1

Building Abstractions with Procedures

> The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.
>
> —John Locke, *An Essay Concerning Human Understanding* (1690)

We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called *data*.

The evolution of a process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. Even small errors (usually called *bugs* or *glitches*) in programs can have complex and unanticipated consequences.

Fortunately, learning to program is considerably less dangerous than learning sorcery, because the spirits we deal with are conveniently contained in a secure way. Real-world programming, however, requires care, expertise, and wisdom. A small bug in a computer-aided design program, for example, can lead to the catastrophic collapse of an airplane or a dam or the self-destruction of an industrial robot.

Master software engineers have the ability to organize programs so that they can be reasonably sure that the resulting processes will perform the tasks intended. They can visualize the behavior of their systems in advance. They know how to structure programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, they can *debug* their programs. Well-designed com-

putational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.

Programming in Lisp

We need an appropriate language for describing processes, and we will use for this purpose the programming language Lisp. Just as our everyday thoughts are usually expressed in our natural language (such as English, French, or Japanese), and descriptions of quantitative phenomena are expressed with mathematical notations, our procedural thoughts will be expressed in Lisp. Lisp was invented in the late 1950s as a formalism for reasoning about the use of certain kinds of logical expressions, called *recursion equations*, as a model for computation. The language was conceived by John McCarthy and is based on his paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine" (McCarthy 1960).

Despite its inception as a mathematical formalism, Lisp is a practical programming language. A Lisp *interpreter* is a machine that carries out processes described in the Lisp language. The first Lisp interpreter was implemented by McCarthy with the help of colleagues and students in the Artificial Intelligence Group of the MIT Research Laboratory of Electronics and in the MIT Computation Center.[1] Lisp, whose name is an acronym for LISt Processing, was designed to provide symbol-manipulating capabilities for attacking programming problems such as the symbolic differentiation and integration of algebraic expressions. It included for this purpose new data objects known as atoms and lists,

[1] The *Lisp 1 Programmer's Manual* appeared in 1960, and the *Lisp 1.5 Programmer's Manual* (McCarthy et al. 1965) was published in 1962. The early history of Lisp is described in McCarthy 1978.

which most strikingly set it apart from all other languages of the period.

Lisp was not the product of a concerted design effort. Instead, it evolved informally in an experimental manner in response to users' needs and to pragmatic implementation considerations. Lisp's informal evolution has continued through the years, and the community of Lisp users has traditionally resisted attempts to promulgate any "official" definition of the language. This evolution, together with the flexibility and elegance of the initial conception, has enabled Lisp, which is the second oldest language in widespread use today (only Fortran is older), to continually adapt to encompass the most modern ideas about program design. Thus, Lisp is by now a family of dialects, which, while sharing most of the original features, may differ from one another in significant ways. The dialect of Lisp used in this book is called Scheme.[2]

Because of its experimental character and its emphasis on symbol manipulation, Lisp was at first very inefficient for numerical computations, at least in comparison with Fortran. Over the years, however,

[2] The two dialects in which most major Lisp programs of the 1970s were written are MacLisp (Moon 1978; Pitman 1983), developed at the MIT Project MAC, and Interlisp (Teitelman 1974), developed at Bolt Beranek and Newman Inc. and the Xerox Palo Alto Research Center. Portable Standard Lisp (Hearn 1969; Griss 1981) was a Lisp dialect designed to be easily portable between different machines. MacLisp spawned a number of subdialects, such as Franz Lisp, which was developed at the University of California at Berkeley, and Zetalisp (Moon and Weinreb 1981), which was based on a special-purpose processor designed at the MIT Artificial Intelligence Laboratory to run Lisp very efficiently. The Lisp dialect used in this book, called Scheme (Steele and Sussman 1975), was invented in 1975 by Guy Lewis Steele Jr. and Gerald Jay Sussman of the MIT Artificial Intelligence Laboratory and later reimplemented for instructional use at MIT. Scheme became an IEEE standard in 1990 (IEEE 1990). The Common Lisp dialect (Steele 1982, Steele 1990) was developed by the Lisp community to combine features from the earlier Lisp dialects to make an industrial standard for Lisp. Common Lisp became an ANSI standard in 1994 (ANSI 1994).

Lisp compilers have been developed that translate programs into machine code that can perform numerical computations reasonably efficiently. And for special applications, Lisp has been used with great effectiveness.[3] Although Lisp has not yet overcome its old reputation as hopelessly inefficient, Lisp is now used in many applications where efficiency is not the central concern. For example, Lisp has become a language of choice for operating-system shell languages and for extension languages for editors and computer-aided design systems.

If Lisp is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them. The most significant of these features is the fact that Lisp descriptions of processes, called *procedures*, can themselves be represented and manipulated as Lisp data. The importance of this is that there are powerful program-design techniques that rely on the ability to blur the traditional distinction between "passive" data and "active" processes. As we shall discover, Lisp's flexibility in handling procedures as data makes it one of the most convenient languages in existence for exploring these techniques. The ability to represent procedures as data also makes Lisp an excellent language for writing programs that must manipulate other programs as data, such as the interpreters and compilers that support computer languages. Above and beyond these considerations, programming in Lisp is great fun.

[3] One such special application was a breakthrough computation of scientific importance—an integration of the motion of the Solar System that extended previous results by nearly two orders of magnitude, and demonstrated that the dynamics of the Solar System is chaotic. This computation was made possible by new integration algorithms, a special-purpose compiler, and a special-purpose computer all implemented with the aid of software tools written in Lisp (Abelson et al. 1992; Sussman and Wisdom 1992).

1.1 The Elements of Programming

A powerful programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework within which we organize our ideas about processes. Thus, when we describe a language, we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

In programming, we deal with two kinds of elements: procedures and data. (Later we will discover that they are really not so distinct.) Informally, data is "stuff" that we want to manipulate, and procedures are descriptions of the rules for manipulating the data. Thus, any powerful programming language should be able to describe primitive data and primitive procedures and should have methods for combining and abstracting procedures and data.

In this chapter we will deal only with simple numerical data so that we can focus on the rules for building procedures.[4] In later chapters we

[4]The characterization of numbers as "simple data" is a barefaced bluff. In fact, the treatment of numbers is one of the trickiest and most confusing aspects of any pro-

will see that these same rules allow us to build procedures to manipulate compound data as well.

1.1.1 Expressions

One easy way to get started at programming is to examine some typical interactions with an interpreter for the Scheme dialect of Lisp. Imagine that you are sitting at a computer terminal. You type an *expression*, and the interpreter responds by displaying the result of its *evaluating* that expression.

One kind of primitive expression you might type is a number. (More precisely, the expression that you type consists of the numerals that represent the number in base 10.) If you present Lisp with a number

```
486
```

the interpreter will respond by printing[5]

```
486
```

gramming language. Some typical issues involved are these: Some computer systems distinguish *integers*, such as 2, from *real numbers*, such as 2.71. Is the real number 2.00 different from the integer 2? Are the arithmetic operations used for integers the same as the operations used for real numbers? Does 6 divided by 2 produce 3, or 3.0? How large a number can we represent? How many decimal places of accuracy can we represent? Is the range of integers the same as the range of real numbers? Above and beyond these questions, of course, lies a collection of issues concerning roundoff and truncation errors—the entire science of numerical analysis. Since our focus in this book is on large-scale program design rather than on numerical techniques, we are going to ignore these problems. The numerical examples in this chapter will exhibit the usual roundoff behavior that one observes when using arithmetic operations that preserve a limited number of decimal places of accuracy in noninteger operations.

[5] Throughout this book, when we wish to emphasize the distinction between the input typed by the user and the response printed by the interpreter, we will show the latter in slanted characters.

Expressions representing numbers may be combined with an expression representing a primitive procedure (such as + or *) to form a compound expression that represents the application of the procedure to those numbers. For example:

```
(+ 137 349)
486

(- 1000 334)
666

(* 5 99)
495

(/ 10 5)
2

(+ 2.7 10)
12.7
```

Expressions such as these, formed by delimiting a list of expressions within parentheses in order to denote procedure application, are called *combinations*. The leftmost element in the list is called the *operator*, and the other elements are called *operands*. The value of a combination is obtained by applying the procedure specified by the operator to the *arguments* that are the values of the operands.

The convention of placing the operator to the left of the operands is known as *prefix notation*, and it may be somewhat confusing at first because it departs significantly from the customary mathematical convention. Prefix notation has several advantages, however. One of them is that it can accommodate procedures that may take an arbitrary number of arguments, as in the following examples:

```
(+ 21 35 12 7)
75

(* 25 4 12)
1200
```

No ambiguity can arise, because the operator is always the leftmost element and the entire combination is delimited by the parentheses.

A second advantage of prefix notation is that it extends in a straightforward way to allow combinations to be *nested*, that is, to have combinations whose elements are themselves combinations:

```
(+ (* 3 5) (- 10 6))
19
```

There is no limit (in principle) to the depth of such nesting and to the overall complexity of the expressions that the Lisp interpreter can evaluate. It is we humans who get confused by still relatively simple expressions such as

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

which the interpreter would readily evaluate to be 57. We can help ourselves by writing such an expression in the form

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

following a formatting convention known as *pretty-printing*, in which each long combination is written so that the operands are aligned vertically. The resulting indentations display clearly the structure of the

expression.[6]

Even with complex expressions, the interpreter always operates in the same basic cycle: It reads an expression from the terminal, evaluates the expression, and prints the result. This mode of operation is often expressed by saying that the interpreter runs in a *read-eval-print loop*. Observe in particular that it is not necessary to explicitly instruct the interpreter to print the value of the expression.[7]

1.1.2 Naming and the Environment

A critical aspect of a programming language is the means it provides for using names to refer to computational objects. We say that the name identifies a *variable* whose *value* is the object.

In the Scheme dialect of Lisp, we name things with `define`. Typing

```
(define size 2)
```

causes the interpreter to associate the value 2 with the name `size`.[8] Once the name `size` has been associated with the number 2, we can refer to the value 2 by name:

```
size
2
```

[6] Lisp systems typically provide features to aid the user in formatting expressions. Two especially useful features are one that automatically indents to the proper pretty-print position whenever a new line is started and one that highlights the matching left parenthesis whenever a right parenthesis is typed.

[7] Lisp obeys the convention that every expression has a value. This convention, together with the old reputation of Lisp as an inefficient language, is the source of the quip by Alan Perlis (paraphrasing Oscar Wilde) that "Lisp programmers know the value of everything but the cost of nothing."

[8] In this book, we do not show the interpreter's response to evaluating definitions, since this is highly implementation-dependent.

```
(* 5 size)
10
```

Here are further examples of the use of `define`:

```
(define pi 3.14159)
(define radius 10)
(* pi (* radius radius))
314.159
(define circumference (* 2 pi radius))
circumference
62.8318
```

`define` is our language's simplest means of abstraction, for it allows us to use simple names to refer to the results of compound operations, such as the `circumference` computed above. In general, computational objects may have very complex structures, and it would be extremely inconvenient to have to remember and repeat their details each time we want to use them. Indeed, complex programs are constructed by building, step by step, computational objects of increasing complexity. The interpreter makes this step-by-step program construction particularly convenient because name-object associations can be created incrementally in successive interactions. This feature encourages the incremental development and testing of programs and is largely responsible for the fact that a Lisp program usually consists of a large number of relatively simple procedures.

It should be clear that the possibility of associating values with symbols and later retrieving them means that the interpreter must maintain some sort of memory that keeps track of the name-object pairs. This memory is called the *environment* (more precisely the *global environment*, since we will see later that a computation may involve a number

of different environments).[9]

1.1.3 Evaluating Combinations

One of our goals in this chapter is to isolate issues about thinking procedurally. As a case in point, let us consider that, in evaluating combinations, the interpreter is itself following a procedure.

To evaluate a combination, do the following:

1. Evaluate the subexpressions of the combination.

2. Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpressions (the operands).

Even this simple rule illustrates some important points about processes in general. First, observe that the first step dictates that in order to accomplish the evaluation process for a combination we must first perform the evaluation process on each element of the combination. Thus, the evaluation rule is *recursive* in nature; that is, it includes, as one of its steps, the need to invoke the rule itself.[10]

Notice how succinctly the idea of recursion can be used to express what, in the case of a deeply nested combination, would otherwise be viewed as a rather complicated process. For example, evaluating

[9] Chapter 3 will show that this notion of environment is crucial, both for understanding how the interpreter works and for implementing interpreters.

[10] It may seem strange that the evaluation rule says, as part of the first step, that we should evaluate the leftmost element of a combination, since at this point that can only be an operator such as + or * representing a built-in primitive procedure such as addition or multiplication. We will see later that it is useful to be able to work with combinations whose operators are themselves compound expressions.

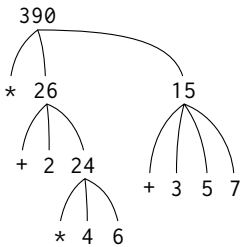

Figure 1.1: Tree representation, showing the value of each subcombination.

```
(* (+ 2 (* 4 6))
   (+ 3 5 7))
```

requires that the evaluation rule be applied to four different combinations. We can obtain a picture of this process by representing the combination in the form of a tree, as shown in Figure 1.1. Each combination is represented by a node with branches corresponding to the operator and the operands of the combination stemming from it. The terminal nodes (that is, nodes with no branches stemming from them) represent either operators or numbers. Viewing evaluation in terms of the tree, we can imagine that the values of the operands percolate upward, starting from the terminal nodes and then combining at higher and higher levels. In general, we shall see that recursion is a very powerful technique for dealing with hierarchical, treelike objects. In fact, the "percolate values upward" form of the evaluation rule is an example of a general kind of process known as *tree accumulation*.

