Out of print; full text available for free at http://www.gustavus.edu/+max/concrete-abstractions.html

CHAPTER THREE

# Iteration and Invariants

## 3.1 Iteration

In the previous chapter, we used a general problem-solving strategy, namely, recursion: solve a smaller problem first, then do the little bit of work that's left. Now we'll turn to a somewhat different problem-solving strategy, known as *iteration*:

> **The iteration strategy:** By doing a little bit of work first, transform your problem into a smaller one with the same solution. Then solve the resulting smaller problem.

Let's listen in on a hypothetical student thinking aloud as she uses this strategy to devise an alternative `factorial` procedure:

> I've got a factorial problem, like $6 \times 5 \times 4 \times 3 \times 2 \times 1$.
>
> Gee, I wonder if I can transform that into a simpler problem with the same answer? What would make it simpler? Well, the problem I've got is six numbers multiplied together. Five numbers multiplied together would be simpler. I wonder if I can find five numbers that when multiplied together give the same result?
>
> $$6 \times 5 \times 4 \times 3 \times 2 \times 1 = ____ \times ____ \times ____ \times ____ \times ____$$
>
> Well, I can't just put the numbers I've got into the blanks, because I've got more numbers than blanks. (That's the whole point.) Because I have one extra number, maybe I can put two numbers into one blank. I guess I can't really get something for nothing—if I only want to have four multiplications left to do, I better do one of my five now. If I multiply two of the numbers together now and put the product in one of the blanks, that would be a way to get two numbers into one blank. Maybe



Excerpted from Concrete Abstractions; copyright © 1999 by Max Hailperin, Barbara Kaiser, and Karl Knight

I'll multiply the first two together, the 6 and the 5, to get 30. So I have

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 = 30 \times 4 \times 3 \times 2 \times 1$$

That was great, I got the problem down from a five multiplication problem to a four multiplication problem. I bet I could transform it the same way into a three-multiplication problem:

$$\ldots = 120 \times 3 \times 2 \times 1$$

If I keep going like this, one multiplication at a time, eventually it will all boil down to a single number:

$$\ldots = 360 \times 2 \times 1$$

$$\ldots = 720 \times 1$$

$$\ldots = 720$$

I guess I could call that last step a "zero multiplication" problem. And that's the answer to the original problem, because it's equal to all the preceding problems, all the way back to the original factorial one.

Now I want to write a procedure that could solve any problem of this form. What specifics do I have to give it to tell it which problem of this form to solve? Well, I could give it the numbers to multiply together. .... No, that's silly, there could be lots of them. I wonder if there is some more concise description of these problems .... Oh, I see, the numbers after the first are always consecutive, down to 1. So I could describe the problems by saying "30 times 4 down to 1" or "120 times 3 down to 1" or that kind of thing. Oh, in fact the "down to 1" part just means it's a factorial, so I've got problems like "30 times 4!" or "120 times 3!." So what I want is a procedure to multiply some number times some factorial:

```
(define factorial-product
  (lambda (a b) ; compute a * b!
    ))
```

What I did with those products was transform them into smaller ones, like this:

```
(define factorial-product
  (lambda (a b) ; compute a * b! as (a*b) * (b-1)!
    (factorial-product (* a b) (- b 1))))
```

Of course, I have to stop making the factorial part smaller eventually, when it can't get any smaller—let's see, that's when there were zero multiplications left—right after multiplying 720 by 1. Because when I had 720 times 1 that was $720 \times 1!$,

I guess the next step is $720 \times 0!$. I never really thought about 0! before, but it would make sense; the factorial is just some consecutive numbers to multiply together, and here there aren't any, so that's 0!. That means I stop when `b` is 0:

```
(define factorial-product
  (lambda (a b) ; compute a * b!
    (if (= b 0)
        a
        (factorial-product (* a b) (- b 1)))))
```

Now I have a general way of solving problems of the form one number times the factorial of the other number. Wait a second, that wasn't what I really wanted: I really wanted just to do factorials. Hmmm . . . , I could just trick my procedure into doing plain factorials by telling it to multiply by 1, because that doesn't change anything:

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

---

A couple things are worth noticing here. One is that the student changed our original problem of finding a factorial to the more general problem of finding the product of a factorial and another number. Our original problem is then just a special case of this latter problem. This is a good example of what Polya calls "the inventor's paradox" in his excellent book on problem solving, *How to Solve It*. Sometimes, trying to solve a more general or "harder" problem actually makes the original problem easier to solve.

Another point to notice is that the student made use of comments (starting with semicolons) to explain her Scheme program. Her comment identifies what `factorial-product` computes (namely, its first argument times the factorial of its second argument). We'll say more about this comment in a bit; it's an extremely important kind of comment that you should definitely make a habit of using.

**Exercise 3.1**

At the very beginning of the above design of the iterative factorial, a choice needed to be made of which two numbers to multiply together, in order to fit the two of them into one blank. In the version shown above, the decision was made to multiply together the leftmost two numbers (the 6 and the 5). However, it would have been equally possible to make some other choice, such as multiplying together the rightmost two numbers. Redo the design, following this alternative path.

The iterative way of doing factorials may not seem very different from the recursive way. In both cases, the multiplications get done one at a time. In both cases, one multiplication is done explicitly, and the others are implicitly done by the procedure reinvoking itself. The only difference is whether all but one of the multiplications are done first and then the remaining one, or whether one multiplication is done first and then all the rest. However, this is actually an extremely important distinction. It is the difference between putting the main problem on hold while a subproblem is solved versus progressively reducing the problem.

The subproblem approach is less efficient because some of the computer's memory needs to be used to remember what the main problem was while it is doing the subproblem. Because the subproblem itself involves a subsubproblem, and so forth, the recursive approach actually uses more and more memory for remembering what it was doing at each level as it burrows deeper. This was illustrated by the diagram of the recursive factorial process shown in Figure 2.1 on page 26. That diagram had one column for the original problem of evaluating `(factorial 3)`, one for the subproblem of evaluating `(factorial 2)`, and one for the sub-subproblem of evaluating `(factorial 1)`. We remarked that a diagram of the recursive evaluation of `(factorial 52)` would have had 52 columns. The number of columns in these diagrams corresponds to the amount of the computer's memory that is used in the evaluation process.

