


CHAPTER TWO

Recursion and Induction

2.1 Recursion

We have used Scheme to write procedures that describe how certain computational processes can be carried out. All the procedures we've discussed so far generate processes of a fixed size. For example, the process generated by the procedure `square` always does exactly one multiplication no matter how big or how small the number we're squaring is. Similarly, the procedure `pinwheel` generates a process that will do exactly the same number of `stack` and `turn` operations when we use it on a basic block as it will when we use it on a huge quilt that's 128 basic blocks long and 128 basic blocks wide. Furthermore, the size of the procedure (that is, the size of the procedure's text) is a good indicator of the size of the processes it generates: Small procedures generate small processes and large procedures generate large processes.

On the other hand, there are procedures of a fixed size that generate computational processes of varying sizes, depending on the values of their parameters, using a technique called *recursion*. To illustrate this, the following is a small, fixed-size procedure for making paper chains that can still make chains of arbitrary length—it has a parameter n for the desired length. You'll need a bunch of long, thin strips of paper and some way of joining the ends of a strip to make a loop. You can use tape, a stapler, or if you use slitted strips of cardstock that look like this , you can just slip the slits together. You'll need some classmates, friends, or helpful strangers to do this with, all of whom have to be willing to follow the same procedure as you. You will need to stand in a line.

To make a chain of length n :

1. If $n = 1$,
 - (a) Bend a strip around to bring the two ends together, and join them.
 - (b) Proudly deliver to your customer a chain of length 1.
2. Otherwise,
 - (a) Pick up a strip.
 - (b) Ask the person next in line to please make you a chain of length $n - 1$.
 - (c) Slip your strip through one of the end links of that chain, bend it around, and join the ends together.
 - (d) Proudly deliver to your customer a chain of length n .

Now you know all there is to know about recursion, you have met a bunch of new people, and if you were ambitious enough to make a long chain, you even have a nice decoration to drape around your room. Despite all these advantages, it is generally preferable to program a computer rather than a person. In particular, using this same recursive technique with a computer comes in very handy if you have a long, tedious calculation to do that you'd rather not do by hand or even ask your friends to do.

For example, imagine that you want to compute how many different outcomes there are of shuffling a deck of cards. In other words, how many different orderings (or *permutations*) of the 52 cards are there? Well, 52 possibilities exist for which card is on top, and for each of those 51 possibilities exist for which card is next, or 52×51 total possibilities for what the top two cards are. This pattern continues similarly on down the deck, leading to a total number of possibilities of $52 \times 51 \times 50 \times \cdots \times 3 \times 2 \times 1$, which is the number that is conventionally called 52 *factorial* and written $52!$. To compute $52!$ we could do a lot of tedious typing, spelling out the 51 multiplications of the numbers from 52 down to 1. Alternatively, we could write a general procedure for computing any factorial, which uses its argument to determine which multiplications to do, and then apply this procedure to 52.

To write this procedure, we can reuse the ideas behind the paper chain procedure. One of these is the following very important general strategy:

The recursion strategy: Do nearly all the work first; then there will only be a little left to do.

Although it sounds silly, it describes perfectly what happened with the paper chain: You (or rather your friends) did most of the work first (making a chain of length $n - 1$), which left only one link for you to do.

Here we're faced with the problem of multiplying 52 numbers together, which will take 51 multiplications. One way to apply the recursion principle is this: Once 50 of the multiplications have been done, only 1 is left to do.

We have many possible choices for which 50 multiplications to do first versus which one to save for last. Almost any choice can be made to work, but some may make us work a bit harder than others. One choice would be to initially multiply together the 51 largest numbers and then be left with multiplying the result by the smallest number. Another possibility would be to initially multiply together the 51 smallest numbers, which would just leave the largest number to multiply in. Which approach will make our life easier? Stop and think about this for a while.

We start out with the problem of multiplying together the numbers from 52 down to 1. To do this, we're going to write a general factorial procedure, which can multiply together the numbers from anything down to 1. Fifty-two down to 1 is just one special case; the procedure will be equally capable of multiplying 105 down to 1, or 73 down to 1, or 51 down to 1.

This observation is important; if we make the choice to leave the largest number as the one left to multiply in at the end, the “nearly all the work” that we need to do first is itself a factorial problem, and so we can use the same procedure. To compute $52!$, we first compute $51!$, and then we multiply by 52. In general, to compute $n!$, for any number n , we'll compute $(n - 1)!$ and then multiply by n . Writing this in Scheme, we get:

```
(define factorial
  (lambda (n)
    (* (factorial (- n 1))
       n)))
```

The strategy of choosing the subproblem to be of the same form as the main problem is probably worth having a name for:

The self-similarity strategy: Rather than breaking off some arbitrary big chunk of a problem to do as a subproblem, break off a chunk that is of the same form as the original.

Will this procedure for computing factorials work? No. It computes the factorial of any number by first computing the factorial of the previous number. That works up to a point; $52!$ can be computed by first computing $51!$, and $51!$ can be computed by first computing $50!$. But, if we keep going like that, we'll never stop. Every factorial will be computed by first computing a smaller one. Therefore $1!$ will be computed in terms of $0!$, which will be computed in terms of $(-1)!$, which will be computed in terms of $(-2)!$, and so on.

When we have a lot of multiplications to do, it makes sense to do all but one and then the one that's left. Even if we only have one multiplication to do, we could do all but one (none) and then the one that's left. But what if we don't have any multiplications at all to do? Then we *can't* do all but one and then the one that's

left—there isn't one to leave for last. The fundamental problem with this procedure is, it tries to always leave one multiplication for last, even when there are none to be done.

