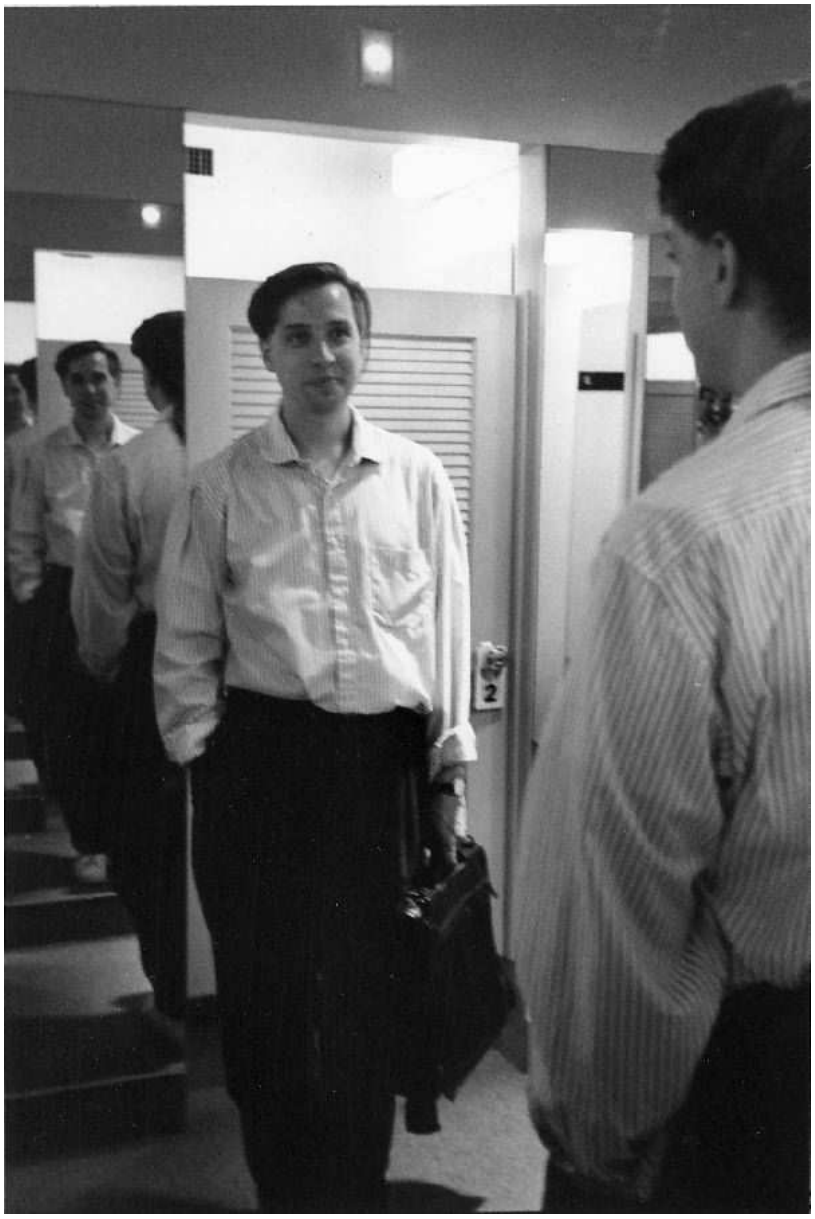

What's the base case?

13 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 Scheme carries them out.

Little People and Recursion

The crowning achievement of the little-people model is explaining recursion. Remember that every time you call a procedure, a little person is hired to compute the result. If you want to know `(+ 2 (+ 3 4))`, there are two separate plus specialists involved.

When we used the combining method, it was probably clear that it's okay for `downup3` to invoke `downup2`, and for `downup2` to invoke `downup1`. But it probably felt like magic when we combined these numbered procedures into a single `downup` procedure that calls *itself.* You may have thought, "How can `downup` do all the different tasks at once without getting confused?" The little-people model answers this question by showing that tasks are done by procedure *invocations,* not by procedures. Each little person handles one task, even though several little people are carrying out the same procedure. The procedure is just a set of instructions; someone has to carry out the instructions.

