9 How Recursion Works

The last two chapters were about how to write recursive procedures. This chapter is about how to *believe in* recursive procedures, and about understanding the process by which Logo carries them out.

Little People and Recursion

In Chapter 3, I introduced you to the metaphor of a computer full of little elves. Each elf is an expert on a particular procedure. I promised that this metaphor would be helpful later, when we'd have to think about two little people carrying out the same procedure at the same time. Well, "later" is now.

I want to use the elf metaphor to think about the `downup` example of the previous chapter:

```
to downup :word
print :word
if equalp count :word 1 [stop]
downup butlast :word
print :word
end
```

Recall that we are imagining the computer to be full of elves, each of whom is a specialist in carrying out some procedure. There are `print` elves, `count` elves, `stop` elves, and so on. Each elf has some number of pockets, used to hold the inputs for a particular invocation of a procedure. So a `print` elf will have one pocket, while an `equalp` elf needs two pockets.

We're going to be most interested in the `downup` elves and the contents of their pockets. To help you keep straight which elf is which, I'm going to name the `downup` elves alphabetically: the first one will be Ann, then Bill, then Cathy, then David, and so on. Since we aren't so interested in the other elves, I won't bother naming them.

☞ If you're reading this with a group of other people, you may find it helpful for each of you to take on the role of one of the `downup` elves and actually stick words in your pockets. If you have enough people, some of you should also serve as elves for the primitive procedures used, like `print` and `if`.

What happens when you type the instruction

```
downup "hello
```

to Logo? The Chief Elf reads this instruction and sees that it calls for the use of the procedure named `downup`. She therefore recruits Ann, an elf who specializes in that procedure. Since `downup` has one input, the Chief Elf has to give Ann something to put in her one pocket. Fortunately, the input you provided is a quoted word, which evaluates to itself. No other elves are needed to compute the input. Ann gets the word `hello` in her pocket.

Ann's task is to carry out the instructions that make up the definition of `downup`. The first instruction is

```
print :word
```

This, you'll remember, is an abbreviation for

```
print thing "word
```

Ann must hire two more elves, a `print` specialist and a `thing` specialist. The `print` elf can't begin his work until he's given something to put in his pocket. Ann asks the `thing` elf to figure out what that input should be. The `thing` elf also gets an input, namely the word `word`. As we saw in Chapter 3, `word` is what's written on the name tag in Ann's pocket, since `word` is the name of `downup`'s input. So the `thing` elf looks in that pocket, where it finds the word `hello`. That word is then given to the `print` elf, who prints it on your computer screen.

Ann is now ready to evaluate the second instruction:

```
if equalp count :word 1 [stop]
```

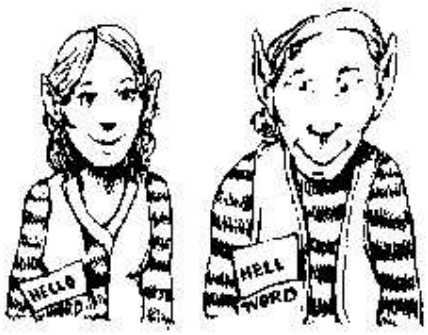
Ann must hire several other elves to help her: an `if` elf, a `count` elf, and a `thing` elf. I won't go through all the steps in computing the inputs to `if`; since the `count` of the word `hello` is not 1, the first input to `if` turns out to be the word `false`. The second input to `if` is, of course, the list `[stop]`. (Notice that Ann does *not* hire a `stop` specialist. A list inside square brackets evaluates to itself, just like a quoted word, without invoking any procedures. If the first input to `if` had turned out to be `true`, it would have been the `if` elf who would have hired a `stop` elf to carry out the instruction inside the list.) Since its first input is `false`, the `if` elf ends up doing nothing.

Ann's third instruction is

```
downup butlast :word
```

Here's where things start to get interesting. Ann must hire *another* `downup` specialist, named Bill. (Ann can't carry out this new `downup` instruction herself because she's already in the middle of a job of her own.) Ann must give Bill an input to put in his pocket; to compute this input, she hires a `butlast` elf and a `thing` elf. They eventually come up with the word `hell` (the `butlast` of `hello`), and that's what Ann puts in Bill's pocket.

We now have two active `downup` elves, Ann and Bill. Each has a pocket. Both pockets are named `word`, but they have different contents: Ann's `word` pocket contains `hello`, while Bill's `word` pocket contains `hell`.



