Mathematics 308—Fall 1996

Loops

I have talked about one way of doing loops in PostScript, but in these notes I shall explain all of them. There are three: **repeat**, **for**, **loop**. In order to show how they work, I shall demonstrate how each of them deals with the problem of drawing 10 squares with one centimeter sides next to each other in a row. In each case the lines I show will be sandwiched in between the lines

```
%!

72 72 scale                % inches
1 2.54 div dup scale        % centimetres

0.5 4 translate

/rectangle {
...  } def

...

showpage
```

where **w h rectangle** builds a rectangle at the origin with width w and height h.

I should point out that in each case there are several ways to draw these squares, and that I have chosen the particular lines I present in order to demonstrate how things can go rather than how they should go.

1. Repeat

This one is very simple.

```
10 {
...
} repeat
```

just causes the procedure in the curly brackets **{ ... }** to be repeated 10 times. Repeat loops are the simplest ones, because they have fewer dangers and are not liable to misuse. But sometimes you may have to go to extraordinary lengths to get them to do the job you want done. The variable used at the head of the loop has to be an integer. Variables defined in the loop can be used outside it.

```
10 {
newpath
1 1 rectangle
stroke
1 0 translate
} repeat
```

2. For

These are the next easiest to use. They look like this:

```
0 1 10 {
...
} for
```

There is a hidden variable which starts with value 0, steps up 1 in each iteration of the loop, and runs through the iteration when this variable is 10. Thus this loop runs 11 times in all. The main danger in this loop is that the variable value at exit has to be reached by the loop variable. Thus

```
1 2 10 {
...
} for
```

will loop forever, since the loop variable takes values 1, 3, 5, ... and never hits 10. The values used at the head of the loop don't have to be integers, but if they are not you will likely meet problems with rounding errors. Therefore I counsel you always to use integers even if you don't have to. There is another peculiarity of **for** loops, which is that in every loop the loop variable will be put on the stack just before the loop is entered. Therefore the lines

```
1 1 5 {
} for
```

have a non-trivial effect: they put the numbers 1, 2, 3, 4, 5 on the stack in that order. You can do any of several things with this value. The simplest is to get rid of it, by popping it.

```
1 1 5 {
pop
...
} for
```

will loop five times and on each iteration pop the loop variable value off the top of the stack.

More likely you want to save the loop variable value and use it somewhere else in the loop. This requires the same trick with **exch** which you used to get values of variables when a procedure is called.

```
1 1 5 {
/i exch def
...
} for
```

Here are some lines to draw the squares which illustrate how things go:

```
0 1 9 {
/i exch def
gsave
i 0 translate
newpath
1 1 rectangle
stroke
grestore
} for
```

3. Loop

This one is the most versatile, but requires more care than the others.

```
{
...  }
loop
```

will perform the procedure in between forever, unless you arrange for it not to. You can do that by putting some kind of test inside and calling **exit** in whatever circumstances you want the loop to stop. This requires using boolean variables in PostScript, which I shall explain in the next section.

```
/i 0 def
{
i 9 gt {
    exit
} if
/i i 1 add def
newpath
1 1 rectangle
stroke
1 0 translate
} loop
```

4. Conditionals

PostScript has a type of varaible with two values, **true** and **false**. They are used in various tests, in conjunction with the operators **if** and **ifelse**.

In the program in the previous section, the operator **gt** (‘**g**reater **t**han’) puts **true** or **false** on the stack depending on the relative size of the two numbers at the top of the stack (and popping them both). Thus **2 3 gt** puts **false** on the stack, while **3 2 gt** puts **true**. These operators act just like arithmetic operations—they remove the stuff they operate on, then place the result on top of the stack.

The command **if** always works this way:

```
true/false { ...  } if
```

performs the procedure if there is a **true** on the stack just before it, otherwise just continues on. The lines

```
i 9 gt {
    exit
} if
```

test to see whether $i > 9$ and if it is then the program leaves the loop.

The other conditional is **ifelse**, which performs one of two procedures depending on whether a **true** or **false** is on the stack. Thus

```
x y gt
{
    (x greater than y) ==
}
{
    (x not greater than y) ==
} ifelse
```

will display the string **x greater than y** on the screen if it is so, otherwise **x not greater than y**.