So what happens when we want to know `(downup 'smile)`? We hire Donna, a `downup` specialist, and she substitutes `smile` for `wd` in the body of `downup`, leaving her with

```
(if (= (count 'smile) 1)
    (se 'smile)
    (se 'smile (downup (bl 'smile)) 'smile)))
```

We'll leave out the details about hiring the `if`, `=`, `count`, and `bl` specialists in this example, so Donna ends up with

```
(se 'smile (downup 'smil) 'smile)
```

In order to evaluate this, Donna needs to know `(downup 'smil)`. She hires David, another `downup` specialist, and waits for his answer.

David's `wd` is `smil`. He substitutes `smil` for `wd` in the body of `downup`, and *he* gets

```
(if (= (count 'smil) 1)
    (se 'smil)
    (se 'smil (downup (bl 'smil)) 'smil)))
```

After some uninteresting work, David has

```
(se 'smil (downup 'smi) 'smil)
```

and he hires Dennis to compute `(downup 'smi)`. There are now three little people, all in the middle of some `downup` computation, and each of them is working on a different word.

Dennis substitutes `smi` for `wd`, and ends up with

```
(se 'smi (downup 'sm) 'smi)
```

He hires Derek to compute `(downup 'sm)`. Derek needs to compute

```
(se 'sm (downup 's) 'sm)
```

Derek hires Dexter to find `downup` of `s`. Now we have to think carefully about the substitution again. Dexter substitutes his actual argument, `s`, for his formal parameter `wd`, and ends up with

```
(if (= (count 's) 1)
    (se 's)
    (se 's (downup (bl 's)) 's)))
```

`Count` of `s` *is* 1. So Dexter hires Simi, a `sentence` specialist, who returns `(s)`. Dexter returns the same answer to Derek.

Derek, you will recall, is trying to compute

```
(se 'sm (downup 's) 'sm)
```

and now he knows the value of `(downup 's)`. So he hires Savita to compute

```
(se 'sm '(s) 'sm)
```

and the answer is `(sm s sm)`. Derek returns this answer to Dennis. By the way, do you remember what question Derek was hired to answer? Dennis wanted to know `(downup 'sm)`. The answer Derek gave him was `(sm s sm)`, which *is* `downup` of `sm`. Pretty neat, huh?

Dennis hires Sigrid to compute

```
(se 'smi '(sm s sm) 'smi)
```

and returns `(smi sm s sm smi)` to David. His answer is the correct value of `(downup 'smi)`. David returns

```
(smil smi sm s sm smi smil)
```

to Donna, who has been waiting all this time to evaluate

```
(se 'smile (downup 'smil) 'smile)
```

Her waiting microseconds are over. She hires a `sentence` specialist and returns

```
(smile smil smi sm s sm smi smil smile)
```

If you have a group of friends whose names all start with "D," you can try this out yourselves. The rules of the game are pretty simple. Remember that each one of you can have only one single value for `wd`. Also, only one of you is in charge of the game at any point. When you hire somebody, that new person is in charge of the game until he or she tells you the answer to his or her question. If some of you have names that don't start with "D," you can be specialists in `sentence` or `butlast` or something. Play hard, play fair, nobody hurt.

Tracing

The little-people model explains recursion very well, as long as you're willing to focus your attention on the job of one little person, taking the next little person's subtask as a "black box" that you assume is carried out correctly. Your willingness to make that assumption is a necessary step in becoming truly comfortable with recursive programming.

Still, some people are very accustomed to a *sequential* model of computing. In that model, there's only one computer, not a lot of little people, and that one computer has to carry out one step at a time. If you're one of those people, you may find it hard to take the subtasks on faith. You want to know exactly what happens when! There's nothing wrong with such healthy scientific skepticism about recursion.

