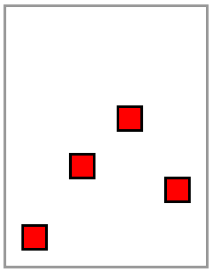
CHAPTER 3

Variables and procedures



At this point, I have explained how to draw only the simplest figures. In particular, I have given no hint of how to use the real programming capability of PostScript.

Before beginning to look at more complex features of the language, place this principle firmly in your mind:

- *To get good results from PostScript, first get a simple picture up on the screen that comes somewhere close to what you want, and then refine it and add to it until it is exactly what you want.*

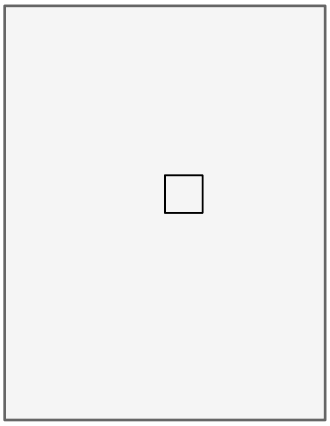
It is the secret to efficient PostScript programming, because *once you have a picture—any picture—you can often visualize your errors*. Another suggestion is that since debugging large chunks of PostScript all at once is extremely painful, you want to keep small the scope of your errors. Yet another thing to keep in mind as you develop programs is flexibility. Ask yourself frequently if you might reuse in another drawing what you are doing in this one. We shall see how to take advantage of reusable code in an efficient way.

The basic technique of this chapter will be to see how one PostScript program evolves according to this process. Technically, the main ingredients we are going to add to our tool kit are **variables** and **procedures**.

3.1. Variables in PostScript

The following program draws a square one inch on a side roughly in the middle of a page.

```
%!

72 72 scale
4.25 5.5 translate
1 72 div setlinewidth

newpath
0 0 moveto
1 0 rlineto
0 1 rlineto
-1 0 rlineto
closepath
stroke

showpage
```



It is extremely simple, and frankly not very interesting.

Among other things, it is not very flexible. Suppose you wanted to change the size of the square? You would have to replace each occurrence of “1” with the new size. This is awkward—you might miss an occurrence, at least if your program were more complicated. It would be better to introduce a **variable** s to control the length of the side of the square.

Variables in PostScript can be just about any sequence of letters and symbols. They are **defined** and **assigned values** in statements like this

```
/s 1 def
```

which sets the variable s to be 1. The `/s` here is the **name** of the variable s. We can't write `s 1 def` because then the **value** of s would go on the stack and its name lost track of, whereas what we want to do is associate the new value 1 with the letter `s`.

- *After a variable is defined in your program, any occurrence of that variable will be replaced by the last value assigned to it.*

We shall see later that this is not quite true in certain local environments.

If you attempt to use the name of a variable that has not been defined, you will get an error message about `/undefined in ...`

Using a variable for the side of the square, the new program would look like this (I include only the interesting parts from now on):

```
/s 1 def

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
```

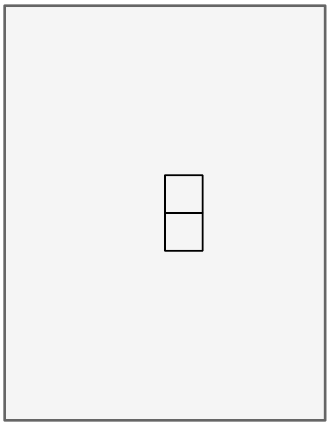
I recall that the command `neg` replaces anything on the top of the stack by its negative. This code is indeed a bit more flexible than the original, because if you want to draw a square of different size you would have to change only the first line.

Technical remark. Definition and assignment in PostScript look the same, and differ only in technical ways. In order to understand how this works, it is helpful to know how PostScript keeps track of the values of variables. It stores them in a **dictionary**, which is a collection of of names and the values assigned to them. There may be several dictionaries currently in use at any given point in a program; they are kept in the **dictionary stack**. When a variable is defined, its name and value are registered in the top dictionary, replacing any value it has had before. When the variable is encountered in a program, all the dictionaries in use are searched until its value is found, starting at the top of the dictionary stack.

3.2. Procedures in PostScript

Suppose you wanted to draw two squares, one of them at $(0, 0)$ and the other at $(0, -1)$ (that is to say, just below the first). Most straightforward:

```
%!

72 72 scale
4.25 5.5 translate
1 72 div setlinewidth

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke

0 -1 translate

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke

showpage
```



The program just repeats the part of the program which actually draws the square, of course. Recall that `translate` shifts the origin of the user's coordinate system in the current units.

