A POSTSCRIPT Tutorial and Reference

Ian Utting

Computing Laboratory, UKC

ABSTRACT

POSTSCRIPT is the de facto standard Page Description Language produced by Adobe Systems Corporation and supported as a printer interface language by many manufacturers of laser printers and phototypesetters. This document provides an illustrated introduction to the major features of the language, it should contain enough information to enable potential users of POSTSCRIPT to determine how much effort is required to perform a task they have in mind, and for some purposes it will serve as a programmer's guide to the language. It will never replace the Adobe Systems books, but then it's nowhere near as expensive.

1. Overview

POSTSCRIPT is a stack-based general purpose programming language (using a postfix notation) with many built-in graphical features. A POSTSCRIPT program typically consists of a number of procedure definitions (a prologue) followed by a set of independent program fragments (a script), each of which use the definitions from the prologue to produce a page of output. The primitive actions in the language are performed by operators, built-in procedures or functions.

Operators which require operands take them from the operand stack, where they have been placed as the results of other operators or functions, or by explicit action taken by the program. There is also a dictionary stack and a graphics state stack, both of which are described in detail below.

POSTSCRIPT views a page as a rectangular grid onto which opaque ink of various colours (including white) may be sprayed, each new coat obscuring the underlying ones. Positions on the page are specified in POSTSCRIPT using a coordinate system (the default user space) which is independent of the resolution of the particular printer being used. This is a normal Cartesian space with the origin at the bottom left hand corner of the page and a resolution of 1/72 of an inch (close to the traditional printers point of 1/72.27 inches). Coordinates are specified as x and y values, where x and y are real numbers which increase rightwards and upwards respectively. The position of the origin, orientation of the axes, resolution and even the relative direction of the axes can be changed via coordinate transformations, see below.

In this document, operators are described (in an informal fashion) in terms of their expected operands and the result of applying an operator to them. In general, the line:

operand(s) operator fi result(s)

implies that operator expects to find operand(s) on the operand stack, with the top-of-stack at the right hand end. After it has completed, the stack will contain result(s) in place of the operands. An operand or result of ''–'' is used to indicate that no operands are required, or that no result is returned.

In examples (set in Courier), the symbol => is used to separate an example from the result of executing it, so that

```
3 4 add => 7
```

indicates that 7 is the result of executing the POSTSCRIPT fragment `3 4 add`.

2. Syntax

POSTSCRIPT programs are written in printable (ASCII) form and are interpreted directly by the printer. There are six distinct syntactic constructs in POSTSCRIPT, delimited by spaces, tabs and newlines. Delimiters are otherwise ignored (outside of strings). These constructs are:

Numbers Both real and integer, with an optional sign. Eg. 123 -98 274.3 -0.0002 1E27 -123.6E10. Integers may also be specified in the form radix#number, indicating that the number is in base radix. Eg. 16#7FFF, 8#377, 2#1011110 and 36#7B45Z (the last being in base 36!).

Strings Strings are sequences of ASCII characters contained in balanced parentheses. All the usual UNIX-like† escape sequences (\n, \t, \037 etc) are recognised, along with \) and \(to provide for unbalanced parentheses. Examples are:

```
(This is a string)
(This contains a newline
but is still one string)
(This contains (balanced) parentheses)
()
```

the last of which was the empty string.

Comments Anything between the character % and the end of the line is ignored by the POSTSCRIPT interpreter. Some comments starting with %% are conventional—used by other processors to preserve useful information of no use to the POSTSCRIPT interpreter.

Names Any string of non-special characters outside a string or comment and not containing a delimiter is interpreted as a name (unless it's a number). So: abc, Offset, 23A, 13-456 and @pattern are all names. The ‘‘/’’ (slash) character is used to indicate a literal name, ie. one that is not to be interpreted.

Procedures Procedures in POSTSCRIPT are sequences of tokens enclosed in curly brackets (‘‘{’’ and ‘‘}’’). For example the POSTSCRIPT code fragment:

```
/average { add 2 div } def
4 6 average
```

defines (using the def operator) a procedure whose name is average (note the use of the literal character when defining a name) and whose body consists of the code to add the top two elements on the operand stack and divide the result by two, leaving it on the operand stack. In the second line, the numbers 4 and 6 are pushed onto the stack, and the average procedure executed (by naming it without the literal character).

Arrays Arrays are heterogeneous collections of POSTSCRIPT objects delimited by square brackets. For example, the array:

```
[ 23 45.2 (a string) /aName [ (abc) 16#7e ] { 2 div } ]
```

consists of six elements:

An integer number: 23
A real number: 45.2
A string: containing the characters a string
A literal name (not evaluated at this point): /aName
An array, itself containing a string and an integer
A procedure body: { 2 div }

The POSTSCRIPT interpreter scans its input (ignoring comments) looking for tokens which are acted upon according to their type. Numbers, strings, arrays, procedure bodies and literal names are simply pushed

† UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.

onto the operand stack. Evaluated names (those not preceded by a literal character) are converted into the corresponding POSTSCRIPT object, and that object is either pushed onto the operand stack or, if it is executable (ie. it is a procedure body or built-in operator), then it is immediately executed.

3. General Purpose Operators

Many of the operators supported by PostScript are general in nature, used to calculate operands for graphical operations and to control the action of the program.

3.1. Arithmetic Operators

All the usual arithmetic operators are supported, as indicated in the following list.

num1 num2 add	fi	(num1 + num2)
num1 num2 sub	fi	(num1 – num2)
num1 num2 mul	fi	(num1 · num2)
num1 num2 div	fi	(num1 / num2)
num1 num2 exp	fi	$num1^{num2}$
num sqrt	fi	Ö$\overline{num}$
num abs	fi	|num|
num neg	fi	–num

The functions sin, cos, atan, ln and log are also supported.