If you're a sequential thinker, you can *trace* procedures to get detailed information about the sequence of events.* But if you're happy with the way we've been talking about recursion up to now, and if you find that this section doesn't contribute to your understanding of recursion, don't worry about it. Our experience shows that this way of thinking helps some people but not everybody.** Before we get to recursive procedures,

* Unfortunately, `trace` isn't part of the Scheme standard, so it doesn't behave the same way in every version of Scheme.

** Even if tracing doesn't help you with recursion, you'll find that it's a useful technique in debugging any procedure.

let's just trace some nonrecursive ones:

```
(define (double wd) (word wd wd))

> (trace double)
> (double 'frozen)
(double frozen)
frozenfrozen
FROZENFROZEN
```

The argument to `trace` specifies a procedure. When you invoke `trace`, that procedure becomes "traced"; this means that every time you invoke the procedure, Scheme will print out the name of the procedure and the actual arguments. When the procedure returns a value, Scheme will print that value.*

Tracing isn't very interesting if we're just invoking a traced procedure once. But look what happens when we trace a procedure that we're using more than once:

```
> (double (double (double 'yum)))
(double yum)
yumyum
(double yumyum)
yumyumyumyum
(double yumyumyumyum)
yumyumyumyumyumyumyumyum

YUMYUMYUMYUMYUMYUMYUMYUM
```

This time, there were three separate invocations of `double`, and we saw each one as it happened. First we `doubled` `yum`, and the answer was `yumyum`. Then we `doubled` `yumyum`, and so on. Finally, after we invoked `double` for the last time, its result was printed by the read-eval-print loop.

When you're finished investigating a procedure, you can turn off tracing by invoking `untrace` with the procedure as argument:

```
> (untrace double)
```

* In this example the return value was printed twice, because the procedure we traced was invoked directly at the Scheme prompt. Its return value would have been printed once anyway, just because that's what Scheme always does. It was printed another time because of the tracing. In this book we've printed the trace-specific output in smaller type and lower-case to help you understand which is what, but of course on the actual computer you're on your own.

Let's try tracing a recursive procedure:

```
(define (downup wd)
  (if (= (count wd) 1)
      (se wd)
      (se wd (downup (bl wd)) wd)))

> (trace downup)

> (downup 'trace)
(downup trace)
|  (downup trac)
|  |  (downup tra)
|  |  |  (downup tr)
|  |  |  |  (downup t)
|  |  |  |  (t)
|  |  |  (tr t tr)
|  |  (tra tr t tr tra)
|  (trac tra tr t tr tra trac)
(trace trac tra tr t tr tra trac trace)
(TRACE TRAC TRA TR T TR TRA TRAC TRACE)
```

When a procedure calls itself recursively, depending on the phase of the moon,* Scheme may indent the trace display to show the levels of procedure calling, and draw a line of vertical bars ("|") from a procedure's invocation to its return value below. This is so you can look at a procedure invocation and see what value it returned, or vice versa.

How does the trace help us understand what is going on in the recursion? First, by reading the trace results from top to bottom, you can see the actual sequence of events when the computer is carrying out your Scheme program. For example, you can see that we start trying to figure out `(downup 'trace)`; the first thing printed is the line that says we're starting that computation. But, before we get a result from that, four more `downup` computations have to begin. The one that begins last finishes first, returning `(t)`; then another one returns a value; the one that started first is the last to return.

You can also read the trace horizontally instead of vertically, focusing on the levels of indentation. If you do this, then instead of a sequence of independent events (such-and-

* That's computer science slang for "depending on a number of factors that I consider too complicated to bother explaining" or "depending on a number of factors that I don't understand myself." Some computer systems automatically print the phase of the moon on program listings as an aid for programmers with "POM-dependent" programs. What we meant in this case is that it depends both on your version of Scheme and on the exact form of your recursive procedure.

such starts, such-and-such returns a value) you see the *inclusion* of processes within other ones. The smallest downup invocation is entirely inside the next-smallest one, and so on. The initial invocation of downup includes all of the others.

