

CHAPTER FIVE

Higher-Order Procedures

5.1 Procedural Parameters

In the earlier chapters, we twice learned how to stop writing lots of specific expressions that differed only in details and instead to write one general expression that captured the commonality:

- In Chapter 1, we learned how to define procedures. That way when we had several expressions that differed only in the specific values being operated on, such as `(* 3 3)`, `(* 4 4)`, and `(* 5 5)`, we could instead define a general procedure:

```
(define square
  (lambda (x)
    (* x x)))
```

This one procedure can be used to do all of the specific calculations just listed; the procedure specifies what operations to do, and the parameter allows us to vary which value is being operated on.

- In Chapter 2, we learned how to generate variable-size computational processes. That way if we had several procedures that generated processes of the same form, but differing in size, such as `(define square (lambda (x) (* x x)))` and `(define cube (lambda (x) (* (* x x) x)))`, we could instead define a general procedure:


```
(define power
  (lambda (b e)
    (if (= e 1)
        b
        (* (power b (- e 1)) b))))
```

This one procedure can be used in place of the more specific procedures listed previously; the procedure still specifies what operations to do, but the parameters now specify how many of these operations to do as well as what values to do them to.

Since learning about these two kinds of variability—variability of values and of computation size—we’ve concentrated on other issues, such as the amount of time and memory that a process consumes. In this chapter, we will learn about a third kind of variability, which, once again, will allow us to replace multiple specific definitions with a single more general one.

Suppose that you replace the operation name `*` with `stack` in the previous definition of `power`. By making this one change, you’ll have a procedure for stacking multiple copies of an image instead of doing exponentiation. That is, the general structure of the two procedures is the same; the only difference is the specific operation being used. (Of course, it would make the procedure easier to understand if you also made some cosmetic changes, such as changing the name from `power` to `stack-copies-of` and changing the name and order of the parameters. If you do this, you’ll probably wind up with the exact same procedure you wrote for Exercise 2.13 on page 40.)

This commonality of structure raises an interesting question: Can we write one general purpose procedure for all computations of this kind and then tell it not only how many copies we want of what but also how they should be combined? If so, we could ask it to stack together 3 copies of rcross-bb, to multiply together 5 copies of 2, or We might use it like this:

```
(together-copies-of stack 3 rcross-bb) ⇒ 
(together-copies-of * 5 2)
32
```

The first argument is a procedure, which is how we specify the kind of combining we want done. The names `stack` and `*` are evaluated, just like the name `rcross-bb` is or any other expression would be. Therefore, the actual argument value is the procedure itself, not the name.

To start writing the procedure `together-copies-of`, we give a name for its procedural parameter in the parameter list, along with the other parameters:

```
(define together-copies-of
  (lambda (combine quantity thing)
```

Here we have three parameters, called `combine`, `quantity`, and `thing`, filling in the blanks in “combine together quantity copies of thing.” We chose to use a verb for the procedural parameter and nouns for the other parameters to remind ourselves how

they are used. Now we can finish writing the procedure, using the parameter names in the body wherever we want to have the specifics substituted in. For example, when we want to check whether the specific quantity requested is 1, we write `(= quantity 1)`. Similarly, when we want to use the specific combining operation that was requested, we write `(combine)`. Here is the resulting procedure:

```
(define together-copies-of
  (lambda (combine quantity thing)
    (if (= quantity 1)
        thing
        (combine (together-copies-of combine
                                   (- quantity 1)
                                   thing)
                 thing))))
```

Once we've got this general purpose procedure, we can use it to simplify the definition of other procedures:

```
(define stack-copies-of
  (lambda (quantity image)
    (together-copies-of stack quantity image)))

(define power
  (lambda (base exponent)
    (together-copies-of * exponent base)))

(define mod-expt
  (lambda (base exponent modulus)
    (together-copies-of (lambda (x y)
                          (remainder (* x y) modulus))
                        exponent base)))
```

(Notice that we didn't bother giving a name, such as `mod*`, to the combining procedure used in `mod-expt`. Typically, using a lambda expression to supply the procedural argument directly is easier than stopping to give it a name with `define` and then referring to it by name.)