By contrast, the iterative approach is only ever solving a single problem—the problem just changes into an easier one with the same answer, which becomes the new single problem to solve. Thus, the amount of memory remains fixed, no matter how many reduction steps the iterative process goes through. If we look at the diagram in Figure 3.1 (on page 53) of the iterative process of evaluating `(factorial 3)`, we can see that the computation stays in a single column. (As usual, we've been selective in showing details.) Even if we were to evaluate `(factorial 52)`, we wouldn't need a wider sheet of paper, just a taller one. (The vertical dimension corresponds to time: It would take longer to compute `(factorial 52)`.) The difference between the two types of processes is less clear if we simply list the computational steps than it is from the diagrams, but with a practiced eye you can also see the iterative nature of the process in this more compact form:

```
(factorial 3)
(factorial-product 1 3)
(if (= 3 0) 1 (factorial-product (* 1 3) (- 3 1)))
(factorial-product (* 1 3) (- 3 1))
(factorial-product 3 2)
(if (= 2 0) 3 (factorial-product (* 3 2) (- 2 1)))
(factorial-product (* 3 2) (- 2 1))
(factorial-product 6 1)
(if (= 1 0) 6 (factorial-product (* 6 1) (- 1 1)))
```

```
(factorial-product (* 6 1) (- 1 1))
(factorial-product 6 0)
(if (= 0 0) 6 (factorial-product (* 6 0) (- 0 1)))
6
```

If we work through the analogous computational steps for `(factorial 6)`, but this time leave out some more steps, namely, those involving the `if`s and the arithmetic, the skeleton we're left with mirrors exactly the hypothetical student's calculation of 6! done at the beginning of this chapter:

| | |
|---|---|
| `(factorial 6)` | $6!$ |
| `(factorial-product 1 6)` | $= 1 \times 6!$ |
| `(factorial-product 6 5)` | $= 6 \times 5!$ |
| `(factorial-product 30 4)` | $= 30 \times 4!$ |
| `(factorial-product 120 3)` | $= 120 \times 3!$ |
| `(factorial-product 360 2)` | $= 360 \times 2!$ |
| `(factorial-product 720 1)` | $= 720 \times 1!$ |
| `(factorial-product 720 0)` | $= 720 \times 0!$ |
| `720` | $= 720$ |

To dramatize the reduced memory consumption of iterative processes, take down the paper chain that is decorating your room, disassemble it, and reassemble it using this new process:

- To make a chain of length $n$,
  - **(a)** Bend one strip around and join its ends together.
  - **(b)** Ask yourself to link $n - 1$ more links onto it.
- To link $k$ links onto a chain,
  - **(a)** If $k = 0$, you are done. Hang the chain back up in your room.
  - **(b)** Otherwise,
    - **i.** Slip one strip through an end link of the chain, bend it around, and join the ends together.
    - **ii.** Ask yourself to link $k - 1$ links onto the chain.

Notice the key difference: You are able to do this one alone, in the privacy of your own room, without having to invite a whole bunch of friends over to stand in line. The reason why the recursive process required one person per link is that you had to stand there with a link in your hand and wait for the rest of the crew

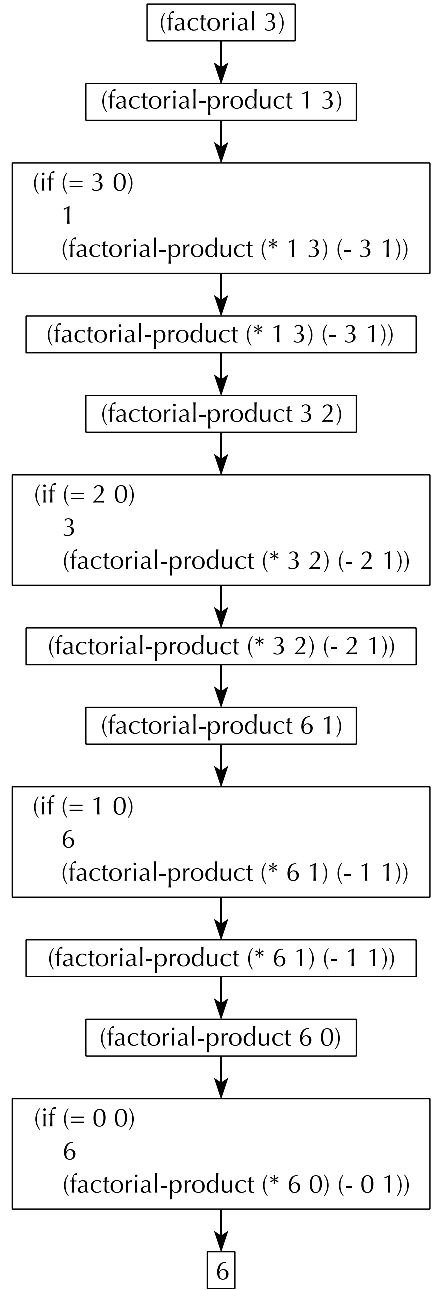


Figure 3.1 The iterative process of evaluating `(factorial 3)`.

to build the chain of length $n - 1$ before you could put your link on. Because the process continued that way, each of your friends in turn wound up having to stand there waiting to put a link on. With the new iterative version, there's no waiting for a subtask to be completed before work can proceed on the main task, so it can all be done singlehandedly (which in a computer would mean with a fixed amount of memory).

Just to confuse everybody, procedures such as the ones we've looked at in this chapter are still called *recursive procedures*, because they invoke themselves. They simply don't generate *recursive processes*. A recursive procedure is any procedure that invokes itself (directly or indirectly). If the self-invocation is to solve a subproblem, where the solution to the subproblem is not the same as the solution to the main problem, the computational process is a recursive process, as in the prior chapter. If, on the other hand, the self-invocation is to solve a reduced version of the original problem (i.e., a simpler version of the problem but with the exact same answer as the original), the process is an iterative process, as in this chapter.

**Exercise 3.2**

Write a new procedure for finding the exponent of 2 in a positive integer, as in Exercise 2.12 on page 40, but this time using an iterative process.

