PART I

Procedural Abstraction

Computer scientists study the processing of information. In this first part of the book, we will focus our attention on specifying the nature of that processing, rather than on the nature of the information being processed. (The latter is the focus of Parts II and III.) For this part of the book, we will look at procedures for processing only a few simple kinds of data, such as numbers and images; in the final chapter of Part I, we will look at procedures for processing other procedures.

We'll examine procedures from several different viewpoints, focusing on the connection between the form of the procedure and the form of the process that results from carrying it out. We'll see how to design procedures so that they have a desired effect and how to prove that they indeed have that effect. We'll see various ways to make procedures generate "expansible" processes that can grow to accommodate arbitrarily large instances of a general problem and see how the form of the procedure and process influences the efficiency of this growth. We'll look at techniques for capturing common processing strategies in general form, for example, by writing procedures that can write any of a family of similar procedures for us.

CHAPTER ONE

Computer Science and Programming

1.1 What's It All About?

Computer science revolves around *computational processes*, which are also called *information processes* or simply *processes*. A process is a dynamic succession of events—a happening. When your computer is busy doing something, a process is going on inside it. What differentiates a computational process from some other kind of process (e.g., a chemical process)? Although computing originally referred to doing arithmetic, that isn't the essence of a computational process: For our purposes, a word, for example, enjoys the same status as a number, and looking up the word in a dictionary is as much a computational process as adding numbers. Nor does the process need to go on inside a computer for it to be a computational process—it could go on in an old-fashioned library, where a patron turns the pages of a dictionary by hand.

What makes the process a computational process is that we study it in ways that ignore its physical nature. If we chose to study how the library patron turns the pages, perhaps by bending them to a certain point and then letting gravity flop them down, we would be looking at a mechanical process rather than a computational one. Here, on the other hand, is a *computational* description of the library patron's actions in looking up *fiduciary*:

1. Because *fiduciary* starts with an *f*, she uses the dictionary's index tabs to locate the *f* section.
2. Next, because the second letter (*i*) is about a third of the way through the alphabet, she opens to a point roughly a third of the way into the *f* section.
3. Finding herself slightly later in the alphabet (*fjord*), she then scans backward in a straightforward way, without any jumping about, until she finds *fiduciary*.

Notice that although there are some apparently physical terms in this description (*index tab* and *section*), the interesting thing about index tabs for the purposes of this process description is not that they are tabs but that they allow one to zoom in on those entries of the dictionary that have a particular initial letter. If the dictionary were stored in a computer, it could still have index tabs in the sense of some structure that allowed this operation, and essentially the same process could be used.

There are lots of questions one can ask about computational processes, such as

1. How do we describe one or specify which one we want carried out?
2. How do we prove that a process has a particular effect?
3. How do we choose a process from among several that achieve the same effect?
4. Are there effects we can't achieve no matter what process we specify?
5. How do we build a machine that automatically carries out a process we've specified?
6. What processes in the natural world are fruitfully analyzed in computational terms?

We'll touch on all these questions in this book, although the level of detail varies from several chapters down to a sentence or two. Our main goal, however, is not so much to answer the questions computer scientists face as to give a feel for the manner in which they formulate and approach those questions.

Because we'll be talking about processes so much, we'll need a notation for describing them. We call our descriptions *programs*, and the notation a *programming language*. For most of this book we'll be using a programming language called *Scheme*. (Two chapters near the end of the book use other programming languages for specialized purposes: assembly language, to illustrate at a detailed level how computers actually carry out computations, and Java, to illustrate how computational processes can interact with other, concurrently active processes.) One advantage of Scheme is that its structure is easy to learn; we will describe its basic structure in Section 1.2. As your understanding of computational processes and the data on which they operate grows, so too will your understanding of how those processes and data can be notated in Scheme.

An added benefit of Scheme (as with most useful programming languages) is that it allows us to make processes happen, because there are machines that can read our notation and carry out the processes they describe. The fact that our descriptions of abstract processes can result in their being concretely realized is a gratifying aspect of computer science and reflects one side of this book's title. It also means that computer science is to some extent an experimental science.