`Together-copies-of` is an example of a *higher-order* procedure. Such procedures have procedural parameters or (as we'll see later) return procedural values. One great benefit of building a higher-order procedure is that the client procedures such as `stack-copies-of` and `mod-expt` are now completely independent of the process used for combining copies. All they say is that so many copies of such and such should be combined with this combiner, without saying how that combining should be orga-

nized. This means that we can improve the technique used by `together-copies-of` and in one fell swoop the performance of `stack-copies-of`, `mod-expt`, and any other client procedures will all be improved.

▶ Exercise 5.1

Write a linear iterative version of `together-copies-of`.

▶ Exercise 5.2

Write a logarithmic-time version of `together-copies-of`. You may assume that the combiner is associative.

▶ Exercise 5.3

What does the following procedure compute? Also, compare its performance with each of the three versions of `together-copies-of` installed, using relatively large values for the first argument, perhaps in the ten thousand to a million range.

```
(define mystery
  (lambda (a b)
    (together-copies-of + a b)))
```

For our second example, note that counting the number of times that 6 is a digit in a number (Exercise 2.9 on page 39) is very similar to counting the number of odd digits in a number (Exercise 2.10 on page 39). In the former case, you're testing to see if each digit is equal to 6 and in the latter you're testing to see if each digit is odd. Thus we can write a general procedure, `num-digits-in-satisfying`, that we can use to define both of these procedures. Its second parameter is the particular test predicate to use on each digit.

```
(define num-digits-in-satisfying
  (lambda (n test?)
    (cond ((< n 0)
           (num-digits-in-satisfying (- n) test?))
          ((< n 10)
           (if (test? n) 1 0))
          ((test? (remainder n 10))
           (+ (num-digits-in-satisfying (quotient n 10) test?)
              1))
          (else
           (num-digits-in-satisfying (quotient n 10) test?))))))
```

We can then define the procedures asked for in Exercises 2.9 and 2.10 as special cases of the more general procedure `num-digits-in-satisfying`:

```
(define num-odd-digits
  (lambda (n)
    (num-digits-in-satisfying n odd?)))

(define num-6s
  (lambda (n)
    (num-digits-in-satisfying n (lambda (n) (= n 6)))))
```

▶ Exercise 5.4

Use `num-digits-in-satisfying` to define the procedure `num-digits`, which was defined “from scratch” in Section 2.3.

▶ Exercise 5.5

Rewrite `num-digits-in-satisfying` so that it generates an iterative process.

Another computational pattern that occurs very frequently involves summing the values of a function over a given range of integers.

▶ Exercise 5.6

Write a general purpose procedure, that when given two integers, *low* and *high*, and a procedure for computing a function *f*, will compute $f(\text{low}) + f(\text{low} + 1) + f(\text{low} + 2) + \dots + f(\text{high})$. Show how it can be used to sum the squares of the integers from 5 to 10 and to sum the square roots of the integers from 10 to 100.

5.2 Uncomputability

Designing general purpose procedures with procedural parameters is an extremely practical skill. It can save considerable programming, because a procedure can be written a single time but reused in many contexts. However, despite this practicality, the single most interesting use of a procedure with a procedural parameter is in a theoretical proof. In this section we’ll take a look at the history and importance of this proof.

By now we’ve seen that procedures are quite powerful. They can be used for doing arithmetic on 200-digit numbers in order to produce digital signatures, for

making a variety of complex images, and for looking words up in dictionaries. You probably know of lots of other things procedures can be used for. There seems to be no limit to what we can do with them. At the beginning of the twentieth century, mathematicians addressed exactly that question: whether a procedure could be found to compute any function that could be precisely mathematically specified. That question was settled in the 1930s by the discovery of several *uncomputable functions* (one of which we'll examine in this section).

The specific function we'll prove uncomputable is a higher-order one and is often called the *halting problem*. It takes a procedure as an argument and returns a true/false value telling whether the given procedure generates a terminating process, as opposed to going into an infinite loop. Now imagine that one of your brilliant friends gives you a procedure, called `halts?`, that supposedly computes this function. You could then use this procedure on the simple procedures `return-seven` and `loop-forever` defined below. Evaluating `(halts? return-seven)` should result in `#t`, whereas `(halts? loop-forever)` should evaluate to `#f`.

```
(define return-seven
  (lambda ()
    7))
```

```
(define loop-forever
  (lambda ()
    (loop-forever)))
```

(`Return-seven` and `loop-forever` happen to be our first examples of procedures with no parameters. This is indicated by the empty parentheses.)

