APPENDIX 1

Summary of PostScript commands

This Appendix offers a summary of PostScript operators useful for producing mathematical figures. Most have already been introduced. In addition, a few that are likely to be more rarely used than the rest are explained here. This is a large list, but by no means a complete list of PostScript commands. The PostScript reference manual ('Red Book') contains a complete list by function as well as a list in alphabetical order in which the operators are described in occasionally invaluable detail.

There are many operators even in this restricted list, but fortunately most commands are very close to normal English usage and should be easy to remember.

The symbol $\emptyset$ means no arguments, or no return value.

1. Mathematical functions

Arguments	Command	Left on stack; side effects
$x\ y$	`add`	$x+y$
$x\ y$	`sub`	$x-y$
$x\ y$	`mul`	xy
$x\ y$	`div`	x/y
$x\ y$	`idiv`	the integral part of x/y
$x\ y$	`mod`	the remainder of x after division by y
x	`abs`	the absolute value of x
x	`neg`	$-x$
x	`ceiling`	the integer just above x
x	`floor`	the integer just below x
x	`round`	x rounded to nearest integer
x	`truncate`	x with fractional part chopped off
x	`sqrt`	square root of x
$y\ x$	`atan`	the polar argument of the point (x, y)
x	`cos`	$\cos x$ (x in degrees)
x	`sin`	$\sin x$ (x in degrees)
$x\ y$	`exp`	x^y
x	`ln`	$\ln x$
x	`log`	$\log x$ (base 10)
	`rand`	a random number

PostScript works with two kinds of numbers, integers and real. Real numbers are floating point, with a limited number of decimals of accuracy. Arguments for some operations, such as `repeat`, must be integers. I leave as an exercise to tell whether `ceiling` etc. return—i. e. leave on the stack—integers or real numbers. Many operations have an implicit range restriction—i. e. `sqrt` must be applied to a non-negative number.

2. Stack operations

x	`pop`	$\emptyset$
$x\ y$	`exch`	$y\ x$
x	`dup`	$x\ x$
$x_{n-1} \ldots x_0\ n\ i$	`roll`	$x_{i-1} \ldots x_0\ x_{n-1} \ldots x_i$

This rolls the top n elements on the stack around by a shift up of i elements. For example, if the stack holds `1 2 3 4 5` (from the bottom up) then `5 2 roll` changes it to `4 5 1 2 3`. It is more efficient if more complicated to do stack operations than access them by variable names, although the extra efficiency is often not worth the inconvenience of having to keep track of what's what on the stack.

$x_{n-1} \ldots x_0\ n$	`copy`	$x_{n-1} \ldots x_0\ x_{n-1} \ldots x_0$

A good trick for debugging is to combine `copy` and `roll` to view in a terminal window the top n items on the stack. The best way to do this (where $n = 3$):

```
3 copy [ 4 1 roll ] ==
```

$x_i \ldots x_0\ i$	`index`	$x_i \ldots x_0\ x_i$

3. Arrays

	`[`	begins an array
	`]`	closes an array
an array a	`length`	number of items in the array a
$a\ i$	`get`	a_i
$a\ i\ x$	`put`	$-$

Sets the i-th entry of a equal to x. The way to remember the order of the arguments here is to think of this as formally equivalent to `a[i] x def`.

$a\ i\ j$	`getinterval`	$a_i \ldots a_j$
n	`array`	an empty array of length n with `null` entries

The `null` item in PostScript is like nothing . . . else.

a	`aload`	$a_0 \ldots a_{\ell-1}\ a$ (ℓ is the length of a)

This essentially just unpacks a onto the stack, but also puts a itself on top. If you want just to unpack a, use the pair `aload pop`.

An array in PostScript is what in other languages is called a **pointer**, which is to say it is stored in PostScript as an address in the machine where the items in the array are stored. The practical importance of this is that if a is an array then the sequence `a dup` doesn't make a new copy of the data stored by a, but only a copy of the address where the data is stored. The sequence

```
a [ exch aload pop ]
```

will make a new array with the same data as a.