However, computer science is not purely experimental, because we can apply mathematical tools to analyze computational processes. Fundamental to this analysis is a way of modeling these evolving processes; we describe the so-called substitution

Responsible Computer Use

If you are using a shared computer system, there are some issues you should think about regarding the social acceptability of your behavior.

The most important point to keep in mind is that the feasibility of an action and its acceptability are quite different matters. You may well be technically capable of rummaging through other people's computer files without their approval. However, this act is generally considered to be like going down the street turning doorknobs and going inside if you find one unlocked.

Sometimes you won't know what is acceptable. If you have any doubts about whether a particular course of action is legal, ethical, and socially acceptable, err on the side of caution. Ask a responsible system administrator or faculty member first.

model in Section 1.2. This abstract model of a concrete process reflects another side of the book's title as it bears on the computational process itself.

As was mentioned above, computational processes do not only deal with numbers. The final section of this chapter applies the concepts of this chapter to an example involving building quilt-cover patterns out of more basic images. We will continue this convention of having the last section of each chapter be an application of that chapter's concepts. Following this application section, each chapter concludes with a collection of review problems, an inventory of the material introduced in the chapter, and notes on reference sources.

1.2 Programming in Scheme

The simplest possible Scheme program is a single number. If you ask the Scheme system to process such a program, it will simply return the number to you as its answer. We call what the Scheme system does finding the *value* of the *expression* you provide, or more simply *evaluation*. Exactly how this looks will vary from one version of Scheme to another; in our book, we'll show it as follows, with `dark, upright type` for your input and *`light, slanted type`* for the computer's output:

```
12
12
```

The first line here was typed by a human, whereas the second line was the computer's response. Other kinds of numbers also work: negative numbers, fractions, and decimals:

```
-7
-7
```

```
1/3
1/3

3.1415927
3.1415927
```

In Scheme, decimals are used for inexact approximations (as in the above approximation to π), and fractions are used for exact rational numbers.

Other kinds of expressions are less boring to evaluate. For example, the value of a *name* is whatever it is a name for. In a moment we'll see how we can name things ourselves, but there are many names already in place when we start up Scheme. Most are names for *procedures*; for example, the name `sqrt` names a procedure, as does the name `+`. If we evaluate either of them, we'll see a printed representation of the corresponding procedure:

```
sqrt
#<procedure>

+
#<procedure>
```

The appearance of procedures varies from one version of Scheme to another; in this book, we'll show them as `#<procedure>`, but you may see something different on your computer. However, this difference generally doesn't matter because procedures aren't meant to be looked at; they're meant to be used.

The way we use a procedure is to *apply* it to some values. For example, the procedure named `sqrt` can be applied to a single number to take its square root, and the procedure named `+` can be applied to two numbers to add them. The way we apply a procedure to values is as follows:

```
(sqrt 9)
3

(+ 3 6)
9
```

In every case, an application consists of a parenthesized list of expressions, separated by spaces. The first expression's value is the procedure to apply; the values of the remaining expressions are what the procedure should be applied to. Applications are themselves expressions, so they can be nested:

```
(sqrt (+ 3 6))
3
```

Here the value of the expression `(+ 3 6)` is 9, and that is the value to which the procedure named `sqrt` is applied. (More succinctly, we say that 9 is the *argument* to the `sqrt` procedure.)

There are any number of other useful procedures that already have names, such as `*` for multiplying, `-` for subtracting, and `/` for dividing.

Exercise 1.1

What is the value of each of the following expressions? You should be able to do them in your head, but checking your answers using a Scheme system will be a good way to get comfortable with the mechanics of using your particular system.

a. `(* 3 4)`

b. `(* (+ 5 3) (- 5 3))`

c. `(/ (+ (* (- 17 14) 5) 6) 7)`

It is customary to break complex expressions, such as in Exercise 1.1c, into several lines with indentation that clarifies the structure, as follows:

```
(/ (+ (* (- 17 14)
         5)
      6)
   7)
```

This arrangement helps make clear what's being multiplied, what's being added, and what's being divided.

