A First Guide to PostScript

Peter Weingartner

24 February, 2006

This is the fifth edition of the *First Guide to PostScript*. It differs from the previous editions in that a number of errors which people have brought to my attention have been fixed and a number of common reader questions have been addressed. I have also added some information on how to work with colors and raster graphics. It is my hope that this document is now stable and reasonably error-free. If you find an error, please send me e-mail and let me know. I can't promise that I'll fix it right away, but I will at least add it to my list of things to do.

My sincere thanks goes out to everyone who has sent me e-mail concerning the guide. Whether you were asking a question, or offering me a correction, I sincerely appreciate it. My only regret is that I have not been able to be as responsive to questions and corrections as I would like.

I left Indiana University quite a long time ago (nearly ten years as of the time of this writing), and while I still have write access to my old account space I can not be sure that I always will have access. I will maintain the original copy at Indiana University while I have access, but from now on the official copy[1] will be maintained at my personal website[2], where you can also find out a *little* more about me, if you are so inclined.

About this Document

This is meant to be a simple introduction to programming in the PostScript page description language from Adobe[3]. This document is not meant to be a comprehensive reference manual (although it does contain an index of some of PostScript's standard operators and a list of various errors). There are far better reference books, if this is what you need. Instead, this is meant as an easily accessible on-line tutorial. It was written with the assumption that you have some experience programming and are familiar with concepts like arrays and variables.

The scope of this document is fairly limited. I cover only a subset of PostScript Level 1 (the earliest version). Since I started this guide, Adobe brought out two revisions to the language: called Level 2 and Level 3. This document was never

[1] http://www.tailrecursive.org/postscript/postscript.html
[2] http://www.tailrecursive.org
[3] http://www.adobe.com

meant to cover these versions of PostScript (although the code I present here should run just fine on a Level 2 or Level 3 capable printer). Likewise, I do not cover any advanced printing concepts like color separations or halftone screens (this is mainly due to ignorance on my part, I am an engineer… not a printer or graphic designer… although I do admire good graphic design when I see it).

I have created this document because I have noticed that many people on the Internet have been asking for some online document to get them started. I decided that this was a good opportunity. I have benefited from the free and open nature of the Internet (most of the software I use is freeware or shareware). This is my opportunity to give something back to the community and to try to perpetuate something of the original community atmosphere that existed when I first started using it.

Contents

Note

Legal questions concerning these issues should be directed to them. Please note that this site is not related to, supported by, or condoned by Adobe in any way. It is an independent site and is not official.

Disclaimer

No warranty or guarantee, either expressed or implied, is made as to the correctness of this document. The author can not be held responsible for any damages that may occur through the use of any code contained herein.

You get what you paid for.

Copyright Information

[4] When I first started this guide, there was no convenient way to put something out there in such a way that you could keep the copyright, but still allow people to make copies or even derivative works. Now there is, through the joys of the Creative Commons[5]. So, since the Creative Commons licenses are now available, this new version is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License[6].

What is PostScript?

PostScript is a programming language optimized for printing graphics and text (whether on paper, film, or CRT is immaterial). In the jargon of the day, it is a **page description language**. It was introduced by Adobe in 1985 and first (to my knowledge) appeared in the Apple LaserWriter. The main purpose of PostScript was to provide a convenient language in which to describe images in a device independent manner. This device independence means that the image is described without reference to any specific device features (*e.g.* printer resolution) so that the same description could be used on any PostScript printer (say, a LaserWriter or a Linotron) without modification. In practice, some PostScript files do make assumptions about the target device (such as its resolution or the number of paper trays it has), but this is bad practice and limits portability.

The language itself, which is typically interpreted, is stack-based in the same manner as an RPN calculator. A program pushes arguments for an operator onto a stack and then invokes the operator. Typically, the operator will have some result which is left at the top of the stack. As an example, let us say we want to multiply 12 and 134. We would use the following PostScript code:

```
12 134 mul
```

[4] http://creativecommons.org/licenses/by-nc-sa/2.5/
[5] http://www.creativecommons.org
[6] http://creativecommons.org/licenses/by-nc-sa/2.5/

The first two words "12" and "134" push the numbers 12 and 134 onto the stack. "mul" invokes the multiply operator which pops two values off the stack, multiplies them, and then pushes the result back onto the stack. The resulting value can be left there to be used by another operator later in the program.

To follow the conventions used by Adobe in their manuals, I will synopsize operators using the following scheme: *arg-1 arg-2* … **operator** *result*. This scheme means that, to use **operator**, you must push arguments *arg-1*, *arg-2*, and so on before invoking the operator. **operator** will return the result: *result*. Many operators return no result (they have some side-effect); these will be shown as returning "**-**".

Graphics Concepts

There are a few concepts that you need to know about before we dive into the language itself. These concepts are the concepts PostScript uses to describe and manipulate images on a page. There are really only a few.

Device Space This is the coordinate space understood by the printer hardware. This coordinate system is typically measured in terms of the device's resolution. There is really nothing else that can be said about this space, as PostScript programs are typically not expressed using it.

User Space This is the coordinate system used by PostScript programs to describe the location of points and lines. User space is essentially the same as the first quadrant of the standard coordinate system used in high school math classes. Point (0, 0) is in the lower left corner. Coordinates are real numbers, so there is no set resolution in user space. The interpreter automatically converts user space coordinates to device space.

Current Transformation Matrix The transformation of user space coordinates to device space coordinates is done through the current transformation matrix. This matrix is a three by three matrix that allows the user to rotate, scale, and translate the entire user space within the device space. This is the source of a lot of PostScript's power, as will be demonstrated later.

Path A path is a collection of (possibly disjoint) line segments and curves arranged on the page. The path does not describe actual ink on the paper; it merely describes an imaginary tracing over the page. There are operators which allow the user to draw ink along the path (stroke), fill an enclosed path with ink (fill), or clip out all future images that are outside the path (clip).

Current Path This is the path that the PostScript program is creating at the moment. The current path is assembled piece by piece.

Clipping Path The PostScript rendering system will ignore any part of a line segment, curve, or bitmap that extends outside a certain region; it will only draw the parts of those elements which are within the region. The region is described by a path called the clipping path. The clipping path is usually a rectangle about a quarter of an inch in from the edge of the page, but it can easily be set by the user to an arbitrary path.

Graphics State This is a collection of various settings that describe the current state of the graphics system. Things like the current path, the current font, and the current transformation matrix make up the graphics state. Often, a program will need to temporarily save a graphics state to be used later. There are a couple of ways of doing this, but the easiest is to push the state onto a special graphics state stack and pop it back later. This can be accomplished with the gsave, and grestore operators.

Language Concepts

As a programming language, PostScript is particularly simple. There are really only a few concepts that need to be sketched out.

Comment A comment in PostScript is any text preceded by a "%". The special comment "%!" as the first two characters of a PostScript program is seen as a tag marking the file as PostScript code by many systems (including Unix's `lpr` command). It is a good idea to start every PostScript document with a "%!"… doing so will ensure that every spooler and printer the document may encounter will recognize it as PostScript code.

Stack There are several stacks in a PostScript system, but only two are important for this guide: the operand stack, and the dictionary stack. The operand stack is where arguments to procedures (or operators, in PostScript jargon) are pushed prior to use. The dictionary stack is for dictionaries, and it provides storage for variables.

Dictionary A dictionary is a collection of name-value pairs. All named variables are stored in dictionaries. Also, all available operators are stored in dictionaries along with their code. The dictionary stack is a stack of all currently open dictionaries. When a program refers to some key, the interpreter wanders down the stack looking for the first instance of that key in a dictionary. In this manner, names may be associated with variables and a simple form of scoping is implemented. Conveniently, dictionaries may be given names and be stored in other dictionaries.

Name A name is any sequence of characters that can not be interpreted as a number. With the exception of spaces and certain reserved characters (the characters "(", ")", "[", "]", "<", ">", "{", "}", "/", and "%") any character may be part of a name. The name may even start with digits (1Z is a name, for example), but you can get into problems with them (1E10 is a real number). A name is seen as being a reference to some value in a dictionary on the dictionary stack.

It should be noted that there are a couple of names that are legal in PostScript which do not follow the above definition. These are the "[" and the "]" operators. Yes, they *are* operators and are stored in the dictionary. Some other names that might surprise you are: "=", "==", "<<", and ">>".

If a name is preceded by a slash, PostScript will place the name on the stack as an operand. If the name has no slash, the interpreter will look up its value in the dictionary stack. If the value is a procedure object, the procedure will be evaluated. If the value is not a procedure, the value will be pushed onto the

operand stack.

Number PostScript supports integers and reals. You can express numbers in two forms: radix form, and scientific notation. Radix form is a number of the form *radix#value* where *radix* specifies the base for *value*. Scientific notation is the standard *mantissa*E*exponent* form used in most languages.

String Strings are, of course, just strings of characters. There are two ways of expressing strings in Level 1 PostScript. The most common way is to wrap your text in parentheses. For example the string "This is a string" would be written as `(This is a string)`. You can also express a string as hexadecimal codes in angle brackets. For example, the string "ABC" would be expressed as `<414243>`. There are several escape codes that may be used in the parenthesis format of strings.

Array Arrays in PostScript are like arrays in any other language. Arrays may contain objects of different type, and they are written as a list of objects surrounded by brackets. For instance, `[12 /Foo 5]` is a three element array containing the number 12, the name Foo, and the number 5.

Procedure A procedure is your way of defining new operators. A procedure is an array that is executable and is written with braces rather than brackets. For example, a procedure to square the top element on the stack might be written as: `{dup mul}`. We can define this procedure to be the square operator with: `/square {dup mul} def`.

Programming in PostScript

Programming in PostScript is really pretty easy. The fundamentals are that you push operands onto the operand stack by naming them, and then you invoke the operator to use them. That's really all there is to it. The real art is knowing which operator to use. Operators to draw and put text on the screen will be covered later, and these make up the bulk of PostScript code, but there are a couple that are used mainly for maintaining the program itself.

The first of these operators is def. "def" is responsible for entering a definition into the top-most dictionary on the dictionary stack. The top operand on the operand stack is the value, and the operand below the value is the key (and should be a name). Let's say that we wanted to define the name "x" to have a value of 5. The PostScript to do this is: `/x 5 def`. Notice the use of the slash on the "x". The slash ensures that the *name* "x" will be pushed onto the stack and not any value it may already have in the dictionary stack.

"def" is also used to define new operators. The value in this case is just a procedure. The following code defines an operator "foo" which adds its top-most two operands and multiplies the result with the next operand on the stack: `/foo {add mul} def`. Remember, operators that return results push them onto the stack, where they may be used later.

An important point to know when defining procedures is that the elements in the procedure are not evaluated until the procedure is invoked. That means that in the procedure `{1 2 add 3 mul}`, the actual names "add" and "mul" are stored in the array that is the procedure. This is different from an actual array

in which the components *are* evaluated when the array is created. For contrast, the array `[1 2 add 3 mul]` contains one object: the *number* 9.

This delayed evaluation of procedure components has two important effects. First, the definition of an operator used in a procedure is the one that is in effect when the procedure is run, *not* when it is defined. Second, because each operator has to be looked up each time the procedure is invoked, things can be a little slow. Fortunately, PostScript provides a handy operator to replace each name in a procedure object with its current definition. This operator is called bind, and it can speed up your program considerably. Bind is typically used as:

```
/foo {add mul} bind def
```

This defines foo to be a procedure array with two components: the procedures for add and mul. Note that, if add or mul is re-defined after defining foo, foo will have the *same* behavior as before. Without the use of bind, foo's behavior would change.