4. Dictionaries

name item	`def`	makes an entry in the current dictionary
n	`dict`	puts a dictionary of n null entries on the stack
dictionary d	`begin`	opens d for use
	`end`	closes the last dictionary opened

Dictionaries in PostScript keep track of variable names and their current values. There may be several dictionaries in use at any moment; they are stored on a stack (the dictionary stack) and searched from the top down. The command `begin` puts a dictionary on this stack and `end` pops it off. So `begin` and `end` should be nested pairs.

something	`bind`	used before `def` to construct a procedure immediately

Normally, when defining a procedure, the names occurring in it are left as strings, without attempting to look up their values when the definition is made. These names are looked up when the procedure is called. But when `bind` is used, the names that do occur in dictionaries are evaluated immediately.

5. Conditionals

The first few return 'boolean' constants `true` or `false`. A few others have boolean values as arguments.

	`false`	`false`	(boolean constant)
	`true`	`true`	(boolean constant)
$x\ y$	`eq`	$x = y$?	
$x\ y$	`ne`	$x \neq y$?	
$x\ y$	`ge`	$x \geq y$?	
$x\ y$	`gt`	$x > y$?	
$x\ y$	`le`	$x \leq y$?	
$x\ y$	`lt`	$x < y$?	
$s\ t$	`and`	s and t are both true?	
$s\ t$	`or`	at least one of s and t is true?	
s	`not`	s is not true?	
$s\,\{\ldots\}$	`if`	executes the procedure if s is true	
$s\,\{\ldots\}\{\ldots\}$	`ifelse`	executes the first procedure if s is true, otherwise the second	

6. Loops

$i\ h\ f\ \{\ldots\}$	`for`	steps through the loop from i to f, incrementing by h

The tricky part of this is that at the start of each loop it leaves the loop variables i, $i + h$, $i + 2h$ on the stack. It is safest to use this only with integer loop variables.

$n\,\{\ldots\}$	`repeat`	executes the procedure n times
$\{\ldots\}$	`loop`	executes the procedure until `exit` is called from within the procedure
$\emptyset$	`exit`	exits the loop it is contained in
$\emptyset$	`quit`	stops everything
$a\,\{\ldots\}$	`forall`	loops through the elements of a, leaving each in turn on the stack
$\{.\}\{.\}\{.\}\{.\}$	`pathforall`	loops through the current path (see below)

The four arguments to `pathforall` are procedures to be called in the course of looking at the current path. This is a tricky command, but it can produce spectacular effects. A path is a special kind of array. Each element in it is one of the four commands `x y moveto`, `x y lineto`, `x[1] y[1] x[2] y[2] x[3] y[3] curveto`, `closepath`. The data are expressed in device coordinates. The command `pathforall` loops through the elements of the current path, pushing its arguments on the stack and then executing the corresponding procedure. For example, the following segment displays the current path.

```
{ [ 3 1 roll (moveto) ] == }
{ [ 3 1 roll (lineto) ] == }
{ [ 7 1 roll (curveto) ] == }
{ [ (closepath) ] == }
pathforall
```

The values of the coordinates are in the current user coordinates.

7. Conversions

$x\ s$	`cvs`	an initial substring of the string s expressing x
x	`cvi`	x converted to integer

8. File handling and miscellaneous

a string s	`run`	executes the file s
	`showpage`	changes a page
a procedure	`exec`	executes a procedure
a name	`load`	loads the value associated to the name
—	`save`	puts a copy of the entire current state on the stack
state	`restore`	restores the state on the stack

Thus

```
save /SavedState exch def
...
SavedState restore
```

will save and restore a snapshot of a state.

	`type`	tells what type the object at the top of the stack is

It pops that object from the stack, so you will likely want to use `dup` and `type` together. This is one of the more complicated PostScript operators. First of all, what it returns is one of the following names

`arraytype`	an array
`booleantype`	a boolean like `true` or `false`
`dicttype`	a dictionary
`fonttype`	a font
`integertype`	an integer like 1
`marktype`	a `[`
`nametype`	a name like `/x`
`nulltype`	a null object
`operatortype`	an operator like `add`
`realtype`	a real number like 3.14159
`stringtype`	a string like `(x)`

or possibly one of a few types I haven't introduced.