3.2. Relational and Boolean Operators

The following relational operators are defined. Relational operators can be applied to numbers or strings. In the case of numbers they have the expected effect, strings are compared for ASCII lexicographic ordering. Any other types of objects can be compared only for (in)equality, that is whether or not they are references to the same object.

any1 any2 eq	fi	(any1 = any2)
any1 any2 ne	fi	(any1 <> any2)
obj1 obj2 ge	fi	(obj1 >= obj2)
obj1 obj2 gt	fi	(obj1 > obj2)
obj1 obj2 le	fi	(obj1 <= obj2)
obj1 obj2 lt	fi	(obj1 < obj2)

The following boolean operators are defined, along with the boolean constants true and false.

bool1 bool2 and	fi	(bool1 AND bool2)
bool1 bool2 or	fi	(bool1 OR bool2)
bool1 bool2 xor	fi	(bool1 XOR bool2)
bool not	fi	not(bool)

3.3. Flow Control

bool proc if fi –

Executes the procedure body proc if bool is true, otherwise ignores it.

bool tproc fproc ifelse fi –

Executes the procedure body tproc if bool is true, otherwise executes the procedure body fproc.

n proc repeat fi –

The procedure body proc is executed n times. Both n and proc are removed from the operand stack before any execution takes place.

```
4 {(abc)} repeat => (abc) (abc) (abc) (abc)
8 4 {1 sub} repeat => 4
```

init inc limit proc for fi –

The equivalent of the C-like for loop:

```
for (i = init; i <= limit; i += inc)
    proc;
```

(if inc is negative, the <= relationship is replaced by >=). The current value of the loop counter (i) is pushed onto the operand stack before each execution of the body proc.

```
0 1 1 4 {add} for => 10
3 -0.5 1 {} for => 3.0 2.5 2.0 1.5 1.0
```

3.4. Operand Stack Handling

any pop fi –

Discard the top element of the operand stack.

any1 any2 exch fi any2 any1

Exchange the top two elements of the operand stack. For example:

```
(a) (b) exch => (b) (a)
```

or:

```
/xdef { /x exch def } def
2 xdef
```

Which associates the value 2 with the name x in the current dictionary. This technique is commonly used for capturing arguments to procedure bodies, see below.

any dup fi any any

Duplicates the top element of the operand stack.

```
/cube { dup dup mul mul } def
3 cube => 27
```

any$_{N-1}$ ... any$_0$ N J roll fi any$_{(J-1) \bmod N}$ ... any$_0$ any$_{N-1}$ ... any$_{J \bmod N}$

(Easier than it looks). Circularly shift the top N elements of the operand stack (treated as a sub-stack) up (towards the top-of-stack) J places. If J is negative, the shift will be down.

```
(a) (b) (c) 3 -1 roll => (b) (c) (a)
(a) (b) (c) 3 1 roll => (c) (a) (b)
(a) (b) (c) 3 2 roll => (b) (c) (a)
```

any == fi –

Destructively but intelligently print the top element of the operand stack onto the job log (not the output page). Useful for debugging.

4. Dictionaries

Dictionaries are associative arrays: that is they contain names (keys) with which are associated data (values), they are used in POSTSCRIPT for storing variables. When a name is mentioned, it is first searched for in the current dictionary, and then in the other dictionaries on the stack in top-down order.

4.1. Dictionary Operators

int dict fi dict

Creates a dictionary with a capacity of int key-value pairs and leaves it on the operand stack.

dict begin fi –

Takes dict from the operand stack, pushes it onto the dictionary stack and makes it the current dictionary.

– end ﬁ –

Discards the top element of the dictionary stack, making the new top element the current dictionary.

key val def ﬁ –

Associates val with key in the current dictionary, overwriting any existing definition. If key is not currently defined, a new entry known as key and containing val is created in the current dictionary.

key val store ﬁ –

Associates val with key in whatever dictionary on the dictionary stack has key defined. If key is not currently defined, a new entry is created in the current dictionary.

dict key known ﬁ boolean

Result is true if key is known in dict, false otherwise.

key load ﬁ val

Result is the (uninterpreted) val associated with key in the current context.

– currentdict ﬁ dict

A copy of the current dictionary is placed on the operand stack.

In the procedure defined below, each instance of the variables called x and y are stored in separate dictionaries (created by the dict operator on every (recursive) call of the procedure). Note the use of exch to capture the ‘‘arguments’’ to the procedure.

```
/pointless {
    2 dict begin
        /x exch def
        /y exch def
        x 1 add y 1 add pointless
    end
} def
```

5. Fonts & Characters

Fonts are dictionaries which contain details of character shapes which are defined by POSTSCRIPT procedures stored therein. They are also expected to contain variables which are used by various operators to decide how and where to place the shapes on the page. Many items in a font dictionary can be altered by POSTSCRIPT programs.

Fonts are stored internally as though they had a ‘size’ of 1 unit in the current user coordinate system, and must be scaled to a more appropriate size before they can be used.

5.1. Font Operators

name findfont ﬁ fdict

The font dictionary fdict, associated with (literal) name is placed on the operand stack.

fdicta int scalefont ﬁ fdictb

The font dictionary fdicta is transformed from its default size to be int units high in the current user coordinate system. The resulting, modified, dictionary fdictb is left on the operand stack.

fdict setfont ﬁ –

The font dictionary fdict is made the current font (part of the graphic state).