Drawing and Filling Shapes

Principles

The main purpose of PostScript is to draw graphics on the page. One of the elegant aspects of PostScript is that even text is a kind of graphic. The main task that must be mastered, then, is constructing paths which may be used to create the image.

To draw and fill shapes, the basic sequence is:

- Start the path with the newpath operator.
- Construct the path out of line segments and curves (the path need not be contiguous).
- Draw the path with the stroke operator or fill it in with the fill operator.

This basic sequence can be modified to do more complicated things as we will see later.

Drawing a Box

In this first example, we will draw a square inch box toward the lower left corner of the page. We start off by defining a function to convert inches into PostScript's main unit, the point (a point is defined in PostScript as 1/72th of an inch, which is slightly shorter than a true printer's point of 1/72.27 inch). The conversion is simple, we just multiply the number of inches by 72. This gives us the function

```
/inch {72 mul} def
```

To actually draw the square, we start a new path and move the current point to a point an inch in from both margins. This is accomplished with the code:

```
newpath
1 inch 1 inch moveto
```

At this point, the path contains only the point (72, 72). We add in line segments leading away from this point with the lineto operator. This operator adds a line segment from the current point to the point specified to lineto and makes that point the new current point. We can build three sides of the box as follows:

```
2 inch 1 inch lineto
2 inch 2 inch lineto
1 inch 2 inch lineto
```

We can add the last line by telling PostScript to close up the path with the smallest possible line segment. The closepath operator does this. This operator is especially useful if you need a closed figure for filling. Once we have closed the path, we can draw it with the stroke operator. We finish off the example by ejecting the page (if you are using a printer). PostScript ejects a page with the showpage operator:

```
closepath
stroke
showpage
```

You can view and try the complete example, if you like.

Refinements

The lineto operator works in absolute coordinates within user space. That is, `72 72 lineto` adds a line segment from the current point to the point (72, 72) in user space. In drawing the box, however, it is more convenient to ignore the absolute coordinates of the box's vertexes and think instead of the lengths and directions of its sides. Fortunately, PostScript provides a version of lineto which takes *relative* coordinates instead. This is the rlineto operator. rlineto adds the coordinates given as operands to the coordinates of the current point in the path to find the destination point. That is, `10 20 rlineto` will draw a line from the current point to a point 10 points to the right and 20 points toward the top of the page. This is in contrast to `10 20 lineto` which adds a line segment which *always* ends at (10, 20).

To see how we can use rlineto, let's replace the lineto lines in the last example with the following code:

```
1 inch 0 inch rlineto
0 inch 1 inch rlineto
-1 inch 0 inch rlineto
```

This new example will draw the same figure, but it draws the lines using relative coordinates instead of absolute. This makes it a little easier to visualize and has the added benefit that the same code can draw the three lines at a different location. Note that a negative relative x coordinate moves the point in the left direction while a negative relative y coordinate moves the point down the page.

Filling Shapes

Filling a shape is just as easy as drawing it. You create the path using the standard path creation operators, but instead of calling stroke at the end, you invoke the fill operator. The fill operator will fill the path with the current ink settings. If you want to fill a shape with a pattern, you will need to do some special tricks which we will cover later. We will use the box from above as an example, but we replace the original invocation of stroke with fill.

Fill uses a simple winding rule (which is described in the *Programming Language Reference Manual*) to determine what parts of the page are inside or outside the path. The regions that are inside are painted. Note that arbitrarily complex shapes can be filled with this operator so long as you have enough memory on your PostScript interpreter. You can easily fill in different shades and even some patterns, but to fill an area with a complex image takes some special effects which we will cover later.

Shading and Width

In PostScript, you can view lines as being drawn by pens that have a given width and ink as having particular shades. You are not restricted to completely black ink and one-point wide lines. PostScript provides two handy operators to change these characteristics.

The setgray operator sets the intensity of the ink used in drawing lines and filling shapes (actually, setgray affects all subsequent markings made on the page). setgray takes a single numerical argument between 0 and 1. "0" signifies black, and "1" signifies white. Numbers between these two values signify various shades of gray.

The setlinewidth operator does just what its name suggests: it sets the width of lines to be drawn. It takes a single numerical argument which is the width of the line in points. setlinewidth affects all lines *stroked* after the operator is invoked.

Both of these operators affect the markings placed on the page after they are called… they do not effect the path until it is stroked or filled. In particular, you can not set the width or gray level for one part of the path and then change it for another… they are the same for all parts of the path, since it is stroked or filled only once. Also, both of these operators affect part of the graphics state and can be saved with gsave and restored with grestore.

I have worked up an example using both these operators. I also demonstrate how you can use gsave and grestore to control the graphics state.

Putting Text on the Page

Printing text on a page is, understandably, a simple process. It consists basically of these main steps:

- Set up a font to use
- Set the current point to where the lower left corner of the text will be
- Give the string to print to the show operator

The show operator is the basic operator for printing strings of text. It takes a string and prints it out in the current font and with the lower left corner at the current point. After the text has been printed, the current point is at the lower right of the string.

Fonts

Fonts in PostScript are actually dictionaries. A font dictionary contains several operators. Most of these operators simply set up the path for a single character in the font. When PostScript needs to typeset an "A" in the current font, it finds the operator specified in the font for "A" and invokes it. This operator goes about the business of drawing the letter. This means that there is no fundamental difference between letters and any other kind of ink on the page: text *is* graphics. Furthermore, since a font is essentially just a program to draw things, the current graphics state applies to text just as much as it applies to lines and curves which your program draws. This is one of the most powerful features of PostScript, as we will see later.

The fonts themselves are stored in a special dictionary of fonts, and they are named. If you want to retrieve a font by name, you need to use the findfont operator. findfont retrieves the font from the dictionary (if it is there) and leaves the font on the stack. You can then specify how big the font should be and make it the current font. The basic process for setting the font is:

- Retrieve the font from the dictionary with findfont,
- Set the size of the font with scalefont,
- Make this new font the current font with setfont

scalefont takes two arguments, the lower argument on the stack is a font dictionary while the second is the size of the new font in points. scalefont returns a new font dictionary which is the same as the old one but scaled to the given size. setfont, on the other hand, takes a font dictionary and makes it the current font.

For example, let us say that we want to start typesetting in Times Roman, and we want it to be set to 20 points. The following code would set up the correct font:

```
/Times-Roman findfont   % Get the basic font
20 scalefont            % Scale the font to 20 points
setfont                 % Make it the current font
```

Since the font "Times-Roman" is stored in a dictionary, we search for it using its PostScript name. Your printer will usually come with a set of built in fonts and will almost always allow you to add more. The names of the fonts available will vary from printer to printer, but Times is almost always present. Fonts typically come in families. "Times" is the name of the family we used here, and it has four member fonts: Times-Roman, Times-Italic, Time-Bold, and Times-BoldItalic.

Showing Text

The show operator is used to typeset text on the page. It takes a single argument: a string containing the text to be typeset. Text can be considered to be part of the path, so you must also have set the current point with call to moveto or an equivalent operator. A typical call to show might look like this:

```
newpath                 % Start a new path
72 72 moveto            % Lower left corner of text at (72, 72)
(Hello, world!) show    % Typeset "Hello, world!"
```

If we ran this code right after the font selection code above, we would get the string "Hello, world!" printed an inch in from the lower left corner, and it would be printed in 20 point Times-Roman. You can actually try this example.

Adding Color

In recent years, color printers have become more common… to the point where it is almost impossible to buy a black-and-white printer. Fortunately, PostScript handles color documents quite easily, and you do not have to learn much to add color to your documents.

There are many, many ways to specify color. The science of color reproduction and perception is very complex and encompasses physics, chemistry, physiology, and psychology. PostScript provides a great deal of flexibility on this front, providing several different methods for specifying color to let you get as close as possible to the color you want specifying it in the way that is most natural in your application. I will discuss only two methods here, however, because—frankly—I know as much about color as a bee knows about ancient Phoenician.

RGB

The first method I will mention is the so-called RGB colorspace. In this model, you specify the red, green, and blue components of the color you want to reproduce. This is model specifies color using the additive primaries and will be very familiar to those who work with monitors and computer graphics.

To specify a color using the RGB color space, you can use the setrgbcolor operator. This operator takes three operands: the red, green, and blue components of the color you want (0 means none, 1 means maximum). For example, to specify red, you can write `1 0 0 setrgbcolor`; and to get a dark yellow, you can write `0.5 0 0.5 setrgbcolor`. Black is `0 0 0 setrgbcolor`, and white is `1 1 1 setrgbcolor`.

CMYK

Another way to specify color is through the CMYK colorspace. In this model, you specify color using the subtractive primaries more common in the printing

world: cyan, magenta, and yellow. To the primaries, a black component is added, to allow you to control the tone of the color (this is not necessary in the RGB model, since you are controlling the intensity of the light inherently).

To specify a color using the CYMK color space, you can use the setcymkcolor operator. This operator takes four operands: the cyan, yellow, magenta, and black components of the color you want (0 means none, 1 means maximum). For example, to specify red, you can write `0 1 1 0 setcymkcolor`; and to get a dark yellow, you can write `0 1 0 0.5 setcymkcolor`.

One convenient property of PostScript is that color is handled as a property of the ink in the graphics state just like the gray level. In fact, the gray level set by setgray is really just color. What this means is that you can set the color of a line, or the color of a fill in exactly the same way as you set the gray level. You just call the operator before you stroke the line or fill the path. You can even use it to set the color of text by calling the appropriate operator before you call show.

Since color works in such a similar way to the setgray operator, I am providing you with a complete example, but I will not break it down here.

A Warning

I have really simplified things considerably in this discussion. There are more than just two color spaces in PostScript, and there are several features in PostScript to allow for accurate color rendition. One thing that I've glossed over is that different devices reproduce color with different degrees of accuracy (and different pigments not only have different color-accuracy but will vary over time). Another thing that I have glossed over is the fact that these two models are focused around color reproduction: it is also possible to specify color based on how it is perceived.

If you do not care about color reproduction to that level of precision, then you do not need to worry about it. The color operators I have described are sufficient. If you do, you know far more about the matter than I do and will likely find the features you need described in the PLRM.

Transformations

The PostScript interpreter keeps track of a matrix called the current transformation matrix. When constructing an image, the interpreter uses this matrix to convert the world coordinates used by the program into device coordinates used by the printer itself. Generally, the actual contents of the matrix are of little interest to a well-written PostScript program; the reason for this is that the specific contents are device-dependent. A program that uses them might not work properly. PostScript does provide a number of operators, however, that transform the matrix in a device-independent way. These operators allow you to transform the way user space maps onto device space, and they modify

the current transformation matrix with a simple matrix transformation. The basic transformation operators are:

- rotate
- translate
- scale

It is useful to realize that the current transformation matrix (and, hence the effect of all these operators) is part of the current graphics state and can be saved and restored using the gsave, and grestore operators. In addition, the transformations on the matrix affect path components constructed *after* the transformation. Even if a path is only partially constructed when a transformation is invoked, the parts of the path that were in place before the transformation will be unaffected.