Second, what it returns is an executable object, which means if you apply to it the `exec` operator it will execute whatever has been defined by you to be associated to that name. Thus after

```
/arraytype { dup length = == } def
/integertype { = } def
```

the sequence `dup type exec` will display and pop the object at the top of the stack if it is an integer, display and pop it and its length if it is an array, and give you an `undefined` error otherwise. This allows you to have a procedure do different things, depending on what kind of arguments you are passing to it. The PostScript operator `transform` behaves like this, for example, detecting whether the top of the stack contains a matrix or a number.

9. Display

x	`=`	pops x from the stack and displays it on the terminal
x	`==`	almost the same as `=`

The most important difference between the two is that the operator `==` displays the contents of arrays, while `=` does not. One curious difference is how they handle strings. Thus `(x) =` displays `x` in the terminal window while `(x) ==` displays `(x)`. In particular, it is useful when using terminal output for debugging to know that `() =` produces an empty line.

...	`stack`	displays the whole stack (but not arrays), not changing it
...	`pstack`	same as `stack`. but also displays arrays
string s	`print`	prints a string; has better format control than the others

The difference between = and == is that == will display arrays and = will not. Sometimes this is a good thing, and sometimes not; sometimes arrays will be huge and displaying them will fill up your screen with garbage. The difference between `stack` and `pstack` is the same.

As for `print`, it is a much fancier way to display items—more difficult to use, but with output under better control. For example

```
(x = ) print
x (      ) cvs print
(\n) print
```

will display "x = " plus the current value of x on a single line. What's tricky is that `print` displays only strings, so everything has to be converted to one first. That's what `cvs` does. The `(\n)` is a string made up of a single carriage return, because otherwise `print` doesn't put one in.

Implicitly the value of x here is converted to a string.

10. Graphics state

$\emptyset$	`gsave`	saves the current graphics state, installs a new copy of it
$\emptyset$	`grestore`	brings back the last graphics state saved

The graphics state holds data such as the current path, current line width, current point, current colour, current font, etc. These data are held on the **graphics stack**, and `gsave` and `grestore` put stuff on this stack and then remove it. They should always occur in nested pairs. All changes to the graphics state have no effect outside a pair. It is a good idea to encapsulate inside a `gsave ... grestore` pair all fragments of a PostScript program that change the graphics state to draw something, unless you really want a long-lasting change.

stack:operator:5 We have seen three stacks used by a PostScript interpreter—the **operator stack** which is used for calculations,
stack:dictionary:5 **stack:graphics:5** the **dictionary stack** which controls access to variable names, and the **graphics stack**. There is one other stack,
stack:execution:5 the **execution stack**, which is used to keep track of what procedures are currently running, but the user has little explicit control over it, and it is not important to know about it.

x	`setlinewidth`	sets current linewidth to x (in current units)
	`currentlinewidth`	the current linewidth in current units
x	`setlinecap`	determines how lines are capped
x	`setlinejoin`	determines how lines are joined
$[\ldots]\ x$	`setdash`	sets current dash pattern

For example `[3 2] 1 setdash` makes it a sequence of dashes 3 units long and blanks 2 units long each, with an offset of 1 unit at the beginning.

Experimentation with `setdash` can be interesting. The initial array specifying the on/off pattern can be long and complicated, and itself produced by a program. Go figure.

g	`setgray`	sets current colour to a shade of grey
$r\ g\ b$	`setrgbcolor`	sets current colour

In both of these, the arguments should be in the range $[0, 1]$.

11. Coordinates

Here, a matrix is an array of 6 numbers. The CTM is the **C**urrent **T**ransformation **M**atrix.

$\emptyset$	`matrix`	puts a matrix on the stack
matrix m	`defaultmatrix`	fills m with the default TM, leaves it on the stack
m	`currentmatrix`	fills the matrix with the current CTM, leaves it
$x\ y$	`translate`	translates the origin by $[x, y]$
$a\ b$	`scale`	scales x by a, y by b
A	`rotate`	rotates by Ac degrees
m	`concat`	multiplies the CTM by m
m	`setmatrix`	sets the current CTM to m
	`identmatrix`	the identity matrix
$x\ y$	`transform`	$x'\ y'$, transform of $x\ y$ by the CTM
$x\ y\ m$	`transform`	$x'\ y'$, transform of $x\ y$ by m
$x\ y$	`itransform`	$x'\ y'$, transform of $x\ y$ by the inverse of the CTM
$x\ y\ m$	`itransform`	$x'\ y'$, transform of $x\ y$ by the inverse of m

idtransform:6 There are also operators `dtransform` and `idtransform` that apply just the linear component of the matrices (to get relative position).