To select 10-point Times Roman as the current font:

```
/Times-Roman findfont 10 scalefont setfont
```

5.2. Character Operators

string show ﬁ –

The characters in string are imaged using the current font, starting at the current position. After each character has been imaged, the current position is shifted (usually right) by the ‘natural’ width of the character as stored in the current font dictionary (actually a (Dx,Dy)vector).

string stringwidth fi wx wy

Places on the operand stack the amounts (wx and wy) by which the current position would be altered in total if string were given to the show operator.

```
(this string) stringwidth => 39.73 0.0
```

Note that the y (vertical) component of the width is 0.0, not the height of the characters composing the string.

numx numy char string widthshow fi –

As show, but every time that the character whose code is char is imaged, the current position is altered by (numx,numy) units in the user coordinate system as well as by any ‘natural’ width it may have.

numx numy string ashow fi –

As widthshow, but the extra movement is associated with each character as it is shown.

numax numay char numbx numby string awidthshow fi –

As both ashow and widthshow. Adds (numbx,numby) to the current position between every pair of characters in string, and also adds (numax,numay) to the current position after every occurrence of char.

proc string kshow fi –

Shows each character from string, adding its ‘natural’ width to the current position. Between every two characters, executes the body proc with the integer codes for the two characters on the operand stack.

For example:

```
{ myproc } (Text) kshow
```

Results in the body myproc being being executed 3 times, first with the integers 16#54 and 16#65 on the stack, then with 16#65 and 16#78, finally with 16#78 and 16#74.

6. String & array handling.

PostScript contains a number of operators for examining the contents of composite objects, usually strings or arrays (called a comp-obj below).

int string fi string

Creates a string of (maximum) length int, leaving it on the operand stack. For example:

```
/mystring 8 string def
```

creates a new string of maximum length 8 characters, which initially contains 8 0s, and stores it in the current dictionary under the name mystring. This is an alternative to the (\0\0\0\0\0\0\0\0) notation.

int array fi array

Performs the same function for an array composite object. 6 array has the same effect as [0 0 0 0 0 0].

comp-obj length fi int

Returns the length of the composite object on top of the operand stack.

```
(abc\n) length => 4
mystring length => 8
[ 0 4.5 (hello) {add} mystring ] length => 5
```

comp-objA comp-objB copy fi subcomp-obj

Copies the contents of comp-objA into the start of comp-objB, leaving on the operand stack a reference to subcomp-obj, the portion of comp-objB containing the copy (comp-objB and subcomp-obj share storage).

comp-obj index get fi element

Returns the indexth element (character or object) from comp-obj (which is indexed from 0).

```
/mystring (Show me) def
/myarray [ 0 4.5 (hello) {add} mystring ] def
mystring 5 get => 16#6d (Hex 6d is the ASCII code for ‘m’).
myarray 1 get => 4.5
```

comp-obj index value put fi –

Makes the indexth element of comp-obj contain value. For instance:

```
mystring 5 16#68 put
```

Leaves mystring containing (Show he) (Hexadecimal 68 is the ASCII code for ‘h’).

```
myarray 3 {sub} put
```

Leaves myarray containing [0 4.5 (hello) {sub} mystring].

comp-obj beg len getinterval fi subcomp-obj

subcomp-obj is created as a reference to the portion of comp-obj starting at index beg and extending on len elements.

```
myarray 1 3 getinterval => [ 4.5 (hello) {sub} ]
/shortstring mystring 5 2 getinterval def
```

The latter defines shortstring to contain the me from mystring. Note that if mystring is now changed, that shortstring will change with it. To create shortstring independently of mystring, say something like:

```
/shortstring mystring 5 2 getinterval dup length string copy def
```

comp-objA index comp-objB putinterval fi –

The inverse of getinterval, comp-objB is copied element by element into comp-objA starting at index.

```
(Show me) dup 5 (it) putinterval => (Show it)
```

Note that there must be enough space in comp-objA to accommodate all of comp-objB:

```
myarray 3 [ {div} shortstring (Oops) ] putinterval
```

causes a fatal rangecheck error.

comp-obj proc forall fi –

Executes the procedure proc once for each element of comp-obj in order, having first placed the element on the operand stack. If comp-obj is a string, the elements pushed are the codes of the individual characters, not single character strings.

```
0 [ 13 29 3 -8 21 ] { add } forall => 58
myarray { } forall => 0 4.5 (hello) {sub} mystring
```

string seek anchorsearch fi

if found: s-post s-match true

else: string false

On strings only. If seek is an initial substring of string, then the substring (s-post) of string which follows seek is pushed onto the operand stack, followed by the matched string (s-match, always the seek), followed by the boolean constant true. If not, string is left on the operand stack, followed by the boolean constant false.

(abbc)	(ab)	anchorsearch	=>	(bc)	(ab)	true
(abbc)	(bb)	anchorsearch	=>	(abbc)	false	
(abbc)	(bc)	anchorsearch	=>	(abbc)	false	
(abbc)	(cc)	anchorsearch	=>	(abbc)	false	

string seek search fi
    if found: s-post s-match s-pre true
    else: string false

As anchorsearch, but seek is not anchored to the start of string. The substring of string occurring before the matched string is also pushed onto the operand stack when the search succeeds.