Rotate

The rotate operator takes a single, numerical operand. This operand specifies how many degrees to rotate the user space around its origin (positive values specify counter clockwise rotations). This transform allows you to do some pretty neat tricks. For example, let's say you have written a routine to draw some complex shape; and you have found that you need to draw it several times at different angles. In a more primitive graphics system, you might need to re-write to routine to take an angle as an argument, but in PostScript you only need to rotate the coordinates with the rotate operator.

As a concrete example, let's say you want to draw lines in a circular pattern so that each line is ten degrees from its neighbors. Rather than figure out the coordinates for each of the 36 lines, we can just draw a horizontal line and rotate it repeatedly to different angles. To do the repeated looping, we can use the for operator. The for operator takes four arguments: an initial index value, a step size, a final index value, and a procedure. The operator increments an index from the initial value to the final value, incrementing it by the step size. For each index value, for will push the index on the stack and execute the procedure. This gives you a simple means of looping.

We start by setting up the for loop. At the beginning of the loop's procedure, we start a new path and save the graphics state.

```
0 10 360 {              % Go from 0 to 360 degrees in 10 degree steps
  newpath               % Start a new path
  gsave                 % Keep rotations temporary
```

We next set the start of the line to (144, 144) and rotate the coordinates, we do not rotate *before* moving because (144, 144) would then be in a different location.

```
  144 144 moveto
  rotate                % Rotate by degrees on stack from 'for'
```

We next draw just a horizontal line:

```
  72 0 rlineto
  stroke
```

Finally, we restore the old graphics state and end the loop.

```
  grestore              % Get back the unrotated state
} for                   % Iterate over angles
```

Translate

The translate operator takes two operands: an x-coordinate, and a y-coordinate. The translate operator sets the origin of user space to the point that was at the given coordinates in user space. The main use of the translate is to draw copies of a shape in different locations. Typically, a shape will be constructed at the origin, and the shape will be translated to the correct location before it is to be drawn. A simple example translates a box constructed at the origin to the point (72, 72) in the original user space.

Scale

The scale operator takes two arguments: an x scale factor, and a y scale factor. The operator scales each coordinate by its associated scale factor. That is, if you have an x scale factor of 0.5 and a y scale factor of 3, the x coordinate will be reduced by a factor of two while the y coordinate will be magnified by a factor of 3. This operator allows you to change the size and dimensions of objects quite easily.

A simple example can just scale text in a couple of ways: We can make things narrow:

```
gsave
  72 72 moveto
  0.5 1 scale           % Make the text narrow
  (Narrow Text) show    % Draw it
grestore
```

We can make things tall:

```
gsave
  72 144 moveto
  1 2 scale             % Make the text tall
  (Tall Text) show      % Draw it
grestore
```

We can distort the text completely:

```
gsave
  72 216 moveto
  2 0.5 scale           % Make the text wide and short
  (Squeezed Text) show  % Draw it
grestore
```

Combining Transformations

Each of these transformations merely modifies the current transformation matrix. This means that these operators can be combined for some interesting effects. For example, you can take a normal document and print two of its pages on a single page (reduced and placed side-by-side) simply by translating the first page to one side, rotating the page by ninety degrees and then reducing the page so that it fits. The second page is handled in the same manner, but is translated to the other side of the page. This can be easily done by PostScript post-processors so long as they know where one page ends and the next begins (this is often accomplished using special comments). A somewhat simpler example is to draw a simple box and some text translated, rotated, and scaled in various ways. An important thing to remember when viewing this example is that translations are always relative to the *current user space*. This means that

```
0.5 0.5 scale
72 72 translate
```

will have a different effect on the image than does

```
72 72 translate
0.5 0.5 scale
```

In the first case, the origin will be half an inch in from the bottom and left margins. In the second case, the origin will be an inch in from the two margins.

Clipping for Effect

Within the graphics state of a PostScript system is a special path called the clipping path. Every bit of ink to be placed on the page is checked against this path. If PostScript determines that the ink would go outside the current clipping path, that portion of ink is ignored. If the ink would be within the clipping path, it is actually placed on the page. For the mathematically inclined, the clipping process is intersection: the set of pixels to be painted is intersected with the set of pixels within the current clipping path to get the set of pixels to paint. For objects that are partly inside and partly outside the clipping path, the natural implication is that only the part that is within the clipping path is drawn.

By default, the clipping path is defined to be a rectangle just within the boundary of the page (usually it is set to about a quarter of an inch). You can set your own clip path by constructing the path with the normal path construction operators and invoking the clip operator. There is only one difficulty: once you reduce the size of the current clipping path, there is no way to expand the size of the clipping path with clip. The only way to go back to a larger clipping path is to save the one you would like to restore with gsave and restore it later with grestore. In fact, it is always good policy to only set a clipping path withing a bracketing gsave/grestore pair. You will always be safe if you do this.

Clipping a Simple Path

As a simple example of clipping, let us say that we want to draw a box and fill it with text in such a way that some text is cut off. The effect we are wanting is that of a hole in a piece of paper over some newsprint, say. This can be done quite simply.

First, we set up the box to act as our window. We can set up the path, stroke it if we want to see it, and then clip to it:

```
gsave                    % Save the old clip path
  72 72 box              % Set up our box
  gsave                  % Don't allow box to be lost after stroke
    stroke
  grestore               % Restore the box path
  clip                   % Clip to the box
```

The clip path is now established, and we can now go on to draw the text that should be clipped (note that there is a leading gsave… this is to keep us from loosing our old clip path which covered the whole page).

```
60 60 moveto
(This is Times-Roman clipped to a box) show
70 90 moveto
(This is Times-Roman clipped to a box) show
50 120 moveto
(This is Times-Roman clipped to a box) show
```

Once we have finished, we can just do a grestore to clean up after ourselves.

While there are some implementation limitations on the complexity of the clip path, in general you can have very complex paths… not just squares. Arcs, lines, even text can be used to create the clip path.

Clipping to Text

There may come a time when you will want to do some special effects with text. For example, you might want to print out "July 4" using letters that look like the flag. This is fairly easy to do using clipping. A somewhat simpler problem would be to draw text that looks like a sunburst (that is, the text is filled with a sunburst pattern). This is also fairly easy to do once you know how to clip to text. The secret is an operator called charpath. This operator takes a string and a Boolean and builds the path at the current point that would trace out the text of the string. The path, once created, can be stroked, filled, clipped, or any other combination of things; it is, after all, just a path. The Boolean which charpath requires is for handling special kinds of fonts, and it is generally left true.

As before, the steps to this example are to build the path, clip to it, and draw the image needing to be clipped.

Here we build the path by setting up our current point and string, and then invoking charpath:

```
    gsave                                    % Save old clip path
      /Times-Roman findfont 60 scalefont setfont
      72 72 moveto (Clipping) true charpath    % Set up the text's path
```

Once we have the path, we can invoke clip to establish the complex path of the text as the current clip path. With the clip path established, we can draw our sunburst, which will be confined to the area inside the text:

```
174 72 translate                         % Set our origin to middle
0 2 360 {                                % For every second degree of circle
  newpath
  gsave
    rotate                               % Rotate to angle
    0 0 moveto                           % From new origin
    300 0 rlineto                        % Setup a 300 point long line
    stroke                               % ... and draw it
  grestore
} for
```

Again, because of our judicious use of gsave and grestore, a simple grestore cleans up the graphics state when we're done.

```
grestore
```

As you might imagine, this sort of effect is very powerful and can make it very easy for you to create some stunning images.

Raster Graphics

One of the common questions I have gotten over the years is how to handle other graphics file formats in PostScript. People often ask about including GIFs, JPEGs, and the like in a PostScript file. While this sort of thing is really beyond the scope of this guide, the basics of handling so-called raster graphics is not.

Raster Graphics Basics

Raster graphics is that style of graphics in which the image is broken up into a matrix of picture elements (pixels). The matrix will have a certain number of rows, each containing a certain number of pixels. Each pixel can be assigned any of a number of colors. The number of colors depending upon the "depth" of the image, often expressed as the number of bits needed to encode all the colors. Typical bit depths used today are: 1 (two colors, usually black and white), 2 (four colors, usually shades of gray), 4 (16 colors), 8 (256 colors), 16 (65,536 colors), 24 (so-called true color), and 32 (more colors than you can shake a stick at).

The numbers of rows and columns gives you the resolution. If there are c columns and r rows, the image is referred to as a r x c image. If the image is to have a certain physical size, this size combined with the number of pixels give you the number of dots-per-inch (DPI) of the image, which is a measure of its

resolution. The higher the DPI, the smaller the dots, and the harder it is to see them as individuals.

Raster graphics are convenient in that they can represent photo-realistic images quite easily, but they have limitations. Because the pixels are arranged in a regular pattern, weird moire patterns can appear if they are displayed on a monitor incorrectly, or if they represent an image with a regular pattern that interacts badly with the pattern of the pixels. Likewise, if the resolution is too low and the contrast is too high, certain pixels can stand out and leave the image with the "jaggies."

Raster Graphics in PostScript

Most graphics work in PostScript is done in vector graphics style. This style of graphics is where the image is composed of lines and curves that are described mathematically. It is the style of graphics we have used throughout the guide. The advantage of vector graphics is that you can do all sorts of mathematical operations on the image (rotate, scale, *etc.*) and still get a decent image. An implication of this is that vector graphics are device independent, since they do not care about the display resolution of the display device. Still, PostScript recognizes the need for support of raster graphics, and so it provides a set of operators just to display raster graphics.

The main operator is image, and it is fairly complex. Go grab yourself a cup of coffee, stretch your legs, and prepare to tuck in.

image takes five arguments that describe the image to be displayed and paints that image in a square with one corner at (0, 0) and the other at (1, 1) in the current coordinate frame. All the operands describe the image data and how it should be used to fill up that square. Of course, you probably do not want to draw images in the unit square at (0, 0) all the time, so you must use scale, rotate, and translate to move the unit square to the desired location (and size).

The image operator is used in the following way: *width height depth matrix data* **image** -. The operands *width* and *height* define the size and shape of image matrix in terms of pixels (in the image data, *not* the display results). The operand *depth* describes the number of bits per pixels and, hence, the number of shades of gray. Legal values here are 1 (black and white), 2 (four shades), 4 (16 shades), 8 (256 shades), and 12 (4,096 shades). The operand *matrix* is a PostScript transform matrix that maps from the unit square to the image's pixel coordinates. The image's coordinates go from (0, 0) in the lower left to (width, height) in the upper right. The last operand, *data* is the source of the actual image data. In Level 1 PostScript, this is a procedure, but it can be any number of things in Level 2 and Level 3. Just sticking with Level 1 for now, this procedure is called to fetch all the data for the image, as needed. The procedure returns a string, and the bits within the string are taken and dismantled to create the image. If the procedure does not return enough data to cover the whole image, it is called repeatedly until all the pixels are accounted for. The order at which the pixels are handled is left to right, bottom to top.

Clear? I didn't think so. Let's take a look at a simple example that will show

you the basics. Let's draw a simple smiley face. First, let's take a look at how the smiley face will be laid out in the matrix:

.	.	X	X	X	X	.	.	C3
.	X	.	.	.	.	X	.	BD
X	.	X	.	.	X	.	X	5A
X	.	.	.	.	.	.	X	7E
X	.	X	.	.	X	.	X	5A
X	.	.	X	X	.	.	X	66
.	X	.	.	.	.	X	.	BD
.	.	X	X	X	X	.	.	C3

The left eight columns are the eight columns of the image. The "X" represents a black pixel, and the "." represents a white pixel. Since black is represented by a 0 in PostScript, and white by a 1, we can convert this 8x8 matrix into an eight byte sequence. The ninth column is the hexadecimal encoding of the row… taking the others columns as a binary number with the dots representing 1's and the X's representing 0's.