**Exercise 3.3**

You have one last chance to quilt. (In the next chapter we'll do something different, but equally pretty.) Rewrite your procedure for making arbitrary sized quilts so that it generates an iterative process. Do the same for your procedure for checkerboard quilts. As before, it helps to start with the one-dimensional case, that is, an iterative version of `stack-copies-of`.

## 3.2 Using Invariants

Comments such as the one on the `factorial-product` procedure—the one that said what the procedure computed, as a function of the argument values—can be very handy. A comment such as this one is called an *invariant* because it describes a quantity that doesn't change. Every time around the `factorial-product` iteration, $b$ decreases by 1, but $a$ increases by a multiple of the old $b$, so the product $a \times b!$ remains constant. In fact, that's another good way to think about the design of such a procedure: Some parameter keeps moving toward the base case, and some other parameter changes in a compensatory fashion to keep the invariant quantity fixed. In this section, we'll show how invariants can be used to write iterative procedures and how they can be used to prove a procedure is correct.

Let's start with `factorial-product`. Because the procedure is already written, we'll prove that it is correct, that is, that it really does compute $a \times b!$, provided $b$ is a nonnegative integer. (Notice how we're focusing on the invariant.)

**Base case:** If $b = 0$, it follows from the way `if` expressions work that the procedure terminates with $a$ as its value. Because $a \times 0! = a \times 1 = a$, the theorem therefore holds in this base case.

**Induction hypothesis:** We will assume that `(factorial-product` *i k*`)` terminates with value $i \times k!$ provided that $k$ is in the range $0 \leq k < b$.

**Inductive step:** Consider the evaluation of `(factorial-product` *a b*`)`, with $b > 0$. Clearly the procedure will terminate with the same value as the expression `(factorial-product (* a b) (- b 1))`, provided that this recursive call terminates. However, because $0 \leq b - 1 < b$, the induction hypothesis allows us to assume that this call will indeed terminate, with $(a \times b) \times (b - 1)!$ as its value. Because $(a \times b) \times (b - 1)! = a \times (b \times (b - 1)!) = a \times b!$, we see that the procedure does indeed terminate with the correct answer in this case, assuming the induction hypothesis.

**Conclusion:** Therefore, by mathematical induction, the evaluation of `(factorial-product` *a b*`)` will terminate with the value $a \times b!$ for any nonnegative integer $b$ (and any number $a$).

Having shown this formal proof by induction, it's illuminating to look back at the comments the hypothetical student included in the `factorial-product` definition. We already identified the primary comment, that the procedure computes $a \times b!$, as the invariant, which is what the proof is proving. However, note that at a critical moment in designing the procedure the student amplified it to say that the procedure "computes $a \times b!$ as $(a \times b) \times (b - 1)!$." This can now be recognized as a simplified version of the inductive step. Proof by induction should be used this way: first as comments written while you are designing the procedure, that give a bare outline of the most important ingredients of the proof. Later, if you need to, you can flesh out the full proof. Of course, leaving the comments in can be helpful to a reader who needs the same points made explicit as you did.

Next we'll look at writing an iterative version of the `power` procedure from Exercise 2.1 on page 28. Finding a power involves doing many multiplications, so it is somewhat similar to `factorial`. For that reason, we'll define `power` and `power-product` analogously to the way we did with `factorial` and we'll use a similar invariant:

```
(define power-product
  (lambda (a b e)    ; returns a times b to the e power
    (if (= e 0)
        ________
        (power-product ________ ________ ________))))

(define power
  (lambda (b e)
    (power-product 1 b e)))
```

If we imagine trying to prove this is correct using induction, filling in the first blank is connected with the base case of the induction proof. Because we're trying to prove that $a \times b^0$ is returned in this case, and because $b^0 = 1$, we should fill in that first blank with `a`.

How should we fill in the remaining three blanks? Think about an induction proof. In order for the induction hypothesis to apply, we've got to fill in the last blank with something that is a nonnegative integer and is strictly less than $e$. We know that $e$ is a positive integer (it was originally only guaranteed to be nonnegative, but we just handled the case of $e = 0$ ). This means that $e - 1$ is a nonnegative integer, and it is of course less than $e$. Therefore, we should probably put $e - 1$ in the last blank. The base goes in the next to last blank. Because we're trying to multiply $e$ copies of $b$ together, the base should probably remain unchanged as $b$. Thus we are left with what to fill in as the first parameter of the recursive call to `power-product`. Our invariant comes in handy here. Suppose we put in some expression, say $x$, here. According to our invariant (and our induction hypothesis), this call to `power-product` will return $x \cdot b^{e-1}$. On the other hand, this is also the value that gets returned from the whole procedure when $e > 0$. But the invariant says that this value should be $a \cdot b^e$. Thus, we can set up an equation and solve it for $x$:

$$
\begin{aligned}
x \cdot b^{e-1} &= a \cdot b^e \\
x &= \frac{a \cdot b^e}{b^{e-1}} \\
x &= a \cdot b
\end{aligned}
$$

Putting this all together gives us:

```
(define power-product
  (lambda (a b e)    ; returns a times b to the e power
    (if (= e 0)
        a
        (power-product (* a b) b (- e 1)))))
```

**Exercise 3.4**

Give a formal induction proof that `power-product` is correct.

**Exercise 3.5**

If when you did Exercise 3.2, you didn't write down the invariant for your iterating procedure, do so now. Next, use induction to prove that your procedure does in fact compute this invariant quantity.

In our final example, we'll write a procedure that a sixteenth-century mathematician, Pierre Fermat, thought would produce prime numbers. A *prime number* is a positive integer with exactly two positive divisors, 1 and itself. Fermat thought that all numbers produced by squaring 2 any number of times and then adding 1 would be prime. Certainly, the numbers $2 + 1 = 3$ (in which 2 isn't squared at all), $2^2 + 1 = 5$, $(2^2)^2 + 1 = 17$, $((2^2)^2)^2 + 1 = 257$, and $(((2^2)^2)^2)^2 + 1 = 65{,}537$ are prime numbers (although checking 65,537 does take some effort). We call these the zeroth through fourth *Fermat numbers*, corresponding to zero through four squarings. Unfortunately, the fifth Fermat number, 4,294,967,297, is not a prime, because it equals $641 \times 6{,}700{,}417$. In fact, the only Fermat numbers known to be prime are the zeroth through fourth. Many Fermat numbers are known to be composite (i.e., not prime); the largest of these is the 23,471st Fermat number. On the other hand, no one knows whether the twenty-fourth Fermat number is prime or composite. (This is the smallest Fermat number for which the primality is unknown.)