Repeating the code to draw the two squares is somewhat inefficient—this technique will lead to a lot of text pasting and turns out to be very prone to error. It is both more efficient and safer to use a PostScript **procedure** to repeat the code for you. A procedure in PostScript is an extremely simple thing.

- *A procedure in PostScript is just any sequence of commands, enclosed in brackets* $\{\ldots\}$.

You can assign procedures to variables just as you can assign any other kind of data. When you insert this variable in your program, it is replaced by the sequence of commands inside the brackets. In other words, using a procedure in PostScript to draw squares proceeds in two steps:

(1) Define a procedure, called say `draw-square`, in the following way:

```
/draw-square {
  newpath
  0 0 moveto
  s 0 rlineto
  0 s rlineto
  s neg 0 rlineto
  closepath
  stroke
} def
```

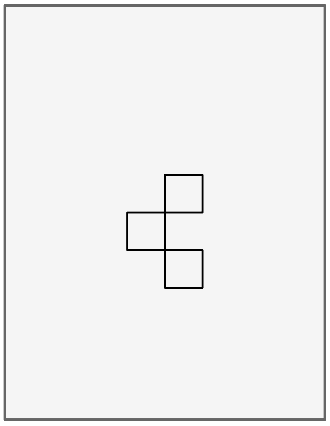
At any point in a program after this definition, whenever the expression `draw-square` occurs, PostScript will simply substitute the lines in between the curly brackets { and }. The effect of calling a procedure in PostScript is always this sort of straightforward substitution.

(2) Call the procedure when it is needed. In this case, the new commands on the page will include the above definition, and also this:

```
draw-square
0 -1 translate
draw-square
```

Of course if we have done things correctly, the page looks the same as before. But we can now change it easily by mixing several translations and calls to `draw-square` like this:

```
draw-square
-1 -1 translate
draw-square
1 -1 translate
draw-square
```



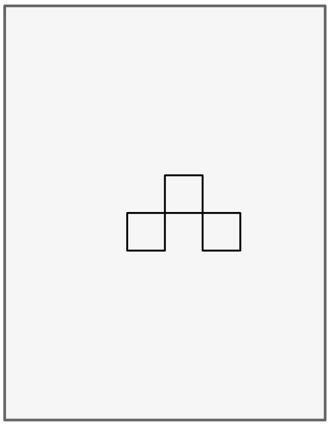
3.3. Keeping track of where you are

In the lines of PostScript above, you can easily forget exactly where you are with all those translations. What you might do is translate back again after each translation and drawing operation to restore the original coordinates. But this would become complicated later on in your work, when you will perform several changes of coordinates and it will be difficult to figure out how to invert them. Instead, you can get PostScript to do the work of remembering where you are. It has a pair of commands that help you do the job easily: `gsave` saves the current coordinate system somewhere (together with a few other things like the current line width) and `grestore` brings back the coordinate system you saved with your last `gsave`.

- *The commands* `gsave` *and* `grestore` *must be used in pairs!*

In this scheme we could write

```
draw-square

gsave
-1 -1 translate
draw-square
grestore

1 -1 translate
draw-square
```



and get something quite different.

To be a bit more precise, `gsave` saves the **current graphics state** and `grestore` brings it back. The graphics state holds data about coordinates, line widths, the way lines are joined together, the current color, and more—in effect everything that you can change easily to affect how things are drawn. You might recall that we saw `gsave` and `grestore` earlier, where we used them to set up successive pages correctly, enclosing each page in a pair of `gsave` and `grestore`.

Incidentally, it is usually—but not always—a bad (very bad) idea to change anything in the graphics state in the middle of drawing a path. Effects of this bad practice are often unintuitive, and therefore unexpected. There are definite exceptions to this rule, but one must be careful. The problem is to know what parts of the graphic state take effect in various commands. The principal exceptions use `translate` and `rotate` to build paths conveniently. For example, the following sequence builds a square.

```
1 0 moveto 90 rotate
1 0 lineto 90 rotate
1 0 lineto 90 rotate
1 0 lineto 90 rotate
1 0 lineto
```

The commands `rotate` etc. change the coordinate system in a figure.

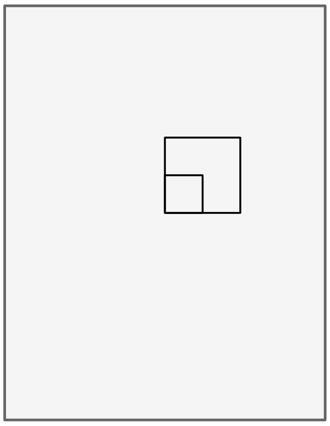
- *The drawing commands* `lineto` *etc. use the coordinate system current when they are applied to build a path in physical coordinates.*

3.4. Passing arguments to procedures

The definition of the procedure `draw-square` has a variable s in it. The variable s is not defined in the procedure itself, but must be defined before the procedure is used. This is awkward—if you want to draw squares of different sizes, you have to redefine s each time you want to use a new size.