Computing $1!$ doesn't require any multiplications; the answer is simply 1. What we can do is treat this *base case* specially, using `if`, just like in the human program for making chains:

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* (factorial (- n 1))
           n))))

(factorial 52)
8065817517094387857166063685640376697528950544088327782400000000
000
```

Thus, base cases are treated separately in recursive procedures. In particular, they result in no further recursive calls. But we also need to guarantee that the recursion will always eventually end in a base case. This is so important that we give it the following name:

The base case imperative: In a recursive procedure, all roads must lead to a base case.

This procedure generates what is called a *recursive process*; a similar but smaller computation is done as a subgoal of solving the main problem. In particular, cases like this with a single subproblem that is smaller by a fixed amount, are called *linear recursions* because the total number of computational steps is a linear function of the problem size. We can see the recursive nature of the process clearly in Figure 2.1, which shows how the evaluation of `(factorial 3)` involves as a subproblem computing `(factorial 2)`, which in turn involves computing `(factorial 1)` as a sub-subproblem. If the original problem had been `(factorial 52)`, the diagram would be 52 columns wide instead of only 3.

This diagram isn't complete—the evaluation of the `if` expression with its equality test isn't explicitly shown and neither is the subtraction of one. These omissions were made to simplify the diagram, leaving the essential information more apparent. If we included all the details, the first three steps (leading from the problem `(factorial 3)` to the subproblem `(factorial 2)`) would expand into the ten steps shown in Figure 2.2.

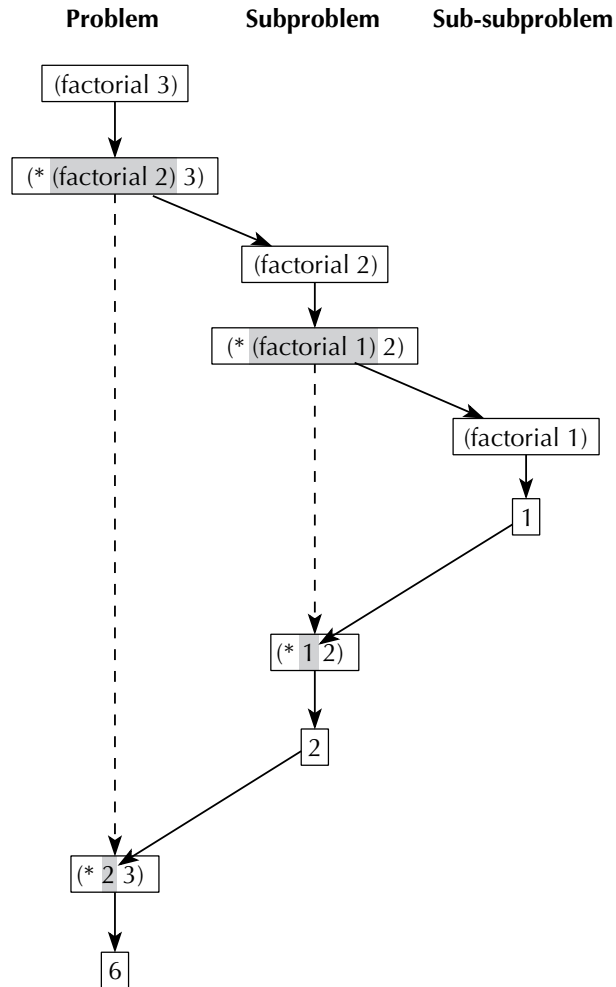


Figure 2.1 The recursive process of evaluating (factorial 3).

Although the recursive nature of the process is most evident in the original diagram, we can as usual save space by instead listing the evaluation steps. If we do this with the same details omitted, we get