We can translate our definition of Fermat numbers into Scheme:

```
(define fermat-number      ; computes the nth Fermat number
  (lambda (n)
    (+ (repeatedly-square 2 n) 1)))
```

Most of the work is done in `repeatedly-square`, which we can outline as follows:

```
(define repeatedly-square  ; computes b squared n times, where
  (lambda (b n)            ;  n is a nonnegative integer
    (if (= n 0)
        b     ;not squared at all
        (repeatedly-square ____________ ____________))))
```

How do we fill in the blanks? Again, to be able to apply the induction hypothesis, we've got to fill in the second blank with something that is a nonnegative integer and is strictly less than $n$. As before, we'll try $n - 1$. This brings us to

```
(define repeatedly-square  ; computes b squared n times, where
  (lambda (b n)            ;  n is a nonnegative integer
    (if (= n 0)
        b     ; not squared at all
        (repeatedly-square ____________ (- n 1)))))
```

Now, whatever we fill in the remaining blank, we know from the induction hypothesis that it will be squared $n-1$ times. We don't need to think about *how* it will be squared $n-1$ times; that's what makes this one-layer thinking. Now the question is, What should be squared $n-1$ times to produce the desired result, $b$ squared $n$ times? The answer is $b^2$; that is, if we square $b$ once and then $n-1$ more times, it will have been squared $n$ times in all. This leads to

```
(define repeatedly-square  ; computes b squared n times, where
  (lambda (b n)            ;  n is a nonnegative integer
    (if (= n 0)
        b     ;not squared at all
        (repeatedly-square (square b) (- n 1)))))
```

We explicitly concern ourselves only with squaring $b$ the first time and trust based on the induction hypothesis that it will be squared the remaining $n-1$ times.

## 3.3 Perfect Numbers, Internal Definitions, and Let

Having seen how iteration works, let's work through an extended example using iteration, both to solidify our understanding and also to provide opportunity for learning a few more helpful features of Scheme.

A number is called *perfect* if the sum of its divisors is twice the number. (Equivalently, a number is perfect if it is equal to the sum of its divisors other than itself.) Although this is a simple definition, lots of interesting questions concerning perfect numbers remain unanswered to date; for example, no one knows whether there are any odd perfect numbers. In this section, we'll use the computer to search for perfect numbers.

A good starting point might be to write a simple `perfect?` predicate, leaving all the hard part for `sum-of-divisors`:

```
(define perfect?
  (lambda (n)
    (= (sum-of-divisors n) (* 2 n))))
```

The simplest way to compute the sum of the divisors of $n$ would be to check each number from 1 to $n$, adding it into a running sum if it divides $n$. This computation

sounds like an iterative process; as we check each number, the range left to check gets smaller, and thus transforms the problem into a smaller one. The running sum changes in a compensatory fashion: Any divisor no longer included in the range to check is instead included in the running sum. The invariant quantity is the sum of the divisors still in the range plus the running sum. The following definition is based on these ideas. Note that `divides?` needs to be written.

```
(define sum-of-divisors
  (lambda (n)
    (define sum-from-plus ; sum of all divisors of n which are
      (lambda (low addend) ; >= low, plus addend
        (if (> low n)
            addend    ; no divisors of n are greater than n
            (sum-from-plus (+ low 1)
                           (if (divides? low n)
                               (+ addend low)
                               addend)))))
    (sum-from-plus 1 0)))
```

The preceding definition illustrates a useful feature of Scheme: It is possible to nest a definition inside a lambda expression, at the beginning of the body. This nesting achieves two results:

- The internally defined name is private to the body in which it appears. This means that we can't invoke `sum-from-plus` directly but rather only by using `sum-of-divisors`. It also means that we're able to use a relatively nondescriptive name (it doesn't specify what it is summing) without fear that we might accidentally give two procedures the same name. As long as the two definitions in question are internal to separate bodies, the same name can be used without problem.
- The `sum-from-plus` procedure is able to make use of `n`, without needing to have it passed as a third argument. This is because a nested procedure can make use of names from the procedure it is nested inside of (or from yet further out, in the case of repeated nesting).

Why didn't we nest `sum-of-divisors` itself inside of the `perfect?` procedure? Although we wrote `sum-of-divisors` for the sake of `perfect?`, it could very well be useful on its own, for other purposes. This is in contrast to `sum-from-plus`, which is hard to imagine as a generally useful procedure rather than merely a means to implement `sum-of-divisors`.

The only detail remaining before we have a working `perfect?` test is the predicate `divides?`. We can implement it using the primitive procedure `remainder`:

```
(define divides?
  (lambda (a b)
    (= (remainder b a) 0)))
```

**Exercise 3.6**

Although the method we use for computing the sum of the divisors is straightforward, it isn't particularly efficient. Any time we find a divisor $d$ of $n$, we can infer that $n/d$ is also a divisor. In particular, all the divisors greater than the square root of $n$ can be inferred from the divisors less than the square root. Make use of this observation to write a more efficient version of `sum-of-divisors` that stops once $low^2 \geq n$. Remember that if $low^2 = n$, $low$ and $n/low$ are the same divisor, not two different ones.

If you start testing numbers for perfectness by trying them out one by one with `perfect?`, you'll quickly grow bored: It seems almost nothing is perfect. Because perfect numbers are so few and far between, we should probably automate the search. The following procedure finds the first perfect number after its argument value:

```
(define first-perfect-after
  (lambda (n)
    (if (perfect? (+ n 1))
        (+ n 1)
        (first-perfect-after (+ n 1)))))
```

Having this start searching with the first number *after* its argument is convenient for using it to search for consecutive perfect numbers, like this:

```
(first-perfect-after 0)
6

(first-perfect-after 6)
28

(first-perfect-after 28)
496
```

Because the search starts after the number we specify, we can specify each time the perfect number we just found, and it will find the next. Unfortunately, starting with the next number causes us to use three copies of the expression `(+ n 1)`, which is a bit ugly.