Clearly `halts?` would be a handy procedure to have, if it really worked. To start with, it could be used to test for a common kind of bug. Never again would you have to guess whether you'd goofed and accidentally written a nonterminating procedure. You could tell the difference between a computation that was taking a long time and one that would never finish.

Above and beyond this, you could answer all sorts of open mathematical questions. For example, we mentioned earlier that no one knows whether there are any odd perfect numbers. It would be easy enough to write a procedure that tested all the odd numbers, one by one, stopping when and if it found one that was perfect. Then all we'd have to do is apply `halts?` to it, and we'd have the answer—if we're told that our search procedure halts, there are odd perfect numbers; otherwise, there aren't. This suggests that such a procedure might be a bit too wonderful to exist—it would make obsolete centuries of mathematicians' hard work. However, this is far from a proof that it doesn't exist.

Another related sense in which `halts?` is a bit too good to be true forms a suitable basis for a proof that it can't be a sure-fire way to determine whether a given

procedure generates a halting process. (In other words, there must be procedures for which it either gives the wrong answer or fails to give an answer.) `Halts?` in effect claims to predict the future: It can tell you now whether a process will terminate or not at some point arbitrarily far into the future. The way to debunk such a fortune-teller is to do the exact opposite of what the fortune-teller foretells (provided that the fortune-teller is willing to give unambiguous answers to any question and that you believe in free will). This will be the essence of our proof that `halts?` can't work as claimed.

What we want is a procedure that asks `halts?` whether it is going to stop and then does the opposite:

```
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        666)))
```

`Debunk-halts?` halts if and only if `debunk-halts?` doesn't halt—provided the procedure `halts?` that it calls upon performs as advertised. But nothing can both halt and not halt, so there is only one possible conclusion: our assumption that such a `halts?` procedure exists must be wrong—there can be no procedure that provides that functionality.

The way we proved that the halting problem is uncomputable is called a *proof by contradiction*. What we did was to assume that it was computable, that is, that a procedure (`halts?`) exists that computes it. We then used this procedure to come up with `debunk-halts?`, which halts if and only if it doesn't halt. In other words, whether we assume that `debunk-halts?` halts or that it doesn't halt, we can infer the opposite; we are stuck with a contradiction either way. Because we arrived at this self-contradictory situation by assuming that we had a `halts?` procedure that correctly solved the halting problem, that assumption must be false; in other words, the halting problem is uncomputable.

This version of proof by contradiction, where the contradiction is arrived at by using an alleged universal object to produce the counterexample to its own universality, is known as a *diagonalization* proof. Another variation on the theme can be used to show that most functions can't even be specified, let alone implemented by a procedure.

We should point out that we've only given what most mathematicians would call a "sketch" of the actual proof that the halting problem is uncomputable. In a formal proof, the notions of what a procedure is, what process that procedure generates, and whether that process terminates need to be very carefully specified in formal mathematical terms. This ensures that the function mapping each procedure to a truth value based on whether or not it generates a terminating process is a well-

defined mathematical function. The mathematician Alan Turing spent considerable effort on these careful specifications when he originated the proof that `halts?` can't exist.

The discovery that there are mathematical functions that can be specified but not computed is one of the wedges that served to split computer science off from mathematics in the middle of the twentieth century. Of course, this was the same period when programmable electronic computers were first being designed and built (by Turing himself, among others). However, we can now see that the fundamental subject matter of mathematics and computer science are distinct: Mathematicians study any abstraction that can be formally specified, whereas computer scientists confine their attention to the smaller realm of the computable. Mathematicians sometimes are satisfied with an answer to the question “is there a ...,” whereas computer scientists ask “How do I find it?”