Now that we've gained some experience using those things for which we already have names, we should learn how to name things ourselves. In Scheme, we do this with a *definition*, such as the following:

```
(define ark-volume (* (* 300 50) 30))
```

Scheme first evaluates the expression `(* (* 300 50) 30)` and gets 450000; it then remembers that `ark-volume` is henceforth to be a name for that value. You may get a response from the computer indicating that the definition has been performed; whether you do and what it is varies from system to system. In this book, we'll show no response. The name you defined can now be used as an expression, either on its own or in a larger expression:

```
ark-volume
450000
```

```
(/ ark-volume 8)
56250
```

Although naming allows us to capture and reuse the results of computations, it isn't sufficient for capturing reusable *methods* of computation. Suppose, for example, we want to compute the total cost, including a 5 percent sales tax, of several different items. We could take the price of each item, compute the sales tax, and add that tax to the original price:

```
(+ 1.29 (* 5/100 1.29))
1.3545
(+ 2.40 (* 5/100 2.40))
2.52
⋮
```

Alternatively, we could define a *procedure* that takes the price of an item (such as $1.29 or $2.40) and returns the total cost of that item, much as `sqrt` takes a number and returns its square root. To define such a total cost procedure we need to specify how the computation is done and give it a name.

We can specify a method of computation by using a *lambda expression*. In our sales tax example, the lambda expression would be as follows:

```
(lambda (x) (+ x (* 5/100 x)))
```

Other than the identifying keyword `lambda`, a lambda expression has two parts: a parameter list and a body. The parameter list in the example is `(x)` and the body is `(+ x (* 5/100 x))`. The value of a lambda expression is a procedure:

```
(lambda (x) (+ x (* 5/100 x)))
#<procedure>
```

Normally, however, we don't evaluate lambda expressions in isolation. Instead, we apply the resulting procedure to one or more argument values:

```
((lambda (x) (+ x (* 5/100 x))) 1.29)
1.3545
((lambda (x) (+ x (* 5/100 x))) 2.40)
2.52
```

When the procedure is applied to a value (such as 1.29), the body is evaluated, but with the parameter (`x` in this example) replaced by the argument value (1.29). In our example, when we apply `(lambda (x) (+ x (* 5/100 x)))` to 1.29, the computation done is `(+ 1.29 (* 5/100 1.29))`. When we apply the same procedure to 2.40, the computation done is `(+ 2.40 (* 5/100 2.40))`, and so on.

Including the lambda expression explicitly each time it is applied is unwieldy, so we usually use a lambda expression as part of a definition. The lambda expression produces a procedure, and `define` simply associates a name with that procedure. This process is similar to what mathematicians do when they say "let $f(x) = x \times x$". In this case, the parameter is x, the body is $x \times x$, and the name is f. In Scheme we would write

```
(define f (lambda (x) (* x x)))
```

or more descriptively

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

Now, whenever we need to square a number, we could just use `square`:

```
(square 3)
9

(square -10)
100
```

Exercise 1.2

a. Create a name for the tax example by using `define` to name the procedure `(lambda (x) (+ x (* 5/100 x)))`.

b. Use your named procedure to calculate the total price with tax of items costing $1.29 and $2.40.

Exercise 1.3

a. In the text example, we defined `f` and `square` in exactly the same way. What happens if we redefine `f`? Does the procedure associated with `square` change also?

b. Suppose we wrote:

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

Fill in the missing values:

```
(f 7)
```

_____

```
(square 7)
```

_____

```
(define f (lambda (x) (+ x 2)))
(f 7)
```

_____

```
(square 7)
```

_____

Here is another example of defining and using a procedure. Its parameter list is `(radius height)`, which means it is intended to be applied to two values. The first should be substituted where `radius` appears in the body, and the second where `height` appears:

```
(define cylinder-volume
  (lambda (radius height)
    (* (* 3.1415927 (square radius))
       height)))
(cylinder-volume 5 4)
314.15927
```

Notice that because we had already given the name `square` to our procedure for squaring a number, we were then able to simply use it by name in defining another procedure. In fact, it doesn't matter which order the two definitions are done in as long as both are in place before an attempt is made to apply the `cylinder-volume` procedure.