Rather than put up with this, or changing how the procedure is used, we can make use of another handy Scheme feature, namely, `let`:

```
(define first-perfect-after
  (lambda (n)
    (let ((next (+ n 1)))
      (if (perfect? next)
          next
          (first-perfect-after next)))))
```

What this means is to first evaluate `(+ n 1)` and then locally let `next` be a name for that value while evaluating the body of the `let`. Not only does this make the code easier to read, it also means that `(+ n 1)` only gets evaluated once. There are two sets of parentheses around `next` and `(+ n 1)` because you can have multiple name/expression pairs. One set of parentheses goes around each name and its corresponding value expression, and another set of parentheses goes around the whole list of pairs. For example,

```
(define distance
  (lambda (x0 y0 x1 y1)
    (let ((xdiff (- x0 x1))
          (ydiff (- y0 y1)))
      (sqrt (+ (* xdiff xdiff)
               (* ydiff ydiff))))))
```

All the value expressions are evaluated before any of the new names are put into place. Those new names may then be used only in the body of the let. Note that a let expression is just like any other expression; in particular, you can use it anywhere you'd use an expression, not just as the body of a lambda expression.

## 3.4 Iterative Improvement: Approximating the Golden Ratio

One important kind of iterative process is the *iterative improvement* of an approximation to some quantity. We start with a crude approximation, successively improve it to better and better approximations, and stop when we have found one that is good enough. Recall that our general definition of an iterative process is that it works by successively transforming the problem into a simpler problem with the same answer. Here the original problem is to get from a crude approximation to one that is good enough. This problem is transformed into the simpler problem of getting from a somewhat less crude approximation to one that is good enough. In other words, our goal is still to get to the good enough approximation, but we move the starting point one improvement step closer to that goal. Our general outline, then, is

```
(define find-approximation-from
  (lambda (starting-point)
    (if (good-enough? starting-point)
        starting-point
        (find-approximation-from (improve starting-point)))))
```

In this section, we'll follow this general outline in order to develop one specific example of an iterative improvement procedure.

Since ancient times, many artists have considered that the most aesthetically pleasing proportion for a work of art has a ratio of the long side to the short side that is the same as the ratio of the sum of the sides to the long side, as illustrated in Figure 3.2. This ratio is called the *golden ratio*.

Among its many interesting properties (which range from pure mathematics to aesthetics and biology), the golden ratio is irrational, that is, it is not equal to any ratio of integers. Real artists, however, are generally satisfied with close approximations. For example, when we drew the illustration in Figure 3.2, we made it 377 points wide and 233 points high. (The point is a traditional printer's unit of distance.) The ratio 377/233 isn't exactly the golden ratio, but it is a quite good approximation: It's off by less than 1/50,000. How do we know that? Or more to the point, how did we

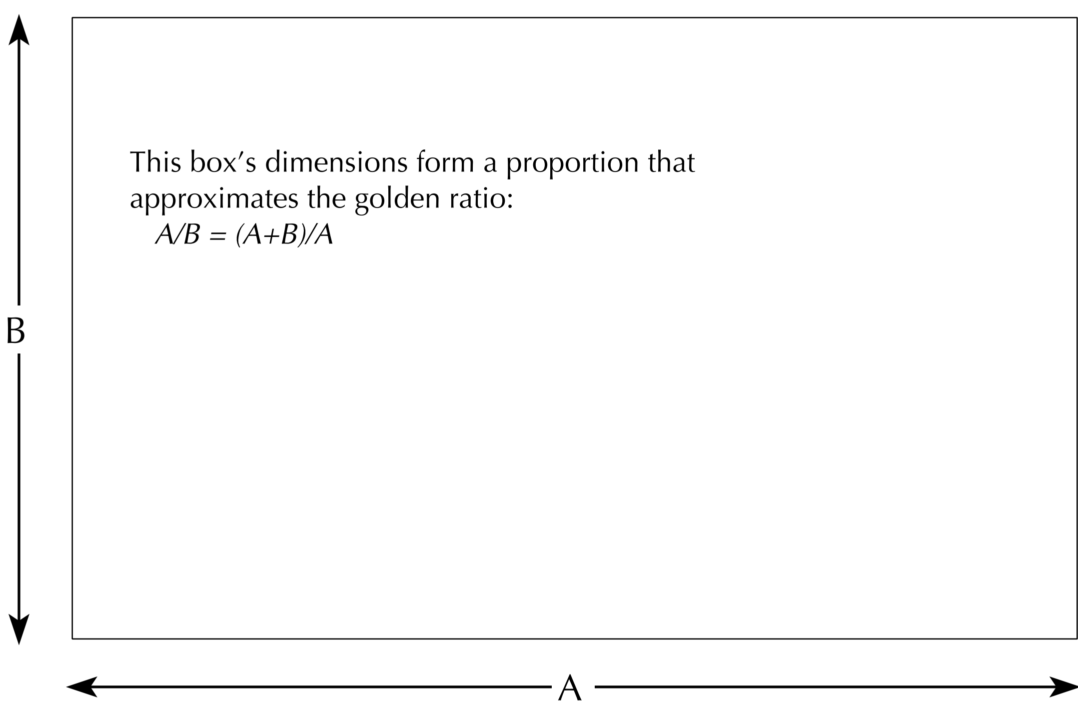


Figure 3.2 An illustration of the golden ratio, said to be the most pleasing proportion.

set about finding a ratio of integers that was that close? That's what we're about to embark on. Our goal is to write a procedure that, given a maximum tolerable error, will produce a rational approximation that is at least that close to the golden ratio. In other words, by the time we're done, you'll be able to get the above answer in the following way:

```
(approximate-golden-ratio 1/50000)
377/233
```