```
(factorial 3)
(* (factorial 2) 3)
(* (* (factorial 1) 2) 3)
(* (* 1 2) 3)
(* 2 3)
6
```

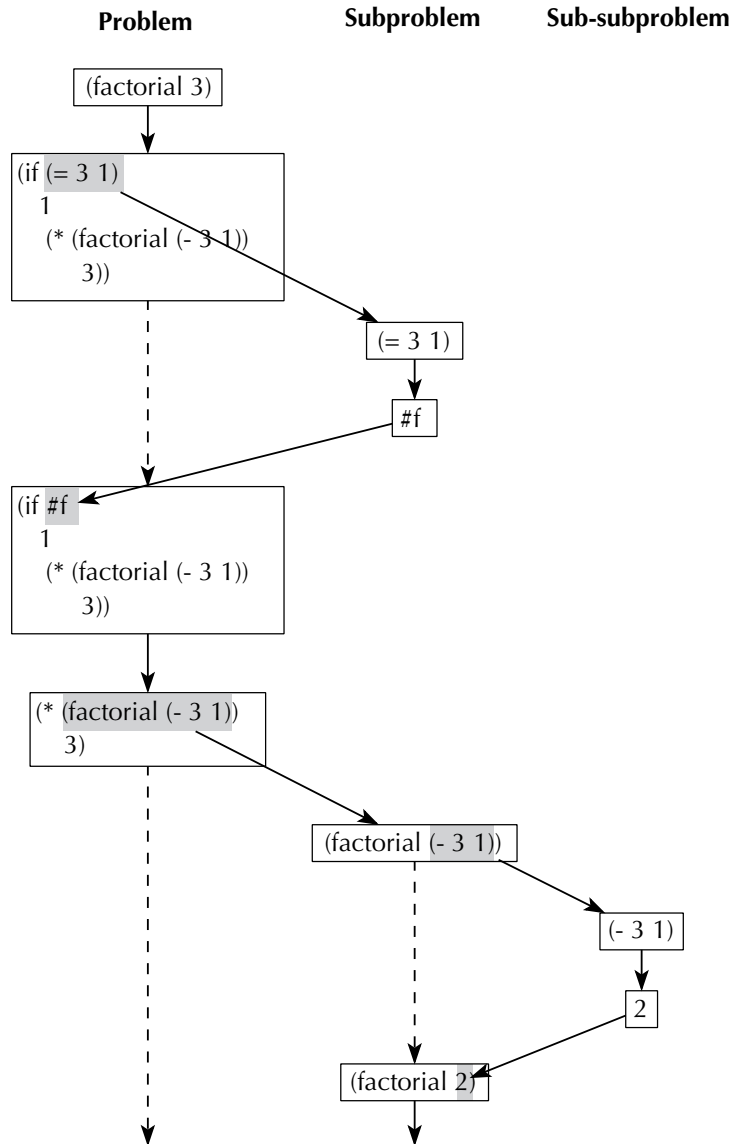


Figure 2.2 Details of the recursive process of evaluating (factorial 3).

Let's sum up what we've done in both the paper chain example and the factorial example. In both, we had to solve a problem by doing something repeatedly, either assembling links or multiplying numbers. We broke off a big chunk of each problem (the recursion principle) that was just like the original problem (the self-similarity principle) except that it was smaller. After that chunk was finished, we only had a little work left to do, either by putting in one more link or multiplying by one more

Exponents

In this book, when we use an exponent, such as the k in x^k , it will almost always be either a positive integer or zero. When k is a positive integer, x^k just means k copies of x multiplied together. That is, $x^k = x \times x \times \cdots \times x$, with k of the x 's. What about when the exponent is zero? We could equally well have said that $x^k = 1 \times x \times x \times \cdots \times x$ with k of the x 's. For example, $x^3 = 1 \times x \times x \times x$, $x^2 = 1 \times x \times x$, and $x^1 = 1 \times x$. If we continue this progression with one fewer x , we see that $x^0 = 1$.

number. In each case, the smaller subproblems must invariably lead to a problem so small that it could be made no smaller (the base case imperative), that is, when we needed to make a chain of length 1 or when we had to compute 1!, which is handled separately.

Exercise 2.1

Write a procedure called `power` such that (`power base exponent`) raises *base* to the *exponent* power, where *exponent* is a nonnegative integer. As explained in the sidebar on exponents, you can do this by multiplying together *exponent* copies of *base*. (You can compare results with Scheme's built-in procedure called `expt`. However, do not use `expt` in `power`. `Expt` computes the same values as `power`, except that it also works for exponents that are negative or not integers.)

2.2 Induction

Do you believe us that the `factorial` procedure really computes factorials? Probably. That's because once we explained the reasoning behind it, there isn't much to it. (Of course, you may also have tried it out on a Scheme system—but that doesn't explain why you believe it works in the cases you didn't try.)

Sometimes, however, it is a bit trickier to convince someone that a procedure generates the right answer. For example, here's another procedure for squaring a number that is rather different from the first one:

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (+ (square (- n 1))
            (- (+ n n) 1))))))
```

Just because it is called `square` doesn't necessarily mean that it actually squares its argument; we might be trying to trick you. After all, we can give any name we want to anything. Why should you believe us? The answer is: You shouldn't, yet, because we haven't explained our reasoning to you. It is not your job as the reader of a procedure to figure it out; it is the job of the writer of a procedure to accompany it with adequate explanation. Right now, that means that we have our work cut out for us. But it also means that when it becomes your turn to write procedures, you are going to have to similarly justify your reasoning.

Earlier we said that the procedure was “for squaring a number.” Now that we're trying to back up that claim, we discover we need to be a bit more precise: This procedure squares any *nonnegative integer*. Certainly it correctly squares 0, because it immediately yields 0 as the answer in that case, and $0^2 = 0$. The real issue is with the positive integers.

We're assuming that `-` subtracts and `+` adds, so `(- n 1)` evaluates to $n - 1$, and `(- (+ n n) 1)` evaluates to $(n + n) - 1$ or $2n - 1$. What if we went one step further and assumed that where `square` is applied to $n - 1$, it squares it, resulting in the value $(n - 1)^2$? In that case, the overall value computed by the procedure is $(n - 1)^2 + 2n - 1$. With a little bit of algebra, we can show that $(n - 1)^2 + 2n - 1 = n^2$, and so in fact the end result is n^2 , just like we said it was.

But wait, not so fast: To show that `square` actually squares n , we had to assume that it actually squares $n - 1$; we seem to need to know that the procedure works in order to show that it works. This apparently circular reasoning isn't, however, truly circular: it is more like a spiral. To show that `square` correctly squares some particular positive integer, we need to assume that it correctly squares some *smaller* particular integer. For example, to show that it squares 10, we need to assume that it can square 9. If we wanted to, though, we could show that it correctly squares 9, based on the assumption that it correctly squares 8. Where does this chain of reasoning end? It ends when we show that `(square 1)` really computes 1^2 , based on the fact that `(square 0)` really computes 0^2 . At that point, the spiraling stops, because we've known since the very beginning that `square` could square 0.

The key point that makes this spiral reasoning work is that the chain of reasoning leads inexorably down to the base case of zero. We only defined `square` in terms of *smaller* squares, so there is a steady progression toward the base case. By contrast, even though it is equally true that $n^2 = (n + 1)^2 - (2n + 1)$, the following procedure does *not* correctly compute the square of any positive integer:

```
(define square      ; This version doesn't work.
  (lambda (n)
    (if (= n 0)
        0
        (- (square (+ n 1))
            (+ (+ n n) 1))))))
```


The reason why this procedure doesn't correctly compute the square of any positive integer isn't that it computes some incorrect answer instead. Rather, it computes no answer at all, because it works its way further and further from the base case, stopping only when the computer runs out of memory and reports failure. We say that the computational process doesn't *terminate*.