Next, observe that the repeated application of the first step brings us to the point where we need to evaluate, not combinations, but primitive expressions such as numerals, built-in operators, or other names. We

take care of the primitive cases by stipulating that

- the values of numerals are the numbers that they name,
- the values of built-in operators are the machine instruction sequences that carry out the corresponding operations, and
- the values of other names are the objects associated with those names in the environment.

We may regard the second rule as a special case of the third one by stipulating that symbols such as + and * are also included in the global environment, and are associated with the sequences of machine instructions that are their "values." The key point to notice is the role of the environment in determining the meaning of the symbols in expressions. In an interactive language such as Lisp, it is meaningless to speak of the value of an expression such as `(+ x 1)` without specifying any information about the environment that would provide a meaning for the symbol `x` (or even for the symbol +). As we shall see in Chapter 3, the general notion of the environment as providing a context in which evaluation takes place will play an important role in our understanding of program execution.

Notice that the evaluation rule given above does not handle definitions. For instance, evaluating `(define x 3)` does not apply `define` to two arguments, one of which is the value of the symbol `x` and the other of which is 3, since the purpose of the `define` is precisely to associate `x` with a value. (That is, `(define x 3)` is not a combination.)

Such exceptions to the general evaluation rule are called *special forms*. `define` is the only example of a special form that we have seen so far, but we will meet others shortly. Each special form has its own evaluation rule. The various kinds of expressions (each with its associated

evaluation rule) constitute the syntax of the programming language. In comparison with most other programming languages, Lisp has a very simple syntax; that is, the evaluation rule for expressions can be described by a simple general rule together with specialized rules for a small number of special forms.[11]

1.1.4 Compound Procedures

We have identified in Lisp some of the elements that must appear in any powerful programming language:

- Numbers and arithmetic operations are primitive data and procedures.
- Nesting of combinations provides a means of combining operations.
- Definitions that associate names with values provide a limited means of abstraction.

Now we will learn about *procedure definitions*, a much more powerful abstraction technique by which a compound operation can be given a name and then referred to as a unit.

[11] Special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways are sometimes called *syntactic sugar*, to use a phrase coined by Peter Landin. In comparison with users of other languages, Lisp programmers, as a rule, are less concerned with matters of syntax. (By contrast, examine any Pascal manual and notice how much of it is devoted to descriptions of syntax.) This disdain for syntax is due partly to the flexibility of Lisp, which makes it easy to change surface syntax, and partly to the observation that many "convenient" syntactic constructs, which make the language less uniform, end up causing more trouble than they are worth when programs become large and complex. In the words of Alan Perlis, "Syntactic sugar causes cancer of the semicolon."

We begin by examining how to express the idea of "squaring." We might say, "To square something, multiply it by itself." This is expressed in our language as

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

We can understand this in the following way:

```
(define (square    x)         (*      x         x))
  |       |       |           |      |         |
 To     square  something,  multiply  it  by  itself.
```

We have here a *compound procedure*, which has been given the name `square`. The procedure represents the operation of multiplying something by itself. The thing to be multiplied is given a local name, `x`, which plays the same role that a pronoun plays in natural language. Evaluating the definition creates this compound procedure and associates it with the name `square`.[12]

The general form of a procedure definition is

```
(define (⟨name⟩ ⟨formal parameters⟩)
  ⟨body⟩)
```

The ⟨*name*⟩ is a symbol to be associated with the procedure definition in the environment.[13] The ⟨*formal parameters*⟩ are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The ⟨*body*⟩ is an expression that will yield the value of

[12] Observe that there are two different operations being combined here: we are creating the procedure, and we are giving it the name `square`. It is possible, indeed important, to be able to separate these two notions—to create procedures without naming them, and to give names to procedures that have already been created. We will see how to do this in Section 1.3.2.

[13] Throughout this book, we will describe the general syntax of expressions by using italic symbols delimited by angle brackets—e.g., ⟨*name*⟩—to denote the "slots" in the expression to be filled in when such an expression is actually used.

the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied.[14] The ⟨*name*⟩ and the ⟨*formal parameters*⟩ are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined `square`, we can now use it:

```
(square 21)
441
(square (+ 2 5))
49
(square (square 3))
81
```

We can also use `square` as a building block in defining other procedures. For example, $x^2 + y^2$ can be expressed as

```
(+ (square x) (square y))
```

We can easily define a procedure `sum-of-squares` that, given any two numbers as arguments, produces the sum of their squares:

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(sum-of-squares 3 4)
25
```

Now we can use `sum-of-squares` as a building block in constructing further procedures:

```
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
(f 5)
136
```

[14] More generally, the body of the procedure can be a sequence of expressions. In this case, the interpreter evaluates each expression in the sequence in turn and returns the value of the final expression as the value of the procedure application.

Compound procedures are used in exactly the same way as primitive procedures. Indeed, one could not tell by looking at the definition of `sum-of-squares` given above whether `square` was built into the interpreter, like `+` and `*`, or defined as a compound procedure.

1.1.5 The Substitution Model for Procedure Application

To evaluate a combination whose operator names a compound procedure, the interpreter follows much the same process as for combinations whose operators name primitive procedures, which we described in Section 1.1.3. That is, the interpreter evaluates the elements of the combination and applies the procedure (which is the value of the operator of the combination) to the arguments (which are the values of the operands of the combination).

We can assume that the mechanism for applying primitive procedures to arguments is built into the interpreter. For compound procedures, the application process is as follows:

> To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

To illustrate this process, let's evaluate the combination

```
(f 5)
```

where `f` is the procedure defined in Section 1.1.4. We begin by retrieving the body of `f`:

```
(sum-of-squares (+ a 1) (* a 2))
```

Then we replace the formal parameter `a` by the argument 5:

```
(sum-of-squares (+ 5 1) (* 5 2))
```

Thus the problem reduces to the evaluation of a combination with two operands and an operator `sum-of-squares`. Evaluating this combination involves three subproblems. We must evaluate the operator to get the procedure to be applied, and we must evaluate the operands to get the arguments. Now `(+ 5 1)` produces 6 and `(* 5 2)` produces 10, so we must apply the `sum-of-squares` procedure to 6 and 10. These values are substituted for the formal parameters `x` and `y` in the body of `sum-of-squares`, reducing the expression to

```
(+ (square 6) (square 10))
```

If we use the definition of `square`, this reduces to

```
(+ (* 6 6) (* 10 10))
```

which reduces by multiplication to

```
(+ 36 100)
```

and finally to

```
136
```

The process we have just described is called the *substitution model* for procedure application. It can be taken as a model that determines the "meaning" of procedure application, insofar as the procedures in this chapter are concerned. However, there are two points that should be stressed:

- The purpose of the substitution is to help us think about procedure application, not to provide a description of how the interpreter really works. Typical interpreters do not evaluate procedure applications by manipulating the text of a procedure to substitute values for the formal parameters. In practice, the "substitution" is accomplished by using a local environment for the formal parameters. We will discuss this more fully in Chapter 3 and

Chapter 4 when we examine the implementation of an interpreter in detail.

- Over the course of this book, we will present a sequence of increasingly elaborate models of how interpreters work, culminating with a complete implementation of an interpreter and compiler in Chapter 5. The substitution model is only the first of these models—a way to get started thinking formally about the evaluation process. In general, when modeling phenomena in science and engineering, we begin with simplified, incomplete models. As we examine things in greater detail, these simple models become inadequate and must be replaced by more refined models. The substitution model is no exception. In particular, when we address in Chapter 3 the use of procedures with "mutable data," we will see that the substitution model breaks down and must be replaced by a more complicated model of procedure application.[15]

Applicative order versus normal order

According to the description of evaluation given in Section 1.1.3, the interpreter first evaluates the operator and operands and then applies the resulting procedure to the resulting arguments. This is not the only way to perform evaluation. An alternative evaluation model would not evaluate the operands until their values were needed. Instead it would

[15] Despite the simplicity of the substitution idea, it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process. The problem arises from the possibility of confusion between the names used for the formal parameters of a procedure and the (possibly identical) names used in the expressions to which the procedure may be applied. Indeed, there is a long history of erroneous definitions of *substitution* in the literature of logic and programming semantics. See Stoy 1977 for a careful discussion of substitution.

first substitute operand expressions for parameters until it obtained an expression involving only primitive operators, and would then perform the evaluation. If we used this method, the evaluation of `(f 5)` would proceed according to the sequence of expansions

```
(sum-of-squares (+ 5 1) (* 5 2))
(+   (square (+ 5 1))      (square (* 5 2))  )
(+   (* (+ 5 1) (+ 5 1))   (* (* 5 2) (* 5 2)))
```

followed by the reductions

```
(+      (* 6 6)      (* 10 10))
(+         36           100)
                136
```

This gives the same answer as our previous evaluation model, but the process is different. In particular, the evaluations of `(+ 5 1)` and `(* 5 2)` are each performed twice here, corresponding to the reduction of the expression `(* x x)` with `x` replaced respectively by `(+ 5 1)` and `(* 5 2)`.

This alternative "fully expand and then reduce" evaluation method is known as *normal-order evaluation*, in contrast to the "evaluate the arguments and then apply" method that the interpreter actually uses, which is called *applicative-order evaluation*. It can be shown that, for procedure applications that can be modeled using substitution (including all the procedures in the first two chapters of this book) and that yield legitimate values, normal-order and applicative-order evaluation produce the same value. (See Exercise 1.5 for an instance of an "illegitimate" value where normal-order and applicative-order evaluation do not give the same result.)

Lisp uses applicative-order evaluation, partly because of the additional efficiency obtained from avoiding multiple evaluations of expressions such as those illustrated with `(+ 5 1)` and `(* 5 2)` above and, more

significantly, because normal-order evaluation becomes much more complicated to deal with when we leave the realm of procedures that can be modeled by substitution. On the other hand, normal-order evaluation can be an extremely valuable tool, and we will investigate some of its implications in Chapter 3 and Chapter 4.[16]

1.1.6 Conditional Expressions and Predicates

The expressive power of the class of procedures that we can define at this point is very limited, because we have no way to make tests and to perform different operations depending on the result of a test. For instance, we cannot define a procedure that computes the absolute value of a number by testing whether the number is positive, negative, or zero and taking different actions in the different cases according to the rule

$$|x| = \begin{cases} x & \text{if} \quad x > 0, \\ 0 & \text{if} \quad x = 0, \\ -x & \text{if} \quad x < 0. \end{cases}$$

This construct is called a *case analysis*, and there is a special form in Lisp for notating such a case analysis. It is called `cond` (which stands for "conditional"), and it is used as follows:

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

The general form of a conditional expression is

[16] In Chapter 3 we will introduce *stream processing*, which is a way of handling apparently "infinite" data structures by incorporating a limited form of normal-order evaluation. In Section 4.2 we will modify the Scheme interpreter to produce a normal-order variant of Scheme.

```
(cond (⟨p1⟩ ⟨e1⟩)
      (⟨p2⟩ ⟨e2⟩)
      ...
      (⟨pn⟩ ⟨en⟩))
```

consisting of the symbol `cond` followed by parenthesized pairs of expressions

```
(⟨p⟩ ⟨e⟩)
```

called *clauses*. The first expression in each pair is a *predicate*—that is, an expression whose value is interpreted as either true or false.[17]

Conditional expressions are evaluated as follows. The predicate $\langle p_1 \rangle$ is evaluated first. If its value is false, then $\langle p_2 \rangle$ is evaluated. If $\langle p_2 \rangle$'s value is also false, then $\langle p_3 \rangle$ is evaluated. This process continues until a predicate is found whose value is true, in which case the interpreter returns the value of the corresponding *consequent expression* $\langle e \rangle$ of the clause as the value of the conditional expression. If none of the $\langle p \rangle$'s is found to be true, the value of the `cond` is undefined.

The word *predicate* is used for procedures that return true or false, as well as for expressions that evaluate to true or false. The absolute-value procedure `abs` makes use of the primitive predicates `>`, `<`, and `=`.[18] These take two numbers as arguments and test whether the first number is, respectively, greater than, less than, or equal to the second number, returning true or false accordingly.

Another way to write the absolute-value procedure is

[17] "Interpreted as either true or false" means this: In Scheme, there are two distinguished values that are denoted by the constants `#t` and `#f`. When the interpreter checks a predicate's value, it interprets `#f` as false. Any other value is treated as true. (Thus, providing `#t` is logically unnecessary, but it is convenient.) In this book we will use names `true` and `false`, which are associated with the values `#t` and `#f` respectively.

[18] `abs` also uses the "minus" operator `-`, which, when used with a single operand, as in `(- x)`, indicates negation.

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

which could be expressed in English as "If x is less than zero return $-x$; otherwise return x." `else` is a special symbol that can be used in place of the ⟨*p*⟩ in the final clause of a `cond`. This causes the `cond` to return as its value the value of the corresponding ⟨*e*⟩ whenever all previous clauses have been bypassed. In fact, any expression that always evaluates to a true value could be used as the ⟨*p*⟩ here.

Here is yet another way to write the absolute-value procedure:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

This uses the special form `if`, a restricted type of conditional that can be used when there are precisely two cases in the case analysis. The general form of an `if` expression is