Alan Turing

One of the surest signs of genius in a computer scientist is the ability to excel in both the theoretical and the practical sides of the discipline. All the greatest computer scientists have had this quality, and most have even gone far beyond the borders of computer science in their breadth. Given the youth of the discipline, most of these greats are still alive, still alternating between theory and application, the computer and the pipe organ. Alan Turing, however, has the dual distinction of having been one of these greats who passed into legend.

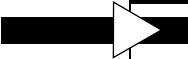
Turing developed one of the first careful theoretical models of the notions of algorithm and process in the 1930s, basing it on a bare-bones computing machine that is still an important theoretical model—the *Turing machine*, as it is called. He did this as the basis of his careful proof of the uncomputability of the halting problem, sketched in this section. In so doing he made a contribution of the first magnitude to the deepest theoretical side of computer science.

During World War II, Turing worked in the British code-breaking effort and successfully designed real-life computing machines dedicated to this purpose. He is given a considerable portion of the credit for the Allied forces' decisive cryptographic edge and in particular for the breaking of the German “Enigma” ciphers.

After the war Turing led the design of the National Physical Laboratory's ACE computer, which was one of the first digital electronic stored-program computers designed anywhere and the first such project started in England.

During this same post-war period of the late forties Turing returned more seriously to a question he had dabbled with for years, the question of *artificial*

Continued

 **Alan Turing (Continued)**

intelligence: whether intelligence is successfully describable as a computational process, such that a computing machine could be programmed to be intelligent. He made a lasting contribution to this area of thought by formulating the question in *operational* terms. In other words, he made the significant choice not to ask “is the machine *really* intelligent inside or just faking” but rather “can the machine be distinguished from a human, simply by looking at its outward behavior.” He formulated this in a very specific way: Can a computer be as successful as a man at convincing an interrogator that it is a woman? He also stipulated that the computer and the man should both be communicated with only through a textual computer terminal (or teletype). In the decades since Turing published this idea in 1950, it has been generalized such that any operational test of intelligence is today referred to as a “Turing test.” Theoretical foundations, applications to code breaking, computer design, and questions of artificial intelligence weren’t all that concerned Turing, however. He also made an important contribution to theoretical biology. His famous 1952 paper “The Chemical Basis of Morphogenesis” showed how chemical reactions in an initially homogeneous substance can give rise to large-scale orderly forms such as are characteristic of life.

Turing’s work on morphogenesis (the origins of form) never reached completion, however, because he tragically took his own life in 1954, at the age of 42. There is considerable uncertainty about exactly why he did this, or more generally about his state of mind. It is documented that he had gone through periods of depression, as well as considerable trauma connected with his sexual orientation. Turing was rather openly homosexual, at a time when sex between men was a crime in England, even if in private and with consent. In 1952 Turing was convicted of such behavior, based on his own frank admission. His lawyer asked the court to put him on probation, rather than sentence him to prison, on the condition that he undergo experimental medical treatment for his homosexuality—paradoxically it was considered an illness as well as a crime. The treatment consisted of large doses of estrogen (female hormones), which caused impotence, depression, and further stigmatization in the form of enlarged breasts. The treatment ended in 1953, but there is circumstantial evidence suggesting that British intelligence agencies kept close tabs on Turing thereafter, including detaining a foreign lover to prevent a rendezvous. (Apparently they were concerned that Turing might divulge his secret information regarding cryptography and related fields.) Although there is no clear evidence, this sequence of events probably played a role in the overall emotional progression leading to Turing’s suicide, cutting off what could have been the entire second half of his career.

Since the 1930s, when Turing showed that there could be no procedure that solves this halting problem, many other functions have been shown to be uncomputable. Many of these proofs have the form: “If I had this procedure, I could use it in this clever way to implement `halts?`. But `halts?` can’t exist, so this procedure must not either.” This is known as a *proof by reduction*.

5.3 Procedures That Make Procedures

Now we can return to the more practical question of what programming techniques are made possible by procedures that operate on procedures. (Recall that this is what *higher-order* means.) So far we have seen procedures that take other procedures as parameters, just as they might take numbers or images. However, procedures don’t just take values in: They also return values as the result of their computations. This carries over to procedural values as well; higher-order procedures can be used to compute procedural results. In other words, we can build procedures that will build procedures. Clearly this could be a very labor-saving device.