(abbc)	(ab)	search	=>	(bc)	(ab)	()	true
(abbc)	(bb)	search	=>	(c)	(bb)	(a)	true
(abbc)	(bc)	search	=>	()	(bc)	(ab)	true
(abbc)	(cc)	search	=>	(abbc)	false		

7. Simple Graphics

Drawing graphical objects using POSTSCRIPT is a two-stage process, first the shape of the object is defined using commands like lineto and arcto, however this in itself won't cause anything to appear on the page. Objects are only ''painted'' by using the fill or stroke operators. When these are used, the current path and the current point are cleared. Until then, everything drawn is added to the current path, which is kept as part of the graphic state. The current path may consist of any number of sub-paths; that is, a path does not need to be continuous, and can contain disjoint figures and lines scattered all over the page.

– currentpoint fi x y
    Pushes the x and y coordinates of the current position onto the operand stack.

x y moveto fi –
    Makes (x,y) the current point.

x y rmoveto fi –
    Makes (current-position + (x,y)) the current point.

x y lineto fi –
    Adds a line from the current position to (x,y) to the current path.

x y rlineto fi –
    Adds a line from the current position to (current-position + (x,y)) to the current path.

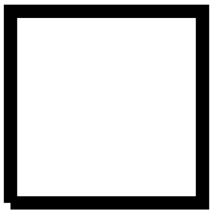
width setlinewidth fi –
    Sets the width of the line used by the stroke operator. A width of 0 generates the thinnest possible line. The linewidth is part of the graphic state.

– stroke fi –
    Draws a line of the current thickness around all of the current path.

– closepath fi –
    Closes the current sub-path, by adding to it a line from the current point to the start of the current sub-path. A new sub-path is started. (An entirely new path can be started by using the newpath operator.) This is important, as it's the only way to tell POSTSCRIPT that the point you end up at is connected with the one you started from. Compare:

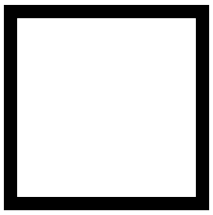
```
/inch  { 72 mul } def
5 setlinewidth    % thick  lines

.5 inch .5 inch moveto
0 inch 1 inch rlineto
1 inch 0 inch rlineto
0 inch -1 inch rlineto
-1 inch 0 inch rlineto
stroke
```



where the current path starts and finishes coincidentally at the same point, with:

```
.5 inch .5 inch moveto
0 inch 1 inch rlineto
1 inch 0 inch rlineto
0 inch -1 inch rlineto
closepath
stroke
```



where the the final (bottom left hand) corner gets the same treatment as all the intermediate ones.

level setgray fi –

Sets the current grey-scale ‘‘colour’’ to level. A level of 0 (the default) corresponds to black, 1 to white and intermediate values to a shade of grey in between. Remember that all shades of ‘‘ink’’ are opaque so that a white object can overwrite a black one. All objects, including text, are imaged using the current colour, which is part of the graphic state.

– fill fi –

Fills the interior of the current path with the current colour, after having closed any open sub-paths. When done, fill implicitly executes the newpath operator.

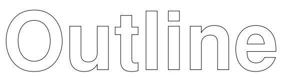
string bool charpath fi –

Appends to the current path the outlines of the characters which would result if string were imaged using show. Most fonts are defined by filled outlines, in which case this operator always appends a series of closed sub-paths. In the case of fonts like Courier which are implemented using the stroke operator, this may result in open sub-paths being generated. To combat this, if the resulting path is to be stroked, then bool should be false, otherwise true.

```
/Helvetica-Bold findfont
30 scalefont setfont

0 0 moveto
0 setlinewidth

(Outline) true charpath stroke
```



– pathbbox fi llx lly urx ury

This operator returns the coordinates of the bottom left (llx,lly) and top right (urx,ury) corners of the smallest box which could contain all of the current path. This can be useful for precisely fitting text, as in the example below.

– gsave fi –
– grestore fi –

These operators always appear as a pair. Gsave makes a copy of the current graphic state, which then becomes the current one. Grestore discards the current graphic state, revealing the saved copy. Ie. they do for the graphic state stack what the dup and pop operators do for the operand stack.

The graphic state contains among other things the current:

- Transformation Matrix (see below),
- position (initially undefined),
- path (initially empty),
- font (initially undefined),
- line width (initially 1),
- colour (initially black).

In the example below, note the use of gsave/grestore to enable the current path to be both filled and stroked. Also that it is the line width and colour in force when the stroke operator is invoked which are used, rather than those in force during definition of the path. The indentation is purely for clarity.

```
/inchbox {
    moveto
    0 1 inch rlineto
    1 inch 0 rlineto
    0 -1 inch rlineto
    closepath
} def

.25 inch .25 inch inchbox
fill

.75 inch .75 inch inchbox
gsave
    .5 setgray fill
grestore
1 setlinewidth stroke

.75 inch .75 inch moveto
/Times-Roman findfont
100 scalefont setfont
1 setgray

% find size of "g"
gsave
    newpath 0 0 moveto
    (g) true charpath pathbbox
grestore
/ury exch def /urx exch def
/lly exch def /llx exch def

1 inch
urx llx sub     % width
sub 2 div       % horiz. centring
llx sub         % x offset from here

1 inch
ury lly sub     % height
sub 2 div       % vertical centring
lly sub         % y offset from here

rmoveto         % go there
(g) show        % and do it
```

In order to specify a circular arc (including a full circle), it is necessary to specify the centre point of the arc, the radius and the start and end angles. The start and end angles are specified, rather than the start and end points, to avoid problems with arcs which don't pass through the specified points. Start and end angles are measured in degrees counterclockwise from the x-axis (''compass'' east). An arc which starts at 0˚ and ends at 360˚ is a circle.