```
(if ⟨predicate⟩ ⟨consequent⟩ ⟨alternative⟩)
```

To evaluate an `if` expression, the interpreter starts by evaluating the ⟨*predicate*⟩ part of the expression. If the ⟨*predicate*⟩ evaluates to a true value, the interpreter then evaluates the ⟨*consequent*⟩ and returns its value. Otherwise it evaluates the ⟨*alternative*⟩ and returns its value.[19]

In addition to primitive predicates such as <, =, and >, there are logical composition operations, which enable us to construct compound

[19] A minor difference between `if` and `cond` is that the ⟨*e*⟩ part of each `cond` clause may be a sequence of expressions. If the corresponding ⟨*p*⟩ is found to be true, the expressions ⟨*e*⟩ are evaluated in sequence and the value of the final expression in the sequence is returned as the value of the `cond`. In an `if` expression, however, the ⟨*consequent*⟩ and ⟨*alternative*⟩ must be single expressions.

predicates. The three most frequently used are these:

- `(and` $\langle e_1 \rangle$ `...` $\langle e_n \rangle$`)`

  The interpreter evaluates the expressions $\langle e \rangle$ one at a time, in left-to-right order. If any $\langle e \rangle$ evaluates to false, the value of the `and` expression is false, and the rest of the $\langle e \rangle$'s are not evaluated. If all $\langle e \rangle$'s evaluate to true values, the value of the `and` expression is the value of the last one.

- `(or` $\langle e_1 \rangle$ `...` $\langle e_n \rangle$`)`

  The interpreter evaluates the expressions $\langle e \rangle$ one at a time, in left-to-right order. If any $\langle e \rangle$ evaluates to a true value, that value is returned as the value of the `or` expression, and the rest of the $\langle e \rangle$'s are not evaluated. If all $\langle e \rangle$'s evaluate to false, the value of the `or` expression is false.

- `(not` $\langle e \rangle$`)`

  The value of a `not` expression is true when the expression $\langle e \rangle$ evaluates to false, and false otherwise.

Notice that `and` and `or` are special forms, not procedures, because the subexpressions are not necessarily all evaluated. `not` is an ordinary procedure.

As an example of how these are used, the condition that a number x be in the range $5 < x < 10$ may be expressed as

```
(and (> x 5) (< x 10))
```

As another example, we can define a predicate to test whether one number is greater than or equal to another as

```
(define (>= x y) (or (> x y) (= x y)))
```

or alternatively as

```
(define (>= x y) (not (< x y)))
```

Exercise 1.1: Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)
(+ (* 2 4) (- 4 6))
(define a 3)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

Exercise 1.2: Translate the following expression into prefix form:

$$\frac{5 + 4 + (2 - (3 - (6 + \frac{4}{5})))}{3(6 - 2)(2 - 7)}.$$

Exercise 1.3: Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Exercise 1.4: Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

Exercise 1.5: Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))
(define (test x y)
  (if (= x 0) 0 y))
```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation

rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

1.1.7 Example: Square Roots by Newton's Method

Procedures, as introduced above, are much like ordinary mathematical functions. They specify a value that is determined by one or more parameters. But there is an important difference between mathematical functions and computer procedures. Procedures must be effective.

As a case in point, consider the problem of computing square roots. We can define the square-root function as

$$\sqrt{x} \;=\; \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 = x.$$

This describes a perfectly legitimate mathematical function. We could use it to recognize whether one number is the square root of another, or to derive facts about square roots in general. On the other hand, the definition does not describe a procedure. Indeed, it tells us almost nothing about how to actually find the square root of a given number. It will not help matters to rephrase this definition in pseudo-Lisp:

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

This only begs the question.

The contrast between function and procedure is a reflection of the general distinction between describing properties of things and describing how to do things, or, as it is sometimes referred to, the distinction

between declarative knowledge and imperative knowledge. In mathematics we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions.[20]

How does one compute square roots? The most common way is to use Newton's method of successive approximations, which says that whenever we have a guess y for the value of the square root of a number x, we can perform a simple manipulation to get a better guess (one closer to the actual square root) by averaging y with x/y.[21] For example, we can compute the square root of 2 as follows. Suppose our initial guess is 1:

```
Guess      Quotient              Average
1          (2/1) = 2             ((2 + 1)/2) = 1.5
1.5        (2/1.5) = 1.3333      ((1.3333 + 1.5)/2) = 1.4167
1.4167     (2/1.4167) = 1.4118   ((1.4167 + 1.4118)/2) = 1.4142
1.4142     ...                   ...
```

[20] Declarative and imperative descriptions are intimately related, as indeed are mathematics and computer science. For instance, to say that the answer produced by a program is "correct" is to make a declarative statement about the program. There is a large amount of research aimed at establishing techniques for proving that programs are correct, and much of the technical difficulty of this subject has to do with negotiating the transition between imperative statements (from which programs are constructed) and declarative statements (which can be used to deduce things). In a related vein, an important current area in programming-language design is the exploration of so-called very high-level languages, in which one actually programs in terms of declarative statements. The idea is to make interpreters sophisticated enough so that, given "what is" knowledge specified by the programmer, they can generate "how to" knowledge automatically. This cannot be done in general, but there are important areas where progress has been made. We shall revisit this idea in Chapter 4.

[21] This square-root algorithm is actually a special case of Newton's method, which is a general technique for finding roots of equations. The square-root algorithm itself was developed by Heron of Alexandria in the first century A.D. We will see how to express the general Newton's method as a Lisp procedure in Section 1.3.4.

Continuing this process, we obtain better and better approximations to the square root.

Now let's formalize the process in terms of procedures. We start with a value for the radicand (the number whose square root we are trying to compute) and a value for the guess. If the guess is good enough for our purposes, we are done; if not, we must repeat the process with an improved guess. We write this basic strategy as a procedure:

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
```

A guess is improved by averaging it with the quotient of the radicand and the old guess:

```
(define (improve guess x)
  (average guess (/ x guess)))
```

where

```
(define (average x y)
  (/ (+ x y) 2))
```

We also have to say what we mean by "good enough." The following will do for illustration, but it is not really a very good test. (See Exercise 1.7.) The idea is to improve the answer until it is close enough so that its square differs from the radicand by less than a predetermined tolerance (here 0.001):[22]

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

[22] We will usually give predicates names ending with question marks, to help us remember that they are predicates. This is just a stylistic convention. As far as the interpreter is concerned, the question mark is just an ordinary character.

Finally, we need a way to get started. For instance, we can always guess that the square root of any number is 1:[23]

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

If we type these definitions to the interpreter, we can use `sqrt` just as we can use any procedure:

```
(sqrt 9)
3.00009155413138

(sqrt (+ 100 37))
11.704699917758145

(sqrt (+ (sqrt 2) (sqrt 3)))
1.7739279023207892

(square (sqrt 1000))
1000.000369924366
```

The `sqrt` program also illustrates that the simple procedural language we have introduced so far is sufficient for writing any purely numerical program that one could write in, say, C or Pascal. This might seem surprising, since we have not included in our language any iterative

[23] Observe that we express our initial guess as 1.0 rather than 1. This would not make any difference in many Lisp implementations. MIT Scheme, however, distinguishes between exact integers and decimal values, and dividing two integers produces a rational number rather than a decimal. For example, dividing 10 by 6 yields 5/3, while dividing 10.0 by 6.0 yields 1.6666666666666667. (We will learn how to implement arithmetic on rational numbers in Section 2.1.1.) If we start with an initial guess of 1 in our square-root program, and x is an exact integer, all subsequent values produced in the square-root computation will be rational numbers rather than decimals. Mixed operations on rational numbers and decimals always yield decimals, so starting with an initial guess of 1.0 forces all subsequent values to be decimals.

(looping) constructs that direct the computer to do something over and over again. `sqrt-iter`, on the other hand, demonstrates how iteration can be accomplished using no special construct other than the ordinary ability to call a procedure.[24]

> **Exercise 1.6:** Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond`?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:
>
> ```
> (define (new-if predicate then-clause else-clause)
>   (cond (predicate then-clause)
>         (else else-clause)))
> ```
>
> Eva demonstrates the program for Alyssa:
>
> ```
> (new-if (= 2 3) 0 5)
> 5
> (new-if (= 1 1) 0 5)
> 0
> ```
>
> Delighted, Alyssa uses `new-if` to rewrite the square-root program:
>
> ```
> (define (sqrt-iter guess x)
>   (new-if (good-enough? guess x)
>           guess
>           (sqrt-iter (improve guess x) x)))
> ```
>
> What happens when Alyssa attempts to use this to compute square roots? Explain.

[24] Readers who are worried about the efficiency issues involved in using procedure calls to implement iteration should note the remarks on "tail recursion" in Section 1.2.1.

Exercise 1.7: The `good-enough?` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how `guess` changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

Exercise 1.8: Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value

$$\frac{x/y^2 + 2y}{3}.$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In Section 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

1.1.8 Procedures as Black-Box Abstractions

`sqrt` is our first example of a process defined by a set of mutually defined procedures. Notice that the definition of `sqrt-iter` is *recursive*; that is, the procedure is defined in terms of itself. The idea of being able to define a procedure in terms of itself may be disturbing; it may seem

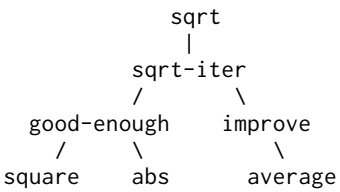

Figure 1.2: Procedural decomposition of the `sqrt` program.

unclear how such a "circular" definition could make sense at all, much less specify a well-defined process to be carried out by a computer. This will be addressed more carefully in Section 1.2. But first let's consider some other important points illustrated by the `sqrt` example.

Observe that the problem of computing square roots breaks up naturally into a number of subproblems: how to tell whether a guess is good enough, how to improve a guess, and so on. Each of these tasks is accomplished by a separate procedure. The entire `sqrt` program can be viewed as a cluster of procedures (shown in Figure 1.2) that mirrors the decomposition of the problem into subproblems.

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, we could take any large program and divide it into parts—the first ten lines, the next ten lines, the next ten lines, and so on. Rather, it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures. For example, when we define the `good-enough?` procedure in terms of `square`, we are able to regard the `square` procedure as a "black box." We are not at that moment concerned with *how* the procedure computes its result, only with the fact that it computes the square. The details of how the square is computed can be suppressed, to be considered at a later time. Indeed, as far as the `good-enough?` pro-

cedure is concerned, `square` is not quite a procedure but rather an abstraction of a procedure, a so-called *procedural abstraction*. At this level of abstraction, any procedure that computes the square is equally good.

Thus, considering only the values they return, the following two procedures for squaring a number should be indistinguishable. Each takes a numerical argument and produces the square of that number as the value.[25]

```
(define (square x) (* x x))
(define (square x) (exp (double (log x))))
(define (double x) (+ x x))
```

So a procedure definition should be able to suppress detail. The users of the procedure may not have written the procedure themselves, but may have obtained it from another programmer as a black box. A user should not need to know how the procedure is implemented in order to use it.

Local names

One detail of a procedure's implementation that should not matter to the user of the procedure is the implementer's choice of names for the procedure's formal parameters. Thus, the following procedures should not be distinguishable:

```
(define (square x) (* x x))
(define (square y) (* y y))
```

[25] It is not even clear which of these procedures is a more efficient implementation. This depends upon the hardware available. There are machines for which the "obvious" implementation is the less efficient one. Consider a machine that has extensive tables of logarithms and antilogarithms stored in a very efficient manner.

This principle—that the meaning of a procedure should be independent of the parameter names used by its author—seems on the surface to be self-evident, but its consequences are profound. The simplest consequence is that the parameter names of a procedure must be local to the body of the procedure. For example, we used `square` in the definition of `good-enough?` in our square-root procedure:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x))
     0.001))
```

The intention of the author of `good-enough?` is to determine if the square of the first argument is within a given tolerance of the second argument. We see that the author of `good-enough?` used the name `guess` to refer to the first argument and `x` to refer to the second argument. The argument of `square` is `guess`. If the author of `square` used `x` (as above) to refer to that argument, we see that the `x` in `good-enough?` must be a different `x` than the one in `square`. Running the procedure `square` must not affect the value of `x` that is used by `good-enough?`, because that value of `x` may be needed by `good-enough?` after `square` is done computing.

If the parameters were not local to the bodies of their respective procedures, then the parameter `x` in `square` could be confused with the parameter `x` in `good-enough?`, and the behavior of `good-enough?` would depend upon which version of `square` we used. Thus, `square` would not be the black box we desired.

A formal parameter of a procedure has a very special role in the procedure definition, in that it doesn't matter what name the formal parameter has. Such a name is called a *bound variable*, and we say that the procedure definition *binds* its formal parameters. The meaning of a procedure definition is unchanged if a bound variable is consistently

renamed throughout the definition.[26] If a variable is not bound, we say that it is *free*. The set of expressions for which a binding defines a name is called the *scope* of that name. In a procedure definition, the bound variables declared as the formal parameters of the procedure have the body of the procedure as their scope.