We've also used this procedure to introduce another feature of the Scheme programming language: *comments*. Any text from a semicolon to the end of the line is ignored by the Scheme system and instead is for use by human readers.

The reasoning technique we've been using is so generally useful that it has a name: *mathematical induction*. Some standard terminology is also used to make arguments of this form more brief. The justification that the base case of the procedure works is called the *base case* of the proof. The assumption that the procedure works correctly for smaller argument values is called the *induction hypothesis*. The reasoning that leads from the induction hypothesis to the conclusion of correct operation is called the *inductive step*. Note that the inductive step only applies to those cases where the base case doesn't apply. For `square`, we only reasoned from $n - 1$ to n in the case where n was positive, not in the case where it was zero.

Putting this all together, we can write an inductive proof of `square`'s correctness in a reasonably conventional format:

Base case: (`square 0`) terminates with the value 0 because of the evaluation rule for `if`. Because $0 = 0^2$, (`square 0`) computes the correct value.

Induction hypothesis: Assume that (`square k`) terminates with the value k^2 for all k in the range $0 \leq k < n$.

Inductive step: Consider evaluating (`square n`), with $n > 0$. This will terminate if the evaluation of (`square (- n 1)`) does and will have the same value as `(+ (square (- n 1)) (- (+ n n) 1))`. Because `(- n 1)` evaluates to $n - 1$ and $0 \leq n - 1 < n$, we can therefore assume by our induction hypothesis that (`square (- n 1)`) does terminate, with the value $(n - 1)^2$. Therefore `(+ (square (- n 1)) (- (+ n n) 1))` evaluates to $(n - 1)^2 + 2n - 1$. Because $(n - 1)^2 + 2n - 1 = n^2$, we see that (`square n`) does terminate with the correct value for any arbitrary positive n , under the inductive hypothesis of correct operation for smaller arguments.

Conclusion: Therefore, by mathematical induction on n , (`square n`) terminates with the value n^2 for any nonnegative integer n .

If you have trouble understanding this, one useful trick is to think of proving one special case of the theorem each day. The first day you prove the base case. On any subsequent day, you prove the next case, making use only of results you've *previously* proven. There is no particular case that you won't eventually show to be true—so the theorem must hold in general.

We wish to point out two things about this proof. First, the proof is relative in the sense that it assumes that other operations (such as `+` and `-`) operate as advertised. But this is an assumption you must make, because you were not there when the people who implemented your Scheme system were doing their work. Second, an important part of verifying that a procedure computes the correct value is showing that it actually terminates for all permissible argument values. After all, if the computation doesn't terminate, it computes no value at all and hence certainly doesn't compute the correct value. This need for *termination* explains our enjoinder in the base case imperative given earlier.

▶ Exercise 2.2

Write a similarly detailed proof of the `factorial` procedure's correctness. What are the permissible argument values for which you should show that it works?

Proving is also useful when you are trying to *debug* a procedure that doesn't work correctly, that is, when you are trying to figure out what is wrong and how to fix it. For example, look at the incorrect version of `square` given earlier. If we were trying to prove that this works by induction, the base case and the inductive hypothesis would be exactly the same as in the proof above. But look at what happens in the inductive step:

Inductive step: Consider evaluating `(square n)`, with $n > 0$. This will terminate if the evaluation of `(square (+ n 1))` does and will have the same value as `(- (square (+ n 1)) (+ (+ n n) 1))`. Because `(+ n 1)` evaluates to $n + 1$ and $0 \leq n + 1 < n \dots$ *Oops ...*

The next time you have a procedure that doesn't work, try proving that it does work. See where you run into trouble constructing the proof—that should point you toward the *bug* (error) in the procedure.

▶ Exercise 2.3

Here's an example of a procedure with a tricky bug you can find by trying to do an induction proof. Try to prove the following procedure also computes n^2 for any nonnegative integer n . Where does the proof run into trouble? What's the bug?

```
(define square ; another version that doesn't work
  (lambda (n)
    (if (= n 0)
        0
        (+ (square (- n 2))
            (- (* 4 n) 4))))))
```

The most important thing to take away from this encounter with induction is a new way of thinking, which we can call *one-layer thinking*. To illustrate what we mean by this, contrast two ways of thinking about what the `square` procedure does in computing 4^2 :

1. You can try thinking about all the layers upon layers of squares, with requests going down through the layers and results coming back up. On the way down, 4^2 requests 3^2 requests 2^2 requests 1^2 requests 0^2 . On the way back up, 0 gets $1 + 1 - 1$ added to it yielding 1, which gets $2 + 2 - 1$ added to it yielding 4, which gets $3 + 3 - 1$ added to it yielding 9, which gets $4 + 4 - 1$ added to it yielding 16, which is the answer.
2. Alternatively, you can just stick with one layer. The computation of 4^2 requests 3^2 and presumably gets back 9, because that's what 3^2 is. The 9 then gets $4 + 4 - 1$ (or 7) added to it, yielding the answer 16.

This is really just an informal version of relying on an induction hypothesis—that's what we were doing when we said "... and presumably gets back 9, because that's what 3^2 is." It saves us having to worry about how the whole rest of the computation is done.

One-layer thinking is much better suited to the limited capacities of human brains. You only have to think about a little bit of the process, instead of the entire arbitrarily large process that you've really got. Plunging down through a whole bunch of layers and then trying to find your way back up through them is a good way to get hopelessly confused. We sum this up as follows:

The one-layer thinking maxim: Don't try to think recursively about a recursive process.