We can model how the computer produced the result `314.15927` by consulting Figure 1.1. In this diagram, the vertical arrows represent the conversion of a problem to an equivalent one, that is, one with the same answer. Alternatively, the same process can be more compactly represented by the following list of steps leading from the original expression to its value:

```
(cylinder-volume 5 4)
(* (* 3.1415927 (square 5)) 4)
(* (* 3.1415927 (* 5 5)) 4)
(* (* 3.1415927 25) 4)
(* 78.5398175 4)
314.15927
```

Whether we depict the evaluation process using a diagram or a sequence of expressions, we say we're using the *substitution model* of evaluation. We use this name because of the way we handle procedure application: The argument values are sub-

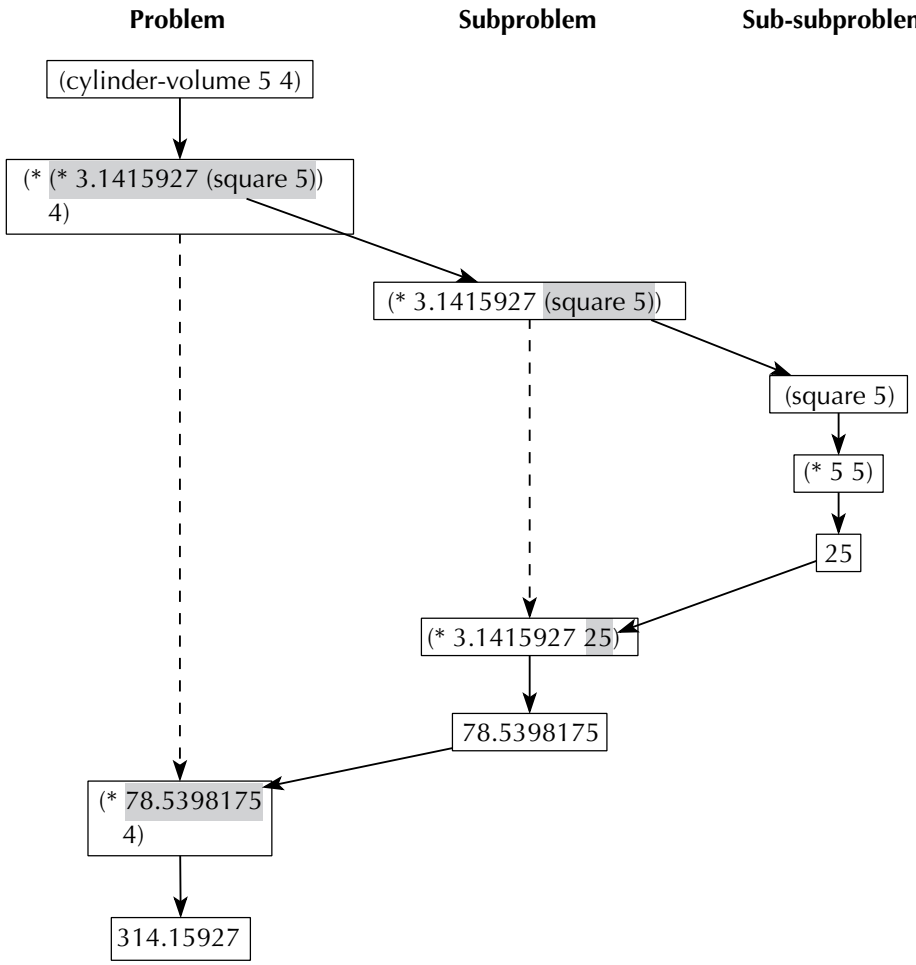


Figure 1.1 The process of evaluating `(cylinder-volume 5 4)`

stituted into the procedure body in place of the parameter names and then the resulting expression is evaluated.