Ann Bill

Here is what this metaphor represents, in more technical language: Although there is only one *procedure* named `downup`, there can be more than one *invocation* of that procedure in progress at a particular moment. (An invocation of a procedure is also sometimes called an *instantiation* of the procedure.) Each invocation has its own local variables; at this moment there are *two* variables named `word`. It is perfectly possible for

two variables to have the same name as long as they are associated with (local to) different procedure invocations.

If you had trouble figuring out how `downup` works in Chapter 7, it's almost certainly because of a misunderstanding about this business of local variables. That's what makes the elf metaphor so helpful. For example, if you're accustomed to programming in BASIC, then you're familiar with *global* variables as the only possibility in the language. If all variables were global in Logo, then there could only be one variable in the entire computer named `word`. Instead of representing variables as pockets in the elves' clothes, we'd have to represent them as safe deposit boxes kept in some central bank and shared by all the elves.

But even if you're familiar with Logo's use of local variables, you may have been thinking of the variables as being local to a *procedure,* instead of understanding that they are local to an *invocation* of a procedure. In that case you may have felt perfectly comfortable with the procedures named `downup1`, `downup2`, and so on, each of them using a separate variable named `word`. But you may still have gotten confused when the *same* variable `word`, the one belonging to the single procedure `downup`, seemed to have several values at once.

If you were confused in that way, here's how to use the elf metaphor to help yourself get unconfused: Suppose the procedure definitions are written on scrolls, which are kept in a library. There is only one copy of each scroll. (That is, there is only one definition for a given procedure.) All the elves who specialize in a particular procedure, like `downup`, have to share the same scroll. Well, if variables were local to a procedure, they'd be pockets in the *scroll,* rather than pockets in the *elves' jackets.* By directing your attention to the elves (the invocations) instead of the scrolls (the procedure definitions), you can see that there can be two variables with the same name (`word`), associated with the same procedure (`downup`), but belonging to different invocations (represented by the elves Ann and Bill).

We still have several more elves to meet, so I'm going to pass over some of the details more quickly now. We've just reached the point where Bill is ready to set to work. For his first instruction he hires a `print` elf, who prints the word `hell` on your screen. Why `hell` and not `hello`? The answer is that when Bill hires a `thing` expert to evaluate the expression `:word`, the `thing` rules say that that expert must look *first* in Bill's pockets, *then* (if Bill didn't have a pocket named `word`) in Ann's pockets.

Bill then carries out the `if` instruction, which again has no effect. Then Bill is ready for the `downup` instruction. He hires a third `downup` elf, named Cathy. Bill puts the word `hel` in Cathy's pocket. There are now three elves, all with pockets named `word`, each with a different word.

Cathy is now ready to get to work. Don't forget, though, that Ann and Bill haven't finished their jobs. Bill is still working on his third instruction, waiting for Cathy to report the completion of her task. Similarly, Ann is waiting for Bill to finish.

Cathy evaluates her first instruction, printing `hel` on the screen. She evaluates the `if` instruction, with no effect. Then she's ready for the `downup` instruction, the third one in the procedure definition. To carry out this instruction, she hires David, a fourth `downup` expert. She puts the word `he` in his pocket.

David's career is like that of the other `downup` elves we've met so far. He starts by printing his input, the word `he`. He evaluates the `if` instruction, with no effect. (The `count` of the word `he` is still not equal to 1.) He then gets to the recursive invocation of `downup`, for which he hires a fifth expert, named Ellen. He puts the word `h` in Ellen's pocket.

Ellen's career is *not* quite like that of the other elves. It starts similarly: she prints her input, the word `h`, on your screen. Then she prepares to evaluate the `if` instruction. This time, though, the first input to `if` turns out to be the word `true`, since the `count` of `h` is, indeed, 1. Therefore, the `if` elf evaluates the instruction contained in its second input, the list `[stop]`. It hires a `stop` elf, whose job is to tell Ellen to stop working. (Why Ellen? Why not one of the other active elves? There are *seven* elves active at the moment: Ann, Bill, Cathy, David, Ellen, the `if` elf, and the `stop` elf. The rule is that a `stop` elf stops the *lowest-level invocation of a user-defined procedure.* `If` and `stop` are primitives, so they don't satisfy the `stop` elf. The remaining five elves are experts in `downup`, a user-defined procedure; of the five, Ellen is the lowest-level invocation.)