In the definition of `good-enough?` above, `guess` and `x` are bound variables but `<`, `-`, `abs`, and `square` are free. The meaning of `good-enough?` should be independent of the names we choose for `guess` and `x` so long as they are distinct and different from `<`, `-`, `abs`, and `square`. (If we renamed `guess` to `abs` we would have introduced a bug by *capturing* the variable `abs`. It would have changed from free to bound.) The meaning of `good-enough?` is not independent of the names of its free variables, however. It surely depends upon the fact (external to this definition) that the symbol `abs` names a procedure for computing the absolute value of a number. `good-enough?` will compute a different function if we substitute `cos` for `abs` in its definition.

Internal definitions and block structure

We have one kind of name isolation available to us so far: The formal parameters of a procedure are local to the body of the procedure. The square-root program illustrates another way in which we would like to control the use of names. The existing program consists of separate procedures:

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
```

[26] The concept of consistent renaming is actually subtle and difficult to define formally. Famous logicians have made embarrassing errors here.

```
      guess
      (sqrt-iter (improve guess x) x)))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
(define (improve guess x)
  (average guess (/ x guess)))
```

The problem with this program is that the only procedure that is important to users of `sqrt` is `sqrt`. The other procedures (`sqrt-iter`, `good-enough?`, and `improve`) only clutter up their minds. They may not define any other procedure called `good-enough?` as part of another program to work together with the square-root program, because `sqrt` needs it. The problem is especially severe in the construction of large systems by many separate programmers. For example, in the construction of a large library of numerical procedures, many numerical functions are computed as successive approximations and thus might have procedures named `good-enough?` and `improve` as auxiliary procedures. We would like to localize the subprocedures, hiding them inside `sqrt` so that `sqrt` could coexist with other successive approximations, each having its own private `good-enough?` procedure. To make this possible, we allow a procedure to have internal definitions that are local to that procedure. For example, in the square-root problem we can write

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x) (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

Such nesting of definitions, called *block structure*, is basically the right solution to the simplest name-packaging problem. But there is a better idea lurking here. In addition to internalizing the definitions of the auxiliary procedures, we can simplify them. Since `x` is bound in the definition of `sqrt`, the procedures `good-enough?`, `improve`, and `sqrt-iter`, which are defined internally to `sqrt`, are in the scope of `x`. Thus, it is not necessary to pass `x` explicitly to each of these procedures. Instead, we allow `x` to be a free variable in the internal definitions, as shown below. Then `x` gets its value from the argument with which the enclosing procedure `sqrt` is called. This discipline is called *lexical scoping*.[27]

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

We will use block structure extensively to help us break up large programs into tractable pieces.[28] The idea of block structure originated with the programming language Algol 60. It appears in most advanced programming languages and is an important tool for helping to organize the construction of large programs.

[27] Lexical scoping dictates that free variables in a procedure are taken to refer to bindings made by enclosing procedure definitions; that is, they are looked up in the environment in which the procedure was defined. We will see how this works in detail in chapter 3 when we study environments and the detailed behavior of the interpreter.

[28] Embedded definitions must come first in a procedure body. The management is not responsible for the consequences of running programs that intertwine definition and use.

1.2 Procedures and the Processes They Generate

We have now considered the elements of programming: We have used primitive arithmetic operations, we have combined these operations, and we have abstracted these composite operations by defining them as compound procedures. But that is not enough to enable us to say that we know how to program. Our situation is analogous to that of someone who has learned the rules for how the pieces move in chess but knows nothing of typical openings, tactics, or strategy. Like the novice chess player, we don't yet know the common patterns of usage in the domain. We lack the knowledge of which moves are worth making (which procedures are worth defining). We lack the experience to predict the consequences of making a move (executing a procedure).

The ability to visualize the consequences of the actions under consideration is crucial to becoming an expert programmer, just as it is in any synthetic, creative activity. In becoming an expert photographer, for example, one must learn how to look at a scene and know how dark each region will appear on a print for each possible choice of exposure and development conditions. Only then can one reason backward, planning framing, lighting, exposure, and development to obtain the desired effects. So it is with programming, where we are planning the course of action to be taken by a process and where we control the process by means of a program. To become experts, we must learn to visualize the processes generated by various types of procedures. Only after we have developed such a skill can we learn to reliably construct programs that exhibit the desired behavior.

A procedure is a pattern for the *local evolution* of a computational process. It specifies how each stage of the process is built upon the previous stage. We would like to be able to make statements about the overall,

or *global*, behavior of a process whose local evolution has been specified by a procedure. This is very difficult to do in general, but we can at least try to describe some typical patterns of process evolution.

In this section we will examine some common "shapes" for processes generated by simple procedures. We will also investigate the rates at which these processes consume the important computational resources of time and space. The procedures we will consider are very simple. Their role is like that played by test patterns in photography: as oversimplified prototypical patterns, rather than practical examples in their own right.

1.2.1 Linear Recursion and Iteration

We begin by considering the factorial function, defined by

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1.$$

There are many ways to compute factorials. One way is to make use of the observation that $n!$ is equal to n times $(n-1)!$ for any positive integer n:

$$n! = n \cdot [(n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n-1)!.$$

Thus, we can compute $n!$ by computing $(n-1)!$ and multiplying the result by n. If we add the stipulation that 1! is equal to 1, this observation translates directly into a procedure:

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

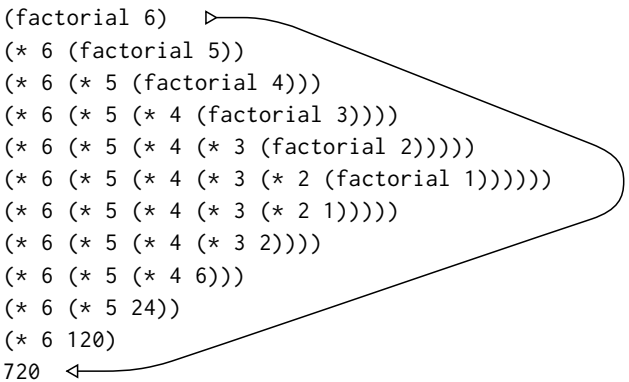

Figure 1.3: A linear recursive process for computing 6!.

We can use the substitution model of Section 1.1.5 to watch this procedure in action computing 6!, as shown in Figure 1.3.

Now let's take a different perspective on computing factorials. We could describe a rule for computing $n!$ by specifying that we first multiply 1 by 2, then multiply the result by 3, then by 4, and so on until we reach n. More formally, we maintain a running product, together with a counter that counts from 1 up to n. We can describe the computation by saying that the counter and the product simultaneously change from one step to the next according to the rule

```
product ← counter * product
counter ← counter + 1
```

and stipulating that $n!$ is the value of the product when the counter exceeds n.

Once again, we can recast our description as a procedure for computing factorials:[29]

[29] In a real program we would probably use the block structure introduced in the last section to hide the definition of `fact-iter`:

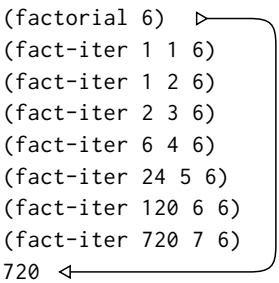

Figure 1.4: A linear iterative process for computing 6!.

```
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

As before, we can use the substitution model to visualize the process of computing 6!, as shown in Figure 1.4.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

We avoided doing this here so as to minimize the number of things to think about at once.

Compare the two processes. From one point of view, they seem hardly different at all. Both compute the same mathematical function on the same domain, and each requires a number of steps proportional to n to compute $n!$. Indeed, both processes even carry out the same sequence of multiplications, obtaining the same sequence of partial products. On the other hand, when we consider the "shapes" of the two processes, we find that they evolve quite differently.

Consider the first process. The substitution model reveals a shape of expansion followed by contraction, indicated by the arrow in Figure 1.3. The expansion occurs as the process builds up a chain of *deferred operations* (in this case, a chain of multiplications). The contraction occurs as the operations are actually performed. This type of process, characterized by a chain of deferred operations, is called a *recursive process*. Carrying out this process requires that the interpreter keep track of the operations to be performed later on. In the computation of $n!$, the length of the chain of deferred multiplications, and hence the amount of information needed to keep track of it, grows linearly with n (is proportional to n), just like the number of steps. Such a process is called a *linear recursive process*.

By contrast, the second process does not grow and shrink. At each step, all we need to keep track of, for any n, are the current values of the variables `product`, `counter`, and `max-count`. We call this an *iterative process*. In general, an iterative process is one whose state can be summarized by a fixed number of *state variables*, together with a fixed rule that describes how the state variables should be updated as the process moves from state to state and an (optional) end test that specifies conditions under which the process should terminate. In computing $n!$, the number of steps required grows linearly with n. Such a process is called a *linear iterative process*.

The contrast between the two processes can be seen in another way. In the iterative case, the program variables provide a complete description of the state of the process at any point. If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables. Not so with the recursive process. In this case there is some additional "hidden" information, maintained by the interpreter and not contained in the program variables, which indicates "where the process is" in negotiating the chain of deferred operations. The longer the chain, the more information must be maintained.[30]

In contrasting iteration and recursion, we must be careful not to confuse the notion of a recursive *process* with the notion of a recursive *procedure*. When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers (either directly or indirectly) to the procedure itself. But when we describe a process as following a pattern that is, say, linearly recursive, we are speaking about how the process evolves, not about the syntax of how a procedure is written. It may seem disturbing that we refer to a recursive procedure such as `fact-iter` as generating an iterative process. However, the process really is iterative: Its state is captured completely by its three state variables, and an interpreter need keep track of only three variables in order to execute the process.

One reason that the distinction between process and procedure may be confusing is that most implementations of common languages (including Ada, Pascal, and C) are designed in such a way that the interpretation of any recursive procedure consumes an amount of memory that

[30] When we discuss the implementation of procedures on register machines in Chapter 5, we will see that any iterative process can be realized "in hardware" as a machine that has a fixed set of registers and no auxiliary memory. In contrast, realizing a recursive process requires a machine that uses an auxiliary data structure known as a *stack*.

grows with the number of procedure calls, even when the process described is, in principle, iterative. As a consequence, these languages can describe iterative processes only by resorting to special-purpose "looping constructs" such as `do`, `repeat`, `until`, `for`, and `while`. The implementation of Scheme we shall consider in Chapter 5 does not share this defect. It will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure. An implementation with this property is called *tail-recursive*. With a tail-recursive implementation, iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar.[31]

> **Exercise 1.9:** Each of the following two procedures defines a method for adding two positive integers in terms of the procedures `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.
>
> ```
> (define (+ a b)
>   (if (= a 0) b (inc (+ (dec a) b))))
> (define (+ a b)
>   (if (= a 0) b (+ (dec a) (inc b))))
> ```
>
> Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

[31] Tail recursion has long been known as a compiler optimization trick. A coherent semantic basis for tail recursion was provided by Carl Hewitt (1977), who explained it in terms of the "message-passing" model of computation that we shall discuss in Chapter 3. Inspired by this, Gerald Jay Sussman and Guy Lewis Steele Jr. (see Steele and Sussman 1975) constructed a tail-recursive interpreter for Scheme. Steele later showed how tail recursion is a consequence of the natural way to compile procedure calls (Steele 1977). The IEEE standard for Scheme requires that Scheme implementations be tail-recursive.