Exercise 1.4

According to the *Joy of Cooking*, candy syrups should be cooked 1 degree cooler than listed in the recipe for each 500 feet of elevation above sea level.

a. Define `candy-temperature` to be a procedure that takes two arguments: the recipe's temperature in degrees and the elevation in feet. It should calculate the temperature to use at that elevation. The recipe for Chocolate Caramels calls for a temperature of 244 degrees; suppose you wanted to make them in Denver, the "mile high city." (One mile equals 5280 feet.) Use your procedure to find the temperature for making the syrup.

b. Candy thermometers are usually calibrated only in integer degrees, so it would be handy if the `candy-temperature` procedure would give an answer rounded to the nearest degree. Rounding can be done using the predefined procedure called `round`. For example, `(round 7/3)` and `(round 5/3)` both evaluate to 2. Insert an application of `round` at the appropriate place in your procedure definition and test it again.

Procedures give us a way of doing the same computation to different values. Sometimes, however, we have a computation we want to do to different values, but not exactly in the same way with each. Instead, we want to choose a particular computation based on the circumstances. For example, consider a simplified income tax, which is a flat 20 percent of income; however, those earning under $10,000 don't have to pay any tax at all. We can write a procedure for calculating this tax as follows:

```
(define tax
  (lambda (income)
    (if (< income 10000)
        0
        (* 20/100 income))))
```

Two things are new in this example. The first is the procedure named `<`. Unlike the procedures we've seen so far, it doesn't calculate a number. Instead it calculates a *boolean* or *truth value*—i.e., either true or false. It's what we call a *test* or *predicate*: a procedure that determines whether some fact is true or not. (In this case, it determines whether the income is less than $10,000.) The other new thing is the `if` expression, which uses the truth value to decide which of the remaining two expressions to evaluate. (As you may have guessed, there are other predefined predicates, including

`>`, `=`, `<=`, `>=`, `even?`, `odd?`, and many others. Of those we mentioned, only `<=` and `>=` are perhaps not self-explanatory; they correspond to the mathematical symbols $\leq$ and $\geq$ respectively.)

We can trace through the steps the computer would take in evaluating `(tax 30000)` as follows:

```
(tax 30000)
(if (< 30000 10000) 0 (* 20/100 30000))
(if #f 0 (* 20/100 30000))
(* 20/100 30000)
6000
```

In going from the second to the third line, the expression `(< 30000 10000)` is evaluated to the false value, which is written `#f`. (Correspondingly, the true value is written `#t`.) Because the `if`'s test evaluated to false, the second subexpression (the `0`) is ignored and the third subexpression (the `(* 20/100 30000)`) is evaluated. We can again show the computational process in a diagram, as in Figure 1.2.

Exercise 1.5

The preceding tax example has (at least) one undesirable property, illustrated by the following: if you earn \$9999, you pay no taxes, so your net income is also \$9999. However, if you earn \$10,000, you pay \$2000 in taxes, resulting in a net income of \$8000. Thus, earning \$1 more results in a net loss of \$1999!

The U.S. tax code deals with this potential inequity by using what is called a *marginal tax rate*. This policy means roughly that each additional dollar of income is taxed at a given percentage rate, but that rate varies according to what income level the dollar represents. In the case of our simple tax, this would mean that the first \$10,000 of a person's income is not taxed at all, but the amount above \$10,000 is taxed at 20 percent. For example, if you earned \$12,500, the first \$10,000 would be untaxed, but the amount over \$10,000 would be taxed at 20 percent, yielding a tax bill of $20\% \times (\$12,500 - \$10,000) = \$500$. Rewrite the procedure `tax` to reflect this better strategy.

Exercise 1.6

The *Joy of Cooking* suggests that to figure out how many people a turkey will serve, you should allow 3/4 of a pound per person for turkeys up to 12 pounds in weight, but only 1/2 pound per person for larger turkeys. Write a procedure, `turkey-servings`, that when given a turkey weight in pounds will calculate the number of people it serves.