For example, if we want two squares of different sizes, we write the code on the left below:

```
/s 1 def
draw-square
/s 2 def
draw-square
```



Let me repeat here: *If you want to assign a new value to a variable you have to define it over again, using the* **name** *of the variable, which begins with* `/`.

Now for a new idea. It is awkward to have to assign a value to s every time we want to draw a square. It would be much better if we could just type something like `2 draw-square` to do the job. This in fact possible, by doing a bit of stack manipulation inside the procedure itself. Let's see—we want to type `2 draw-square` and draw a square of side 2. This means that the procedure should have access to the item that's put on the stack just before it's called, and assign its value to a variable. This requires a trick. Normally we assign a value to a variable by putting the name of the variable on the stack, then the new value, then calling `def`, like this: `/s 2 def`. In order to assign a value to the variable s inside the procedure, we must somehow get the name `/s` on the stack *below* the value on the stack when the procedure is called. The way to do this is to put `/s` on the stack *after* the new value and then call `exch`. The command `exch` exchanges the top two items on the stack. Therefore `2 /s exch` makes the stack `/s 2`, and then `2 /s exch def` has exactly the same effect as `/s 2 def`. Thus the lines

```
/draw-square {
  /s exch def
  newpath
```

```
    0 0 moveto
    s 0 rlineto
    0 s rlineto
    s neg 0 rlineto
    closepath
    stroke
  } def

  2 draw-square
```

do exactly what we want—the side of the square is picked up off the stack, assigned to the variable s, and then used for drawing. The important point is that the procedure itself now handles the assignment of a value to s, and all we do is pass the value of s to the procedure as an **argument** to it by putting it on the stack before the procedure is called. If you are familiar with how programming language compilers work, you will recognize this as what programming languages do to pass arguments, but behind the scenes. The difference is that PostScript does it in the open, and effectively forces you to do a bit more work yourself.

If you wanted to draw rectangles with different width and height, you would pass two arguments in a similar way:

```
  /draw-rectangle {
    /h exch def
    /w exch def
    newpath
    0 0 moveto
    w 0 rlineto
    0 h rlineto
    w neg 0 rlineto
    closepath
    stroke
  } def

  2 3 draw-rectangle
```

draws a rectangle of width 2 and height 3. Notice that the stuff on the stack is removed in the order *opposite* to that in which you placed it there.

- *In PostScript the* **arguments** *of a procedure are data that go on the stack before the procedure is called.*

By the way, it seems to me that one of PostScript's principal mistakes in design is the order of the arguments to `def`. There would be several advantages to having the command work like `2 /s def` rather than the way it does. For one thing, reading the arguments of procedures would be more sensible. For another, programmers would be encouraged to make programs more readable. Very often you want to make a very long calculation and then assign the result to a variable. The most common way to do this is to make the calculation and then pick the result off the stack just as above, with `... exch def`. So here, too, programs would benefit from the change. And they would be more readable, because the name of the variable would be close to where it is used. The program would be more *local* in a sense. Locality is one important feature of happy programming.

3.5. Procedures as functions

A procedure can be a function. That is to say, it can accept some kind of input, calculate something depending on the input, and pass back or **return** the results of the calculation. There are of course several built in functions in PostScript, for example the mathematical functions `neg`, `add` etc. The way these work is that you put **arguments** on the stack, call the function, and then next get the **return value** (or values) on the stack. For example, the sequence `30 sin` in your program, when the program is executed, puts 30 on the stack (the argument), calls the `sin` procedure, and leaves 0.5 on the stack (the return value). Some others, like `atan` have two arguments. I repeat:

- *The* **return value** *of a procedure is what it leaves on the stack.*

It can in fact leave lots of stuff on the stack, and can have several return values.

There is only a formal difference between functions and procedures.