Perhaps you're thinking that downup's pattern of inclusion is the only one possible for recursive procedures. That is, perhaps you're thinking that every invocation includes exactly one smaller invocation, and *that* one includes a yet-smaller one, and so on. But actually the pattern can be more complicated. Here's an example. The *Fibonacci numbers* are a sequence of numbers in which the first two numbers are 1 and each number after that is the sum of the two before it:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

(They're named after Leonardo Pisano. You'd think they'd be called "Pisano numbers," but Leonardo had a kind of alias, Leonardo Fibonacci, just to confuse people.) Here's a procedure to compute the *n*th Fibonacci number:

```
(define (fib n)
  (if (<= n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Here's a trace of computing the fourth Fibonacci number:

```
> (fib 4)
(fib 4)
|  (fib 2)
|  1
|  (fib 3)
|  |  (fib 1)
|  |  1
|  |  (fib 2)
|  |  1
|  2
3
3
```

(By the way, this trace demonstrates that in the dialect of Scheme we used, the argument subexpressions of the + expression in fib are evaluated from right to left, because the smaller fib arguments come before the larger ones in the trace.)

As you can see, we still have invocations within other invocations, but the pattern is not as simple as in the downup case. If you're having trouble making sense of this pattern, go back to thinking about the problem in terms of little people; who hires whom?

Pitfalls

⇒ Whenever you catch yourself using the words "go back" or "goes back" in describing how some procedure works, bite your tongue. A recursive invocation isn't a going back; it's a separate process. The model behind "go back" is that the same little person starts over again at the beginning of the procedure body. What actually happens is that a new little person carries out the same procedure. It's an important difference because when the second little person finishes, the first may still have more work to do.

For example, when we used little people to show the working of `downup`, Dennis computes the result `(smi sm s sm smi)` and returns that value to David; at that point, David still has work to do before returning his own result to Donna.

⇒ The `trace` mechanism doesn't work for special forms. For example, you can't say

```
(trace or)
```

although you can, and often will, trace primitive procedures that aren't special forms.

Boring Exercises

13.1 Trace the `explode` procedure from page 183 and invoke

```
(explode 'ape)
```

How many recursive calls were there? What were the arguments to each recursive call? Turn in a transcript showing the `trace` listing.

13.2 How many `pigl`-specialist little people are involved in evaluating the following expression?

```
(pigl 'throughout)
```

What are their arguments and return values, and to whom does each give her result?

13.3 Here is our first version of `downup` from Chapter 11. It doesn't work because it has no base case.

```
(define (downup wd)
  (se wd (downup (bl wd)) wd))
```

```
> (downup 'toe)
ERROR: Invalid argument to BUTLAST: ""
```

Explain what goes wrong to generate that error. In particular, why does Scheme try to take the `butlast` of an empty word?

13.4 Here is a Scheme procedure that never finishes its job:

```
(define (forever n)
  (if (= n 0)
      1
      (+ 1 (forever n))))
```

Explain why it doesn't give any result. (If you try to trace it, make sure you know how to make your version of Scheme stop what it's doing and give you another prompt.)

Real Exercises

13.5 It may seem strange that there is one little person per *invocation* of a procedure, instead of just one little person per procedure. For certain problems, the person-per-procedure model would work fine.

Consider, for example, this invocation of `pigl`:

```
> (pigl 'prawn)
AWNPRAY
```

Suppose there were only one `pigl` specialist in the computer, named Patricia. Alonzo hires Patricia and gives her the argument `prawn`. She sees that it doesn't begin with a vowel, so she moves the first letter to the end, gets `rawnp`, and tries to `pigl` that. Again, it doesn't begin with a vowel, so she moves another letter to the end and gets `awnpr`. That *does* begin with a vowel, so she adds an `ay`, returning `awnpray` to Alonzo.

Nevertheless, this revised little-people model doesn't always work. Show how it fails to explain what happens in the evaluation of

```
(downup 'smile)
```

13.6 As part of computing `(factorial 6)`, Scheme computes `(factorial 2)` and gets the answer `2`. After Scheme gets that answer, how does it know what to do next?