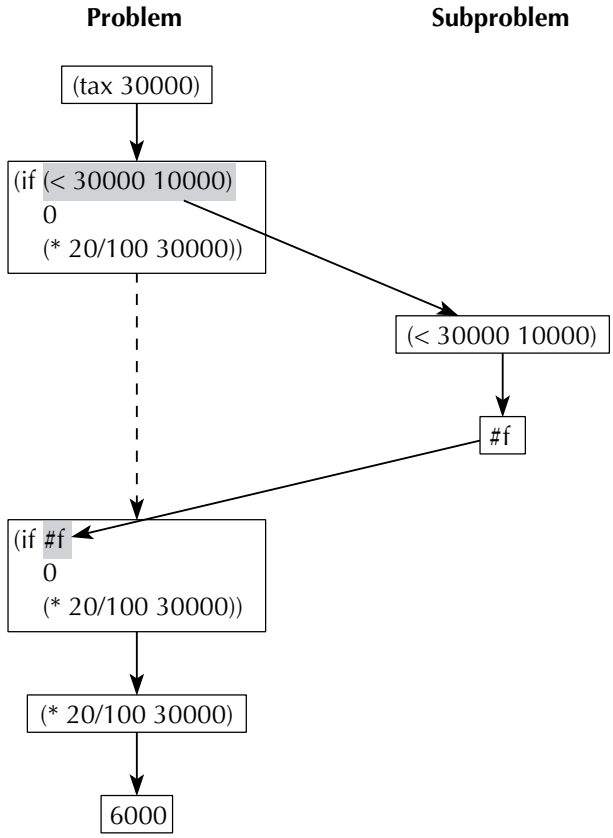


Figure 1.2 The process of evaluating `(tax 30000)`

Exercise 1.7

Write a succinct English description of the effect of each of the following procedures. Try to express *what* each calculates, not *how* it calculates that.

a.
```
(define puzzle1
  (lambda (a b c)
    (+ a (if (> b c)
             b
             c))))
```

b.
```
(define puzzle2
  (lambda (x)
    ((if (< x 0)
         -
         +)
     0 x)))
```

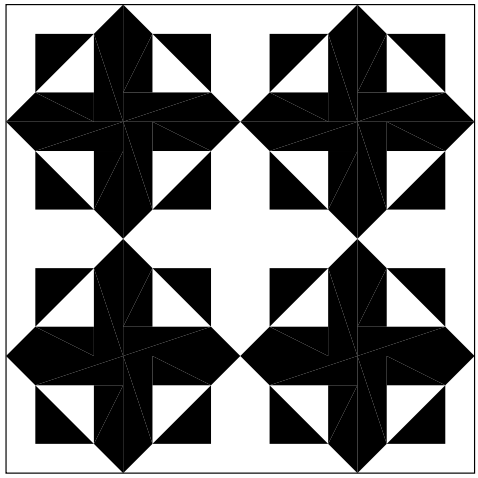


Figure 1.3 A sample of the *Repeating Crosses* quilt

1.3 An Application: Quilting

Now we turn our attention to building procedures that operate on rectangular images, rather than numbers. Using these procedures we can produce geometric quilt patterns, such as the *Repeating Crosses* pattern shown in Figure 1.3.

In doing numeric computations, the raw materials are numbers you type in and some primitive numeric procedures, such as `+`. (By *primitive procedures*, we mean the fundamental predefined procedures that are built into the Scheme system.) The situation here is similar. We will build our images out of smaller images, and we will build our image procedures out of a few primitive image procedures that are built into our Scheme system. Unfortunately, image procedures are not as standardized as numeric procedures, so you can't count on these procedures to work in all versions of Scheme; any Scheme used with this book, however, should have the procedures we use here. There is also the problem of how to input the basic building-block images that are to be manipulated. Graphic input varies a great deal from computer to computer, so rather than tell you how to do it, we've provided a file on the web site for this book that you can load into Scheme to define some sample images. Loading that file defines each of the names shown in Figure 1.4 as a name for the corresponding image. (Exercise 1.11 at the end of this section explains how these blocks are produced.)

We'll build our quilts by piecing together small square images called *basic blocks*. The four examples in Figure 1.4 are all basic blocks; the one called `rcross-bb` was used to make the Repeating Crosses quilt. The quilt was made by piecing together copies of the basic block, with some of them turned.