How do we get a procedure to return a new procedure? We do it in the same way that we get a procedure to return a number. Recall that in order to ensure that a procedure returns a number when it is applied, its body must be an expression that evaluates to a number. Similarly, for a procedure to create a new procedure when it is applied, its body must be an expression that evaluates to a procedure. At this point, we know of only one kind of expression that can evaluate to a new procedure—a lambda expression. For example, here is a simple “procedure factory” with examples of its use:

```
(define make-multiplier
  (lambda (scaling-factor)
    (lambda (x)
      (* x scaling-factor))))

(define double (make-multiplier 2))
(define triple (make-multiplier 3))

(double 7)
14

(triple 12)
36
```

When we evaluate the definition of `make-multiplier`, the outer lambda expression is evaluated immediately and has as its value the procedure named `make-multiplier`. That procedure is waiting to be told what the scaling factor is. When we evaluate

```
(define double (make-multiplier 2))
```

the body of the procedure named `make-multiplier` is evaluated, with the value 2 substituted for `scaling-factor`. In other words, the expression `(lambda (x) (* x scaling-factor))` is evaluated with 2 substituted for `scaling-factor`. The result of this evaluation is the procedure that is named `double`, just as though the definition had been `(define double (lambda (x) (* x 2)))`. When we apply `double` to 7, the procedure `(lambda (x) (* x 2))` is applied to 7, and the result is, of course, 14.

▶ Exercise 5.7

Write a procedure `make-exponentiator` that is passed a single parameter e (an exponent) and returns a function that itself takes a single parameter, which it raises to the e power. You should use the built-in Scheme procedure `expt`. As examples, you could define `square` and `cube` as follows:

```
(define square (make-exponentiator 2))
```

```
(define cube (make-exponentiator 3))
```

```
(square 4)
```

```
16
```

```
(cube 4)
```

```
64
```

For another example of a procedure factory, suppose that we want to automate the production of procedures like `repeatedly-square`, from Section 3.2. That procedure took two arguments, the number to square and how many times it should be squared. We could make a procedure factory called `make-repeated-version-of` that would be able to make `repeatedly-square` out of `square`:

```
(define make-repeated-version-of
  (lambda (f) ; make a repeated version of f
    (define the-repeated-version
      (lambda (b n) ; which does f n times to b
        (if (= n 0)
            b
            (the-repeated-version (f b) (- n 1))))
      the-repeated-version))
  (define square (lambda (x) (* x x)))
```

```
(define repeatedly-square
  (make-repeated-version-of square))

(repeatedly-square 2 3) ; 2 squared squared squared
256
```

One thing worth noticing in this example is that we used an internal definition of `the-repeated-version` to provide a name for the generated procedure. That way we can refer to it by name where it reinvokes itself to do the $n - 1$ remaining repetitions. Having internally defined this name, we then return the procedure it is a name for.

▶ Exercise 5.8

Define a procedure that can be used to produce `factorial` (Section 2.1) or `sum-of-first` (Section 2.3). Show how it can be used to define those two procedures.

▶ Exercise 5.9

Generalize your solution to the previous exercise so it can also be used to produce `sum-of-squares` and `sum-of-cubes` from Exercise 2.8 on page 38.

5.4 An Application: Verifying ID Numbers

Does this scenario sound familiar?

May I have your credit card number please?

Yes, it's 6011302631452178.

I'm sorry, I must have typed that wrong. Could you please say it again?

How did the sales representative know the number was wrong?

Credit card numbers are one of the most common examples of *self-verifying numbers*. Other examples include the ISBN numbers on books, the UPC (Universal Product Code) numbers on groceries, the bank numbers on checks, the serial numbers on postal money orders, the membership numbers in many organizations, and the student ID numbers at many universities.

Self-verifying numbers are designed in such a way that any valid number will have some specific numerical property and so that most simple errors (such as getting two digits backward or changing the value of one of the digits) result in numbers that don't have the property. That way a legitimate number can be distinguished from one that is in error, even without taking the time to search through the entire list of valid numbers.

What interests us about self-verifying numbers is that there are many different systems in use, but they are almost all of the same general form. Therefore, although we will need separate procedures for checking the validity of each kind of number, we can make good use of a higher-order procedure to build all of the verifiers for us.