$m_1\ m_2$	`invertmatrix`	m_2 (the matrix m_2 is filled by the inverse of m_1

12. Drawing

$\emptyset$	`newpath`	starts a new path, deleting the old one
$\emptyset$	`currentpoint`	the current point $x\ y$ in device coordinates

In order for there to be a current point, a current path must have been started. Every path must begin with a `moveto`, so an error message complaining that there is no current point probably means you forgot a `moveto`.

$x\ y$	`moveto`	begins a new piece of the current path
$x\ y$	`lineto`	adds a line to the current path
$dx\ dy$	`rmoveto`	relative move
$dx\ dy$	`rlineto`	relative line
$x\ y\ r\ a\ b$	`arc`	adds an arc from angle a to angle b, centre (x, y), radius r
$x\ y\ r\ a\ b$	`arcn`	negative direction

The operators `arc` and `arcn` are a bit complicated. If there is no current path under construction, it starts off at the first angle and makes the arc to the second. If there is a current path already it adds to it a line from where it ends to the beginning of the arc, before it adds the arc to the current path.

$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	`curveto`	adds a Bezier curve to the current path
$dx_1\ dy_1\ dx_2\ dy_2\ dx_3\ dy_3$	`rcurveto`	coordinates relative to the current point
$\emptyset$	`closepath`	closes up the current path back to the last point moved to
$\emptyset$	`stroke`	draws the current path
$\emptyset$	`fill`	fills the outline made by the current path
$\emptyset$	`clip`	clips drawing to the region outlined by the current path
$\emptyset$	`pathbbox`	$x_\ell\ y_\ell\ x_u\ y_u$

This returns four numbers `llx lly urx ury` on the stack which specify the lower left and upper right corners of a rectangle just containing the current path.

$\emptyset$	`strokepath`	replaces the current path by its outline
a special dictionary	`shfill`	used for gradient fill

13. Displaying text

font name	`findfont`	puts the font on the stack
font s	`scalefont`	sets the size of the font (in current units), & leaves it on the stack
font	`setfont`	sets that font to be the current font

So that

```
/Helvetica-Bold findfont
12 scalefont
setfont
```

sets the current font equal to Helvetica-Bold at approximate height 12 units.

string s	`show`	displays s

The string is placed at the current point, and moves that current point to the end of the string. Usually it is prefaced by a `moveto`. There must also be a current font set.

string s	`stringwidth`	w_x w_y, the shift caused by showing s

I.e. displaying a string moves the current point. This returns the shift in that point.

string s boolean t	`charpath`	the path this string would make if displayed.

Use `true` for filling or clipping the path, `false` for stroking it. In some circumstances these will produce somewhat different results, and in particular the path produced by `true` might not be what you want to see stroked.

14. Errors

When a program encounters an error it displays a key word describing the type of error it has met. Here are some of the more likely ones, roughly in the order of frequency, along with some typical situations that will cause them.

`undefined`	A word has been used that is undefined. Often a typing error.
`rangecheck`	An attempt has been made to apply an operation to something not in its range.

For example, `-1 sqrt` or `[0 1] 2 get`.

`syntaxerror`	Probably an `(` or `{` without matching `)` or `}`.
`typecheck`	An attempt to perform an operation on an unsuitable type of datum.
`undefinedfilename`	An attempt to `run` a file that doesn't exist.
`undefinedresult`	`5 0 div`
`unmatchedmark`	`]` without a previous `[`.
`dictstackoverflow`	Dictionaries have not been closed. Probably a `begin` without `end`.

15. Alphabetical list

Here is a list of all the operators described above, along with the section it can be found in.

`=`	**9**
`==`	**9**
`[`	**4**
`]`	**4**
`abs`	**1**
`add`	**1**
`aload`	**3**
`and`	**5**
`arc`	**12**
`arcn`	**12**
`array`	**3**
`atan`	**1**
`begin`	**4**
`bind`	**4**
`ceiling`	**1**
`charpath`	**13**
`clip`	**12**
`closepath`	**12**