To make the Repeating Crosses quilt, we need at least two primitive procedures: one that will produce an image by piecing together two smaller images and one

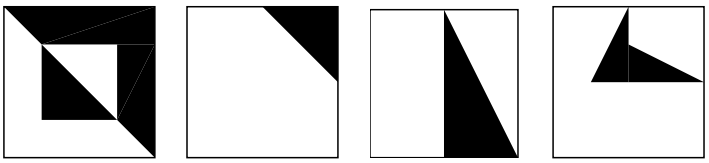


Figure 1.4 Predefined images

that will turn an image a quarter turn to the right. These procedures, which are built into the Scheme systems recommended for this book, are called `stack` and `quarter-turn-right`.

Exercise 1.8

Try evaluating the following expressions:

```
(stack rcross-bb corner-bb)
(quarter-turn-right test-bb)
```

What happens if you nest several expressions, such as in the following:

```
(stack (stack rcross-bb corner-bb) test-bb)
(stack (stack rcross-bb corner-bb)
       (stack (quarter-turn-right test-bb) test-bb))
```

Can you describe the effect of each primitive?

Exercise 1.9

Before undertaking anything so ambitious as making an actual quilt, it may pay to have a few more tools in our kit. For example, it would be nice if we could turn an image to the left, or half way around, as well as to the right. Similarly, it would be desirable to be able to join two images side by side as well as stacking them on top of one another.

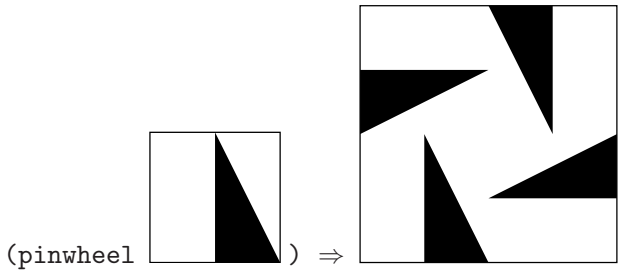
a. Define procedures `half-turn` and `quarter-turn-left` that do as their names suggest. Both procedures take a single argument, namely, the image to turn. You will naturally need to use the built-in procedure `quarter-turn-right`.

b. Define a procedure `side-by-side` that takes two images as arguments and creates a composite image having the first image on the left and the second image on the right.

If you don't see how to build the three additional procedures out of `quarter-turn-right` and `stack`, you may want to play more with combinations of those two. Alternatively, try playing with paper squares with basic blocks drawn on them. (The web site for this book has some basic blocks you can print out, but hand-drawn ones work just as well.)

Exercise 1.10

Each dark cross in the repeating crosses pattern is formed by joining together four copies of the basic block, each facing a different way. We can call this operation *pinwheeling* the basic block; here is an example of the same operation performed on the image `test-bb`:



Define the `pinwheel` procedure and show how you can use it to make a cross out of the basic block.

Now try pinwheeling the cross—you should get a sample of the quilt, with four dark crosses, as shown at the beginning of the section. If you pinwheel that, how big is the quilt you get?

Try making other pinwheeled quilts in the same way, but using the other basic blocks. What do the designs look like?

Although you have succeeded (through the exercises) in making the Repeating Crosses quilt described at the beginning of this section, there are at least two questions you may have. First, how are the basic blocks constructed in the first place? And second, how could we create quilts that aren't pinwheels of pinwheels? This latter question will be dealt with in the next two chapters, which introduce new programming techniques called *recursion* and *iteration*. The former question is addressed in the following exercise.

Exercise 1.11

All four basic blocks shown previously can be produced using two primitive graphics procedures supported by all the Scheme systems recommended for this book. The

first of these procedures, `filled-triangle`, takes six arguments, which are the *x* and *y* coordinates of the corners of the triangle that is to be filled in. The coordinate system runs from −1 to 1 in both dimensions. For example, here is the definition of `test-bb`:

```
(define test-bb
  (filled-triangle 0 1 0 -1 1 -1))
```

The second of these procedures, `overlay`, combines images. To understand how it works, imagine having two images on sheets of transparent plastic laid one on top of the other so that you see the two images together. For example, here is the definition of `nova-bb`, which is made out of two triangles:

```
(define nova-bb
  (overlay (filled-triangle 0 1 0 0 -1/2 0)
           (filled-triangle 0 0 0 1/2 1 0)))
```

a. Use these primitive graphics procedures to define the other two basic blocks from Figure 1.4.

b. Now that you know how it is done, be inventive. Come up with some basic blocks of your own and make pinwheeled quilts out of them. Of course, if your system supports direct graphical input, you can also experiment with freehand images, or images from nature. You might find it interesting to try experiments such as overlaying rotated versions of an image on one another.