Suppose we call the rightmost digit of a number d_1 , the second digit from the right d_2 , etc. All of the kinds of identifying numbers listed previously possess a property of the following kind:

$$f(1, d_1) + f(2, d_2) + f(3, d_3) + \dots \text{ is divisible by } m$$

All that is different between a credit card and a grocery item, or between a book and a money order, is the specific function f and the divisor m .

How do we define a procedure factory that will construct verifiers for us? As we did in Section 5.3, we will first look at one of the procedures that this factory is supposed to produce. This verifier checks to see whether the sum of the digits is divisible by 17; in other words, the divisor is 17 and the function is just $f(i, d_i) = d_i$. To write the verifier, we'll first write a procedure to add the digits. Recall from Chapter 2 that we can get at the individual digits in a number by using division by 10. The remainder when we divide by 10 is the rightmost digit, d_1 , and the quotient is the rest of the digits. For example, here is how we could compute the sum of the digits in a number (as in Exercise 2.11 on page 39) using an iterative process:

```
(define sum-of-digits
  (lambda (n)
    (define sum-plus ;(sum of n's digits) + addend
      (lambda (n addend)
        (if (= n 0)
            addend
            (sum-plus (quotient n 10)
                      (+ addend (remainder n 10))))))
      (sum-plus n 0)))
```

▶ Exercise 5.10

Write a predicate that takes a number and determines whether the sum of its digits is divisible by 17.

▶ Exercise 5.11

Write a procedure `make-verifier`, which takes f and m as its two arguments and returns a procedure capable of checking a number. The argument f is itself a

procedure, of course. Here is a particularly simple example of a verifier being made and used:

```
(define check-isbn (make-verifier * 11))

(check-isbn 0262010771)
#t
```

The value `#t` is the “true” value; it indicates that the number is a valid ISBN.

As we just saw, for ISBN numbers the divisor is 11 and the function is simply $f(i, d_i) = i \times d_i$. Other kinds of numbers use slightly more complicated functions, but you will still be able to use `make-verifier` to make a verifier much more easily than if you had to start from scratch.

▶ Exercise 5.12

For UPC codes (the barcodes on grocery items), the divisor is 10, and the function $f(i, d_i)$ is equal to d_i itself when i is odd, but to $3d_i$ when i is even. Build a verifier for UPC codes using `make-verifier`, and test it on some of your groceries. (The UPC number consists of all the digits: the one to the left of the bars, the ones underneath the bars, and the one on the right.) Try making some mistakes, like switching or changing digits. Does your verifier catch them?

▶ Exercise 5.13

Credit card numbers also use a divisor of 10 and also use a function that yields d_i itself when i is odd. However, when i is even, the function is a bit fancier: It is $2d_i$ if $d_i < 5$, and $2d_i + 1$ if $d_i \geq 5$. Build a verifier for credit card numbers. In the dialog at the beginning of this section, did the order taker really mistype the number, or did the customer read it incorrectly?

▶ Exercise 5.14

The serial number on U.S. postal money orders is self-verifying with a divisor of 9 and a very simple function: $f(i, d_i) = d_i$, with only one exception, namely, $f(1, d_1) = -d_1$. Build a verifier for these numbers, and find out which of these two money orders is mistyped: 48077469777 or 48077462766.

Actually, *both* of those money order numbers were mistyped. In one case the error was that a 0 was replaced by a 7, and in the other case two digits were reversed. Can you figure out which kind of error got caught and which didn't? Does this help explain why the other kinds of numbers use fancier functions?