Now, let's take this image data and try to build up an actual image. First, we need to map the unit square to the location we want to show the image in. Let's make the image a 1 inch square image with the lower left corner at (72, 72).

```
gsave                 % We're mucking about with graphics state... save the original
  72 72 translate     % position the lower left at (72, 72)
  72 72 scale         % make the image 1 inch square
```

Now, we set up the actual image data.

```
  8                   % 8 columns in the image
  8                   % 8 rows in the image
  1                   % 1-bit per pixel: black and white
  [8 0 0 8 0 0]       % map the unit square to (0, 0) - (8, 8)
  {<c3bd665a7e5abdc3>} % the image data as a hex-encoded string
  image               % actually draw the image
grestore
```

Note that the pixel data maps left to right the same way the bits do when you write them in binary. That is, the left-most pixel for a given byte in the data maps to the left-most bit in the byte. Also, note the funny way the string is specified. It is written in hexadecimal using the < and > notation instead of the more usual parenthesis notation. This notation indicates that what is contained between the < and > is a string of 8-bit data bytes encoded in hex. You will see this notation fairly often in working PostScript, since it is a convenient way to store binary data in an ASCII format.

A Gradient

This example is all well and good, but it is just black and white. How do you deal with gray scale images? The procedure is similar, you just specify a different bit depth and lay out your data in a slightly different manner (instead

of a single bit per pixel, you will now need to map multiple bits per pixel. As an example, let's try to make a horizontal gradient fill that goes through sixteen different shades:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sixteen different shades implies 4 bits per pixel, which means that this data can be represented in 8 bytes again (2 pixels per byte). Encoded in hexadecimal again, the data for the above gradient is: 0123456789ABCDEF. Now, we just need to do with this data what we did before, with one exception. The image data no longer represents an 8x8 image, instead it represents a 16x1 image that is 4 bits deep… we need to modify the settings accordingly. Also, instead of scaling this image to a 1 inch square, let's make it 2 inches high by 1 wide and set it next to the smiley face:

```
gsave
  216 72 translate     % lower-left of images at (216, 72)
  72 144 scale         % size of rendered image is 72 points by 72 points
  16                   % 16 pixels wide
  1                    % 1 pixel high
  4                    % 4 bits per pixel
  [16 0 0 1 0 0]       % transform array... maps unit square to pixels
  {<0123456789ABCDEF>} % the image data itself
  image                % let's draw!
grestore
```

Note that, even though the image data is for a one line image, we can scale that single line to fill just about any area. By the way, drawing with the image operator is very much like drawing with any other operator. In particular, clipping can be used to control what parts of the page can be filled by image, so you can use it to do interesting effects like shapes (or text) with gradient fills or with a photographic image as a fill pattern.

The Basics of Color

So, these examples are nice and all, but they are in dull monochrome. What if you want to do color? PostScript does include a handy operator for color images called, creatively enough colorimage. The colorimage operator adds a number of operands to handle the addition of color and to provide for a number of ways of supplying the color data.

The first thing to consider is how you want to specify the colors. Is your image grayscale? RGB? CMYK? This information will indicate the number of color channels you need. Grayscale takes one. RGB takes three (one for red, one for green, and one for blue). Finally, CMYK takes four: cyan, magenta, yellow, and black. Next, you need to think about how these channels are going to be provided to the colorimage operator; will they be interleaved into a single data source string, or will they be broken out into separate data sources. These two decisions will determine the number of additional operands to colorimage: one

specifies the number of channels, one specifies whether the channels are separate or interleaved, and then there is one for each channel data source.

To make this a bit more concrete, let's take a look at a color gradient, which will be a variant of the previous gradient:

Gradient:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Red:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Green:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Blue:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

So, we will now lay this information out in a very similar manner to the original gradient. The example, however, will use three color channels, and they will be provided by separate data sources:

```
gsave
  360 72 translate     % set lower left of image at (360, 72)
  72 144 scale         % size of rendered image is 72 points by 72 points
  16                   % number of columns per row
  1                    % number of rows
  4                    % bits per color channel (1, 2, 4, or 8)
  [16 0 0 1 0 0]       % transform array... maps unit square to pixels
  {<0000000000000000>} % the red image data
  {<FEDCBA9876543210>} % the green image data
  {<0123456789ABCDEF>} % the blue image data
  true                 % pull channels from separate sources
  3                    % 3 color channels (RGB)
  colorimage
grestore
```

Note that the bits-per-pixel operand is really a bits-per-pixel-per-channel operand. This is really no different than for image, it is just that image always works with a single channel. In fact the call:

```
  w h bpp matrix data image
```

is exactly equivalent to:

```
  w h bpp matrix data true 1 colorimage
```

Wrap-up

You have now seen a basic example of three different types of raster graphics in PostScript (you can view a complete example of all three). This is only a start, however.

Encapsulated PostScript

What is Encapsulated PostScript?

At some point, you may want to include some nice PostScript image into a document. There are a number of problems associated with this, but the main one is that your page layout program needs to know how big the image is, and how to move it to the correct place on the page. Encapsulated PostScript (EPS) is that part of Adobe's Document Structuring Convention that provides this information.

What Is the Document Structuring Convention?

The DSC is a special file format for PostScript documents. The full details for the DSC can (and should) be gotten from Adobe. If you are writing a PostScript printer driver or other utility which will be used by a large number of people to create or manipulate PostScript documents, do not even think about writing it without making it DSC-compliant. You will save yourself and your users a lot of headaches.

Although the full DSC is beyond the scope of this guide, the most basic rules can be explained. A DSC-compliant document is an ordinary PostScript document with a number of comments added. These comments provide information to any post-processors which work with the files. Some comments strictly provide information, others are used to structure the document into sections, which may be shuffled or processed in other ways by the post-processor.

Every DSC-compliant document is indicated by having the comment `%!PS-Adobe 3.0` as the first line. This comment is a flag to indicate that the document is compliant. You should never use this comment unless your document really is DSC compliant. There are many other parts to proper DSC. A document which follows the DSC can be manipulated in many ways. In particular, post-processors can shuffle the pages, print two or more pages on a side, and so on. The printer drivers from some notable companies do not follow the DSC, and their PostScript documents are, therefore, impossible to work with once they've been generated.

Now, What About EPS?

An EPS file is a PostScript file which follows the DSC and which follows a couple of other rules. These rules can be summarized as follows:

- The first line must be `%!PS-Adobe EPSF-3.0`
- The file must make use of the BoundingBox comment
- The file should be a single page image (in DSC terms, the `%%Pages` comment must have a value of 0 or 1).
- The file should not use any operators which affect the global state.
- Finally, the EPS file should not use showpage. Actually, Adobe says that it is fine to use showpage in your EPS files. Officially, it is the responsibility

of the importing application to redefine showpage so that the EPS file does not actually eject the page. Still, in creating EPS files, it would be wise not to use this operator.

BoundingBox

The BoundingBox comment is used in DSC to indicate where the actual image will be on a page. The comment describes a rectangle which completely encloses the image. The form of the comment is: **%%BoundingBox:** *llx lly urx ury*. For instance, suppose I have an image which extends from x=72 to x=144 and from y=150 to y=170. The BoundingBox comment in the document should then be: `%%BoundingBox:  72 150 144 170`.

Funky Stuff

There are often times when you will want to take an existing PostScript document and manipulate it in some way. For example, you may be publishing a book, and you want to print the pages with wide margins for proofing notes (but you don't want to modify the book's layout). Maybe you are printing out some 100-page manual, and you want to avoid using most of a rain forest to print it. Maybe you want to print out some document with the word "Draft" stamped *beneath* the pages. All of these things can be done in PostScript by a post-processor (that is, a program which manipulates an existing PostScript file). Moreover, these are all things which may be difficult to manage in the program you used to generate the files.

In this section, I'll show you the basic PostScript code to do each of these jobs and how to use EPS comments to find the right places to insert the additional PostScript.

You are welcome to use these programs as you will. Bear in mind, however, that there are professionally written programs that do these jobs and more. I strongly suggest that you look into buying such a program rather than writing your own. Generally, they have already solved most of the problems. Also, these packages usually come with tools you did not even know were possible. These examples, therefore, are more to give you a taste of what is possible and how to do it, in case you want to roll your own post-processing utility.

By the way, being an American who rarely gets out of the country (or, indeed, off the sofa), I should warn you that both the "Galley Proofs" and "Two Up" examples are set up for the U.S. Letter paper size (8.5 inches by 11 inches), since this is the kind of paper I have and which my printer uses. They can be adapted for other paper sizes, of course, just the particular scale factors and translation coordinates will have to be adjusted appropriately.

Galley Proofs

A galley proof is a printout of a document in which the margins are especially large. The idea is that you can read over what you have printed and have room for writing comments. This system was important in the days of manuscripts and lead-cast type, because the layout of your document was under the control of the publisher's typesetter. You would receive a galley proof from your publisher and make comments about mistakes or changes to be made.

When you have control of the typesetting, galley proofs are not so important, but you may still want to have them. Many systems will not let you make a galley proof, but fortunately it is not hard to do.

The main idea is that you want to scale each page down (to make room for the extra big margins) and then translate the document up and to the right.

Let us say that we want to give ourselves an extra inch of margin on the vertical margins (and scale the horizontals to keep the proportions correct). Here is the PostScript code to do that:

```
gsave
  8.5 6.5 sub 2 div inch               % Center page horizontally...
  11 11 6.5 8.5 div mul sub 2 div inch %  and vertically
  translate
  6.5 8.5 div              % Scale page horizontally...
  dup                      %  and vertically
  scale
  % original page code here...
grestore
```

Here is a slightly faster version. Here, we allow the post-processor to do the math for us. This will print more quickly, since each page does not need to do its own division.

```
gsave
  72 93 translate
  .7647 .7647 scale
  % original page code here...
grestore
```

Why the gsave and grestore? Well, a good rule of thumb is to *always* save the graphics state before you go about changing it (and remember to restore it when you are done). Also, one of the rules of the document structuring convention is that each page should restore the state of the system to what it was when the page was about to start. In other words, the code to layout a page should not alter the permanent state of the system (graphics or otherwise). This assures that pages can be reordered after the PostScript has been generated.

The Hard Part

The hard part of all of this is knowing *where* to insert the new code. Where does one page begin and another end? You could look for calls to showpage,

but many programs define their own versions of this operator (in the code that is generated by `dvips`, for instance, it is called `eop`).

So, how *do* we go about recognizing pages? The document structuring conventions provide us with some handy comments for flagging page information. The most important is the **%%Page:** comment. This comment specifies that the next piece of code is the first one for the new page (in fact, it also tells you which page it is). The end of the document should also be marked with a **%%Trailer:** comment and a **%%EOF** comment. The **%%Trailer:** comment specifies that code to be run at the end of the document is about to be given (so, we are done with the pages). The **%%EOF** comment specifies that we are done with the file. Again, this specifies that we have processed the last page.