Review Problems

Exercise 1.12

Find two integers such that applying `f` to them will produce 16 as the value, given that `f` is defined as follows:

```
(define f
  (lambda (x y)
    (if (even? x)
        7
        (* x y))))
```

Exercise 1.13

Write a Scheme expression with no multidigit numbers in it that has 173 as its value.

Exercise 1.14

Write a procedure that takes two arguments and computes their average.

Exercise 1.15

What could be filled into the blank in the following procedure to ensure that no division by zero occurs when the procedure is applied? Give several different answers.

```
(define foo
  (lambda (x y)
    (if __________
        (+ x y)
        (/ x y))))
```

Exercise 1.16

A 10-foot-long ladder leans against a wall, with its base 6 feet away from the bottom of the wall. How high on the wall does it reach? This question can be answered by evaluating `(ladder-height 10 6)` after entering the following definition. Make a diagram such as the one in Figure 1.1 showing the evaluation of `(ladder-height 10 6)` in the context of this definition:

```
(define ladder-height
  (lambda (ladder-length base-distance)
    (sqrt (- (square ladder-length)
             (square base-distance)))))
```

Chapter Inventory

Vocabulary

computer science
computational process
information process
process
program
programming language
Scheme
value
expression
evaluation
procedure
apply
argument
parameter
substitution model
boolean
truth value
test
predicate
primitive procedure

New Predefined Scheme Names

The dagger symbol (†) indicates a name that is not part of the R^{4}RS standard for Scheme.

`sqrt`
`+`
`*`
`-`
`/`
`round`
`<`
`>`
`=`
`<=`
`>=`
`even?`
`odd?`
`stack`†
`quarter-turn-right`†
`filled-triangle`†
`overlay`†

New Scheme Syntax

number
name
application
definition
lambda expression
parameter list
body
if expression
`#f`
`#t`

Scheme Names Defined in This Chapter

`ark-volume`
`square`
`cylinder-volume`
`candy-temperature`
`tax`
`turkey-servings`
`puzzle1`
`puzzle2`
`rcross-bb`
`corner-bb`
`test-bb`
`nova-bb`
`half-turn`
`quarter-turn-left`
`side-by-side`
`pinwheel`
`ladder-height`

Sidebars

Responsible Computer Use

Notes

The identifying keyword `lambda`, which indicates that a procedure should be created, has a singularly twisted history. This keyword originated in the late 1950s in a programming language (an early version of Lisp) that was a direct predecessor to Scheme. Why? Because it was the name of the Greek letter λ, which Church had used in the 1930s to abstract mathematical functions from formulas [12]. For

example, where we write `(lambda (x) (* x x))`, Church might have written $\lambda x.x \times x$. Because the computers of the 1950s had no Greek letters, the λ needed to be spelled out as `lambda`. This development wasn't the first time that typographic considerations played a part in the history of `lambda`. Barendregt [6] tells "what seems to be the story" of how Church came to use the letter λ. Apparently Church had originally intended to write $\hat{x}.x \times x$, with a circumflex or "hat" over the x. (This notation was inspired by a similar one that Whitehead and Russell used in their *Principia Mathematica* [53].) However, the typesetter of Church's work was unable to center the hat over the top of the x and so placed it before the x, resulting in $\^{}x.x \times x$ instead of $\hat{x}.x \times x$; a later typesetter then turned that hat with nothing under it into a λ, presumably based on the visual resemblance.

The formula for candy-making temperatures at higher elevations, the recipe for chocolate caramels, and the formula for turkey servings are all from the *Joy of Cooking* [42]. The actual suggested formula for turkey servings gives a range of serving sizes for each class of turkeys; we've chosen to use the low end of each range, because we've never had a shortage of turkey.

The quilting application is rather similar to the "Little Quilt" language of Sethi [49]. The Repeating Crosses pattern is by Helen Whitson Rose [43].