In order that lines can be properly joined to arcs, two facts should be noted:

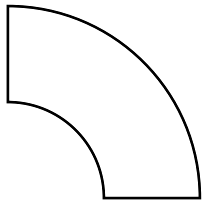
- Arcs are added to the current path by drawing a line from the current position to the start point of the arc. In order to prevent this happening, start a new path immediately before invoking an arc operator. See the example below.
- After an arc has been added to the current path, the end point of the arc becomes the current position.

Between any two angles, there are two possible arcs and hence two arc drawing operators:

cx cy rad sang eang arc fi –
cx cy rad sang eang arcn fi –

arc draws an arc in a positive (anti-clockwise) sense from sang to eang. The arc has its centre at (cx,cy) and its radius is rad.
arcn does just the same, but the sense of the arc is negative (clockwise).

```
newpath
0 0 1 inch 0 90 arc
0 0 0.5 inch 90 0 arcn
closepath
stroke
```



The newpath inhibits the joining of the (unspecified) current point to the start of the first arc, the start point of this arc thus becomes the beginning of the current path. The absence of a newpath before the arcn call means that the then current point (the end point of the first arc) is joined to the start point of the second arc. The closepath joins the end of the second arc back to the start of the first and the stroke causes the path to be painted.

– showpage fi –

Just as no graphical object will appear on the page unless filled or stroked, nothing at all will appear on the output page unless the showpage operator is invoked.
Showpage prints a copy of the current page image, erases the contents of the page and resets the graphic state to its initial values.

8. The Coordinate System

Since mentioning the existence of the default user space in the introduction, and hinting that it was flexible, we have resolutely referred to the POSTSCRIPT coordinate system as being first quadrant Cartesian with a resolution of 1/72 inches. Much of the graphical power of POSTSCRIPT comes from the ability to minutely manipulate (transform) the coordinate system via a powerful matrix method.*

Below, we will use as an example a 2cm square ''chunk'' of coordinate system with a 50-point Times Italic ''Z'' positioned at the point (0.5cm, 0.25cm) within it. In order to emphasise the transformations taking place, a grid has been drawn behind the character in black to represent the eventual coordinate system and in various shades of grey to represent its predecessors. The POSTSCRIPT code used to generate the grid is not shown.

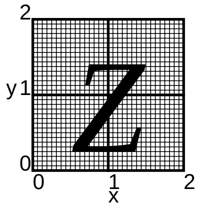
---

* This approach is explained, in far more detail than here, in any reasonable book on Computer Graphics, eg. Foley & van Dam: Fundamentals of Interactive Computer Graphics, Addison Wesley, 1982; or Newman & Sproull: Principles of Interactive Computer Graphics, McGraw-Hill, 1981; etc.

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

.5 cm .25 cm moveto

(Z) show
```



Fortunately, it is not always necessary to employ the matrix manipulation methods directly, there are a number of ‘‘shorthand’’ operators to make simple and common alterations to the coordinate system. These are the operations which rotate, stretch or shrink (scale) and shift (translate) the coordinate system.

angle rotate fi –

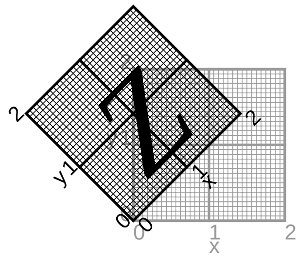
Rotate the current coordinate system through angle degrees relative to the current system. Angle is measured counter-clockwise, as in the arc etc. commands above. All graphical operations, including moves and character imaging, are performed in the current coordinate system.

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

45 rotate

.5 cm .25 cm moveto

(Z) show
```



xs ys scale fi –

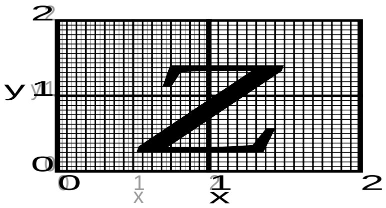
Expand or shrink the coordinate system by xs in the x-direction and ys in the y-direction. Effectively, after executing this operator, every x-coordinate value will be multiplied by xs and every y value by ys. For instance, below, under the influence of the 2 1 scale operation, the Z becomes twice as wide as before, and .5 cm .25 cm moveto has the effect of 1 cm .5 cm moveto in the default coordinate system.

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

2 1 scale

.5 cm .25 cm moveto

(Z) show
```



xt yt translate fi –

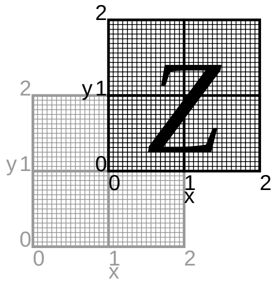
Shift the origin of the coordinate system to (xt,yt) (measured in the current system), eg.

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

1 cm 1 cm translate

.5 cm .25 cm moveto

(Z) show
```



Note that translation does not alter the current physical position (although the numbers returned by the currentpoint operator will alter), i.e. the translate operator does not perform an implicit 0 0 moveto.