So, using these comments, how can we add the needed PostScript? Well, we can start by looking for the first **%%Page:** comment. When we find it, we insert the translate and scale commands right after it. Thereafter, we will proceed each **%%Page:** with a grestore and insert the translate and scale code after the comment. This process continues until we find either a **%%Trailer:** or a **%%EOF** comment. The first of these we find is proceeded by a grestore.

This is all we need to do. To make things a bit more concrete, here is a PERL script to do the job (to make things a bit more interesting, I have added a light line around the original page's image, so you can know how big it is):

```
#!/usr/local/bin/perl
$flag = 0;
while (<>) {
    if (/^%%Page:/) {
        if ($flag) {
            print "grestore\n";
        }
        $flag = 1;
        print $_;
        print "gsave 72 93 translate .7647 .7647 scale\n";
        print "gsave .75 setgray newpath -1 -1 moveto 614 0 rlineto\n";
        print "0 794 rlineto -614 0 rlineto closepath stroke grestore\n";
    } elsif (/^%%Trail/) {
        if ($flag) {
            print "grestore\n";
        }
        print $_;
        $flag = 0;
    } elsif (/^%%EOF/) {
        if ($flag) {
            print "grestore\n";
        }
        print $_;
        $flag = 0;
    } else {
        print;
    }
}
```

Now, this script is not perfect. Many PostScript files do not conform as they

should. This script can, however, serve as a starting point for your own, more robust code.

Two Up

There are occasions when you might want to print more than one page of a PostScript document on a piece of paper. For example, you may have a collection of slides for a presentation, and you may want to print them out in condensed form for a kind of digest hand-out. This kind of printing, where two pages are printed side-by-side on a piece of paper is called "two-up," for the two pages facing up. This idea generalizes readily to any number of pages (though, of course, legibility goes down quickly as the number of pages goes up). In its general form, it is called "n-up."

The PostScript

What is necessary to print in two-up mode? First, we need to translate and rotate each page into the right location of the page, then we need to make sure that the page fits in the new area reserved for it (we will need to scale it down to about half its original size). If we place the two pages side by side, we will get proper two-up form.

The code I will present here will place the odd pages on the left (as you're looking at the page in landscape orientation) and the even pages on the right. You could do it the other way around, if that makes more sense to you.

Here is the code we must wrap around the odd pages:

```
gsave
 504 30 translate  % Position page in middle of region
 90 rotate         % Aim it in the right direction
 .5 .5 scale       % make it small enough
 % original page code here...
grestore
```

And here is the code for the even pages:

```
gsave
 504 426 translate % Position page in middle of region
 90 rotate         % Aim it in the right direction
 .5 .5 scale       % make it small enough
 % original page code here...
grestore
```

Now, you will notice that I used some curious numbers in the translate command. The reason I chose these particular numbers was that I wanted to center each page in its half of the page. I knew I was going to scale by 0.5, so I computed how much white-space was left and added in the appropriate fudge-factor to center the pages.

You may also notice that I wrap a gsave and a grestore around the page and the additional code? The reason for this is that each page *must* leave the state of

the printer unchanged when it has been printed. If you permanently change the state, that state change will be in affect for all subsequent pages. By following this rule, you make the pages independent of order. Some print servers must shuffle page order in order to print the document correctly; since my pages are independent (at least as far as my code is concerned), they will print correctly.

The Hard Part

Now comes the hard part of recognizing where the pages begin. The technique is essentially the same as what we used for galley proofs, so I will spare you the logic here. Essentially, we will look for **%%Page:** comments. We will, however, need to keep track of whether the current page is an odd page or an even page and insert the correct translation code. Also, as before, we must be careful about inserting grestores before subsequent pages and before the **%%Trailer** or **%%EOF** comments.

Here is the PERL script to do the job:

```
#!/usr/local/bin/perl
$flag = 0;                      # We have not yet found a page
$even = 0;                      # First page is an odd page
$page = 1;                      # Start at page #1
$pages = 1;                     # Allow %%Pages comment
while (<>) {
    if (/^%%Pages:/ && $pages) {
        print "%%Pages: (atend)\n";
        $pages = 0;
    } elsif (/^%%Page:/) {      # We have found a page
        if ($flag) {
            print "restore\n";  # restore if it isn't the first
        }
        $flag = 1;
        if ($even) {            # Translate for even pages
            print "save\n";     # gsave
            print "504 426 translate\n";
            $even = 0;
            $page++;
        } else {                # Translate for odd pages
            printf("%%%%Page: %d %d\n", $page, $page);
            print "save /showpage {} def\n";
            print "504 30 translate\n";
            $even = 1;
        }                       # Code to rotate and shrink
        print "90 rotate .5 .5 scale\n";
    } elsif (/^%%Trail/) {      # Cleanup if a %%Trailer is found
        if ($flag) {
            print "restore\n";
        }
        print $_;
        printf("%%%%Pages: %d\n", $page);
        $flag = 0;
    } elsif (/^%%EOF/) {        # Cleanup if an %%EOF is found
        if ($flag) {
```

```
            print "restore\n";
        }
        print $_;
        $flag = 0;
    } else {
        print;
    }
}
```

Note the basic similarity with the script for the galley proofs. There are some additions, however. Because we are taking two pages and printing them on one page, we need to modify the page numbers. The **%%Pages:** comment specifies how many pages are in the document. If you specify "%%Pages: (atend)", you are specifying that you do not know the exact number of pages, but you will give the information later.

An additional complication is the use of save and restore rather than gsave grestore. These operators save the *entire* state of the printer and restore it just as gsave and grestore work with the *graphics* state. In fact, an implicit gsave is done by save; and an implicit grestore is done by restore. The reason these are used is so that I can redefine showpage to a do-nothing procedure (`/showpage {} def`) for the odd pages. This trick prevents the page from being ejected when the odd page does its end of page routines. Unfortunately, this trick only works if the document calls showpage by name. If the document bound showpage up or calls some of the lower level operators, this program would need to be more sophisticated.

Draft

There are times when you will need to stamp a document as a draft. That is, you will want to mark the document so that no one can mistake it for a finished document, but you do not want to make it illegible. Watermarks are perfect for this task.

A watermark is any marking which appears *behind* the text of the page and is generally quite light in appearance. The main text of the page should be legible above it, and the watermark should be visible beneath.

The PostScript

The PostScript for generating a watermark is quite simple. After each **%%Page:** comment (and before the actual PostScript code for the page, you should insert the code to draw the watermark (safely wrapped between a gsave and grestore pair.

As a concrete example, let us say we want to print the word "Draft" down the page beneath the actual text of the page. Such a watermark would be suitable for printing drafts of documents.

Here is the PostScript code to print the watermark:

```
gsave
 .75 setgray
 /Helvetica-Bold findfont 72 scalefont setfont
 80 80 800 {
  306 exch moveto                      % move to the center of the line
  (Draft) dup
  stringwidth pop 2 div neg 0 rmoveto  % Center the text horizontally
  show                                 % Show the text
 } for                                 % and keep doing it
grestore
```

The Hard Part

The hard part of the job is to find the pages. Fortunately, we can use the same technique we used for the galley proofs. Actually, our requirements are simpler. We do not need to wrap the original page code in a gsave, grestore pair, as we did before.

And here is the PERL script to do the job:

```
#!/usr/local/bin/perl
$flag = 0;
while (<>) {
    if (/^%%Page:/) {
        if ($flag) {
            print "grestore\n";
        }
        $flag = 1;
        print $_;
        print "gsave\n";
        print ".75 setgray\n";
        print "/Helvetica-Bold findfont 72 scalefont setfont\n";
        print "80 80 800 { 306 exch moveto\n";
        print "(Draft) dup\n";
        print "stringwidth pop 2 div neg 0 rmoveto show } for\n";
        print "grestore\n";
    } else {
        print;
    }
}
```

Index of Examples

- Scale
- Shade and Width
- Transformations
- Translate
- Drawing a Raster Image

Clipping Text to a Box

```
%!
% operator box: xcoord ycoord box -
% Creates one inch box at xcoord, ycoord
/box {
   newpath
   moveto
   72 0 rlineto
   0 72 rlineto
   -72 0 rlineto
   closepath
} def

/Times-Roman findfont 30 scalefont setfont

gsave                       % Save the old clip path
  72 72 box                 % Set up our box
  gsave                     % Don't allow box to be lost after stroke
    stroke
  grestore                  % Restore the box path
  clip                      % Clip to the box

  60 60 moveto
  (This is Times-Roman clipped to a box) show
  70 90 moveto
  (This is Times-Roman clipped to a box) show
  50 120 moveto
  (This is Times-Roman clipped to a box) show

grestore                    % Get the clip path back

showpage
```

Clipping to Text

```
%!
gsave                                   % Save old clip path
  /Times-Roman findfont 60 scalefont setfont

  72 72 moveto (Clipping) true charpath   % Set up the text's path
  clip                                  % Clip to it

  174 72 translate                      % Set our origin to middle
  0 2 360 {                             % For every second degree of circle
```

```
    newpath

    gsave
      rotate                            % Rotate to angle
      0 0 moveto                        % From new origin
      300 0 rlineto                     % Setup a 300 point long line
      stroke                            % ... and draw it
    grestore
  } for

grestore                                % Restore old clip path

showpage
```

Drawing a Box

```
%!
%% Draws a one square inch box and inch in from the bottom left

/inch {72 mul} def      % Convert inches->points (1/72 inch)

newpath                 % Start a new path
1 inch 1 inch moveto    % an inch in from the lower left
2 inch 1 inch lineto    % bottom side
2 inch 2 inch lineto    % right side
1 inch 2 inch lineto    % top side
closepath               % Automatically add left side to close path
stroke                  % Draw the box on the paper
showpage                % We're done... eject the page
```

Drawing a Box with rlineto

```
%!
%% Draws a one square inch box and inch in from the bottom left
%% This example uses relative coordinates on the lines

/inch {72 mul} def      % Convert inches->points (1/72 inch)

newpath                 % Start a new path
1 inch 1 inch moveto    % an inch in from the lower left
1 inch 0 inch rlineto   % bottom side
0 inch 1 inch rlineto   % right side
-1 inch 0 inch rlineto  % top side
closepath               % Automatically add left side to close path
stroke                  % Draw the box on the paper
showpage                % We're done... eject the page
```

Filled Box Example

```
%!
```

```
%% Draws a one square inch box and inch in from the bottom left

/inch {72 mul} def      % Convert inches->points (1/72 inch)

newpath                 % Start a new path
1 inch 1 inch moveto    % an inch in from the lower left
2 inch 1 inch lineto    % bottom side
2 inch 2 inch lineto    % right side
1 inch 2 inch lineto    % top side
closepath               % Automatically add left side to close path
fill                    % Fill in the box on the paper
showpage                % We're done... eject the page
```

Text Example

```
%!
% Sample of printing text

/Times-Roman findfont   % Get the basic font
20 scalefont            % Scale the font to 20 points
setfont                 % Make it the current font

newpath                 % Start a new path
72 72 moveto            % Lower left corner of text at (72, 72)
(Hello, world!) show    % Typeset "Hello, world!"

showpage
```

