign of #1

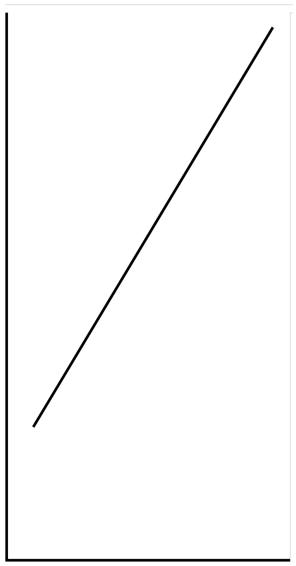
• **Control operators:** (Example #4 will be a tremendous help here)

#1 #2	**eq**	bool	test if #1 & #2 are equal
#1 #2	**ne**	bool	test if #1 & #2 are not equal
#1 #2	**gt**	bool	true if #1 is greater than #2
#1 #2	**ge**	bool	true if #1 is greater than or equal to #2
#1 #2	**lt**	bool	true if #1 is less than #2
#1 #2	**le**	bool	true if #1 is less than or equal to #2
—	**true**	bool	push true onto the stack
—	**false**	bool	push false onto the stack
bool proc	**if**	—	execute proc if bool is true
#1 proc	**repeat**	—	execute proc #1 times

Examples

Example 1

`moveto, lineto, stroke, showpage`



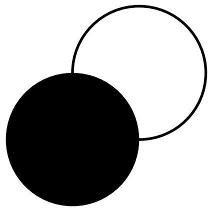
Note: The axis is drawn merely to show the coordinate system the line is drawn in. It is not drawn by the PostScript code shown.

```
%!                 % standard beginning of a PostScript file
10 50 moveto       % make point (10,50) current
100 200 lineto     % draw an invisible line from the current point to (100,200)
stroke             % paint line black
showpage           % show this page
```

When drawing your picture, PostScript will ignore anything after a "%" on a line. This is called a "comment" and simply makes the code easier for us humans to read.

Example 2

`rlineto, fill, arc`

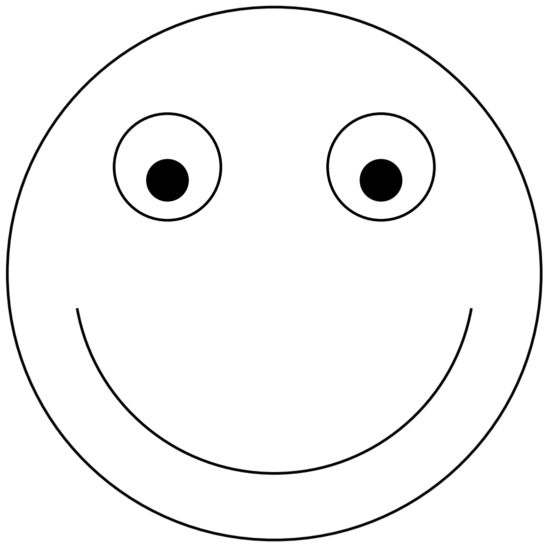


```
%!
100 150 25 0 360 arc       % draw an invisible circle radius 25 at (100,150)
stroke                     % paint the circle black
75 125 25 0 360 arc        % draw an invisible circle radius 25 at (75,125)
fill                       % fill this circle with black

40 dup moveto              % put 40 on the stack, duplicate it, moveto (40,40)
0 100 exch rlineto         % make line to (100,0)  (points are exchanged!)
stroke                     % paint this line black

showpage                   % show this page
```

Example 3

Shows a neat quality of stack-based languages—you can build up the stack and *then* call all of the operators...even though this is not good programming practice since it is very difficult for humans to read.

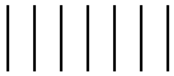
```
%!

250 250 100 0 360
210 290 20 0 360
210 285 8 0 360
290 290 20 0 360
290 285 8 0 360
250 250 75 190 -10

arc stroke newpath
arc fill newpath
arc stroke newpath
arc fill newpath
arc stroke newpath
arc stroke newpath

showpage
```



Just keep the stack in your mind, and you will get the idea of what is happening.

Example 4

Conditional statements and looping (`repeat`, `if`, etc.), procedures



```
%!
/line { moveto 0 25 rlineto stroke } def     % define line to draw a vertical line 25
                                             % points tall starting at a point on the stack.
/variable 10 def                             % variable = 10

10 {                                         % place 10 on the stack for the repeat
    4 variable le {                          % check if 4 is less then variable
        variable 10 mul 10 line              % draw line at (variable*10, 10)
    } if                                     % perform previous step only if
                                             % 4 variable le was true.
    /variable variable 1 sub def             % subtract 1 from variable
} repeat                                     % repeat the loop (only 10 times)
```

This is a rather simple program that shows looping and conditionals. The picture is stupid, but the number of lines drawn is significant. Each time the program goes through the loop, it draws a line. Trace through this code and make sure you understand why it is drawing 7 lines.

This program also uses the `def` operator several times. All that the first line of the program does is say that wherever the program sees `line` it should act like `moveto 0 25 rlineto stroke` is there instead. Likewise, wherever there the program sees `variable`, it should act like there is a `10` there in its place. Notice also that `variable` is redefined later in the program; you can define `variable` by saying “subtract one from whatever `variable` currently is, then make `variable` equal to that.” In short, subtract one from `variable`. `def` is an extremely useful operator—well worth the time spent understanding it.