Exercise 1.10: The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1) (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)
(A 2 4)
(A 3 3)
```

Consider the following procedures, where `A` is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures `f`, `g`, and `h` for positive integer values of n. For example, `(k n)` computes $5n^2$.

1.2.2 Tree Recursion

Another common pattern of computation is called *tree recursion*. As an example, consider computing the sequence of Fibonacci numbers, in which each number is the sum of the preceding two:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots.$$

In general, the Fibonacci numbers can be defined by the rule

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise.} \end{cases}$$

We can immediately translate this definition into a recursive procedure for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Consider the pattern of this computation. To compute `(fib 5)`, we compute `(fib 4)` and `(fib 3)`. To compute `(fib 4)`, we compute `(fib 3)` and `(fib 2)`. In general, the evolved process looks like a tree, as shown in Figure 1.5. Notice that the branches split into two at each level (except at the bottom); this reflects the fact that the `fib` procedure calls itself twice each time it is invoked.

This procedure is instructive as a prototypical tree recursion, but it is a terrible way to compute Fibonacci numbers because it does so much redundant computation. Notice in Figure 1.5 that the entire computation of `(fib 3)`—almost half the work—is duplicated. In fact, it is not hard to show that the number of times the procedure will compute `(fib 1)` or `(fib 0)` (the number of leaves in the above tree, in general) is precisely $\text{Fib}(n+1)$. To get an idea of how bad this is, one can show that the value of $\text{Fib}(n)$ grows exponentially with n. More precisely (see Exercise 1.13), $\text{Fib}(n)$ is the closest integer to $\varphi^n/\sqrt{5}$, where

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180$$

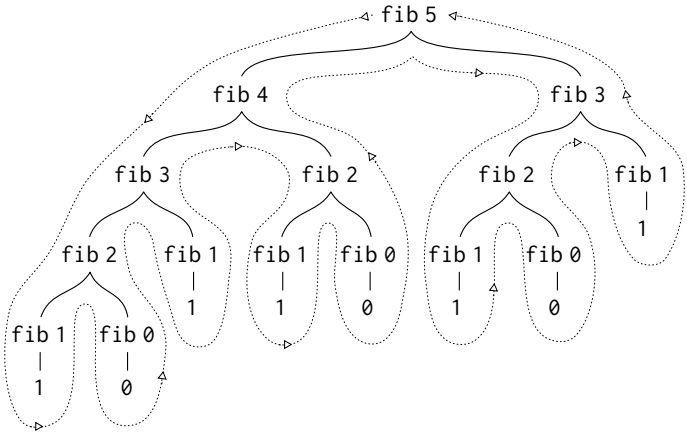

Figure 1.5: The tree-recursive process generated in computing `(fib 5)`.

is the *golden ratio*, which satisfies the equation

$$\varphi^2 = \varphi + 1.$$

Thus, the process uses a number of steps that grows exponentially with the input. On the other hand, the space required grows only linearly with the input, because we need keep track only of which nodes are above us in the tree at any point in the computation. In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

We can also formulate an iterative process for computing the Fibonacci numbers. The idea is to use a pair of integers a and b, initialized to Fib(1) = 1 and Fib(0) = 0, and to repeatedly apply the simultaneous

transformations

$$a \leftarrow a + b,$$
$$b \leftarrow a.$$

It is not hard to show that, after applying this transformation n times, a and b will be equal, respectively, to Fib(n + 1) and Fib(n). Thus, we can compute Fibonacci numbers iteratively using the procedure

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

This second method for computing Fib(n) is a linear iteration. The difference in number of steps required by the two methods—one linear in n, one growing as fast as Fib(n) itself—is enormous, even for small inputs.

One should not conclude from this that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool.[32] But even in numerical operations, tree-recursive processes can be useful in helping us to understand and design programs. For instance, although the first `fib` procedure is much less efficient than the second one, it is more straightforward, being little more than a translation into Lisp of the definition of the Fibonacci sequence. To formulate the iterative algorithm required noticing that the computation could be recast as an iteration with three state variables.

[32] An example of this was hinted at in Section 1.1.3. The interpreter itself evaluates expressions using a tree-recursive process.

Example: Counting change

It takes only a bit of cleverness to come up with the iterative Fibonacci algorithm. In contrast, consider the following problem: How many different ways can we make change of $1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a procedure to compute the number of ways to change any given amount of money?

This problem has a simple solution as a recursive procedure. Suppose we think of the types of coins available as arranged in some order. Then the following relation holds:

The number of ways to change amount a using n kinds of coins equals

- the number of ways to change amount a using all but the first kind of coin, plus
- the number of ways to change amount $a - d$ using all n kinds of coins, where d is the denomination of the first kind of coin.

To see why this is true, observe that the ways to make change can be divided into two groups: those that do not use any of the first kind of coin, and those that do. Therefore, the total number of ways to make change for some amount is equal to the number of ways to make change for the amount without using any of the first kind of coin, plus the number of ways to make change assuming that we do use the first kind of coin. But the latter number is equal to the number of ways to make change for the amount that remains after using a coin of the first kind.

Thus, we can recursively reduce the problem of changing a given amount to the problem of changing smaller amounts using fewer kinds of coins. Consider this reduction rule carefully, and convince yourself

that we can use it to describe an algorithm if we specify the following degenerate cases:[33]

- If a is exactly 0, we should count that as 1 way to make change.
- If a is less than 0, we should count that as 0 ways to make change.
- If n is 0, we should count that as 0 ways to make change.

We can easily translate this description into a recursive procedure:

```
(define (count-change amount) (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination
                         kinds-of-coins))
                     kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

(The `first-denomination` procedure takes as input the number of kinds of coins available and returns the denomination of the first kind. Here we are thinking of the coins as arranged in order from largest to smallest, but any order would do as well.) We can now answer our original question about changing a dollar:

[33] For example, work through in detail how the reduction rule applies to the problem of making change for 10 cents using pennies and nickels.

```
(count-change 100)
292
```

`count-change` generates a tree-recursive process with redundancies similar to those in our first implementation of `fib`. (It will take quite a while for that 292 to be computed.) On the other hand, it is not obvious how to design a better algorithm for computing the result, and we leave this problem as a challenge. The observation that a tree-recursive process may be highly inefficient but often easy to specify and understand has led people to propose that one could get the best of both worlds by designing a "smart compiler" that could transform tree-recursive procedures into more efficient procedures that compute the same result.[34]

Exercise 1.11: A function f is defined by the rule that

$$f(n) = \begin{cases} n & \text{if } n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{if } n \geq 3. \end{cases}$$

Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

Exercise 1.12: The following pattern of numbers is called *Pascal's triangle*.

[34] One approach to coping with redundant computations is to arrange matters so that we automatically construct a table of values as they are computed. Each time we are asked to apply the procedure to some argument, we first look to see if the value is already stored in the table, in which case we avoid performing the redundant computation. This strategy, known as *tabulation* or *memoization*, can be implemented in a straightforward way. Tabulation can sometimes be used to transform processes that require an exponential number of steps (such as `count-change`) into processes whose space and time requirements grow linearly with the input. See Exercise 3.27.

```
            1
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
          . . .
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.[35] Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

Exercise 1.13: Prove that Fib(n) is the closest integer to $\varphi^n/\sqrt{5}$, where $\varphi = (1 + \sqrt{5})/2$. Hint: Let $\psi = (1 - \sqrt{5})/2$. Use induction and the definition of the Fibonacci numbers (see Section 1.2.2) to prove that $\text{Fib}(n) = (\varphi^n - \psi^n)/\sqrt{5}$.

1.2.3 Orders of Growth

The previous examples illustrate that processes can differ considerably in the rates at which they consume computational resources. One convenient way to describe this difference is to use the notion of *order of growth* to obtain a gross measure of the resources required by a process as the inputs become larger.

[35] The elements of Pascal's triangle are called the *binomial coefficients*, because the n^{th} row consists of the coefficients of the terms in the expansion of $(x + y)^n$. This pattern for computing the coefficients appeared in Blaise Pascal's 1653 seminal work on probability theory, *Traité du triangle arithmétique*. According to Knuth (1973), the same pattern appears in the *Szu-yuen Yü-chien* ("The Precious Mirror of the Four Elements"), published by the Chinese mathematician Chu Shih-chieh in 1303, in the works of the twelfth-century Persian poet and mathematician Omar Khayyam, and in the works of the twelfth-century Hindu mathematician Bháscara Áchárya.

Let n be a parameter that measures the size of the problem, and let $R(n)$ be the amount of resources the process requires for a problem of size n. In our previous examples we took n to be the number for which a given function is to be computed, but there are other possibilities. For instance, if our goal is to compute an approximation to the square root of a number, we might take n to be the number of digits accuracy required. For matrix multiplication we might take n to be the number of rows in the matrices. In general there are a number of properties of the problem with respect to which it will be desirable to analyze a given process. Similarly, $R(n)$ might measure the number of internal storage registers used, the number of elementary machine operations performed, and so on. In computers that do only a fixed number of operations at a time, the time required will be proportional to the number of elementary machine operations performed.

We say that $R(n)$ has order of growth $\Theta(f(n))$, written $R(n) = \Theta(f(n))$ (pronounced "theta of $f(n)$"), if there are positive constants k_1 and k_2 independent of n such that $k_1 f(n) \leq R(n) \leq k_2 f(n)$ for any sufficiently large value of n. (In other words, for large n, the value $R(n)$ is sandwiched between $k_1 f(n)$ and $k_2 f(n)$.)

For instance, with the linear recursive process for computing factorial described in Section 1.2.1 the number of steps grows proportionally to the input n. Thus, the steps required for this process grows as $\Theta(n)$. We also saw that the space required grows as $\Theta(n)$. For the iterative factorial, the number of steps is still $\Theta(n)$ but the space is $\Theta(1)$—that is, constant.[36] The tree-recursive Fibonacci computation requires $\Theta(\varphi^n)$

[36] These statements mask a great deal of oversimplification. For instance, if we count process steps as "machine operations" we are making the assumption that the number of machine operations needed to perform, say, a multiplication is independent of the size of the numbers to be multiplied, which is false if the numbers are sufficiently large. Similar remarks hold for the estimates of space. Like the design and description of a process, the analysis of a process can be carried out at various levels of abstraction.

steps and space $\Theta(n)$, where φ is the golden ratio described in Section 1.2.2.

Orders of growth provide only a crude description of the behavior of a process. For example, a process requiring n^2 steps and a process requiring $1000n^2$ steps and a process requiring $3n^2 + 10n + 17$ steps all have $\Theta(n^2)$ order of growth. On the other hand, order of growth provides a useful indication of how we may expect the behavior of the process to change as we change the size of the problem. For a $\Theta(n)$ (linear) process, doubling the size will roughly double the amount of resources used. For an exponential process, each increment in problem size will multiply the resource utilization by a constant factor. In the remainder of Section 1.2 we will examine two algorithms whose order of growth is logarithmic, so that doubling the problem size increases the resource requirement by a constant amount.

Exercise 1.14: Draw the tree illustrating the process generated by the `count-change` procedure of Section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

Exercise 1.15: The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin x \approx x$ if x is sufficiently small, and the trigonometric identity

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$$

to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
```

a. How many times is the procedure p applied when `(sine 12.15)` is evaluated?

b. What is the order of growth in space and number of steps (as a function of a) used by the process generated by the `sine` procedure when `(sine a)` is evaluated?

1.2.4 Exponentiation

Consider the problem of computing the exponential of a given number. We would like a procedure that takes as arguments a base b and a positive integer exponent n and computes b^n. One way to do this is via the recursive definition

$$b^n = b \cdot b^{n-1},$$
$$b^0 = 1,$$

which translates readily into the procedure

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

This is a linear recursive process, which requires $\Theta(n)$ steps and $\Theta(n)$ space. Just as with factorial, we can readily formulate an equivalent linear iteration:

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                 (- counter 1)
                 (* b product))))
```

This version requires $\Theta(n)$ steps and $\Theta(1)$ space.

We can compute exponentials in fewer steps by using successive squaring. For instance, rather than computing b^8 as

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))\,,$$

we can compute it using three multiplications:

$$\begin{aligned} b^2 &= b \cdot b, \\ b^4 &= b^2 \cdot b^2, \\ b^8 &= b^4 \cdot b^4. \end{aligned}$$

This method works fine for exponents that are powers of 2. We can also take advantage of successive squaring in computing exponentials in general if we use the rule

$$\begin{aligned} b^n &= (b^{n/2})^2 \qquad \text{if } n \text{ is even}, \\ b^n &= b \cdot b^{n-1} \qquad \text{if } n \text{ is odd}. \end{aligned}$$

We can express this method as a procedure:

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

where the predicate to test whether an integer is even is defined in terms of the primitive procedure `remainder` by

```
(define (even? n)
  (= (remainder n 2) 0))
```

The process evolved by `fast-expt` grows logarithmically with n in both space and number of steps. To see this, observe that computing b^{2n} using `fast-expt` requires only one more multiplication than computing b^n. The size of the exponent we can compute therefore doubles (approximately) with every new multiplication we are allowed. Thus, the number of multiplications required for an exponent of n grows about as fast as the logarithm of n to the base 2. The process has $\Theta(\log n)$ growth.[37]

The difference between $\Theta(\log n)$ growth and $\Theta(n)$ growth becomes striking as n becomes large. For example, `fast-expt` for $n = 1000$ requires only 14 multiplications.[38] It is also possible to use the idea of successive squaring to devise an iterative algorithm that computes exponentials with a logarithmic number of steps (see Exercise 1.16), although, as is often the case with iterative algorithms, this is not written down so straightforwardly as the recursive algorithm.[39]

> **Exercise 1.16:** Design a procedure that evolves an iterative exponentiation process that uses successive squaring

[37] More precisely, the number of multiplications required is equal to 1 less than the log base 2 of n plus the number of ones in the binary representation of n. This total is always less than twice the log base 2 of n. The arbitrary constants k_1 and k_2 in the definition of order notation imply that, for a logarithmic process, the base to which logarithms are taken does not matter, so all such processes are described as $\Theta(\log n)$.

[38] You may wonder why anyone would care about raising numbers to the 1000th power. See Section 1.2.6.

[39] This iterative algorithm is ancient. It appears in the *Chandah-sutra* by Áchárya Pingala, written before 200 B.C. See Knuth 1981, section 4.6.3, for a full discussion and analysis of this and other methods of exponentiation.

and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

Exercise 1.17: The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in `b`. Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to `fast-expt` that uses a logarithmic number of steps.

Exercise 1.18: Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.[40]

Exercise 1.19: There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the `fib-iter` process of Section 1.2.2: $a \leftarrow a+b$ and $b \leftarrow a$. Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $\mathrm{Fib}(n+1)$ and $\mathrm{Fib}(n)$. In other words, the Fibonacci numbers are produced by applying T^n, the n^{th} power of the transformation T, starting with the pair $(1,0)$. Now consider T to be the special case of $p=0$ and $q=1$ in a family of transformations T_{pq}, where T_{pq} transforms the pair (a,b) according to $a \leftarrow bq + aq + ap$ and $b \leftarrow bp + aq$. Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring, as in the `fast-expt` procedure. Put this all together to complete the following procedure, which runs in a logarithmic number of steps:[41]

[40] This algorithm, which is sometimes known as the "Russian peasant method" of multiplication, is ancient. Examples of its use are found in the Rhind Papyrus, one of the two oldest mathematical documents in existence, written about 1700 B.C. (and copied from an even older document) by an Egyptian scribe named A'h-mose.

[41] This exercise was suggested to us by Joe Stoy, based on an example in Kaldewaij 1990.

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   ⟨??⟩    ; compute p′
                   ⟨??⟩    ; compute q′
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.2.5 Greatest Common Divisors

The greatest common divisor (GCD) of two integers a and b is defined to be the largest integer that divides both a and b with no remainder. For example, the GCD of 16 and 28 is 4. In Chapter 2, when we investigate how to implement rational-number arithmetic, we will need to be able to compute GCDs in order to reduce rational numbers to lowest terms. (To reduce a rational number to lowest terms, we must divide both the numerator and the denominator by their GCD. For example, 16/28 reduces to 4/7.) One way to find the GCD of two integers is to factor them and search for common factors, but there is a famous algorithm that is much more efficient.

The idea of the algorithm is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are precisely the same as the common divisors of b and r. Thus, we can

use the equation