Color

```
%!

% Convert inches->points (1/72 inch)
/inch {72 mul} def

% Build a 1 inch square path
/box {
    newpath
    moveto
    1 inch 0 inch rlineto
    0 inch 1 inch rlineto
    -1 inch 0 inch rlineto
    closepath
} def

% Pick a font
/Helvetica findfont 24 scalefont setfont

gsave
    72 72 box               % Build a box
    1 0 0 setrgbcolor       % Set the color to full red
    fill                    % Fill the box
```

```
grestore

gsave
    154 72 box              % Build a box
    1 0 0 setrgbcolor       % Set the color to full red
    stroke                  % Outline the box
grestore

gsave
    1 0 0 setrgbcolor       % Set the color to full red
    236 72 moveto
    (Full Red) show        % Print some text
grestore

gsave
    72 154 box              % Build a box
    1 0.5 0 setrgbcolor     % Set the color to orange
    fill                    % Fill the box
grestore

gsave
    154 154 box             % Build a box
    1 0.5 0 setrgbcolor     % Set the color to orange
    stroke                  % Outline the box
grestore

gsave
    1 0.5 0 setrgbcolor     % Set the color to orange
    236 154 moveto
    (Orange) show           % Print some text
grestore

gsave
    72 236 box              % Build a box
    1 0 1 0.5 setcmykcolor  % Set the color to dark green
    fill                    % Fill the box
grestore

gsave
    154 236 box             % Build a box
    1 0 1 0.5 setcmykcolor  % Set the color to dark green
    stroke                  % Outline the box
grestore

gsave
    1 0 1 0.5 setcmykcolor  % Set the color to dark green
    236 236 moveto
    (Dark Green) show       % Print some text
grestore

showpage
```

Rotation Example

```
%!
% Example of rotation... draws 36 lines in a circular pattern

0 10 360 {              % Go from 0 to 360 degrees in 10 degree steps
  newpath               % Start a new path

  gsave                 % Keep rotations temporary
    144 144 moveto
    rotate              % Rotate by degrees on stack from 'for'
    72 0 rlineto
    stroke
  grestore              % Get back the unrotated state

} for                   % Iterate over angles

showpage
```

Scale

```
%!
% Example of scaled image (text)

/Times-Roman findfont 40 scalefont setfont

gsave
  72 72 moveto
  0.5 1 scale           % Make the text narrow
  (Narrow Text) show    % Draw it
grestore

gsave
  72 144 moveto
  1 2 scale             % Make the text tall
  (Tall Text) show      % Draw it
grestore

gsave
  72 216 moveto
  2 0.5 scale           % Make the text wide and short
  (Squeezed Text) show  % Draw it
grestore

showpage
```

Shade and Width Example

```
%!
% Demonstrate shading and width in drawing lines and filling shapes
```

```
% Define an operator box which builds a path for a one inch square box
% Note that box does not draw or fill the box.

/box {
  newpath
  moveto                                % Current point is on stack
  0 72 rlineto                          % Left
  72 0 rlineto                          % Top
  0 -72 rlineto                         % Right
  closepath                             % Bottom
} def

0 setgray                               % 100% black
1 setlinewidth                          % One point thick lines
72 72 moveto 72 144 lineto stroke       % Draw a one inch line

gsave                                   % Save a copy of the current settings
  0.5 setgray                           % 50% black
  10 setlinewidth                       % 10 point wide lines
  144 72 moveto 144 144 lineto stroke   % Draw a one inch wide line
  216 72 box                            % Build a square path...
  0.35 setgray                          %   make it a little darker...
  fill                                  %   and fill it.
grestore                                % Go back to the original settings

3 setlinewidth                          % Make the box lines wider
300 72 box stroke                       % Draw a black box

showpage
```

If you try this example, you should note a couple of things. Firstly, the black outlined box is a little larger than the gray filled one. This extra width comes from the 3 point wide lines used to draw it—they are *centered* about the path of the box. The ink filling the gray box, however, is completely within the path of the box. Also, when shading objects, you must be careful. PostScript makes shades through a process called halftoning. Basically, uniform dots are placed in various patterns to simulate different shades of grey. Unfortunately, various considerations limit how many shades a printer can produce. So some grey tones may come out the same. This may be the case with the filled box and the outlined box when viewed on your screen or printer.

Transformations Examples

```
%!
% Example to demonstrate translate, rotate, and scale

% operator box: xcoord ycoord box -
% Creates one inch box at xcoord, ycoord
/box {
   newpath
   moveto
   72 0 rlineto
```

```
   0 72 rlineto
   -72 0 rlineto
   closepath
} def

% Specify font for text labels
/Helvetica findfont 40 scalefont setfont

gsave
  40 40 translate               % Set origin to (40, 40)
  0 0 box stroke                % Draw box at new origin...
  77 0 moveto
  (Translated) show             %   and label
grestore

gsave
  100 150 translate             % Translate origin to (100, 150)
  30 rotate                     % Rotate counter-clockwise by 30 degrees
  0 0 box stroke                % Draw box...
  75 0 moveto
  (Translated & Rotated) show   %   and label
grestore

gsave
  40 300 translate              % Translate to  (40, 300)
  0.5 1 scale                   % Reduce x coord by 1/2, y coord left alone
  0 0 box stroke                % Draw box...
  75 0 moveto
  (Translated & Squished) show  %   and label
grestore

gsave
  100 450 translate             % Set origin to (300, 300)
  30 rotate                     % Rotate coordinates by 45 degrees
  0.5 1 scale                   % Scale coordinates
  0 0 box stroke                % Draw box
  75 0 moveto
  (Everything) show
grestore

showpage
```

Translate

```
%!
% Draw a box at 72, 72 using translate

% operator box: xcoord ycoord box -
% Creates one inch box at xcoord, ycoord
/box {
   newpath
   moveto
   72 0 rlineto
```

```
   0 72 rlineto
   -72 0 rlineto
   closepath
} def

gsave                          % Preserve the old coordinates
  72 72 translate              % Set origin to (72, 72)
  0 0 box stroke               % Draw the box at the new origin
grestore                       % Restore the old coordinates

showpage
```

Drawing a Raster Image

```
%!PS-Adobe-3.0

% A simple smiley face

gsave                  % We're mucking about with graphics state... save the original
  72 72 translate      % position the lower left at (72, 72)
  72 72 scale          % make the image 1 inch square
  8                    % 8 columns in the image
  8                    % 8 rows in the image
  1                    % 1-bit per pixel: black and white
  [8 0 0 8 0 0]        % map the unit square to (0, 0) - (8, 8)
  {<c3bd665a7e5abdc3>} % the image data as a hex-encoded string
  image                % actually draw the image
grestore

% A 16-shade horizontal gradient

gsave
  216 72 translate     % lower-left of images at (216, 72)
  72 144 scale         % size of rendered image is 72 points by 72 points
  16                   % 16 pixels wide
  1                    % 1 pixel high
  4                    % 4 bits per pixel
  [16 0 0 1 0 0]       % transform array... maps unit square to pixels
  {<0123456789ABCDEF>} % the image data itself
  image                % let's draw!
grestore

% A 16-tone color gradient from green to blue

gsave
  360 72 translate
  72 144 scale         % size of rendered image is 72 points by 72 points
  16                   % number of samples per line
  1                    % number of lines
  4                    % bits per color channel (1, 2, 4, or 8)
  [16 0 0 1 0 0]       % transform array... maps unit square to pixels
  {<0000000000000000>} % the red image data
  {<FEDCBA9876543210>} % the green image data
```

```
  {<0123456789ABCDEF>} % the blue image data
  true                 % pull channels from separate sources

  colorimage
grestore
```

Index of Operators

- translate

Frequently Asked Questions

I have received a number of questions from readers, and many of them are the same. Since these seem to be popular questions, I thought I would list them here, along with my usual answers. Of course, you will also find the FAQ[7] for the comp.lang.postscript Usenet group to be useful. There are far more FAQs in that list than are here. By the way, if anyone knows of a better answer to any of these question, let me know.

- Is there a utility to convert a PostScript file into my favorite word processors format?
- I have a Hawat-Pickford 520xz ink jet printer without PostScript, is there anyway I can print PostScript files on it?
- Can you point me to a good previewer for my computer?
- Can you tell me X about Acrobat and PDF files?
- How do I print out a PostScript file from my computer?
- My company has an Acme Laz-o-Tron typesetter. We're having problems printing out a set of color separations for a TIFF photograph processed by FotoWerks Pro+. Why are the separations coming out wrong?
- I'm looking at two printers, one has PostScript while the other does not. Which should I buy?
- Is it possible to concatenate two PostScript files together into a single file?
- Is it possible to create a PDF from a PostScript file?

Is there a utility to convert a PostScript file into my favorite word processors format?

Short answer: no.

Long answer: Maybe. There is a utility to convert PostScript files into ASCII files (it tries to extract the text), but it can not work on every PostScript file. The problem here is that PostScript is a full programming language, and there are many ways to accomplish a given thing. It would be next to impossible for a program to look at some piece of PostScript and decide what the contents are. It would be possible to write a program which would accept some subset of PostScript files and convert them to some useful format, but it would be difficult to write (and it could not handle all possible PostScript files).

I have a Hawat-Pickford 520xz ink jet printer without PostScript, is there anyway I can print PostScript files on it?

Yep. There are a number of PostScript interpreters which run on your computer and can print out PostScript files. There are versions of these kinds of utilities for the Mac and for DOS/Windows machines. I have never used one of these utilities, so I can not recommend any particular one. Go to your friendly neighborhood dealer or your favorite catalog to see what they have. There is a

[7] http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.postscript.html

section in the PostScript FAQ on this issue. You may also want to have a look at GhostScript[8]. GhostScript can print PostScript files on certain printers.

Can you point me to a good previewer for my computer?

My first recommendation is GhostScript[9]. Hey, it's free; and it does a good job. It is also able to convert PostScript files to a number of other graphics formats, so it can be handy there. There are also a number of commercial previewers. I have not used any of these commercial packages, so I can not recommend any of them in particular.

Can you tell me X about Acrobat and PDF files?

Nope. I don't know anything about Acrobat or PDF files. Check out the Adobe[10] website. You may find the information you need there.

How do I print out a PostScript file from my computer?

The procedures and tools vary, depending upon the machine. I'm going to assume that you have a PostScript enabled printer and either received or wrote the PostScript file (if you have the application that generated the file, you should just use your application's print command or menu).

DOS/PC Assuming your PostScript printer is on port LPT1: (it really doesn't matter), all you need to do is:
`COPY FILE.PS LPT1:`
where `FILE.PS` is whatever your file is.

Mac OS X Mac OS X comes with a built-in utility to convert PostScript files to PDF. Just opening the PostScript file from the Finder is sufficient to start the conversion. If you want to print the file and you do not have a PostScript printer, you can just print the PDF file you got normally. If you do have a PostScript printer, you will probably have to follow the instructions for Unix below.

UNIX and its cousins Depending upon your system, just printing the file as if it were a text file *should* send it to the printer correctly. Most UNIX systems are clever enough to recognize the PostScript file from the `%!` comment at the beginning of the file. For example, on a BSD system:
`lpr file.ps`
should do the trick.

My company has an Acme Laz-o-Tron typesetter. We're having problems printing out a set of color separations for a TIFF photograph processed by FotoWerks Pro+. Why are the separations coming out wrong?

Beats me.

I'm afraid the basic fact of the matter is that I'm not very bright. Nope. Nope. Nope. I'm not very bright. I also don't know a heck of a lot (you know, there's a joke: a dog, sitting in front of a computer says to his doggy friend, "You know, on the Internet, no one knows you're a dog…" for "a dog" substitute "not an

[8] http://www.cs.wisc.edu/~ghost/index.html
[9] http://www.cs.wisc.edu/~ghost/index.html
[10] http://www.adobe.com

expert," and you've got something). I do know a fair amount about PostScript. Despite my deep and abiding love for fine typography, however, and despite the fact that ink runs in my blood, I don't know much about the printing industry or of the equipment they use.