Recall that the definition of the golden ratio is that it is the ratio $A/B$ such that $A/B = (A + B)/A$. Doing a little algebra, it follows that $A/B = 1 + B/A = 1 + 1/(A/B)$. In other words, if we take the golden ratio, divide it into 1, and then add 1, we'll get the golden ratio back again. For brevity, let's start calling the golden ratio not $A/B$ but instead $\phi$, the Greek letter phi, which is in honor of the sculptor Phidias, who is known to have consciously used the golden ratio in his work. This makes our equation

$$\phi = 1 + \frac{1}{\phi}$$

Because this is an equation, we can substitute the right-hand side for $\phi$ anywhere it occurs. In particular, we can substitute it for the $\phi$ on the right-hand side of the same equation:

$$\phi = 1 + \frac{1}{1 + \frac{1}{\phi}}$$

We could keep doing this over and over again, and we would get the infinite *continued fraction* for $\phi$:

$$\phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\ddots}}}$$

It turns out that this continued fraction is the key to finding rational approximations to $\phi$. All we have to do is calculate some finite part of that infinite tower. In particular, the following are better and better approximations of $\phi$:

$$\phi_0 = 1$$

$$\phi_1 = 1 + \frac{1}{\phi_0}$$

$$\phi_2 = 1 + \frac{1}{\phi_1}$$

$$\phi_3 = 1 + \frac{1}{\phi_2}$$

$$\vdots$$

**Exercise 3.7**

Using this technique, write a procedure, `improve`, that takes one of the approximations of $\phi$ and returns then next one. For example, given $\phi_2$, it would return $\phi_3$.

The only remaining problem is to figure out how good each of these approximations is, so we know when we've got a good enough one and can stop. Using some number theory, it is possible to show that the error of each approximation is less than 1 over the square of its denominator. So, for example, it follows that 377/233 is within $1/233^2$ of $\phi$. We can stop when this is less than our acceptable error, or *tolerance* as it is called. We'll do this by setting up the overall `approximate-golden-ratio` procedure as follows:

```
(define approximate-golden-ratio
  (lambda (tolerance)
    (define find-approximation-from
      (lambda (starting-point)
        (if (good-enough? starting-point)
            starting-point
            (find-approximation-from (improve starting-point)))))
    (define good-enough?
      (lambda (approximation)
        (< (/ 1 (square (denominator approximation)))
           tolerance)))
    (find-approximation-from 1)))
```

The Scheme procedure `denominator` is used here, which returns the denominator of a rational number. (To be precise, it computes the denominator the number has when written in lowest terms; the denominator is always positive, even when the rational number is negative.)

**Exercise 3.8**

Presumably any art work needs to be made out of something, and there are only about $10^{79}$ electrons, neutrons, and protons in the universe. Therefore, we can conservatively assume that no artist will ever need to know $\phi$ to better than one part in $10^{79}$. Calculate an approximation that is within a tolerance of $1/10^{79}$, which can also be expressed as $10^{-79}$. (To calculate this tolerance in Scheme, you could use the `expt` procedure, as in `(/ 1 (expt 10 79))` or `(expt 10 -79)`.)

## 3.5 An Application: The Josephus Problem

In the fast-paced world of computing, about the most damning comment you can make regarding the relevance of a technique is to say that it is of purely historical interest. At the other extreme of the relevance spectrum, you could say that something is a matter of life or death. In this section, we'll see an application of iterative processes that is quite literally both a matter of life or death *and* of purely historical interest—because we'll be deciding the life or death fate of people who lived in Galilee nearly 2000 years ago. Our real goal is to provide you with a memorable illustration that iterative processes operate by progressively reducing a problem to a smaller problem with the same answer: in this case, the problem will be made smaller by the drastic means of killing someone.

Josephus was a Jewish general who was in the city of Jotapata, in Galilee, when it fell after a brutal 47-day siege by the Roman army under Vespasian, in 67 CE. The Romans massacred the inhabitants of Jotapata, but Josephus initially evaded capture by hiding (by day) in a cavern. Forty other "persons of distinction" were already hiding in that cavern. One of these nameless other people was captured while out and about and revealed the location where Josephus and the others still hid. The Romans sent word that Josephus was to be captured alive, rather than killed. Josephus himself was all for this and ready to go over to the Romans. However, the others with him advocated mass suicide as preferable to enslavement by the Romans. They were sufficiently angered by Josephus's preference for surrender that he was barely able to keep them from killing him themselves. In order to satisfy them, Josephus orchestrated a scheme whereby they all drew lots (Josephus among them) to determine their order of death and then proceeded to kill themselves, with the second killing the first, the third the second, etc. However, Josephus managed to be one of the last two left and convinced the other who was left with him that they should surrender together.

How did Josephus wind up being one of the two who survived? The Greek version of Josephus's account attributes it to fortune or the providence of God. However, the Slavonic version (which shows some signs of originating from an earlier manuscript than the Greek) has a more interesting story: "He counted the numbers cunningly, and so deceived them all." The Slavonic version also doesn't specifically mention the drawing of lots, instead leaving it open exactly how the order was determined in which the cornered Jews killed one another. Thus, we have a tantalizing suggestion that Josephus used his mathematical ability to arrange what appeared to be a chance ordering, but in fact was rigged so that he would be one of the last two.

Out of this historical enigma has come a well-known mathematical puzzle. Suppose this is how Josephus's group did their self-killing: They stood in a circle and killed every third person, going around the circle. It is fairly clear who will get killed early on: the third, sixth, ninth, etc. However, once the process wraps around the circle, the situation is much less clear, because it will be every third still-surviving

person who will be killed, skipping over those who are already dead from the previous round. Can you determine which people will live and which will die?

Our goal is to write a procedure that will determine the fate of a person, given his or her position in the circle and the total number of people in the circle. Rather than using any advanced mathematical ideas, we'll simply simulate the killing process, stopping when the position we are interested in is either killed or is one of the last two who are left.