(By the way, the insistence of `stop` on a user-defined procedure to stop is one of the few ways in which Logo treats such procedures differently from primitive procedures. If you think about it, you'll see that it would be useless for `stop` to stop just the invocation of `if`. That would mean that the `if` instruction would never do anything of interest and there would be no way to stop a procedure of your own conditionally. But you can imagine other situations in which it would be nice to be able to `stop` a primitive. Here's one:

```
repeat 100 [print "hello if equalp random 5 0 [stop]]
```

If it worked, this instruction would print the word `hello` some number of times, up to 100, but with a 20 percent chance of stopping after each time. In fact, though, you can't use `stop` to stop a `repeat` invocation.)

Let's review what's been printed so far:

```
hello   printed by Ann
hell    printed by Bill
hel     printed by Cathy
he      printed by David
h       printed by Ellen
```

Ellen has just stopped. She reports back to David, the elf who hired her. He's been waiting for her; now he can continue with his own work. David is up to the fourth and final instruction in the definition of `downup`:

```
print :word
```

What word will David print? For David, `:word` refers to the contents of *his own* pocket named `word`. That is, when David hires a `thing` expert, that expert looks first in David's pockets, before trying Cathy's, Bill's, and Ann's. The word in David's `word` pocket is `he`. So that's what David prints.

Okay, now David has reached the end of his instructions. He reports back to his employer, Cathy. She's been waiting for him, so that she can continue her own work. She, too, has one more `print` instruction to evaluate. She has the word `hel` in her `word` pocket, so that's what she prints.

Cathy now reports back to Bill. He prints his own word, `hell`. He reports back to Ann. She prints her word, `hello`.

When Ann finishes, she reports back to the Chief Elf, who prints a question mark on the screen and waits for you to type another instruction.

Here is the complete effect of this `downup` instruction:

```
hello   printed by Ann
hell    printed by Bill
hel     printed by Cathy
he      printed by David
h       printed by Ellen
he      printed by David
hel     printed by Cathy
hell    printed by Bill
hello   printed by Ann
```

☞ You might want to see if the little person metaphor can help you understand the working of the `inout` procedure from Chapter 7. Remember that each elf carrying out the recursive procedure needs two pockets, one for each input.

Tracing

Many people find the idea of multiple, simultaneous invocations of a single procedure confusing. To keep track of what's going on, you have to think about several "levels" of evaluation at once. "Where is `downup` up to right now?" — "Well, it depends what you mean. The lowest-level `downup` invocation has just evaluated its first `print` instruction. But there are three other invocations of `downup` that are in the middle of evaluating their recursive `downup` instructions." This can be especially confusing if you've always been taught that the computer can only do one thing at a time. People often emphasize the *sequential* nature of the computer; what we've been saying about recursion seems to violate that nature.

If this kind of confusion is a problem for you, it may help to think about a procedure like `downup` by *tracing* its progress. That is, we can tell the procedure to print out extra information each time it's invoked, to help you see the sequence of events.

Just for reference, here's `downup` again:

```
to downup :word
print :word
if equalp count :word 1 [stop]
downup butlast :word
print :word
end
```

The trace command takes a procedure name (or a list of procedure names, to trace more than one) as its input. It tells Logo to notify you whenever that procedure is invoked:

```
? trace "downup
? downup "logo
( downup "logo )
logo
 ( downup "log )
log
  ( downup "lo )
lo
   ( downup "l )
l
   downup stops
lo
  downup stops
log
 downup stops
logo
downup stops
```

To make this result a little easier to read, I've printed the lines that are generated by the tracing in smaller letters than the lines generated by `downup` itself. Of course the actual computer output all looks the same.

Each line of tracing information is indented by a number of spaces equal to the number of traced procedure invocations already active—the *level* of procedure invocation. By looking only at the lines between one `downup` invocation and the equally-indented stopping line, you can see how much is accomplished by each recursive call. For example, the innermost invocation (at level 4) prints only the letter `l`.

Level and Sequence

The result of tracing `downup` is most helpful if you think about it two-dimensionally. If you read it *vertically,* it represents the *sequence* of instructions that fits the traditional model of computer programming. That is, the order of the printed lines represents the order of events in time. First the computer enters `downup` at level 1. Then it prints the word `logo`. Then it enters `downup` at level 2. Then it prints `log`. And so on. Each printed line, including the "official" lines as well as the tracing lines, represents a particular instruction, carried out at a particular moment. Reading the trace vertically will help you fit `downup`'s recursive method into your sequential habits of thought.