```
GCD(a,b) = GCD(b,r)
```

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

```
GCD(206,40) = GCD(40,6)
            = GCD(6,4)
            = GCD(4,2)
            = GCD(2,0)
            = 2
```

reduces GCD(206, 40) to GCD(2, 0), which is 2. It is possible to show that starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair. This method for computing the GCD is known as *Euclid's Algorithm*.[42]

It is easy to express Euclid's Algorithm as a procedure:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

This generates an iterative process, whose number of steps grows as the logarithm of the numbers involved.

[42]Euclid's Algorithm is so called because it appears in Euclid's *Elements* (Book 7, ca. 300 B.C.). According to Knuth (1973), it can be considered the oldest known nontrivial algorithm. The ancient Egyptian method of multiplication (Exercise 1.18) is surely older, but, as Knuth explains, Euclid's algorithm is the oldest known to have been presented as a general algorithm, rather than as a set of illustrative examples.

The fact that the number of steps required by Euclid's Algorithm has logarithmic growth bears an interesting relation to the Fibonacci numbers:

> **Lamé's Theorem:** If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the k^{th} Fibonacci number.[43]

We can use this theorem to get an order-of-growth estimate for Euclid's Algorithm. Let n be the smaller of the two inputs to the procedure. If the process takes k steps, then we must have $n \geq \text{Fib}(k) \approx \varphi^k / \sqrt{5}$. Therefore the number of steps k grows as the logarithm (to the base φ) of n. Hence, the order of growth is $\Theta(\log n)$.

[43] This theorem was proved in 1845 by Gabriel Lamé, a French mathematician and engineer known chiefly for his contributions to mathematical physics. To prove the theorem, we consider pairs (a_k, b_k), where $a_k \geq b_k$, for which Euclid's Algorithm terminates in k steps. The proof is based on the claim that, if $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ are three successive pairs in the reduction process, then we must have $b_{k+1} \geq b_k + b_{k-1}$. To verify the claim, consider that a reduction step is defined by applying the transformation $a_{k-1} = b_k, b_{k-1} =$ remainder of a_k divided by b_k. The second equation means that $a_k = qb_k + b_{k-1}$ for some positive integer q. And since q must be at least 1 we have $a_k = qb_k + b_{k-1} \geq b_k + b_{k-1}$. But in the previous reduction step we have $b_{k+1} = a_k$. Therefore, $b_{k+1} = a_k \geq b_k + b_{k-1}$. This verifies the claim. Now we can prove the theorem by induction on k, the number of steps that the algorithm requires to terminate. The result is true for $k = 1$, since this merely requires that b be at least as large as $\text{Fib}(1) = 1$. Now, assume that the result is true for all integers less than or equal to k and establish the result for $k + 1$. Let $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$ be successive pairs in the reduction process. By our induction hypotheses, we have $b_{k-1} \geq \text{Fib}(k - 1)$ and $b_k \geq \text{Fib}(k)$. Thus, applying the claim we just proved together with the definition of the Fibonacci numbers gives $b_{k+1} \geq b_k + b_{k-1} \geq \text{Fib}(k) + \text{Fib}(k - 1) = \text{Fib}(k + 1)$, which completes the proof of Lamé's Theorem.

Exercise 1.20: The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in Section 1.1.5. (The normal-order-evaluation rule for `if` is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating `(gcd 206 40)` and indicate the `remainder` operations that are actually performed. How many `remainder` operations are actually performed in the normal-order evaluation of `(gcd 206 40)`? In the applicative-order evaluation?

1.2.6 Example: Testing for Primality

This section describes two methods for checking the primality of an integer n, one with order of growth $\Theta(\sqrt{n})$, and a "probabilistic" algorithm with order of growth $\Theta(\log n)$. The exercises at the end of this section suggest programming projects based on these algorithms.

Searching for divisors

Since ancient times, mathematicians have been fascinated by problems concerning prime numbers, and many people have worked on the problem of determining ways to test if numbers are prime. One way to test if a number is prime is to find the number's divisors. The following program finds the smallest integral divisor (greater than 1) of a given number n. It does this in a straightforward way, by testing n for divisibility by successive integers starting with 2.

```
(define (smallest-divisor n) (find-divisor n 2))
```

```
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b) (= (remainder b a) 0))
```

We can test whether a number is prime as follows: n is prime if and only if n is its own smallest divisor.

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

The end test for `find-divisor` is based on the fact that if n is not prime it must have a divisor less than or equal to $\sqrt{n}$.[44] This means that the algorithm need only test divisors between 1 and $\sqrt{n}$. Consequently, the number of steps required to identify n as prime will have order of growth $\Theta(\sqrt{n})$.

The Fermat test

The $\Theta(\log n)$ primality test is based on a result from number theory known as Fermat's Little Theorem.[45]

[44] If d is a divisor of n, then so is n/d. But d and n/d cannot both be greater than $\sqrt{n}$.

[45] Pierre de Fermat (1601-1665) is considered to be the founder of modern number theory. He obtained many important number-theoretic results, but he usually announced just the results, without providing his proofs. Fermat's Little Theorem was stated in a letter he wrote in 1640. The first published proof was given by Euler in 1736 (and an earlier, identical proof was discovered in the unpublished manuscripts of Leibniz). The most famous of Fermat's results—known as Fermat's Last Theorem—was jotted down in 1637 in his copy of the book *Arithmetic* (by the third-century Greek mathematician Diophantus) with the remark "I have discovered a truly remarkable proof, but this margin is too small to contain it." Finding a proof of Fermat's Last Theorem became one of the most famous challenges in number theory. A complete solution was finally given in 1995 by Andrew Wiles of Princeton University.

> **Fermat's Little Theorem:** If n is a prime number and a is any positive integer less than n, then a raised to the n^{th} power is congruent to a modulo n.

(Two numbers are said to be *congruent modulo n* if they both have the same remainder when divided by n. The remainder of a number a when divided by n is also referred to as the *remainder of a modulo n*, or simply as *a modulo n*.)

If n is not prime, then, in general, most of the numbers $a < n$ will not satisfy the above relation. This leads to the following algorithm for testing primality: Given a number n, pick a random number $a < n$ and compute the remainder of a^n modulo n. If the result is not equal to a, then n is certainly not prime. If it is a, then chances are good that n is prime. Now pick another random number a and test it with the same method. If it also satisfies the equation, then we can be even more confident that n is prime. By trying more and more values of a, we can increase our confidence in the result. This algorithm is known as the Fermat test.

To implement the Fermat test, we need a procedure that computes the exponential of a number modulo another number:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m))))
```

This is very similar to the `fast-expt` procedure of Section 1.2.4. It uses successive squaring, so that the number of steps grows logarithmically with the exponent.[46]

The Fermat test is performed by choosing at random a number a between 1 and $n-1$ inclusive and checking whether the remainder modulo n of the n^{th} power of a is equal to a. The random number a is chosen using the procedure `random`, which we assume is included as a primitive in Scheme. `random` returns a nonnegative integer less than its integer input. Hence, to obtain a random number between 1 and $n-1$, we call `random` with an input of $n-1$ and add 1 to the result:

```
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

The following procedure runs the test a given number of times, as specified by a parameter. Its value is true if the test succeeds every time, and false otherwise.

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

[46] The reduction steps in the cases where the exponent e is greater than 1 are based on the fact that, for any integers x, y, and m, we can find the remainder of x times y modulo m by computing separately the remainders of x modulo m and y modulo m, multiplying these, and then taking the remainder of the result modulo m. For instance, in the case where e is even, we compute the remainder of $b^{e/2}$ modulo m, square this, and take the remainder modulo m. This technique is useful because it means we can perform our computation without ever having to deal with numbers much larger than m. (Compare Exercise 1.25.)

Probabilistic methods

The Fermat test differs in character from most familiar algorithms, in which one computes an answer that is guaranteed to be correct. Here, the answer obtained is only probably correct. More precisely, if n ever fails the Fermat test, we can be certain that n is not prime. But the fact that n passes the test, while an extremely strong indication, is still not a guarantee that n is prime. What we would like to say is that for any number n, if we perform the test enough times and find that n always passes the test, then the probability of error in our primality test can be made as small as we like.

Unfortunately, this assertion is not quite correct. There do exist numbers that fool the Fermat test: numbers n that are not prime and yet have the property that a^n is congruent to a modulo n for all integers $a < n$. Such numbers are extremely rare, so the Fermat test is quite reliable in practice.[47]

There are variations of the Fermat test that cannot be fooled. In these tests, as with the Fermat method, one tests the primality of an integer n by choosing a random integer $a < n$ and checking some condition that depends upon n and a. (See Exercise 1.28 for an example of such a test.) On the other hand, in contrast to the Fermat test, one can prove that, for any n, the condition does not hold for most of the integers $a < n$ unless n is prime. Thus, if n passes the test for some random choice of

[47] Numbers that fool the Fermat test are called *Carmichael numbers*, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

a, the chances are better than even that n is prime. If n passes the test for two random choices of a, the chances are better than 3 out of 4 that n is prime. By running the test with more and more randomly chosen values of a we can make the probability of error as small as we like.

The existence of tests for which one can prove that the chance of error becomes arbitrarily small has sparked interest in algorithms of this type, which have come to be known as *probabilistic algorithms*. There is a great deal of research activity in this area, and probabilistic algorithms have been fruitfully applied to many fields.[48]

> **Exercise 1.21:** Use the `smallest-divisor` procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

> **Exercise 1.22:** Most Lisp implementations include a primitive called `runtime` that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following `timed-prime-test` procedure, when called with an integer n, prints n and checks to see if n is prime. If n is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

[48] One of the most striking applications of probabilistic prime testing has been to the field of cryptography. Although it is now computationally infeasible to factor an arbitrary 200-digit number, the primality of such a number can be checked in a few seconds with the Fermat test. This fact forms the basis of a technique for constructing "unbreakable codes" suggested by Rivest et al. (1977). The resulting *RSA algorithm* has become a widely used technique for enhancing the security of electronic communications. Because of this and related developments, the study of prime numbers, once considered the epitome of a topic in "pure" mathematics to be studied only for its own sake, now turns out to have important practical applications to cryptography, electronic funds transfer, and information retrieval.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime) start-time))))
(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

Using this procedure, write a procedure `search-for-primes` that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of $\Theta(\sqrt{n})$, you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the $\Theta(\sqrt{n})$ prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

Exercise 1.23: The `smallest-divisor` procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for `test-divisor` should not be 2, 3, 4, 5, 6, ..., but rather 2, 3, 5, 7, 9, ...

To implement this change, define a procedure `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest-divisor` procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With `timed-prime-test` incorporating this modified version of `smallest-divisor`, run the test for each of the 12 primes found in Exercise 1.22. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

Exercise 1.24: Modify the `timed-prime-test` procedure of Exercise 1.22 to use `fast-prime?` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has $\Theta(\log n)$ growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

Exercise 1.25: Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

Exercise 1.26: Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the `expmod` procedure to use an explicit multiplication, rather than calling `square`:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                       (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base
                       (expmod base (- exp 1) m))
                    m))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

Exercise 1.27: Demonstrate that the Carmichael numbers listed in Footnote 1.47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

Exercise 1.28: One variant of the Fermat test that cannot be fooled is called the *Miller-Rabin test* (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little

Theorem, which states that if n is a prime number and a is any positive integer less than n, then a raised to the $(n-1)$-st power is congruent to 1 modulo n. To test the primality of a number n by the Miller-Rabin test, we pick a random number $a < n$ and raise a to the $(n-1)$-st power modulo n using the `expmod` procedure. However, whenever we perform the squaring step in `expmod`, we check to see if we have discovered a "nontrivial square root of 1 modulo n," that is, a number not equal to 1 or $n-1$ whose square is equal to 1 modulo n. It is possible to prove that if such a nontrivial square root of 1 exists, then n is not prime. It is also possible to prove that if n is an odd number that is not prime, then, for at least half the numbers $a < n$, computing a^{n-1} in this way will reveal a nontrivial square root of 1 modulo n. (This is why the Miller-Rabin test cannot be fooled.) Modify the `expmod` procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

1.3 Formulating Abstractions with Higher-Order Procedures

We have seen that procedures are, in effect, abstractions that describe compound operations on numbers independent of the particular numbers. For example, when we