Let's call the number of people in the circle $n$ and number the positions from 1 to $n$. We'll assume that the killing of every third person starts with killing the person in position number 3. That is, we start by skipping over 1 and 2 and killing 3. We want to write a procedure, `survives?`, that takes as its arguments the position number and $n$ and returns `#t` if the person in that position is one of the last two survivors; otherwise it returns `#f`. For example, we've already figured out that position 3 doesn't survive:

```
(survives? 3 40)
#f
```

Recall that Josephus called the killing off when he was left with only one other; thus we will say that if there are fewer than three people left, everybody remaining is a survivor:

```
(define survives?
  (lambda (position n)
    (if (< n 3)
        #t
        we still need to write this part)))
```

On the other hand, if there are three or more people left, we still have some killing left to do. As we saw above, if the person we care about is in position 3, that person is the one killed and hence definitely not a survivor:

```
(define survives?
  (lambda (position n)
    (if (< n 3)
        #t
        (if (= position 3)
            #f
            we still need to write this part))))
```

Suppose we aren't interested in the person in position number 3 but rather in some other person—let's say J. Doe. The person in position number 3 got killed, so

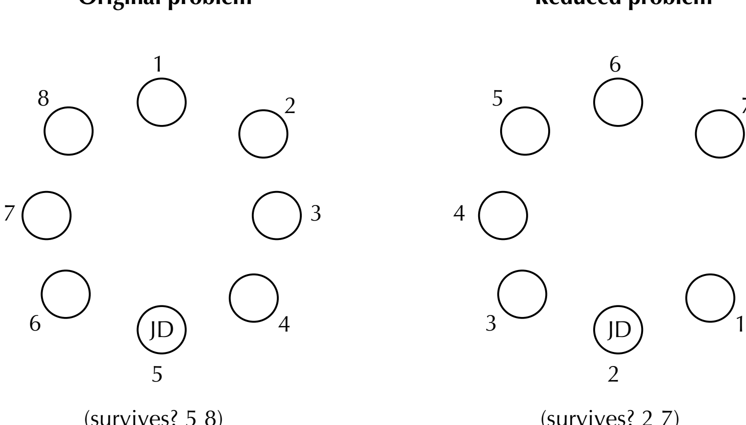


Figure 3.3 Determining the fate of J. Doe, who is initially in position 5 out of 8, can be reduced to finding the fate of position 2 out of 7.

now we only have $n-1$ people left. Of that smaller group of $n-1$, there will still be two survivors, and we still want to know if J. Doe is one of them. In other words, we have reduced our original problem (Is J. Doe among the survivors from this group of $n$?) to a smaller problem (Is J. Doe among the survivors from this group of $n-1$?).

We can solve this smaller problem by using `survives?` again. However, the `survives?` procedure assumes that the positions are numbered so that we start by skipping over positions 1 and 2 and killing the person in position 3. Yet we don't really want to start back at position 1—we want to keep going from where we left off, skipping over 4 and 5 and killing 6. To solve this problem, we can renumber all the survivors' positions. The survivor who was in position 4 (just after the first victim) will get renumbered to be in position 1, because he is now the first to be skipped over. The survivor who was in position 5 gets renumbered to position 2, etc. For example, suppose we are interested in the fate of a specific person, let's say J. Doe, who is in position 5 out of a group of 40 people. Then we are initially interested in `(survives? 5 40)`. Neither of the base cases applies, because $40 \geq 3$ and $5 \neq 3$. Therefore, we reduce the problem size by 1 (by killing off one of J. Doe's companions) and ask `(survives? 2 39)`. The answer to this question will be the same as the answer to our original question `(survives? 5 40)`, because J. Doe is now in position number 2 in our new renumbered circle of 39 people. Figure 3.3 illustrates this, but rather than going from 40 to 39 people, it goes from 8 to 7.

**Exercise 3.9**

How about the people who were in positions 1 and 2; what position numbers are they in after the renumbering?

**Exercise 3.10**

Write a procedure for doing the renumbering. It should take two arguments: the old position number and the old number of people ($n$). (It can assume that the old position number won't ever be 3, because that person is killed and hence doesn't get renumbered.) It should return the new position number.

**Exercise 3.11**

Finish writing the `survives?` procedure, and carefully test it with a number of cases that are small enough for you to check by hand but that still cover an interesting range of situations.

**Exercise 3.12**

Write a procedure, analogous to `first-perfect-after`, that can be used to systematically search for surviving positions. What are the two surviving positions starting from a circle of 40 people? (Presumably Josephus chose one of these two positions.)

Now that you have settled the urgent question of where Josephus should stand, we can take some time to point out additional features of the Scheme programming language that would have simplified the procedure a little bit. You may recall that the overall form of the procedure involved two `if` expressions. The outer one checked to see if the killing was over; if it was, then the person we cared about was definitely a survivor, so the answer was `#t`. The inner `if` took care of the case where more killing was still needed. If our person of interest was in position number 3, and so was the next to go, the answer was `#f`. This succession of tests can be reformulated in a different way. Our person of interest is a survivor if the killing is over (i.e., $n < 3$) *or* we are *not* interested in position number 3 *and* the person survives in the reduced circle of $n - 1$ people. Writing this in Scheme, we get

```
(define survives?
  (lambda (position n)
    (or (< n 3)
        (and (not (= position 3))
             your part with n − 1 people goes here))))
```

This procedure illustrates three new features of Scheme: `or`, `and`, and `not`. Of these, `not` is just an ordinary procedure, which we could write ourselves, although it happens to be predefined. If its argument is `#f`, it returns `#t`; otherwise it returns `#f`. That way, it returns the true/false opposite of its argument. The other two logical

operations, `or` and `and`, are not procedures. They are special language constructs like `if`. In fact, you can see their close relationship to `if` by comparing our new version of `survives?` with our old one. In particular, if $n < 3$, the computation is immediately over, with the answer of `#t`; the second part of the `or` is not evaluated. Similarly, if we don't have $n < 3$, but the position is equal to 3, the computation is immediately over with the answer of `#f`; the second part of the `and` is not evaluated.

The official definitions of `or` and `and` are made a bit more complicated by two factors:

- There needn't be just two expressions in an `or` or `and`. There can be more than two or even none or one.
- Any value other than `#f` counts as true in Scheme, not just `#t`. Thus we need to be careful about which true value gets returned.

The resolution to these issues is as follows. The expressions listed inside an `or` get evaluated one by one in order. As soon as one that produces a true value is found, that specific true value is returned as the value of the `or` expression. If none is found, the false value produced by the last expression is returned. If there are no expressions at all, `#f` is immediately returned. Similarly for `and`, the expressions are evaluated one by one in order. As soon as one that produces a false value is found, that false value is returned as the value of the `and` expression. If none is found, the specific true value produced by the last expression is returned. If there are no expressions in the `and`, `#t` is returned as the value.