One-layer thinking is more than just a way to think about the process a procedure will generate; it is also the key to writing the procedure in the first place. For example, when we presented our recursive version of `square` at the beginning of this section, you may well have wondered where we got such a strange procedure. The answer is that we started with the idea of computing squares recursively, using smaller squares. We knew we would need to have a base case, which would probably be when $n = 0$. We also knew that we had to relate the square of n to the square of some smaller number. This led to the following template:

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (____ (square _____)
              _____))))
```

We knew that the argument to `square` would have to be less than n for the induction hypothesis to apply; on the other hand, it would still need to be a nonnegative integer. The simplest way to arrange this is to use `(- n 1)`; thus we have

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (____ (square (- n 1))
              _____))))
```

At this point, our one-layer thinking tells us not to worry about the specific computational process involved in evaluating `(square (- n 1))`. Instead, we assume that the value will be $(n - 1)^2$. Thus the only remaining question is, What do we need to do to $(n - 1)^2$ to get n^2 ? Because $(n - 1)^2 = n^2 - 2n + 1$, it becomes clear that we need to add $2n - 1$. This lets us fill in the remaining two blanks, arriving at our procedure:

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (+ (square (- n 1))
           (- (+ n n) 1))))))
```

Exercise 2.4

Use one-layer thinking to help you correctly fill in the blanks in the following version of `square` so that it can square any nonnegative integer:

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (if (even? n)
            (____ (square (/ n 2))
                  _____)
            (+ (square (- n 1))
               (- (+ n n) 1))))))
```

2.3 Further Examples

Recursion adds great power to Scheme, and the recursion strategy will be fundamental to the remainder of the book. However, if this is your first encounter with recursion, you may find it confusing. Part of the confusion arises from the fact that recursion seems “circular.” However, it really involves spiraling down to a firm foundation at the base case (or base cases). Another problem at this point is simply lack of familiarity. Therefore, we devote this section to various examples of numerical procedures involving recursion. And the next section applies recursion to quilting.

As our first example, consider the built-in Scheme procedure `quotient`, which computes how many times one integer divides another integer. For example,

```
(quotient 9 3)
```

```
3
```

```
(quotient 10 3)
```

```
3
```

```
(quotient 11 3)
```

```
3
```

```
(quotient 12 3)
```

```
4
```

Even though `quotient` is built into Scheme, it is instructive to see how it can be written in terms of a more “elementary” procedure, in this case subtraction. We’ll write a procedure that does the same job as `quotient`, but we’ll call it `quot` instead so that the built-in `quotient` will still be available. (Nothing stops you from redefining `quotient`, but then you lose the original until you restart Scheme.) In order to simplify the discussion, suppose we want to compute `(quot n d)`, where $n \geq 0$ and $d > 0$. If $n < d$, d doesn’t divide n at all, so the result would be 0. If, however, $n \geq d$, d will divide n one more time than it divides $n - d$. Writing this in Scheme, we have

```
(define quot
  (lambda (n d)
    (if (< n d)
        0
        (+ 1 (quot (- n d) d)))))
```

The built-in version of `quotient`, unlike the `quot` procedure just shown, allows either or both of the arguments to be negative. The value when one or both arguments are negative is defined by saying that negating either argument negates the quotient. For example, because the quotient of 13 and 3 is 4, it follows that the

quotient of -13 and 3 is -4 , and so is the quotient of 13 and -3 . Because negating either argument negates the quotient, negating both of them negates the quotient twice, or in other words leaves it unchanged. For example, the quotient of -13 and -3 is 4 .

In order to negate a number in Scheme, we could subtract it from zero; for example, to negate the value of `n`, we could write `(- 0 n)`. However, it is more idiomatic to instead write `(- n)`, taking advantage of a special feature of the predefined procedure named `-`, namely, that it performs negation if only given a single argument. Note that `(- n)` is quite different in form from `-5`: The former applies a procedure to an argument, whereas the latter is a single number. It is permissible to apply the procedure named `-` to a number, as in `(- 5)`, but you can't put a negative sign on a name the way you would on a number: `-n` isn't legal Scheme.

We could build these ideas into our procedure as follows:

```
(define quot
  (lambda (n d)
    (if (< d 0)
        (- (quot n (- d)))
        (if (< n 0)
            (- (quot (- n) d))
            (if (< n d)
                0
                (+ 1 (quot (- n d) d)))))))
```

Notice that our first version of `quot` corresponds to the innermost `if`; the outer two `if`'s deal with negative values for n and d .

This new, more general, `quot` procedure is our first example of a procedure with `ifs` nested within one another so deeply that they jeopardize the readability of the procedure. Procedures like this can be clarified by using another form of conditional expression that Scheme offers as an alternative to `if`: `cond`. Here is how we can rewrite `quot` using `cond`:

```
(define quot
  (lambda (n d)
    (cond ((< d 0) (- (quot n (- d))))
          ((< n 0) (- (quot (- n) d)))
          ((< n d) 0)
          (else (+ 1 (quot (- n d) d))))))
```

A `cond` consists of a sequence of parenthesized *clauses*, each providing one possible case for how the value might be calculated. Each clause starts with a test expression, except that the last clause can start with the keyword `else`. Scheme evaluates each

test expression in turn until it finds one that evaluates to true, to decide which clause to use. Once a test evaluates to true, the remainder of that clause is evaluated to produce the value of the `cond` expression; the other clauses are ignored. If the `else` clause is reached without any true test having been found, the `else` clause's expression is evaluated. If, on the other hand, no test evaluates to true and there is no `else` clause, the result is not specified by the Scheme language standard, and each system is free to give you whatever result it pleases.

▶ Exercise 2.5

Use addition to write a procedure `multiply` that calculates the product of two integers (i.e., write `*` for integers in terms of `+`).