On the other hand, if you read the trace *horizontally,* it shows you the hierarchy of *levels* of `downup`'s invocations. To see this, think of the trace as divided into two overlapping columns. The left column consists of the official pattern of words printed by the original `downup`. In the right column, the pattern of entering and exiting from each level is shown. The lines corresponding to a particular level are indented by a number of spaces that corresponds to the level number. For example, find the line

```
( downup "log )
```

and the matching

```
downup stops
```

Between these two lines you'll see this:

```
log
  ( downup "lo )
lo
   ( downup "l )
l
   downup stops
lo
  downup stops
log
```

What this shows is that levels 3 and 4 are *part of* level 2. You can see that the traced invocation and stopping lines for levels 3 and 4 begin further to the right than the ones for level 2. Similarly, the lines for level 4 are further indented than the ones for level 3. This variation in indentation is a graphic display of the superprocedure/subprocedure relationships among the various invocations.

There are two ways of thinking about the lines that aren't indented. One way is to look at all such lines within, say, level 2:

```
log
lo
l
lo
log
```

This tells you that those five lines are printed somehow within the activity of level 2. (In terms of the little people metaphor, those lines are printed by Bill, either directly or through some subordinate elf.) Another way to look at it is this:

```
 ( downup "log )
log
  ( downup "lo )
  ...
  downup stops
log
 downup stops
```

What this picture is trying to convey is that only the two `log` lines are *directly* within the control of level 2. The three shorter lines (`lo`, `l`, `lo`) are delegated to level 3.

We've seen three different points of view from which to read the trace, one vertical and two horizontal. The vertical point of view shows the sequence of events in time. The horizontal point of view can show either the *total* responsibility of a given level or the *direct* responsibility of the level. To develop a full understanding of recursion, the trick is to be able to see all of these aspects of the program at the same time.

☞ Try invoking the traced `downup` with a single-letter input. Make a point of reading the resulting trace from all of these viewpoints. Then try a two-letter input.

Instruction Stepping

Perhaps you are comfortable with the idea of levels of invocation, but confused about the particular order of instructions within `downup`. Why should the `if` instruction be where it is, instead of before the first `print`, for example? Logo's `step` command will allow you to examine each instruction line within `downup` as it is carried out:

```
? step "downup
? downup "ant
[print :word] >>>
ant
[if equalp count :word 1 [stop]] >>>
[downup butlast :word] >>>
[print :word] >>>
an
[if equalp count :word 1 [stop]] >>>
[downup butlast :word] >>>
[print :word] >>>
a
[if equalp count :word 1 [stop]] >>>
[print :word] >>>
an
[print :word] >>>
ant
```

After each of the lines ending with **>>>**, Logo waits for you to press the RETURN or ENTER key.

You can combine `trace` and `step`:

```
? step "downup
? trace "downup
? downup "ant
( downup "ant )
[print :word] >>>
ant
[if equalp count :word 1 [stop]] >>>
[downup butlast :word] >>>
 ( downup "an )
 [print :word] >>>
an
 [if equalp count :word 1 [stop]] >>>
 [downup butlast :word] >>>
  ( downup "a )
  [print :word] >>>
a
  [if equalp count :word 1 [stop]] >>>
  downup stops
 [print :word] >>>
an
 downup stops
[print :word] >>>
ant
downup stops
```

In this case, the `step` lines are indented to match the `trace` lines.

Once a procedure is `trace`d or `step`ped, it remains so until you use the `untrace` or `unstep` command to counteract the tracing or stepping.

☞ Try drawing a vertical line extending between the line

```
( downup "an )
```

and the equally indented

```
downup stops
```

Draw the line just to the left of the printing, after the indentation. The line you drew should also touch exactly four instruction lines. These four lines make up the entire definition of the `downup` procedure. If we restrict our attention to one particular invocation of `downup`, like the one you've marked, you can see that each of `downup`'s instructions is, indeed, evaluated in the proper sequence. Below each of these instruction lines, you can see the effect of the corresponding instruction. The two `print` instructions each print one line in the left (unindented) column. (In this case, they both print the word `an`.) The `if` instruction has no visible effect. But the recursive invocation of `downup` has quite a large effect; it brings into play the further invocation of `downup` with the word `a` as input.

One way to use the stepping information is to "play computer." Pretend you are the Logo interpreter, carrying out a `downup` instruction. Exactly what would you do, step by step? As you work through the instructions making up the procedure definition, you can check yourself by comparing your activities to what's shown on the screen.