## Review Problems

### Exercise 3.13

In Exercises 2.12 and 3.2 you saw that any positive integer $n$ can be expressed as $2^j k$ where $k$ is odd, and you wrote a procedure to compute $j$, the exponent of 2. The following procedure instead computes $k$, the odd factor (which is the largest odd divisor of $n$). Does it generate a recursive process or an iterative process? Justify your answer.

```
(define largest-odd-divisor
  (lambda (n)
    (if (odd? n)
        n
        (largest-odd-divisor (/ n 2)))))
```

### Exercise 3.14

Here is a procedure that finds the largest number $k$ such that $b^k \leq n$, assuming that $n$ and $b$ are integers such that $n \geq 1$ and $b \geq 2$. For example, `(closest-power 2 23)` returns 4:

```
(define closest-power
  (lambda (b n)
    (if (< n b)
        0
        (+ 1 (closest-power b (quotient n b))))))
```

**a.** Explain why this procedure generates a recursive process.

**b.** Write a version of `closest-power` that generates an iterative process.

### Exercise 3.15

Consider the following two procedures:

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

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

**a.** Use the substitution model to evaluate each of `(f 1)`, `(f 2)`, and `(f 3)`.

**b.** Can you predict `(f 4)`? `(f 5)`? In general, which arguments cause `f` to return 0 and which cause it to return 1? (You need only consider nonnegative integers.)

**c.** Is the process generated by `f` iterative or recursive? Explain.

### Exercise 3.16

Consider the following two procedures:

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

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

**a.** Use the substitution model to illustrate the evaluation of `(f 2)`, `(f 3)`, and `(f 4)`.

**b.** Is the process generated by `f` iterative or recursive? Explain.

**c.** Predict the values of `(f 5)` and `(f 6)`.

### Exercise 3.17

*Falling factorial powers* are similar to normal powers and also similar to factorials. We write them as $n^{\underline{k}}$ and say "$n$ to the $k$ falling." This means that $k$ consecutive numbers should be multiplied together, starting with $n$ and working downward. For example, $7^{\underline{3}} = 7 \times 6 \times 5$ (i.e., three consecutive numbers from 7 downward multiplied together).

Write a procedure for calculating falling factorial powers that generates an iterative process.

### Exercise 3.18

We've already seen how to raise a number to an integer power, provided that the exponent isn't negative. We could extend this to allow negative exponents as well by using the following definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b^{n-1} \times b & \text{if } n > 0 \\ b^{n+1}/b & \text{if } n < 0 \end{cases}$$

**a.** Using this idea, write a procedure `power` such that `(power` $b$ $n$`)` raises $b$ to the $n$ power for any integer $n$.

**b.** Use the substitution model to show how `(power 2 -3)` would be evaluated. (You can leave out steps that just determine which branch of a `cond` or `if` should be taken.) Does your procedure generate a recursive process or an iterative one?

**Exercise 3.19**

Prove that, for all nonnegative integers $n$ and numbers $a$, the following procedure computes the value $2^n \times a$:

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

**Exercise 3.20**

Consider the following two procedures:

```
(define factorial
  (lambda (n)
    (product 1 n)))

(define product
  (lambda (low high)
    (if (> low high)
        1
        (* low
           (product (+ low 1) high)))))
```

**a.** Use the substitution model to illustrate the evaluation of `(factorial 4)`.

**b.** Is the process generated by `factorial` iterative or recursive? Explain.

**c.** Describe exactly what `product` computes in terms of its parameters.

## Chapter Inventory

### Vocabulary

iteration
the inventor's paradox
recursive procedure
invariant
prime number
Fermat number
perfect number
iterative improvement
golden ratio
continued fraction
tolerance
falling factorial powers

### Slogans

The iteration strategy

### New Predefined Scheme Names

`denominator`
`not`

### New Scheme Syntax

| | |
|---|---|
| nested (or internal) definitions | `or` |
| `let` | `and` |

### Scheme Names Defined in This Chapter

| | |
|---|---|
| `factorial` | `divides?` |
| `factorial-product` | `first-perfect-after` |
| `stack-copies-of` | `distance` |
| `power` | `approximate-golden-ratio` |
| `power-product` | `improve` |
| `fermat-number` | `survives?` |
| `repeatedly-square` | `largest-odd-divisor` |
| `perfect?` | `closest-power` |
| `sum-of-divisors` | `product` |

## Notes

The way we have used the term *invariant* is superficially different from the way most other authors use it, but the notions are closely related. Most authors use *invariant assertions* rather than *invariant quantities*. That is, they focus not on a numerical quantity that remains constant but rather on a logical assertion such as $a = (b + 1) \times (b + 2) \times \cdots \times n$, the truth of which remains unchanged from one iteration to the next. The other difference is that most authors focus on what the computation has already accomplished rather than on what it is going to compute. So, although we say that `factorial-product` will compute $a \times b!$, others say that when `factorial-product` is entered, it is already the case that $a = (b + 1) \times (b + 2) \times \cdots \times n$. The relationship between these two becomes clear when we recognize that `factorial-product` is ultimately being used to compute $n!$. This gives the equation $n! = a \times b!$, which is equivalent to $a = (b+1) \times (b+2) \times \cdots \times n$.

Polya's *How to Solve It* introduced the phrase “inventor's paradox” for the idea that some problems can be made easier to solve by generalizing them [40]. Our information regarding which Fermat numbers are known to be prime or composite is from Ribenboim's *The New Book of Prime Number Records* [41]. For information on Fermat numbers, perfect numbers, continued fractions, and rational approximations,

any good textbook on number theory should do. The classic is Hardy and Wright [25]. Perhaps the most accessible source of golden-ratio trivia is in Martin Gardner's second collection of mathematical recreations [22]. Our source for the number of subatomic particles in the universe is Davis's *The Lore of Large Numbers* [15]. For the story of Josephus, see his *Jewish Wars*, Book III, for example in the translation of H. St. J. Thackeray [28] or G. A. Williamson [29]; both these translations have appendixes pointing out the relevant deviation in the Slavonic version.