If you have questions about your software or your printer, I recommend contacting the manufacturer or a posting a question to a newsgroup (if there is a relevant one). I just do not know the particulars on different printers or software (unless it's something I own or use).

I'm looking at two printers, one has PostScript while the other does not. Which should I buy?

This is a common and very good question (there is a related one on whether or not to buy a PostScript extension for an existing printer). The answer, as it usually does, boils down to a definite, "It depends."

If your main printing task consists of printing the monthly report or letters to clients, friends, family members, or whomever, then you will probably find PostScript to be an extravagance. This is especially true if you have no interest in mucking about with fonts or graphics. In other words, for light-duty, mainly text, print once and send to whomever kind of work, the answer is, "No."

If you are a desktop publisher, you are writing a book and want to send the book to your publisher electronically, or you do a lot of graphics work and want the graphics to look good regardless of the printer, the answer is yes. In all of these cases, you have a complex printing task and may want to proof your document at home or in the office but then send it out for final (higher quality) printing. Generally speaking, the same PostScript file will look the same regardless of which PostScript printer you use, with one exception: if one printer is capable of better print quality (finer lines, smoother curves, gentler shades of grey) than the other, your document should benefit from these increased capabilities without the PostScript file's needing to be changed. You are benefiting from PostScript's device independence.

As for graphics intensive work, I find the EPS format to be the best for line drawing type graphics (*i.e.* no bitmap images) that I will want to include in a document. I very often want to print an image generated by one package when the word processor may be from a different vendor. Many times (incredible to tell) I sometimes need to include a graphic made on one computer system in a document on a completely different system! In such a heterogeneous environment, EPS graphics are just about the only reasonable option. Also, many top-quality drafting/painting programs generate their best output in EPS (on some windowing systems, the built-in graphic format can have a limited resolution that results in badly displaced elements in a printed image). If you use EPS graphics, you *must* have a PostScript enabled printer if you want to print them out with any quality at all.

As with most things in our complex universe, it all depends upon what you are going to do. You must sit down and evaluate your needs and probable work habits. If you think you will benefit from PostScript's unique characteristics enough to justify the cost, then go for it.

Is it possible to concatenate two PostScript files together into a single

file?

Yes and no. If you do not care about being DSC compliant, then all you have to do is to slap the two files together into one. If that does not work, you could wrap each in a save/restore pair:

```
%!
save
  % contents of the first file
restore
save
  % contents of the second file
restore
```

The problem with this approach is that it is not compliant with the DSC, so you can not do anything with the concatenated file. Previewers and print spoolers will have problems recognizing pages and would not be able to shuffle them appropriately. Other post-processing engines would fail to work with them too (for instance, you would not be able to display the file in two up format.

If the two PostScript files are DSC compliant, came from the same tool, and use the same resources (the acid test is if they have the same preamble… the definitions at the start of the file before you get to the first page), then you could concatenate the files by starting with the common preamble, followed by the pages of the first file, followed by the pages of the second file. You would have to recognize the pages by looking at the page comments much like we did in the post-processing examples.

You can not play this trick with files from different tools (or with different preambles). The problem is that a page in a file depends upon the definitions in the preamble, if you miss a definition, or if two files have different definitions for the same name, you really can not concatenate the files and keep the result compliant with the DSC.

There are various scripts available out there to do the concatenation under these restrictions. Look around for tools like "psmerge," "psconcat," and the like, and see what might work for you (psmerge is part of the psutils package and is available on many Unix work-alikes, and comes bundled with Mac OS X).

Is it possible to create a PDF from a PostScript file?

Yes. GhostScript[11] comes with a utility called ps2pdf which will do the job. Also, Mac OS X has the ability built-in: just double-click on the PostScript file in the Finder, and it will convert the file to PDF for you and give you an option to save the results. I'm sure there are other tools to do the job, too.

Is it true that you are one of the most stunningly attractive men on the net?

Why, yes. How did you know?

Okay, so maybe no one has ever asked me that. It was worth a shot. Maybe you wouldn't have noticed.

[11] http://www.cs.wisc.edu/~ghost/index.html

By the way, you finished reading the FAQ, so here's a mailto link for you to send me e-mail[12] if you don't see your question here.

PostScript Operators

Operator: add

num1 num2 **add** *num3*

This operator returns the addition of the two arguments.

- stackunderflow
- typecheck
- undefinedresult

See also:

- div
- mul
- sub

Operator: arc

x-coord y-coord r ang1 ang2 **arc -**

This operator adds an arc to the current path. The arc is generated by sweeping a line segment of length *r*, and tied at the point (*x-coord y-coord*), in a counter-clockwise direction from an angle *ang1* to an angle *ang2*. Note: a straight line segment will connect the current point to the first point of the arc, if they are not the same.

- limitcheck
- stackunderflow
- typecheck

Operator: begin

dict **begin -**

This operator pushes the dictionary *dict* onto the dictionary stack. Where it can be used for def and name lookup. This operator allows an operator to set up a dictionary for its own use (e.g. for local variables).

Errors:

- dictstackoverflow
- invalidaccess
- stackunderflow
- typecheck

[12] mailto:pjw@tailrecursive.org?subject=First%20Guide%20to%20PostScript

Operator: bind

procedure1 **bind** *procedure2*

The bind operator goes through *procedure1* and replaces any operator names with their associate operators. Names which do not refer to operators are left alone. Operators within *procedure1* which have unrestricted access will have bind called on themselves before they are inserted into the procedure. The new procedure with operators instead of operator names is returned on the stack as *procedure2*.

The main effect and use of this operator is to reduce the amount of name lookup done by the interpreter. This speeds up execution and ties down the behavior of operators.

Errors:

- typecheck

Operator: clip

- clip -

This operator intersects the current clipping path with the current path and sets the current clipping path to the results. Any part of a path drawn after calling this operator which extends outside this new clipping area will simply not be drawn. If the given path is open, clip will treat it as if it were closed. Also, clip does not destroy the current path when it is finished… it may be used for other activities.

It is important to note that there is no easy way to restore the clip path to a larger size once it has been set. The best way to set the clip path is to wrap it in a gsave and grestore pair.

Errors:

- limitcheck

Operator: closepath

- closepath -

This operator adds a line segment to the current path from the current point to the first point in the path. This closes the path so that it may be filled.

Errors:

- limitcheck

Also see the following operators:

- **newpath**
- **moveto**
- **lineto**

Operator: charpath

string bool **charpath -**

This operator takes the given string and appends the path which the characters define to the current path. The result is can be used as any other path for stroking, filling, or clipping.

The boolean argument informs charpath what to do if the font was not designed to be stoked. If the boolean is true, the path will be modified to be filled and clipped (but not stroked). If the boolean is false, the path will be suitable to be stroked (but not filled or clipped).

- limitcheck
- nocurrentpoint
- stackunderflow
- typecheck

See also:

- clip
- fill
- show
- stroke

Operator: colorimage

width height bppc data_1 … data_n separate channels **colorimage -**

This operator draws a color image in the unit square from (0,0) to (1, 1). The source information is a raster image *width* pixels wide by *height* pixels high. The image is composed of *channels* color channels (1, 3, or 4), and each pixel is represented by *bppc* bits in each channel. If *separate* is false, there will be only one *data* operand. Otherwise, there will be one for each channel. The *data* operand can be a number of things, but is usually a procedure that returns a string of bytes with the channel data each time it is called. The procedure will be called repeatedly until all pixels have been processed. The image is processed from left-to-right, top-to-bottom.

- invalidaccess
- ioerror
- limitcheck
- rangecheck
- stackunderflow
- typecheck
- undefined
- undefinedresult

See also:

- image

Operator: curveto

x1 y1 x2 y2 x3 y3 **curveto -**

This operator draws a curve from the current point to the point (*x3*, *y3*) using points (*x1*, *y1*) and (*x2*, *y2*) as control points. The curve is a Bézier cubic curve. In such a curve, the tangent of the curve at the current point will be a line segment running from the current point to (*x1*, *y1*) and the tangent at (*x3*, *y3*) is the line running from (*x3*, *y3*) to (*x2*, *y2*).

- limitcheck
- nocurrentpoint
- stackunderflow
- typecheck

See also:

- arc
- lineto
- moveto

Operator: def

name value **def -**

This operator associates the *name* with *value* in the dictionary at the top of the dictionary stack. This operator essentially defines names to have values in the dictionary and is used to define variables and operators.

Errors:

- dictfull
- invalidaccess
- limitcheck
- stackunderflow
- typecheck
- VMerror

Operator: div

num1 num2 **div** *num3*

This operator returns the result of dividing *num1* by *num2*. The result is always a real.

- stackunderflow
- typecheck
- undefinedresult

See also:

- add
- mul

- sub

Operator: dup

object **dup** *object object*

This operate pushes a second copy of the topmost object on the operand stack. If the object is a reference to an array, string, or similar composite object, only the reference is duplicated; both references will still refer to the same object.

See also:

- exch
- index
- pop

Errors:

- stackoverflow
- stackunderflow

Operator: end

- end -

This operator pops the topmost dictionary off of the dictionary stack. The dictionary below it becomes the new current dictionary.

Errors:

- dictstackunderflow

Operator: exch

value1 value2 **exch** *value2 value1*

This operator simply exchanges the top two items on the operand stack. It does not matter what the operands are.

See also:

- dup
- index
- pop

Errors:

- stackunderflow

Operator: fill

- fill -

This operator closes and fills the current path with the current color. Any ink within the path is obliterated. Note that fill blanks out the current path as if it had called newpath. If you want the current path preserved, you should use gsave and grestore to preserve the path.

Errors:

- limitcheck

Operator: findfont

name **findfont** *font*

This operator looks for the named font in the font dictionary. If it finds the font, it pushes the font on the stack for later processing. It signals an error if the font can not be found.

Errors:

- invalidfont
- stackunderflow
- typecheck

Also see the following operators:

- **scalefont**
- **setfont**

Operator: for

initial increment limit proc **for -**

This operator will execute *proc* repeatedly. The first time *proc* is executed, it will be given *initial* as the top operand. Each time it is executed after that, the top operand will be incremented by *increment*. This process will continue until the argument would have exceeded *limit*.

- stackoverflow
- stackunderflow
- typecheck

See also:

- if
- ifelse

Operator: grestore

- grestore -

Sets the current graphics state to the topmost graphics state on graphics state stack and pops that state off the stack. This operator is almost always used in conjunction with gsave.

Operator: gsave

- gsave -

This operator pushes a copy of the current graphics state onto the graphics state stack. The graphics state consists of (among other things):

- Current Transformation Matrix
- Current Path
- Clip Path
- Current Color
- Current Font
- Current Gray Value

gsave is typically used with grestore whenever you need to change the graphics state temporarily and return to the original.

Errors:

- limitcheck

Operator: if

bool proc **if -**

This operator will execute *proc* if *bool* is true.

- stackunderflow
- typecheck

Operator: ifelse

bool proc1 proc2 **ifelse -**

This operator will execute *proc1* if *bool* is true and *proc2* otherwise.