Review Problems

▶ Exercise 5.15

Write a higher-order procedure called `make-function-with-exception` that takes two numbers and a procedure as parameters and returns a procedure that has the same behavior as the procedural argument except when given a special argument. The two numerical arguments to `make-function-with-exception` specify what that exceptional argument is and what the procedure made by `make-function-with-exception` should return in that case. For example, the `usually-sqrt` procedure that follows behaves like `sqrt`, except that when given the argument 7, it returns the result 2:

```
(define usually-sqrt
  (make-function-with-exception 7 2 sqrt))
```

```
(usually-sqrt 9)
3
```

```
(usually-sqrt 16)
4
```

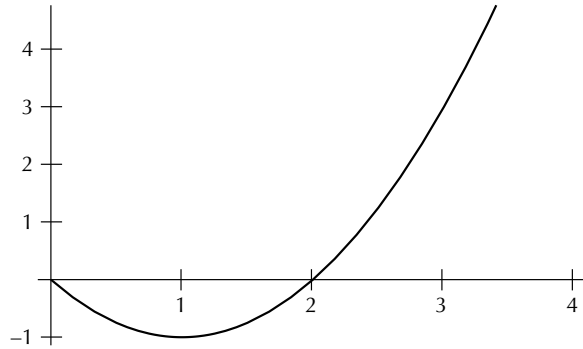
```
(usually-sqrt 7)
2
```

▶ Exercise 5.16

If two procedures f and g are both procedures of a single argument such that the values produced by g are legal arguments to f , the *composition* of f and g is defined to be the procedure that first applies g to its argument and then applies f to the result. Write a procedure called `compose` that takes two one-argument procedures and returns the procedure that is their composition. For example, `((compose sqrt abs) -4)` should compute the square root of the absolute value of -4 .

▶ Exercise 5.17

Suppose you have a function and you want to find at what integer point in a given range it has the smallest value. For example, looking at the following graph of the function $f(x) = x^2 - 2x$, you can see that in the range from 0 to 4, this function has the smallest value at 1.



We could write a procedure for answering questions like this; it could be used as follows for this example:

```
(integer-in-range-where-smallest (lambda (x)
                                  (- (* x x) (* 2 x)))
  0 4)
1
```

Here is the procedure that does this; fill in the two blanks to complete it:

```
(define integer-in-range-where-smallest
  (lambda (f a b)
    (if (= a b)
        a
        (let ((smallest-place-after-a
                _____))
          (if _____
              a
              smallest-place-after-a))))))
```

► Exercise 5.18

Consider the following definitions:

```
(define make-scaled
  (lambda (scale f)
    (lambda (x)
      (* scale (f x)))))

(define add-one
  (lambda (x)
    (+ 1 x)))
```



```
(define mystery
  (make-scaled 3 add-one))
```

For the following questions, be sure to indicate how you arrived at your answer:

- a. What is the value of `(mystery 4)`?
- b. What is the value of the procedural call `((make-scaled 2 (make-scaled 3 add-one)) 4)`?

▷ Exercise 5.19

If l and h are integers, with $l < h$, we say f is an increasing function on the integer range from l to h if $f(l) < f(l+1) < f(l+2) < \dots < f(h)$. Write a procedure, `increasing-on-integer-range?`, that takes f , l , and h as its three arguments and returns true or false (that is, `#t` or `#f`) as appropriate.

▷ Exercise 5.20

Suppose the following have been defined:

```
(define f
  (lambda (m b)
    (lambda (x) (+ (* m x) b))))

(define g (f 3 2))
```

For each of the following expressions, indicate whether an error would be signaled, the value would be a procedure, or the value would be a number. If an error is signaled, explain briefly the nature of the error. If the value is a procedure, specify how many arguments the procedure expects. If the value is a number, specify which number.

- a. `f`
- b. `g`
- c. `(* (f 3 2) 7)`
- d. `(g 6)`
- e. `(f 6)`
- f. `((f 4 7) 5)`

 **Exercise 5.21**

We saw in Section 5.3 the following procedure-generating procedure:

```
(define make-multiplier
  (lambda (scaling-factor)
    (lambda (x)
      (* x scaling-factor))))
```

You were also asked in Exercise 5.7 to write the procedure `make-exponentiater`.

Notice that these two procedures are quite similar. We could abstract out the commonality into an even more general procedure `make-generator` such that we could then just write:

```
(define make-multiplier (make-generator *))

(define make-exponentiater (make-generator expt))
```

Write `make-generator`.

 **Exercise 5.22**

The function `halts?` was defined as a test of whether a procedure with no parameters would generate a terminating process. That is, `(halts? f)` returns true if and only if the evaluation of `(f)` would terminate. What about procedures that take arguments? Suppose we had a procedure `halts-on?` that tests whether a one-argument procedure generates a terminating process when given some particular argument. That is, `(halts-on? f x)` returns true if and only if the evaluation of `(f x)` would terminate.