Example. We will make up a procedure `hypotenuse` that has two arguments and returns the square root of the sum of their squares. In fact, we shall see two versions of this. The first will use variables, the second will do all of its work directly with the stack. Both are used in the same way: `3 4 hypotenuse` will leave a `5` on the stack. Here is the first, using variables.

```
/hypotenuse {
  /b exch def
  /a exch def
  a dup mul b dup mul add sqrt
} def
```

This is reasonably easy to read and understand. There is a problem with it we shall deal with later. The second version is not so readable:

```
/hypotenuse {
  dup mul
  exch
  dup mul
  add sqrt
} def
```

This is more efficient than the first—PostScript is generally very efficient when operating directly with stuff on the stack (as opposed to using variables). Still, the cost in terms of readability here is high enough that my general advice is to imitate the first one of these styles, rather than the second. If you do want to be more efficient, it is a good idea to add lots of comments, so as to trace what's on the stack.

```
/hypotenuse {    % a b
  dup mul       % a b*b
  exch          % b*b a
  dup mul       % b*b a*a
  add           % b*b+a*a
  sqrt          % the square root of the sum is left on the stack
} def
```

3.6. Local variables

There is another problem lurking in our present definition of `draw-square`, that of **variable name conflicts**. If you have a large program with lots of different figures being drawn in various orders, you might very well have several places where you use w and h with different meanings. This can cause a lot of trouble. The way around this is a technique in PostScript that I suggest you use without trying to understand too much about it in detail. We want the variables we use in a procedure to be **local** to that procedure, so that assignments we make to them inside that procedure don't affect other variables with the same name outside the procedure. To do this we add some lines to the procedure:

```
/draw-rectangle { 2 dict begin
  /h exch def
  /w exch def
  newpath
  0 0 moveto
  w 0 rlineto
  0 h rlineto
  w neg 0 rlineto
  0 h neg rlineto
  closepath
  stroke
end } def

2 3 draw-rectangle
```

There are exactly two new lines, in fact, one at either end of the procedure. The line `2 dict begin` sets up a local variable mechanism, and `end` restores the original environment. The 2 is in the statement because we are defining 2 local variables.

- *You should set up a local variable mechanism in all procedures in which you make variable definitions.*

One tricky thing to be aware of is that *all variables defined within this pair will be local variables, so that it is impossible to change the value of global variables within them*. It is not usually a good idea to redefine global variables within a procedure, anyway. It is only slightly too strong to say that you should *never* assign a value to a global variable inside a PostScript procedure. Again: `begin` *and* `end` *have to come in pairs*. If they don't, the effect will be that a certain block of space in the computer will fill up. You might get away with it for a while, but unless you are careful sooner or later some awful error is bound to occur. Another good rule to follow is that if you change coordinates inside a procedure, say in order to build a path, you should restore the original coordinates before it exits—unless the explicit purpose of the procedure is to change coordinates. The general principle is that

- *Procedures should have as few side effects as possible, and those side effects should always be explicit.*

Another thing to keep in mind is that while local variables are indispensable in procedures, they can in fact be useful anywhere in a program where you want to avoid name conflicts. Encapsulating a segment of code with `1 dict begin ... end` is often a great way to handle variables safely.

Dictionaries are expensive to set up, in the programmer's sense. They are costly in time. Sometimes, in critical places, it is definitely worthwhile not to introduce any variables at all in a procedure and rely entirely on stack manipulations alone. Or at least use them sparingly. The first of the two procedures

```
/h1 {
  dup mul exch dup mul add sqrt
} def

/h2 { 2 dict begin
```

```
/y exch def
/x exch def
x x mul y y mul add sqrt
end } def
```

takes noticeably less time than the second: $10,000,000$ calls on the first take about 5 seconds on my machine, *versus* about 35 seconds for the second. Recall that to run PostScript programs with no display, as is often useful, you can use Ghostscript in command-line mode as `gsnd`.

At any rate, don't worry too much about exactly what you have to do to set up local variables. Just copy the pattern without thinking about it. The `2` in `2 dict begin` could have been `3` or `4` or `20`. In the earliest version of PostScript, it had to be at least as large as the number of variables you are about to define, but in more recent versions a dictionary will expand to whatever size is needed. So even a `1` would be acceptable. In most of my code I am pretty sloppy about this.

3.7. A final improvement

I want to mention here a subtle but valuable point about procedures. It is only rarely a good idea in PostScript to do any actual drawing inside a procedure. This is part of the general principle that the side effects of procedures ought to be severely restricted. Instead, it is usually a good idea to use procedures to build paths without drawing them. Furthermore, it is *always* a good idea to tell in a comment what you have to do to use a procedure, and what its effect is. Thus

```
% Builds a rectangular path with
% first corner at origin.
% On stack at entry:  width height
/rectangle { 2 dict begin
  /h exch def
  /w exch def
  0 0 moveto
  w 0 rlineto
  0 h rlineto
  w neg 0 rlineto
  0 h neg rlineto
  closepath
end } def

newpath
2 3 rectangle
stroke
```

is the preferable way to use a procedure to draw rectangles. This way you can `fill` them or `clip` them as well as `stroke` them. (We shall meet clipping later.) You can also link paths together to make more complicated paths.