- stackunderflow
- typecheck

Operator: image

width height bpp data **image -**

This operator draws a grayscale image in the unit square from (0,0) to (1, 1). The source information is a raster image *width* pixels wide by *height* pixels high. Each pixel is represented by *bpp* bits. The *data* operand can be a number of things, but is usually a procedure that returns a string of bytes with the grayscale data each time it is called. The procedure will be called repeatedly until all pixels have been processed. The image is processed from left-to-right, top-to-bottom.

- invalidaccess
- ioerror
- limitcheck
- rangecheck
- stackunderflow
- typecheck
- undefined
- undefinedresult

See also:

- colorimage

Operator: index

value_n … value_0 n **index** *value_n … value_0 value_n*

This operator grabs the nth item off the operand stack (item 0 is the one just under the index you push on the stack for the operator) and pushes it on top of the stack.

See also:

- dup
- exch
- pop

Errors:

- rangecheck
- stackunderflow
- typecheck

Operator: lineto

x-coord y-coord **lineto** -

This operator adds a line into the path. The line is from the current point to the point (*x-coord y-coord*). After the line is added to the path, the current point is set to (*x-coord y-coord*). It is an error to call **lineto** without having a current point.

Errors:

- limitcheck

- nocurrentpoint
- stackunderflow
- typecheck

Also see the following operators:

- **rlineto**
- **moveto**
- **rmoveto**
- **curveto**
- **arc**
- **closepath**

Operator: moveto

x-coord y-coord **moveto** -

This operator moves the current point of the current path to the given point in user space. If a **moveto** operator immediately follows another **moveto** operator, the previous one is erased.

Errors:

- limitcheck
- stackunderflow
- typecheck

Also see the following operators:

- **rmoveto**
- **lineto**
- **curveto**
- **arc**
- **closepath**

Operator: mul

value1 value2 **mul** *product*

This operator multiplies the first two operands on the stack and pushes the result back onto the stack. The result is an integer if both operands are integers and the product is not out of range. If the product is too big, or one of the operands is a real, the result will be a real.

Errors:

- stackunderflow
- typecheck
- undefinedresult

Operator: newpath

- newpath -

The newpath operator clears out the current path and prepares the system to start a new current path. This operator should be called before starting *any* new path, even though some operators call it implicitly.

Operator: pop

value **pop -**

This operator just removes the top-most item off of the operand stack.

See also:

- dup
- exch
- index

Errors:

- stackunderflow

Operator: restore

state **restore** -

This restores the total state of the PostScript system to the state saved in *state*.

Errors:

- invalidrestore
- stackunderflow
- typecheck

See also:

- save

Operator: rlineto

dx dy **rlineto** -

This operator adds a line into the path. The line is from the current point to a point found by adding *dx* to the current *x* and *dy* to the current *y*. After line is added to the path, the current point is set to the new point. It is an error to call **lineto** without having a current point.

Errors:

- limitcheck
- nocurrentpoint
- stackunderflow

- typecheck

Also see the following operators:

- **lineto**
- **moveto**
- **rmoveto**
- **curveto**
- **arc**
- **closepath**

Operator: rmoveto

dx dy **rmoveto** -

This operator moves the current point of the current path by adding *dx* to the current *x* and *dy* to the current *y*.

Errors:

- limitcheck
- stackunderflow
- typecheck

Also see the following operators:

- **moveto**
- **lineto**
- **curveto**
- **arc**
- **closepath**

Operator: rotate

angle **rotate -**

This operator has the effect of rotating the user space counter-clockwise by *angle* degrees (negative angles rotate clockwise). The rotation occurs around the current origin.

- rangecheck
- stackunderflow
- typecheck

See also:

- scale
- translate

Operator: save

- **save** *state*

This operator gathers up the complete state of the PostScript system and saves it in *state*. Errors:

- limitcheck
- stackoverflow

See Also:

- restore

Operator: scale

sx sy **scale -**

This operator has the effect of scaling the user coordinates. All coordinates will be multiplied by *sx* in the horizontal direction, and *sy* in the vertical.

The origin will not be affected by this operation.

- rangecheck
- stackunderflow
- typecheck

See also:

- rotate
- translate

Operator: scalefont

font size **scalefont** *font*

This operator takes the given font and scales it by the given scale factor. The resulting scaled font is pushed onto the stack. A *size* of one produces the same sized characters as the original font, 0.5 produces half-size characters, and so on.

Errors:

- invalidfont
- stackunderflow
- typecheck
- undefined

Also see the following operators:

- **findfont**
- **setfont**

Operator: setfont

font **setfont** -

This operator sets the current font to be *font*. This font can be the result of any font creation or modification operator. This font is used in all subsequent character operations like show.

- invalidfont
- stackunderflow
- typecheck

Also see:

- findfont
- scalefont

Operator: setcymkcolor

cyan magenta yellow black **setcymkcolor -**

Sets the color of the ink to the color in the CYMK color space specified by *cyan*, *magenta*, *yellow*, and *black*. The components must be between 0 (none) to 1 (full).

- stackunderflow
- typecheck
- undefined

See also:

- setgray
- setrgbcolor

Operator: setgray

gray-value **setgray** - This operator sets the current intensity of the ink to *gray-value*. *gray-value* must be a number from 0 (black) to 1 (white). This will affect all markings stroked or filled onto the page. This applies even to path components created before the call to **setgray** as long as they have not yet been stroked.

- stackunderflow
- typecheck
- undefined

See also:

- setcymkcolor
- setrgbcolor

Operator: setlinewidth

width **setlinewidth** - This operator sets the width of all lines to be stroked to *width*, which must be specified in points. A line width of zero is possible and is interpreted to be a hairline, as thin as can be rendered on the given device.

- stackunderflow
- typecheck

Operator: setrgbcolor

red green blue **setrgbcolor -**

Sets the color of the ink to the color in the RGB color space specified by *red*, *green*, and *blue*. The components must be between 0 (none) to 1 (full).

- stackunderflow
- typecheck
- undefined

See also:

- setcymkcolor
- setgray

Operator: show

string **show -**

This operator draws the given string onto the page. The current graphics state applies, so the current font, fontsize, gray value, and current transformation matrix all apply.

The location for the text is set by the current point. The current point will specify the leftmost point of the baseline for the text.

- invalidaccess
- invalidfont
- nocurrentpoint
- rangecheck
- stackunderflow
- typecheck

See also:

- charpath
- moveto
- setfont

Operator: showpage

- showpage -

This operator commits the current page to print and ejects the page from printing device. **showpage** also prepares a new blank page.

Operator: stroke

- stroke -

This operator draws a line along the current path using the current settings. This includes the current line thickness, current pen color, current dash pattern, current settings for how lines should be joined, and what kind of caps they should have. These settings are the settings at the time the stroke operator is invoked.

A closed path consisting of two or more points at the same location is a degenerate path. A degenerate path will be drawn only if you have set the line caps to round caps. If your line caps are not round caps, or if the path is not closed, the path will not be drawn. If the path is drawn, it will appear as a filled circle center at the point.

Errors:

- limitcheck

Operator: sub

num1 num2 **sub** *num3*

This operator returns the result of subtracting *num2* from *num1*.

- stackunderflow
- typecheck
- undefinedresult

See also:

- add
- div
- mul

Operator: translate

x-coord y-coord **translate -**

This operator has the affect of moving the origin to the point (*x-coord*, *y-coord*) in the current user space.

- rangecheck
- stackunderflow
- typecheck

See also:

- rotate
- scale

Errors You Might Encounter

configurationerror setpagedevice request can not be satisfied
dictfull dictionary is full
dictstackoverflow too many begins
dictstackundeflow too many ends
invalidaccess access attribute violated (e.g. attempted to write a read-only object)
invalidfont bad font name or dictionary
invalidrestore the saved state object is too old to restore
ioerror some kind of error during input or output
limitcheck some implementation-dependent size restriction has been exceeded
nocurrentpoint the current point is not defined, yet
rangecheck operand is too big or too small
stackoverflow the stack was full before the last push
stackunderflow you tried to pop from an empty stack
syntaxerror PostScript's syntax has been violated
typecheck operand is of the wrong type
undefined name is not defined in any dictionary on the stack
undefinedresult the result of the last numeric operation is invalid (*e.g.* division by zero)
VMerror virtual memory full

String Escape Codes

`\n` Newline
`\r` Carriage return
`\t` Horizontal TAB
`\b` Backspace
`\f` Form feed
`\\` Backslash
`\(` Left parenthesis
`\)` Right parenthesis
`\ddd` The character code ddd, where ddd is in octal.

In addition to these basic codes, a backslash just before a newline allows you to break a string across two lines without inserting a newline into the string. That is, the string:

```
(This is a \
string \
that has no \
newlines)
```

is equivalent to the string:`(This is a string that has no newlines)`.

Forbidden Operators in EPS

Here is the list of operators which are not allowed within EPS files. The fact that none of these operators are listed in my index of operators should tell you something: they are not operators you would use very often, anyway. There is also a list of operators you *can* use but you should probably avoid.

Forbidden Operators banddevice, clear, cleardictstack, copypage, erasepage, exitserver, framedevice, grestoreall, initclip, initgraphics, initmatrix, quit, renderbands, setglobal, setpagedevice, setshared, startjob

To Avoid nulldevice, setgstate, sethalftone, setmatrix, setscreen, settransfer, undefinedfont

For more details, please see the red book.

PostScript Books

There are a number of good books on PostScript. If you do much PostScript programming at all, I highly recommend that you get one of these print books.

PostScript Language Reference Manual: Third Edition This book is put out by Adobe Systems Incorporated and is published by Addison Wesley. It is *the* reference manual and pretty much defines the language. The operator reference guide I have here pretty well follows Adobe's reference in this manual. Their reference is, however, far more detailed (and accurate). This book is known as the "Red and White Book." Adobe also provides other books (including the "Blue" tutorial and cookbook). The *PLRM*'s ISBN is: 0-201-37922-8

PostScript Language Tutorial and Cookbook This book is also put out by Adobe and is published by Addison Wesley. (Are you starting to see a pattern here?) This is a very neat book on PostScript and gives a number of handy examples (two which come to mind are a set of routines to set text in a circle and a simple text formatter). This is the so-called blue book. The ISBN is 0-201-10179-3.

Learning PostScript: a Visual Approach This book is a fine beginning text for PostScript. The book emphasizes PostScript's fantastic graphics abilities while illustrating basic language constructs. The language concepts are illustrated with graphic design examples, most of which you will be tempted to use in your own documents. The book also includes a number of interesting and useful utilities. Author: Ross Smith. Publisher: Peachpit Press. ISBN 0-938-151-12-6. I do not know if this book is still in print, but there are copies available on Amazon at the time I write this. You can probably find it elsewhere, too.

Contacting Me

If you wish to contact me with a question, comment, or uncontrollable adulation, please feel free to e-mail me. My address is `pjw@tailrecursive.org`. I

recommend that you check the FAQ first, however, since I have listed answers for many of the common questions I've received. I must also warn you not to expect a quick answer. I do not check my e-mail with notable frequency, so it may be a while before I see your message.

About the Author

I was a graduate student at Indiana University in lovely Bloomington, Indiana. While there, I learned to love the closure and learned to hate the PDP-8 (well, not really).

Currently, I am wasting my talents as a software developer for a certain large company in California you've all heard about and generally discovering to my horror that Dilbert is 100% accurate. Still, as a compensation, I get to live on the Pacific Rim.