```
(define (cube x) (* x x x))
```

we are not talking about the cube of a particular number, but rather about a method for obtaining the cube of any number. Of course we could get along without ever defining this procedure, by always writing expressions such as

```
(* 3 3 3)
(* x x x)
(* y y y)
```

and never mentioning `cube` explicitly. This would place us at a serious disadvantage, forcing us to work always at the level of the particular operations that happen to be primitives in the language (multiplication, in this case) rather than in terms of higher-level operations. Our programs would be able to compute cubes, but our language would lack the ability to express the concept of cubing. One of the things we should demand from a powerful programming language is the ability to build abstractions by assigning names to common patterns and then to work in terms of the abstractions directly. Procedures provide this ability. This is why all but the most primitive programming languages include mechanisms for defining procedures.

Yet even in numerical processing we will be severely limited in our ability to create abstractions if we are restricted to procedures whose parameters must be numbers. Often the same programming pattern will be used with a number of different procedures. To express such patterns as concepts, we will need to construct procedures that can accept procedures as arguments or return procedures as values. Procedures that manipulate procedures are called *higher-order procedures*. This section shows how higher-order procedures can serve as powerful abstraction mechanisms, vastly increasing the expressive power of our language.

1.3.1 Procedures as Arguments

Consider the following three procedures. The first computes the sum of the integers from a through b:

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

The second computes the sum of the cubes of the integers in the given range:

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
```

The third computes the sum of a sequence of terms in the series

$$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \cdots,$$

which converges to $\pi/8$ (very slowly):[49]

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2)))
         (pi-sum (+ a 4) b))))
```

[49] This series, usually written in the equivalent form $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$, is due to Leibniz. We'll see how to use this as the basis for some fancy numerical tricks in Section 3.5.3.

These three procedures clearly share a common underlying pattern. They are for the most part identical, differing only in the name of the procedure, the function of a used to compute the term to be added, and the function that provides the next value of a. We could generate each of the procedures by filling in slots in the same template:

```
(define (⟨name⟩ a b)
  (if (> a b)
      0
      (+ (⟨term⟩ a)
         (⟨name⟩ (⟨next⟩ a) b))))
```

The presence of such a common pattern is strong evidence that there is a useful abstraction waiting to be brought to the surface. Indeed, mathematicians long ago identified the abstraction of *summation of a series* and invented "sigma notation," for example

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b),$$

to express this concept. The power of sigma notation is that it allows mathematicians to deal with the concept of summation itself rather than only with particular sums—for example, to formulate general results about sums that are independent of the particular series being summed.

Similarly, as program designers, we would like our language to be powerful enough so that we can write a procedure that expresses the concept of summation itself rather than only procedures that compute particular sums. We can do so readily in our procedural language by taking the common template shown above and transforming the "slots" into formal parameters:

```
(define (sum term a next b)
  (if (> a b)
```

```
    0
    (+ (term a)
       (sum term (next a) next b))))
```

Notice that `sum` takes as its arguments the lower and upper bounds `a` and `b` together with the procedures `term` and `next`. We can use `sum` just as we would any procedure. For example, we can use it (along with a procedure `inc` that increments its argument by 1) to define `sum-cubes`:

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (sum cube a inc b))
```

Using this, we can compute the sum of the cubes of the integers from 1 to 10:

```
(sum-cubes 1 10)
3025
```

With the aid of an identity procedure to compute the term, we can define `sum-integers` in terms of `sum`:

```
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b))
```

Then we can add up the integers from 1 to 10:

```
(sum-integers 1 10)
55
```

We can also define `pi-sum` in the same way:[50]

[50] Notice that we have used block structure (Section 1.1.8) to embed the definitions of `pi-next` and `pi-term` within `pi-sum`, since these procedures are unlikely to be useful for any other purpose. We will see how to get rid of them altogether in Section 1.3.2.

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```

Using these procedures, we can compute an approximation to π:

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

Once we have `sum`, we can use it as a building block in formulating further concepts. For instance, the definite integral of a function f between the limits a and b can be approximated numerically using the formula

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \ldots \right] dx$$

for small values of dx. We can express this directly as a procedure:

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))

(integral cube 0 1 0.01)
.24998750000000042

(integral cube 0 1 0.001)
.249999875000001
```

(The exact value of the integral of `cube` between 0 and 1 is 1/4.)

Exercise 1.29: Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n),$$

where $h = (b - a)/n$, for some even integer n, and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f, a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

Exercise 1.30: The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if ⟨??⟩
        ⟨??⟩
        (iter ⟨??⟩ ⟨??⟩))))
  (iter ⟨??⟩ ⟨??⟩))
```

Exercise 1.31:

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-

order procedures.[51] Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to π using the formula[52]

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}.$$

b. If your `product` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Exercise 1.32:

a. Show that `sum` and `product` (Exercise 1.31) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

[51] The intent of Exercise 1.31 through Exercise 1.33 is to demonstrate the expressive power that is attained by using an appropriate abstraction to consolidate many seemingly disparate operations. However, though accumulation and filtering are elegant ideas, our hands are somewhat tied in using them at this point since we do not yet have data structures to provide suitable means of combination for these abstractions. We will return to these ideas in Section 2.2.3 when we show how to use *sequences* as interfaces for combining filters and accumulators to build even more powerful abstractions. We will see there how these methods really come into their own as a powerful and elegant approach to designing programs.

[52] This formula was discovered by the seventeenth-century English mathematician John Wallis.

`accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null-value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`.

b. If your `accumulate` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Exercise 1.33: You can obtain an even more general version of `accumulate` (Exercise 1.32) by introducing the notion of a *filter* on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting `filtered-accumulate` abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write `filtered-accumulate` as a procedure. Show how to express the following using `filtered-accumulate`:

a. the sum of the squares of the prime numbers in the interval a to b (assuming that you have a `prime?` predicate already written)

b. the product of all the positive integers less than n that are relatively prime to n (i.e., all positive integers $i < n$ such that $\text{GCD}(i, n) = 1$).

1.3.2 Constructing Procedures Using `lambda`

In using `sum` as in Section 1.3.1, it seems terribly awkward to have to define trivial procedures such as `pi-term` and `pi-next` just so we can use them as arguments to our higher-order procedure. Rather than define `pi-next` and `pi-term`, it would be more convenient to have a way to directly specify "the procedure that returns its input incremented by 4" and "the procedure that returns the reciprocal of its input times its input plus 2." We can do this by introducing the special form `lambda`, which creates procedures. Using `lambda` we can describe what we want as

```
(lambda (x) (+ x 4))
```

and

```
(lambda (x) (/ 1.0 (* x (+ x 2))))
```

Then our `pi-sum` procedure can be expressed without defining any auxiliary procedures as

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b))
```

Again using `lambda`, we can write the `integral` procedure without having to define the auxiliary procedure `add-dx`:

```
(define (integral f a b dx)
  (* (sum f
          (+ a (/ dx 2.0))
          (lambda (x) (+ x dx))
          b)
     dx))
```

In general, `lambda` is used to create procedures in the same way as `define`, except that no name is specified for the procedure:

```
(lambda (⟨formal-parameters⟩) ⟨body⟩)
```

The resulting procedure is just as much a procedure as one that is created using `define`. The only difference is that it has not been associated with any name in the environment. In fact,

```
(define (plus4 x) (+ x 4))
```

is equivalent to

```
(define plus4 (lambda (x) (+ x 4)))
```

We can read a `lambda` expression as follows:

```
(lambda                    (x)     (+   x     4))
    |                       |       |   |     |
the procedure of an argument x that adds x and 4
```

Like any expression that has a procedure as its value, a `lambda` expression can be used as the operator in a combination such as

```
((lambda (x y z) (+ x y (square z)))
 1 2 3)
12
```

or, more generally, in any context where we would normally use a procedure name.[53]

[53] It would be clearer and less intimidating to people learning Lisp if a name more obvious than `lambda`, such as `make-procedure`, were used. But the convention is firmly entrenched. The notation is adopted from the λ-calculus, a mathematical formalism introduced by the mathematical logician Alonzo Church (1941). Church developed the λ-calculus to provide a rigorous foundation for studying the notions of function and function application. The λ-calculus has become a basic tool for mathematical investigations of the semantics of programming languages.

Using `let` to create local variables

Another use of `lambda` is in creating local variables. We often need local variables in our procedures other than those that have been bound as formal parameters. For example, suppose we wish to compute the function

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y),$$

which we could also express as

$$\begin{aligned} a &= 1 + xy, \\ b &= 1 - y, \\ f(x, y) &= xa^2 + yb + ab. \end{aligned}$$

In writing a procedure to compute f, we would like to include as local variables not only x and y but also the names of intermediate quantities like a and b. One way to accomplish this is to use an auxiliary procedure to bind the local variables:

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

Of course, we could use a `lambda` expression to specify an anonymous procedure for binding our local variables. The body of `f` then becomes a single call to that procedure:

```
(define (f x y)
  ((lambda (a b)
```

```
     (+ (* x (square a))
        (* y b)
        (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

This construct is so useful that there is a special form called `let` to make its use more convenient. Using `let`, the `f` procedure could be written as

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

The general form of a `let` expression is

```
(let ((⟨var1⟩ ⟨exp1⟩)
      (⟨var2⟩ ⟨exp2⟩)
      ...
      (⟨varn⟩ ⟨expn⟩))
   ⟨body⟩)
```

which can be thought of as saying

```
let ⟨var1⟩ have the value ⟨exp1⟩ and
    ⟨var2⟩ have the value ⟨exp2⟩ and
    ...
    ⟨varn⟩ have the value ⟨expn⟩
in  ⟨body⟩
```

The first part of the `let` expression is a list of name-expression pairs. When the `let` is evaluated, each name is associated with the value of the corresponding expression. The body of the `let` is evaluated with these names bound as local variables. The way this happens is that the `let` expression is interpreted as an alternate syntax for

```
((lambda (⟨var_1⟩ ... ⟨var_n⟩)
    ⟨body⟩)
 ⟨exp_1⟩
 ...
 ⟨exp_n⟩)
```

No new mechanism is required in the interpreter in order to provide local variables. A `let` expression is simply syntactic sugar for the underlying `lambda` application.

We can see from this equivalence that the scope of a variable specified by a `let` expression is the body of the `let`. This implies that:

- `let` allows one to bind variables as locally as possible to where they are to be used. For example, if the value of `x` is 5, the value of the expression

  ```
  (+ (let ((x 3))
       (+ x (* x 10)))
     x)
  ```

  is 38. Here, the `x` in the body of the `let` is 3, so the value of the `let` expression is 33. On the other hand, the `x` that is the second argument to the outermost `+` is still 5.

- The variables' values are computed outside the `let`. This matters when the expressions that provide the values for the local variables depend upon variables having the same names as the local variables themselves. For example, if the value of `x` is 2, the expression

  ```
  (let ((x 3)
        (y (+ x 2)))
    (* x y))
  ```

will have the value 12 because, inside the body of the `let`, `x` will be 3 and `y` will be 4 (which is the outer `x` plus 2).

Sometimes we can use internal definitions to get the same effect as with `let`. For example, we could have defined the procedure `f` above as

```
(define (f x y)
  (define a (+ 1 (* x y)))
  (define b (- 1 y))
  (+ (* x (square a))
     (* y b)
     (* a b)))
```

We prefer, however, to use `let` in situations like this and to use internal `define` only for internal procedures.[54]

Exercise 1.34: Suppose we define the procedure

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

Then we have

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
6
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

[54] Understanding internal definitions well enough to be sure a program means what we intend it to mean requires a more elaborate model of the evaluation process than we have presented in this chapter. The subtleties do not arise with internal definitions of procedures, however. We will return to this issue in Section 4.1.6, after we learn more about evaluation.

1.3.3 Procedures as General Methods

We introduced compound procedures in Section 1.1.4 as a mechanism for abstracting patterns of numerical operations so as to make them independent of the particular numbers involved. With higher-order procedures, such as the `integral` procedure of Section 1.3.1, we began to see a more powerful kind of abstraction: procedures used to express general methods of computation, independent of the particular functions involved. In this section we discuss two more elaborate examples—general methods for finding zeros and fixed points of functions—and show how these methods can be expressed directly as procedures.