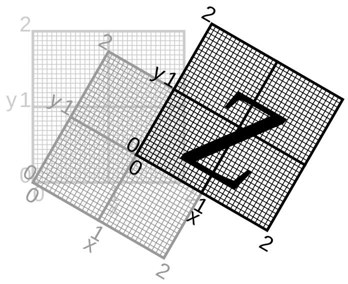
Rotation and translation are often applied in succession, but are not necessarily commutative, compare:

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

-30 rotate
1 cm 1 cm translate

.5 cm .25 cm moveto

(Z) show
```



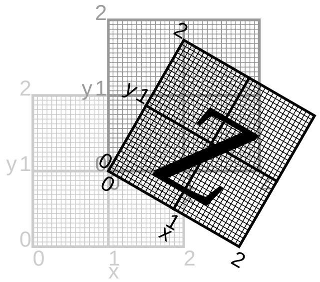
and:

```
/cm { 28.35 mul } def
/Times-Italic findfont
50 scalefont setfont

1 cm 1 cm translate
-30 rotate

.5 cm .25 cm moveto

(Z) show
```



This mechanism can be used to ‘‘rotate’’ images so that the long edge of the apparent page lies in the positive x-direction, that is to produce a landscape rather than portrait page image. If the length of the pages long edge is 297mm (A4), then the instructions:

```
21.0 cm 0 translate
90 rotate
```

will result in a landscape page with the origin in the bottom left corner.

The rotate, translate and scale operators all manipulate an underlying Current Transformation Matrix of the form:

$$\begin{bmatrix} a_x & a_y & 0 \\ b_x & b_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

which, in POSTSCRIPT is represented by the array [a_x a_y b_x b_y c_x c_y]. The device coordinates x¢and y¢can then be derived from the given coordinates x and y according to the following equations:

$$x\text{¢} = a_x x+b_x y+c_x$$

$$y\text{¢} = a_y x+b_y y+c_y$$

The POSTSCRIPT concat operator multiplies the CTM by the ‘‘matrix’’ (six element array) which is given as an argument, producing a new CTM. From the above equations, it can be derived that:

```
2 3 scale
```

is equivalent to:

```
[ 2 0 0 3 0 0 ] concat
```

and:

```
72   144   translate
```

is the same as:

```
[  1  0  0  1  72  144  ]  concat
```

With more effort, it should be apparent that:

```
45   rotate
```

is also expressible as:

```
[  45  cos   45  sin   -45  sin   45  cos   0  0  ]  concat
```

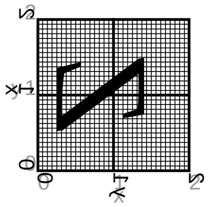
Certain operations can be performed with concat which are not easily obtainable via the shorthand operators, such as coordinate flipping or mirroring. Below, the operation [0 1 1 0 0 0] concat sets up the equations: x¢ = y and y¢ = x

```
/cm   {  28.35   mul  }  def
/Times-Italic        findfont
50  scalefont       setfont

[  0  1  1  0  0  0  ]  concat

.5   cm  .25   cm  moveto

(Z)   show
```



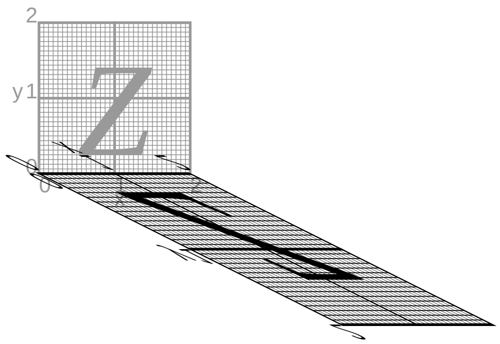
Below a combination of mirroring and differential scaling (the equations are x¢ = x+2y and y¢ = -y) is used to produce a stretched shadow effect (the old Z is shown for comparison).

```
/cm   {  28.35   mul  }  def
/Times-Italic        findfont
50  scalefont       setfont

[  1  0  2  -1   0  0  ]  concat

.5   cm  .25   cm  moveto

(Z)   show
```



Such operations can be used to produce quite sophisticated graphics effects, note the use of gsave/grestore to isolate the changes to the CTM.

```
/Helvetica      findfont      45  scalefont       setfont

gsave
        .03    .09    rmoveto
        [   1  0  2  -1   0  0  ]  concat
        .7   setgray     (Shadow)     show
grestore