Suppose we want to write a procedure that computes the sum of the first n integers, where n is itself a positive integer. This is a very similar problem to `factorial`; the difference is that we are adding up the numbers rather than multiplying them. Because the base case $n = 1$ should yield the value 1, we come up with a solution identical in form to `factorial`:

```
(define sum-of-first
  (lambda (n)
    (if (= n 1)
        1
        (+ (sum-of-first (- n 1))
           n))))
```

But why should $n = 1$ be the base case for `sum-of-first`? In fact, we could argue that the case $n = 0$ makes good sense: The sum of the first 0 integers is the “empty sum,” which could reasonably be interpreted as 0. With this interpretation, we can extend the allowable argument values as follows:

```
(define sum-of-first
  (lambda (n)
    (if (= n 0)
        0
        (+ (sum-of-first (- n 1))
           n))))
```

This extension is reasonable because it computes the same values as the original version whenever $n \geq 1$. (Why?) A similar extension for `factorial` would be

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* (factorial (- n 1))
           n))))
```

It is not as clear that the “empty product” should be 1; however, we’ve seen empty products when we talked about exponents (see the sidebar, Exponents). The product of zero copies of x multiplied together is 1; similarly the product of the first zero positive integers is also 1. Not coincidentally, this agrees with the mathematical convention that $0! = 1$.

Exercise 2.6

Let’s consider some variants of the basic form common to `factorial` and `sum-of-first`.

- a. Describe precisely what the following procedure computes in terms of n :

```
(define subtract-the-first
  (lambda (n)
    (if (= n 0)
        0
        (- (subtract-the-first (- n 1))
           n))))
```

- b. Consider what happens when you exchange the order of multiplication in `factorial`:

```
(define factorial2
  (lambda (n)
    (if (= n 0)
        1
        (* n
           (factorial2 (- n 1))))))
```

Experimentation with various values of n should persuade you that this version computes the same value as did the original `factorial`. Why is this so? Would the same be true if you switched the order of addition in `sum-of-first`?

- c. If you reverse the order of subtraction in **subtract-the-first**, you will get a different value in general. Why is this so? How would you precisely describe the value returned by this new version?

One way to generalize **sum-of-first** is to sum up the integers between two specified integers (e.g., from 4 to 9). This would require two parameters and could be written as follows:

```
(define sum-integers-from-to
  (lambda (low high)
    (if (> low high)
        0
        (+ (sum-integers-from-to low (- high 1))
           high))))
```

Note that this could also be accomplished by increasing **low** instead of decreasing **high**.

▶ Exercise 2.7

Rewrite **sum-integers-from-to** in this alternative way.

▶ Exercise 2.8

Another type of generalization of **sum-of-first** can be obtained by varying what is being summed, rather than just the range of summation:

- Write a procedure **sum-of-squares** that computes the sum of the first n squares, where n is a nonnegative integer.
- Write a procedure **sum-of-cubes** that computes the sum of the first n cubes, where n is a nonnegative integer.
- Write a procedure **sum-of-powers** that has two parameters n and p , both nonnegative integers, such that **(sum-of-powers n p)** computes $1^p + 2^p + \cdots + n^p$.

In the factorial procedure, the argument decreases by 1 at each step. Sometimes, however, the argument needs to decrease in some other fashion. Consider, for example, the problem of finding the number of digits in the usual decimal way of writing an integer. How would we compute the number of digits in n , where n is a nonnegative integer? If $n < 10$, the problem is easy; the number of digits would be 1. On the other hand, if $n \geq 10$, the quotient when it is divided by 10 will be all

but the last digit. For example, the quotient when 1234 is divided by 10 is 123. This lets us define the number of digits in n in terms of the number of digits in a smaller number, namely, (`quotient n 10`). Putting this together, we have

```
(define num-digits
  (lambda (n)
    (if (< n 10)
        1
        (+ 1 (num-digits (quotient n 10))))))
```

We could extend `num-digits` to negative integers using `cond`:

```
(define num-digits
  (lambda (n)
    (cond ((< n 0) (num-digits (- n)))
          ((< n 10) 1)
          (else (+ 1 (num-digits (quotient n 10)))))))
```

If we want to do more with the digits than count how many there are, we need to find out what each digit is. We can do this using the remainder from the division by 10; for example, when we divide 1234 by 10, the remainder is 4. A built-in procedure called `remainder` finds the remainder; for example, (`remainder 1234 10`) evaluates to 4.

▶ Exercise 2.9

Write a procedure that computes the number of 6s in the decimal representation of an integer. Generalize this to a procedure that computes the number of d 's, where d is another argument.

▶ Exercise 2.10

Write a procedure that calculates the number of odd digits in an integer. (Reminder: There is a built-in predicate called `odd?`.)

▶ Exercise 2.11

Write a procedure that computes the sum of the digits in an integer.

▶ Exercise 2.12

Any positive integer i can be expressed as $i = 2^n k$, where k is odd, that is, as a power of 2 times an odd number. We call n the exponent of 2 in i . For example, the exponent of 2 in 40 is 3 (because $40 = 2^3 5$) whereas the exponent of 2 in 42 is 1. If i itself is odd, then n is zero. If, on the other hand, i is even, that means it can be divided by 2. Write a procedure for finding the exponent of 2 in its argument.

2.4 An Application: Custom-Sized Quilts

At the end of the previous chapter we made some quilts by pinwheeling basic blocks. The only problem is that the quilts only come in certain sizes: You could make a single cross by pinwheeling `rcross-bb`, or a quilt that is two crosses wide and high by pinwheeling the cross, or four wide and high by pinwheeling that, or But we want a quilt that is four crosses wide and *three* high. We're not being stubborn; we have a paying customer whose bed isn't square. In fact, given that there are lots of different sizes of beds in the world, it would probably be best if we wrote a general purpose procedure that could make a quilt any number of crosses wide and any number high. We know how to make a cross; the challenge is how to replicate an image a desired number of times.

▶ Exercise 2.13

We can often simplify a problem by first considering a one-dimensional version of it. Here, this means we should look at the problem of stacking a specified number of copies of an image one on top of another in a vertical column. Write a procedure `stack-copies-of` so that, for example, `(stack-copies-of 5 rcross-bb)` produces a tall, thin stack of five basic blocks. By the way, the name `stack-copies-of` illustrates a useful trick for remembering the order of the arguments. We chose the name so that it effectively has blanks in it for the arguments to fill in: "stack _____ copies of _____."

▶ Exercise 2.14

Use your `stack-copies-of` from the previous exercise to define a procedure called `quilt` so that `(quilt (pinwheel rcross-bb) 4 3)` makes our desired quilt. In general, `(quilt image w h)` should make a quilt that is w images wide and h images high. Try this out.

Some quilts have more subtle patterns, such as checkerboard-style alternation of light and dark regions. Consider, for example, the *Blowing in the Wind* pattern,

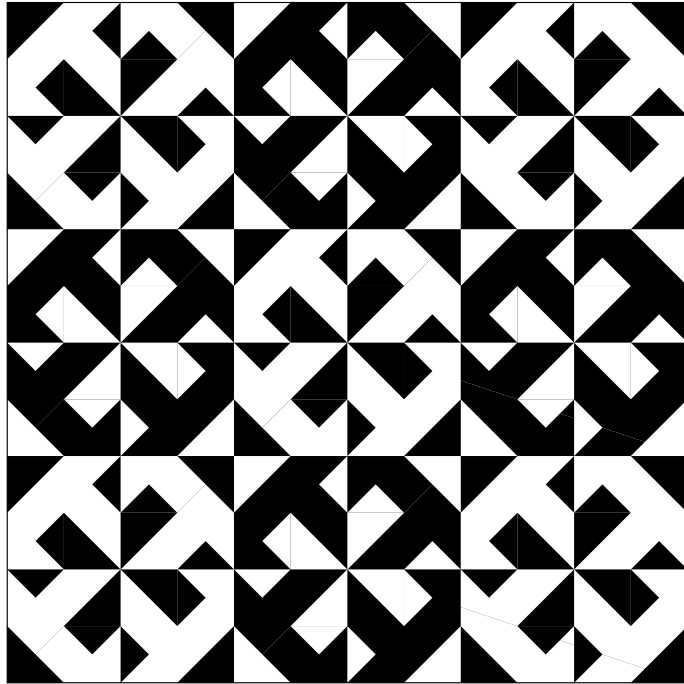
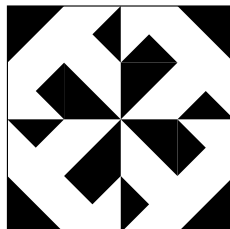


Figure 2.3 The *Blowing in the Wind* quilt pattern.

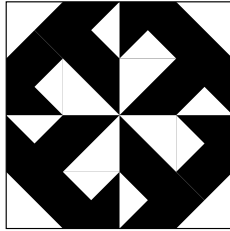
shown in Figure 2.3. This is again made out of pinwheels of a basic block; the basic block, which we've defined as `bitw-bb`, is



and the result of pinwheeling it is



Five copies of this pinwheel appear as the white-on-black regions in the corners and the center of the quilt. The four black-on-white regions of the quilt are occupied by a black/white reversal of the pinwheel, namely,



This “inverted” version of the pinwheel can be produced using the primitive procedure `invert` as follows: `(invert (pinwheel bitw-bb))`.

The trick is to make a checkerboard out of alternating copies of `(pinwheel bitw-bb)` and `(invert (pinwheel bitw-bb))`. We can approach this in many different ways, because so many algebraic identities are satisfied by `invert`, `stack`, and `quarter-turn-right`. For example, inverting an inverted image gives you the original image back, and inversion “distributes” over stacking (inverting a stack gives the same result as stacking the inverses).

Before you write a procedure for alternating inverted and noninverted copies of an image, you should pin down exactly what *alternating* means. For example, you might specify that the image in the lower left corner is noninverted and that the images within each row and column alternate. Or, you could specify that the alternation begins with a noninverted image in the upper left, the upper right, or the lower right. For a three-by-three checkerboard such as is shown here, all of these are equivalent; only if the width or height is even will it make a difference. Nonetheless, it is important before you begin to program to be sure you know which version you are programming.

Exercise 2.15

One way or another, develop a procedure `checkerboard` for producing arbitrarily sized checker-boarded quilts of images. Making a call of the form `(checkerboard (pin-wheel bitw-bb) 3 3)` should result in the Blowing in the Wind pattern of Figure 2.3. The `checkerboard` procedure also produces an interesting “boxed crosses” pattern if you pinwheel `rcross-bb` instead of `bitw-bb` (check it out), although we hadn’t intended it for that purpose, and it can be used with a black (or white) image to make a regular checkerboard. You might be interested to try it on some of your own basic blocks as well.

Review Problems

▶ Exercise 2.16

Consider the following procedure `foo`:

```
(define foo
  (lambda (x n)
    (if (= n 0)
        1
        (+ (expt x n) (foo x (- n 1))))))
```

Use induction to prove that `(foo x n)` terminates with the value

$$\frac{x^{n+1} - 1}{x - 1}$$

for all values of $x \neq 1$ and for all integers $n \geq 0$. You may assume that `expt` works correctly, (i.e., `(expt b m)` returns b^m). *Hint:* The inductive step will involve some algebra.

▶ Exercise 2.17

Perhaps you have heard the following Christmas song:

On the first day of Christmas
My true love gave to me
A partridge in a pear tree.

On the second day of Christmas
My true love gave to me
Two turtle doves
And a partridge in a pear tree.

On the third day of Christmas
My true love gave to me
Three French hens,
Two turtle doves,
And a partridge in a pear tree.

And so on, through the twelfth day of Christmas. Note that on the first day, my true love gave me one present, on the second day three presents, on the third day six presents, and so on. The following procedure determines how many presents I received from my true love on the n th day of Christmas:

```
(define presents-on-day
  (lambda (n)
    (if (= n 1)
        1
        (+ n (presents-on-day (- n 1))))))
```

How many presents did I receive total over the 12 days of Christmas? This can be generalized by asking how many presents I received in total over the first n days. Write a procedure called `presents-through-day` (which may naturally use `presents-on-day`) that computes this as a function of n . Thus, `(presents-through-day 1)` should return 1, `(presents-through-day 2)` should return $1 + 3 = 4$, `(presents-through-day 3)` should return $1 + 3 + 6 = 10$, etc.

► Exercise 2.18

Prove by induction that for every nonnegative integer n the following procedure computes $2n$:

```
(define f
  (lambda (n)
    (if (= n 0)
        0
        (+ 2 (f (- n 1))))))
```

► Exercise 2.19

Prove that for all nonnegative integers n the following procedure computes the value $2^{(2^n)}$:

```
(define foo
  (lambda (n)
    (if (= n 0)
        2
        (expt (foo (- n 1)) 2))))
```

Hint: You will need to use certain laws of exponents, in particular that $(2^a)^b = 2^{ab}$ and $2^a 2^b = 2^{a+b}$.

► Exercise 2.20

Prove that the following procedure computes $n/(n+1)$ for any nonnegative integer n . That is, `(f n)` computes $n/(n+1)$ for any integer $n \geq 0$.

```
(define f
  (lambda (n)
    (if (= n 0)
        0
        (+ (f (- n 1))
            (/ 1 (* n (+ n 1)))))))
```

▶ Exercise 2.21

- a. Appendix A describes the predefined procedure `stack` by saying (among other things) that `(stack image1 image2)` produces an image, the height of which is the sum of the heights of `image1` and `image2`. How would you describe the height of the image that is the value of `(stack-on-itself image)`, given the following definition of `stack-on-itself`?

```
(define stack-on-itself
  (lambda (image)
    (stack image image)))
```

- b. Use induction to prove that given the definition in part a and the following definition of `f`, the value of `(f image n)` is an image 2^n times as high as `image`, provided n is a nonnegative integer.

```
(define f
  (lambda (image n)
    (if (= n 0)
        image
        (stack-on-itself (f image (- n 1))))))
```

▶ Exercise 2.22

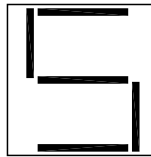
Consider the following procedure:

```
(define foo
  (lambda (n)
    (if (= n 0)
        0
        (+ (foo (- n 1))
            (/ 1 (- (* 4 (square n)) 1))))))
```

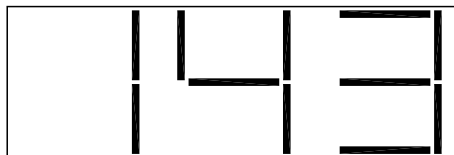

- a. What is the value of (foo 1)? Of (foo 2)? Of (foo 3)?
- b. Prove by induction that for every nonnegative integer n , (foo n) computes $n/(2n + 1)$.

► **Exercise 2.23**

Suppose we have made images for each of the digits 0–9, which we name `zero-bb`, `one-bb`, `...`, `nine-bb`. For example, if you evaluate `five-bb`, you get the following image:



- a. Write a procedure `image-of-digit` that takes a single parameter d that is an integer satisfying $0 \leq d \leq 9$ and returns the image corresponding to d . You should definitely use a `cond`, because you would otherwise have to nest the `ifs` ridiculously deep.
- b. Using the procedure `image-of-digit`, write another procedure `image-of-number` that takes a single parameter n that is a nonnegative integer and returns the image corresponding to it. Thus, `(image-of-number 143)` would return the following image:



Hint: Use the Scheme procedures `quotient` and `remainder` to break n apart. Also, you may use the procedure `side-by-side` from Exercise 1.9b without redefining it here.

Chapter Inventory

Vocabulary

recursion
 permutations
 factorial
 base case (of a procedure)

recursive process
 linear recursion
 mathematical induction
 base case (of a proof)

induction hypothesis	debug
inductive step	bug
termination	one-layer thinking

Slogans

The recursion strategy	The base case imperative
The self-similarity strategy	One-layer thinking maxim

New Predefined Scheme Names

The dagger symbol (†) indicates a name that is not part of the R⁴RS standard for Scheme.

expt	remainder
quotient	invert†

New Scheme Syntax

comments	clauses (of a cond)
cond	else

Scheme Names Defined in This Chapter

factorial	sum-of-powers
power	num-digits
square	stack-copies-of
quot	quilt
multiply	bitw-bb
sum-of-first	checkerboard
subtract-the-first	presents-on-day
factorial2	presents-through-day
sum-integers-from-to	image-of-digit
sum-of-squares	image-of-number
sum-of-cubes	

Sidebars

Exponents

Notes

The Blowing in the Wind pattern is by Rose [43]. The image of mathematical induction in terms of successive days is used very powerfully by Knuth in his fascinating “mathematical novelette” *Surreal Numbers* [32].