- a. Use `halts-on?` in a definition of `halts?`.
- b. What does this tell you about the possibility of `halts-on?`

 **Exercise 5.23**

Consider the following example:

```
(define double (lambda (x) (* x 2)))
(define square (lambda (x) (* x x)))
(define new-procedure
  (make-averaged-procedure double square))
```

```
(new-procedure 4)
12
(new-procedure 6)
24
```

In the first example, the `new-procedure` that was made by `make-averaged-procedure` returned 12 because 12 is the average of 8 (twice 4) and 16 (4 squared). In the second example, it returned 24 because 24 is the average of 12 (twice 6) and 36 (6 squared). In general, `new-procedure` will return the average of whatever `double` and `square` return because those two procedures were passed to `make-averaged-procedure` when `new-procedure` was made.

Write the higher-order procedure factory `make-averaged-procedure`.

▶ Exercise 5.24

Consider the following procedure:

```
(define positive-integer-upto-where-smallest
  (lambda (n f) ; return an integer i such that
    ; 1 <= i <= n and for all integers j
    ; in that same range, f(i) <= f(j)
    (define loop
      (lambda (where-smallest-so-far next-to-try)
        (if (> next-to-try n)
            where-smallest-so-far
            (loop (if (< (f next-to-try)
                       (f where-smallest-so-far))
                    next-to-try
                    where-smallest-so-far)
                  (+ next-to-try 1))))))
    (loop 1 2)))
```

- Write a mathematical formula involving n that tells how many times this procedure uses the procedure it is given as its second argument. Justify your answer.
- Give a simple Θ order of growth for the quantity you determined in part a. Justify your answer.
- Suppose you were to rewrite this procedure to make it more efficient. What (in terms of n) is the minimum number of times it can invoke `f` and still always determine the correct answer? Justify your answer. (You are *not* being asked to actually rewrite the procedure.)

Chapter Inventory

Vocabulary

procedural parameter	Turing machine
higher-order procedure	artificial intelligence
uncomputable function	operational test of intelligence
halting problem	Turing test
proof by contradiction	self-verifying number
diagonalization proof	composition
proof by reduction	

Scheme Names Defined in This Chapter

together-copies-of	repeatedly-square
stack-copies-of	factorial
power	sum-of-first
mod-expt	sum-of-squares
mystery	sum-of-cubes
num-digits-in-satisfying	sum-of-digits
num-odd-digits	make-verifier
num-6s	check-isbn
num-digits	make-function-with-exception
return-seven	compose
loop-forever	integer-in-range-where-smallest
debunk-halts?	make-scaled
make-multiplier	add-one
double	increasing-on-integer-range?
triple	make-generator
make-exponentiater	new-procedure
square	make-averaged-procedure
cube	positive-integer-upto-
make-repeated-version-of	where-smallest

Sidebars

Alan Turing

Notes

Turing's original proof that `halts?` can't exist is in [51]. The standard biography of Turing is Hodges's [26], and we heartily recommend it.

We remarked in passing that diagonalization can also be used to prove that most functions can't even be specified, let alone implemented by a procedure. To give some flavor for this, let's restrict ourselves to functions mapping positive integers to positive integers and show that any notational scheme must miss at least one of them. Consider an infinitely long list of all possible function specifications in the notational scheme under consideration, arranged in alphabetical order; call the first one f_1 , the second one f_2 , etc. Now consider the function f that has the property that for all n , $f(n) = f_n(n) + 1$. Clearly there is no n for which f is identical to f_n , because it differs from each of them in at least one place (namely, at n). Thus f is a function that is not on the list.

Our information about the various schemes used for self-verifying numbers is gleaned in small part from experimentation but primarily from two articles by Gallian, [20] and [19]. Those articles contain more of the mathematical underpinnings and citations for additional sources. We confess that the ISBN checker we defined as an example will only work for those ISBNs that consist purely of digits; one-eleventh of all ISBNs end with an X rather than a digit. This X is treated as though it were a digit with value 10.