(Shadow)     show
```

9. Structuring Conventions

It was noted above that POSTSCRIPT programs generally consist of a prologue, usually hand-written and containing definitions of procedures and data to be used by a script, usually program-generated and containing definitions of page images. POSTSCRIPT does not enforce this structure (and indeed others are used) but in order to enable inter-working between applications such as the inclusion of POSTSCRIPT fragments from one source within programs generated by another, it is necessary to follow certain conventions as to the structure of a program.

The POSTSCRIPT language is context dependent, that is the current state of a program's environment is dependent on it's execution history. Although the graphical state is reset by the showpage operator, any procedures defined or data items altered are available throughout the rest of the program. If these problems are avoided and the program is cleanly divided into a defining prologue and independent page descriptions in a script, then, for example, a general-purpose post-processor can select individual pages, or change the order of page printing, with impunity. To this end, there exists a Document Structuring Convention, implemented by a system of conventional comments, to mark the boundaries of the component parts of the program and to signal its conformity.

The precise format of the comments described below is critical, the leading % must start a line, there must be no space between the %% and the keyword, precisely one space between the : and the first value and one space between values. A newline must immediately follow the last value.

Comments marked with a ‘‘†’’ are compulsory.

%!PS-Adobe-1.0 †

> The first two characters of this comment (%!) must be the first two characters of any POSTSCRIPT program. They are used to indicate to the operating system and sometimes to the printer itself that the program is to be interpreted as POSTSCRIPT, rather than as some other data format to be translated into POSTSCRIPT (cf. the idea of a ‘‘magic number’’ on UNIX). If these two characters are followed by the string PS-Adobe-, then the program is taken to be minimally conforming, ie. it will contain all the compulsory structuring comments noted here. The trailing 1.0 indicates that the program conforms fully to version 1.0 of the structuring convention, ie. this one.

The following comments, called header comments, extend from the line after the above version identifier to the first line not starting with %%.Their order of appearance is not significant. In some cases (marked below with a *), the reporting of the values associated with a keyword may be deferred until the end of the document. In this case the value (atend) should be given in the header comment, and the keyword repeated with the correct values as part of the trailer comments (q.v.).

%%DocumentFonts: font1 font2 ... † *

> Where font1, font2 etc. are the POSTSCRIPT names of the fonts used by the document. This comment is used by systems which must down-load fonts which are not usually resident on the printer.

%%Title: title

> For identification purposes, the title of the document.

%%Creator: name

> The name of the program or person responsible for the creation of the document.

%%CreationDate: date

> The date and time on which this document were created. There is no particular format specified for date.

%%For: name

> The name of the intended recipient. If this is missing, the Creator is assumed.

%%Pages: number *

> The total number of pages (number of showpage operations performed) produced by this document. Must be ‡ 0.

%%BoundingBox: llx lly urx ury *

> The coordinates (in the default user coordinate system) of the bottom left (llx,lly) and top right urx,ury) corners of the notional box surrounding all the marks made on the page by this program.

This is used by systems such which attempt to include POSTSCRIPT programs in others to position the contents on the page relative to the surrounding marks. If the document produces more than one page, these coordinates should take the maximum values produced, or this comment should be omitted.

%%EndComments
: Explicitly ends the header comments.

The following body comments are used to mark the boundaries between the various parts of a POSTSCRIPT program.

%%EndProlog †
: Marks the end of the prologue and the beginning of the script section of the document.

%%Page: label ordinal †
: Signals the start of the script section for an individual page image. The page has number ordinal in the sequence of pages in this document (ie. is an integer between 1 and n for an n page document), but is known as label (eg. xiv or B.7) in the numbering scheme of the generating system. Unknown values should be denoted by a ?.

%%PageFonts: font1 font2 ...
: Specifies the fonts used on the current page, they should of course be a subset of those given in the %%DocumentFonts list. If present, this comment should immediately follow a %%Page comment.

%%Trailer †
: Marks the end of the last page of the document, any code which follows is assumed to be part of the document itself rather than a particular page. Following any such code will be the header comments for which the (atend) value was given.

10. Operator Reference

Below is an alphabetical listing of all portable POSTSCRIPT operators, together with a note of their expected parameters and results (all on the operand stack).

abs	num Þ |num|
add	num1 num2 Þ (num1+num2)
aload	array Þ elem1..elem2.. array
anchorsearch	string seek Þ found: spost smatch true not found: string false
and	a b Þ aANDb (bitwise if a,b are integers)
arc	x y r ang1 ang2 Þ –
arcn	x y r ang1 ang2 Þ –
arcto	x1 y1 x2 y2 r Þ xt1 yt1 xt2 yt2
array	int Þ arry-of-size-int
ashow	ax ay string Þ –
astore	elem1..elemk k Þ arry[elem1..elemk]
atan	a b Þ angle-whose-tan-is-(a/b)
awidthshow	cx cy char ax ay string Þ –