Finding roots of equations by the half-interval method

The *half-interval method* is a simple but powerful technique for finding roots of an equation $f(x) = 0$, where f is a continuous function. The idea is that, if we are given points a and b such that $f(a) < 0 < f(b)$, then f must have at least one zero between a and b. To locate a zero, let x be the average of a and b, and compute $f(x)$. If $f(x) > 0$, then f must have a zero between a and x. If $f(x) < 0$, then f must have a zero between x and b. Continuing in this way, we can identify smaller and smaller intervals on which f must have a zero. When we reach a point where the interval is small enough, the process stops. Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as $\Theta(\log(L/T))$, where L is the length of the original interval and T is the error tolerance (that is, the size of the interval we will consider "small enough"). Here is a procedure that implements this strategy:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
```

```
(if (close-enough? neg-point pos-point)
    midpoint
    (let ((test-value (f midpoint)))
      (cond ((positive? test-value)
             (search f neg-point midpoint))
            ((negative? test-value)
             (search f midpoint pos-point))
            (else midpoint))))))
```

We assume that we are initially given the function f together with points at which its values are negative and positive. We first compute the midpoint of the two given points. Next we check to see if the given interval is small enough, and if so we simply return the midpoint as our answer. Otherwise, we compute as a test value the value of f at the midpoint. If the test value is positive, then we continue the process with a new interval running from the original negative point to the midpoint. If the test value is negative, we continue with the interval from the midpoint to the positive point. Finally, there is the possibility that the test value is 0, in which case the midpoint is itself the root we are searching for.

To test whether the endpoints are "close enough" we can use a procedure similar to the one used in Section 1.1.7 for computing square roots:[55]

```
(define (close-enough? x y) (< (abs (- x y)) 0.001))
```

`search` is awkward to use directly, because we can accidentally give it points at which f's values do not have the required sign, in which case

[55] We have used 0.001 as a representative "small" number to indicate a tolerance for the acceptable error in a calculation. The appropriate tolerance for a real calculation depends upon the problem to be solved and the limitations of the computer and the algorithm. This is often a very subtle consideration, requiring help from a numerical analyst or some other kind of magician.

we get a wrong answer. Instead we will use `search` via the following procedure, which checks to see which of the endpoints has a negative function value and which has a positive value, and calls the `search` procedure accordingly. If the function has the same sign on the two given points, the half-interval method cannot be used, in which case the procedure signals an error.[56]

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b)))))
```

The following example uses the half-interval method to approximate π as the root between 2 and 4 of $\sin x = 0$:

```
(half-interval-method sin 2.0 4.0)
3.14111328125
```

Here is another example, using the half-interval method to search for a root of the equation $x^3 - 2x - 3 = 0$ between 1 and 2:

```
(half-interval-method (lambda (x) (- (* x x x) (* 2 x) 3))
                      1.0
                      2.0)
1.89306640625
```

[56] This can be accomplished using `error`, which takes as arguments a number of items that are printed as error messages.

Finding fixed points of functions

A number x is called a *fixed point* of a function f if x satisfies the equation $f(x) = x$. For some functions f we can locate a fixed point by beginning with an initial guess and applying f repeatedly,

$$f(x), \quad f(f(x)), \quad f(f(f(x))), \quad \ldots,$$

until the value does not change very much. Using this idea, we can devise a procedure `fixed-point` that takes as inputs a function and an initial guess and produces an approximation to a fixed point of the function. We apply the function repeatedly until we find two successive values whose difference is less than some prescribed tolerance:

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

For example, we can use this method to approximate the fixed point of the cosine function, starting with 1 as an initial approximation:[57]

```
(fixed-point cos 1.0)
.7390822985224023
```

Similarly, we can find a solution to the equation $y = \sin y + \cos y$:

[57] Try this during a boring lecture: Set your calculator to radians mode and then repeatedly press the cos button until you obtain the fixed point.

```
(fixed-point (lambda (y) (+ (sin y) (cos y)))
             1.0)
1.2587315962971173
```

The fixed-point process is reminiscent of the process we used for finding square roots in Section 1.1.7. Both are based on the idea of repeatedly improving a guess until the result satisfies some criterion. In fact, we can readily formulate the square-root computation as a fixed-point search. Computing the square root of some number x requires finding a y such that $y^2 = x$. Putting this equation into the equivalent form $y = x/y$, we recognize that we are looking for a fixed point of the function[58] $y \mapsto x/y$, and we can therefore try to compute square roots as

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

Unfortunately, this fixed-point search does not converge. Consider an initial guess y_1. The next guess is $y_2 = x/y_1$ and the next guess is $y_3 = x/y_2 = x/(x/y_1) = y_1$. This results in an infinite loop in which the two guesses y_1 and y_2 repeat over and over, oscillating about the answer.

One way to control such oscillations is to prevent the guesses from changing so much. Since the answer is always between our guess y and x/y, we can make a new guess that is not as far from y as x/y by averaging y with x/y, so that the next guess after y is $\frac{1}{2}(y + x/y)$ instead of x/y. The process of making such a sequence of guesses is simply the process of looking for a fixed point of $y \mapsto \frac{1}{2}(y + x/y)$:

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

[58] $\mapsto$ (pronounced "maps to") is the mathematician's way of writing `lambda`. $y \mapsto x/y$ means `(lambda (y) (/ x y))`, that is, the function whose value at y is x/y.

(Note that $y = \frac{1}{2}(y + x/y)$ is a simple transformation of the equation $y = x/y$; to derive it, add y to both sides of the equation and divide by 2.)

With this modification, the square-root procedure works. In fact, if we unravel the definitions, we can see that the sequence of approximations to the square root generated here is precisely the same as the one generated by our original square-root procedure of Section 1.1.7. This approach of averaging successive approximations to a solution, a technique that we call *average damping*, often aids the convergence of fixed-point searches.

> **Exercise 1.35:** Show that the golden ratio φ (Section 1.2.2) is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute φ by means of the `fixed-point` procedure.

> **Exercise 1.36:** Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log(1000)/\log(x)$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log(1) = 0$.)

> **Exercise 1.37:**
>
> a. An infinite *continued fraction* is an expression of the

form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \ldots}}}.$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\varphi$, where φ is the golden ratio (described in Section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called *k-term finite continued fraction*—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}.$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\varphi$ using

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

b. If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

Exercise 1.38: In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... Write a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

Exercise 1.39: A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \ldots}}},$$

where x is in radians. Define a procedure `(tan-cf x k)` that computes an approximation to the tangent function based on Lambert's formula. `k` specifies the number of terms to compute, as in Exercise 1.37.

1.3.4 Procedures as Returned Values

The above examples demonstrate how the ability to pass procedures as arguments significantly enhances the expressive power of our programming language. We can achieve even more expressive power by creating procedures whose returned values are themselves procedures.

We can illustrate this idea by looking again at the fixed-point example described at the end of Section 1.3.3. We formulated a new version of the square-root procedure as a fixed-point search, starting with the observation that $\sqrt{x}$ is a fixed-point of the function $y \mapsto x/y$. Then we used average damping to make the approximations converge. Average damping is a useful general technique in itself. Namely, given a function f, we consider the function whose value at x is equal to the average of x and $f(x)$.

We can express the idea of average damping by means of the following procedure:

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

`average-damp` is a procedure that takes as its argument a procedure `f` and returns as its value a procedure (produced by the `lambda`) that, when applied to a number `x`, produces the average of `x` and `(f x)`. For example, applying `average-damp` to the `square` procedure produces a procedure whose value at some number x is the average of x and x^2. Applying this resulting procedure to 10 returns the average of 10 and 100, or 55:[59]

[59] Observe that this is a combination whose operator is itself a combination. Exercise 1.4 already demonstrated the ability to form such combinations, but that was only a toy example. Here we begin to see the real need for such combinations—when applying a procedure that is obtained as the value returned by a higher-order procedure.

```
((average-damp square) 10)
55
```

Using `average-damp`, we can reformulate the square-root procedure as follows:

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

Notice how this formulation makes explicit the three ideas in the method: fixed-point search, average damping, and the function $y \mapsto x/y$. It is instructive to compare this formulation of the square-root method with the original version given in Section 1.1.7. Bear in mind that these procedures express the same process, and notice how much clearer the idea becomes when we express the process in terms of these abstractions. In general, there are many ways to formulate a process as a procedure. Experienced programmers know how to choose procedural formulations that are particularly perspicuous, and where useful elements of the process are exposed as separate entities that can be reused in other applications. As a simple example of reuse, notice that the cube root of x is a fixed point of the function $y \mapsto x/y^2$, so we can immediately generalize our square-root procedure to one that extracts cube roots:[60]

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

Newton's method

When we first introduced the square-root procedure, in Section 1.1.7, we mentioned that this was a special case of *Newton's method*. If $x \mapsto g(x)$

[60] See Exercise 1.45 for a further generalization.

is a differentiable function, then a solution of the equation $g(x) = 0$ is a fixed point of the function $x \mapsto f(x)$, where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

and $Dg(x)$ is the derivative of g evaluated at x. Newton's method is the use of the fixed-point method we saw above to approximate a solution of the equation by finding a fixed point of the function f.[61]

For many functions g and for sufficiently good initial guesses for x, Newton's method converges very rapidly to a solution of $g(x) = 0$.[62]

In order to implement Newton's method as a procedure, we must first express the idea of derivative. Note that "derivative," like average damping, is something that transforms a function into another function. For instance, the derivative of the function $x \mapsto x^3$ is the function $x \mapsto 3x^2$. In general, if g is a function and dx is a small number, then the derivative Dg of g is the function whose value at any number x is given (in the limit of small dx) by

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}.$$

Thus, we can express the idea of derivative (taking dx to be, say, 0.00001) as the procedure

```
(define (deriv g)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))
```

[61] Elementary calculus books usually describe Newton's method in terms of the sequence of approximations $x_{n+1} = x_n - g(x_n)/Dg(x_n)$. Having language for talking about processes and using the idea of fixed points simplifies the description of the method.

[62] Newton's method does not always converge to an answer, but it can be shown that in favorable cases each iteration doubles the number-of-digits accuracy of the approximation to the solution. In such cases, Newton's method will converge much more rapidly than the half-interval method.

along with the definition

```
(define dx 0.00001)
```

Like `average-damp`, `deriv` is a procedure that takes a procedure as argument and returns a procedure as value. For example, to approximate the derivative of $x \mapsto x^3$ at 5 (whose exact value is 75) we can evaluate

```
(define (cube x) (* x x x))
((deriv cube) 5)
75.00014999664018
```

With the aid of `deriv`, we can express Newton's method as a fixed-point process:

```
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

The `newton-transform` procedure expresses the formula at the beginning of this section, and `newtons-method` is readily defined in terms of this. It takes as arguments a procedure that computes the function for which we want to find a zero, together with an initial guess. For instance, to find the square root of x, we can use Newton's method to find a zero of the function $y \mapsto y^2 - x$ starting with an initial guess of 1.[63]

This provides yet another form of the square-root procedure:

```
(define (sqrt x)
  (newtons-method
   (lambda (y) (- (square y) x)) 1.0))
```

[63] For finding square roots, Newton's method converges rapidly to the correct solution from any starting point.

Abstractions and first-class procedures

We've seen two ways to express the square-root computation as an instance of a more general method, once as a fixed-point search and once using Newton's method. Since Newton's method was itself expressed as a fixed-point process, we actually saw two ways to compute square roots as fixed points. Each method begins with a function and finds a fixed point of some transformation of the function. We can express this general idea itself as a procedure:

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

This very general procedure takes as its arguments a procedure g that computes some function, a procedure that transforms g, and an initial guess. The returned result is a fixed point of the transformed function.

Using this abstraction, we can recast the first square-root computation from this section (where we look for a fixed point of the average-damped version of $y \mapsto x/y$) as an instance of this general method:

```
(define (sqrt x)
  (fixed-point-of-transform
   (lambda (y) (/ x y)) average-damp 1.0))
```

Similarly, we can express the second square-root computation from this section (an instance of Newton's method that finds a fixed point of the Newton transform of $y \mapsto y^2 - x$) as

```
(define (sqrt x)
  (fixed-point-of-transform
   (lambda (y) (- (square y) x)) newton-transform 1.0))
```

We began Section 1.3 with the observation that compound procedures are a crucial abstraction mechanism, because they permit us to express general methods of computing as explicit elements in our programming

language. Now we've seen how higher-order procedures permit us to manipulate these general methods to create further abstractions.

As programmers, we should be alert to opportunities to identify the underlying abstractions in our programs and to build upon them and generalize them to create more powerful abstractions. This is not to say that one should always write programs in the most abstract way possible; expert programmers know how to choose the level of abstraction appropriate to their task. But it is important to be able to think in terms of these abstractions, so that we can be ready to apply them in new contexts. The significance of higher-order procedures is that they enable us to represent these abstractions explicitly as elements in our programming language, so that they can be handled just like other computational elements.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the "rights and privileges" of first-class elements are:[64]

- They may be named by variables.
- They may be passed as arguments to procedures.
- They may be returned as the results of procedures.
- They may be included in data structures.[65]

Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation,

[64] The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916-1975).

[65] We'll see examples of this after we introduce data structures in Chapter 2.

but the resulting gain in expressive power is enormous.[66]

Exercise 1.40: Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

Exercise 1.41: Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

Exercise 1.42: Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

```
((compose square inc) 6)
49
```

[66] The major implementation cost of first-class procedures is that allowing procedures to be returned as values requires reserving storage for a procedure's free variables even while the procedure is not executing. In the Scheme implementation we will study in Section 4.1, these variables are stored in the procedure's environment.

Exercise 1.43: If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if f is the function $x \mapsto x + 1$, then the n^{th} repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n-th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

Hint: You may find it convenient to use `compose` from Exercise 1.42.

Exercise 1.44: The idea of *smoothing* a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x - dx)$, $f(x)$, and $f(x + dx)$. Write a procedure `smooth` that takes as input a procedure that computes f and returns a procedure that computes the smoothed f. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the *n-fold smoothed function*. Show how to generate the n-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

Exercise 1.45: We saw in Section 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute n^{th} roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{n-1}$. Use this to implement a simple procedure for computing n^{th} roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

Exercise 1.46: Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as *iterative improvement*. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is

good enough. Rewrite the `sqrt` procedure of Section 1.1.7 and the `fixed-point` procedure of Section 1.3.3 in terms of `iterative-improve`.