begin	dict Þ –
bind	proc Þ proc
bitshift	int shift Þ int-shifted (right: +, left: -)
bytesavailable	file Þ int (-1 if indeterminate)
cachestatus	– Þ bsize bmax msize mmax csize cmax maxbits
ceiling	number Þ least-integ-grtr-than-or-eq-to
charpath	string strokepath-bool Þ –
clear	a..b..c.. Þ –
cleartomark	stuff mark a..b..c.. Þ stuff
clip	– Þ –
clippath	– Þ –
closefile	file Þ –
closepath	– Þ –
concat	matrix Þ –
concatmatrix	mtrx1 mtrx2 mtrx3 Þ mtrx3 (=mtrx1·mtrx2)
copy	a..b..c.. n Þ a..b..c.. a..b..c.. (top n elem)
copypage	– Þ –
cos	a Þ cosine(a)
count	a..b..c.. Þ a..b..c.. count
countdictstack	– Þ count
countexecstack	– Þ count
counttomark	mark a..b..c.. Þ mark a..b..c..count
currentdash	– Þ array offset
currentdict	– Þ dict
currentfile	– Þ file
currentflat	– Þ number
currentfont	– Þ font-dict
currentgray	– Þ number
currenthsbcolor	– Þ hue satur bright
currentlinecap	– Þ integer
currentlinejoin	– Þ integer
currentlinewidth	– Þ number
currentmatrix	matrix Þ CTM-in-matrix
currentmiterlimit	– Þ number
currentpoint	– Þ x y
currentrgbcolor	– Þ red green blue
currentscreen	– Þ freq rot spot-funct
currenttransfer	– Þ gray-trans-funct
curveto	x0 y0 x1 y1 x2 y2 Þ –
cvi	num Þ integ or strng Þ int
cvlit	any Þ literal (not-exec)
cvn	string Þ name
cvr	num Þ real or strng Þ real
cvrs	num base string Þ substring
cvs	any string Þ substring
cvx	any Þ executable

def	key value Þ –
defaultmatrix	matrix Þ def-matrix
definefont	key dict Þ font-dict
dict	int Þ dict (maximum-capacity: int)
dictstack	array Þ subarray
div	num1 num2 Þ (num1/num2)
dtransform	x y Þ xt yt or x y matrix Þ xt yt
dup	any Þ any any
echo	bool Þ –
end	– Þ –
eoclip	– Þ –
eofill	– Þ –
eq	a b Þ bool (true if a=b)
erasepage	– Þ –
exch	a b Þ b a
exec	any Þ –
execstack	array Þ subarray
executeonly	arry Þ exec-only-arry (or string)
exit	– Þ –
exp	num1 num2 Þ num1-to-the-pwr-num2
false	– Þ false
file	string1 string2 Þ file (string2: r, w)
fill	– Þ –
findfont	key Þ font-dict
flattenpath	– Þ –
floor	number Þ greatest-int-less-than-or-eq-to
flush	– Þ –
flushfile	file Þ –
for	init incr limit proc Þ –
forall	array proc Þ –
ge	num1 num2 Þ bool (true if num1>=num2)
get	array index Þ elem or dict key Þ value
getinterval	arry beg len Þ subarry (also strings)
grestore	– Þ –
grestoreall	– Þ –
gsave	– Þ –
gt	num1 num2 Þ bool (true if num1>num2)
identmatrix	matrix Þ id-transf-mtrx
idiv	int1 int2 Þ int-part-of(int1/int2)
idtransform	xdt ydt Þ xd yd (xdt ydt mtrx Þ xd yd)
if	bool proc Þ –
ifelse	bool true-proc false-proc Þ –
image	scan-len scan-lns bits/pixl mtrx proc Þ –
imagemask	scan-len scan-lns invrt mtrx proc Þ –
index	a1..a2..a3...ak t Þ a1..a2..a3..ak a(k-t)
initclip	– Þ –
initgraphics	– Þ –
initmatrix	– Þ –
invertmatrix	mtrx1 mtrx Þ mtrx (inverted-mtrx1)
itransform	xt yt Þ x y (xt yt mtrx Þ x y)
known	dict key Þ bool
kshow	proc string Þ –
le	num1 num2 Þ bool (true if num1<=num2)
length	array Þ length-of-arry (also strings)
lineto	x y Þ –
ln	num Þ natural-log-of-num
load	key Þ value
log	num Þ common-log-of-num
loop	proc Þ –
lt	num1 num2 Þ bool (true if num1<num2)
makefont	font-dict matrix Þ transformed-font-dict
mark	– Þ mark
matrix	– Þ matrix
maxlength	dict Þ int
mod	int1 int2 Þ int1MODint2
moveto	x y Þ –
mul	num1 num2 Þ num1·num2
ne	num1 num2 Þ bool (false if num1=num2)
neg	num Þ –num
newpath	– Þ –
not	a Þ NOTa (bitwise if a is integer)
null	– Þ null
or	a b Þ aORb (bitwise if a,b are integers)
pathbbox	– Þ llx lly urx ury
pathforall	mveto-p lneto-p crveto-p clsepth-p Þ –
pop	any Þ –
print	string Þ –
prompt	– Þ –
pstack	a..b..c.. Þ –
put	array index value Þ – (also strings)
putinterval	arry1 beg arry2 Þ arry1 (also strings)
quit	– Þ –
rand	– Þ int
rcheck	array Þ bool (true if readable)
rcurveto	dx0 dy0 dx1 dy1 dx2 dy2 Þ –
read	file Þ byte bool (false if EOF)
readhexstring	file string Þ substring bool
readline	file string Þ substring bool
readonly	array Þ ReadOnly-array
readstring	file string Þ substr bool (false if EOF)
repeat	count proc Þ –
restore	save-objct Þ –
reversepath	– Þ –
rlineto	dx dy Þ –
rmoveto	dx dy Þ –
roll	a..b..c.. N R Þ a..b..c.. (N rolled by R)
rotate	angle Þ – or angle mtrx Þ mtrx
round	num Þ num-rounded
rrand	– Þ current-random-nr-seed-state
run	string Þ –
save	– Þ save-object
scale	sx sy Þ – or sx sy mtrx Þ mtrx
scalefont	font-dict number Þ transformed-font-dict
search	string seek Þ found: spost smatch spre true not found: string false
setcachedevice	wx wy llx lly urx ury Þ –
setcachelimit	maxbytes Þ –
setcharwidth	wx wy Þ –
setdash	array offset Þ –
setflat	num Þ –
setfont	font-dict Þ –
setgray	num Þ –
sethsbcolor	hue satur bright Þ –
setlinecap	integer Þ –
setlinejoin	integer Þ –
setlinewidth	num Þ –
setmatrix	matrix Þ –
setmiterlimit	num Þ –
setrgbcolor	red green blue Þ –
setscreen	freq rotation spot-function Þ –
settransfer	gray-transfer-funct Þ –
show	string Þ –
showpage	– Þ –
sin	num Þ sine(num)
sqrt	num Þ square-root-of-num
srand	int Þ –
stack	a..b..c.. Þ a..b..c..
start	– Þ –
status	file Þ bool (true if open)
stop	– Þ –
stopped	proc Þ bool (false if proc stopped)

store	key value Þ –
string	int Þ string-of-length-int
stringwidth	string Þ wx wy
stroke	– Þ –
strokepath	– Þ –
sub	num1 num2 Þ num1-num2
systemdict	– Þ system-dict
token	file Þ any bool (true if found)
token	string Þ found: spost token true not found: false
transform	x y Þ xt xy or x y mtrx Þ xt yt
translate	tx ty Þ – or tx ty mtrx Þ mtrx
true	– Þ true
truncate	num Þ num-truncated
type	any Þ type-name-of-a
userdict	– Þ user-dict
usertime	– Þ time-in-msecs
version	– Þ soft-&-hard-version-string
vmstatus	– Þ save-level bytes-used bytes-avail
wcheck	array Þ bool (true if writeable)
where	key Þ found: dict true not found: false
widthshow	dx dy char-code string Þ –
write	file int Þ –
writehexstring	file string Þ –
writestring	file string Þ –
xcheck	any Þ bool (true if a is executable)
xor	a b Þ aXORb (bitwise if a,b are integers)
=	a..b..c.. Þ –
==	a..b..c.. Þ –