# Chapter 4

# Procedures

*A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem may be modest, but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery.*

George Pólya

Computers are tools for performing computations. The reason we want to perform a computation is to solve a problem. In this chapter, we consider what it means to solve a problem and explore some strategies for constructing procedures that solve problems.

## 4.1 Solving Problems

Traditionally, a problem is an obstacle to overcome or some question to answer. Once the question is answered or the obstacle circumvented, the problem is solved and we can move on to the next one. When we talk about writing programs to solve problems, however, we usually have a larger goal. We don't just want to solve one instance of a problem, we want a procedure that can solve *all* instances of a problem.

A *problem* is defined by its inputs and the desired property of the output. A solution to a problem is a procedure that produces a correct output for all possible

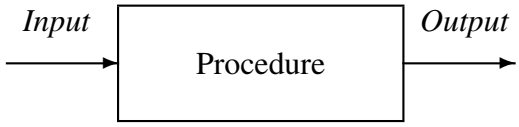


Figure 4.1: **Procedure.**

inputs to the problem, as shown in Figure 4.1.

For the procedure to be useful, it should also have the property that it always produces an output — that is, it always finishes executing. A procedure that solves a problem and is guaranteed to always finish is called an *algorithm*. The name algorithm is a Latinization of the name of the Persian mathematician and scientist, Muhammad ibn Mūsā al-Khwārizmī, who published a book in 825 on calculation with Hindu numerals. Al-Khwārizmī was also the responsible for defining algebra. Although the name was not adopted until after al-Khwārizmī's book, algorithms go back much further than that. The ancient Babylonians had algorithms for finding square roots more than 3500 years ago[1].

Some programs are intended to provide a service rather than just one output. For example, a web server is a program that is intended to run forever, not to finish and produce an output. It keeps running and responding to new requests as they come in. Even programs that are services, however, involve algorithms. Each individual web request is a problem, and the server program should respond to it with a particular output (sending the requested page) within a finite amount of time.

Our goal in solving a problem is to devise a procedure that takes inputs that define a problem instance, and produces as output the solution to the problem. There is no magic wand for solving all problems, but at its core most problem solving involves breaking problems you does not yet know how to solve into simpler problems until you find problems simple enough that you already know how to solve them. The trick is to find the right subproblems so that they can be combined to

[1]Donald E. Knuth, *Ancient Babylonian Algorithms*, Communications of the ACM, July 1972.

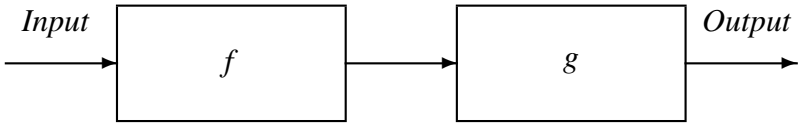


Figure 4.2: **Composition.**

solve the original problem. The following sections describe a few techniques for solving problems, and illustrate them with some simple examples. We will use these same problem solving techniques over and over throughout this book. In the examples here, we limit ourselves to simple data: all of the problems have one or more numbers as their input, and produce a number as output. In the next chapter, we introduce structured data and revisit these problem solving techniques.

## 4.2 Composing Procedures

One way to divide a problem is to split it into steps where the output of the first step is the input to the second step, and the output of the second step is the solution to the problem. Each step can be defined by one procedure, and the two procedures can be combined to create one procedure that solves the problem.

Figure 4.2 shows a composition of two functions, $f$ and $g$. The output of $f$ is the input to $g$. We can express this composition with the Scheme expression `(g (f x))` where $x$ is the input.

The order appears to be reversed from Figure 4.2. That is because we apply a procedure to the values of its subexpressions. That means although `f` appears to the right of `g` in the expression, the subexpression `(f x)` is evaluated first since the evaluation rule for the outer application expression is to first evaluate all the subexpressions.

We can define a procedure that implements the composed procedure by making `x` a parameter:

```
(define fog (lambda (x) (g (f x))))
```

This defines `fog` as a procedure that takes one input and produces as output the composition of *f* and *g* applied to the input parameter. The works for any two procedures, as long as both procedures take a single input parameter. For example, we could compose the `square` and `cube` procedures from Chapter 3 as:

```
(define sixth-power
    (lambda (x) (cube (square x))))
```

So, `(sixth-power 2)` evaluates to 64.

### 4.2.1 Procedures as Inputs and Outputs

So far, all the inputs we have seen have been numbers. But, the subexpressions of an application can be any expression including a procedure. We call a procedure that takes other procedures as inputs or that produces a procedure as its output a *higher-order procedure*. Higher-order procedures give us the ability to write general procedures that behave differently based on not just number value inputs, but based on procedures that are passed as inputs to the general procedure.

For example, we can create a generic composition procedure by making *f* and *g* parameters:

```
(define fog (lambda (f g x) (g (f x))))
```

The `fog` procedure takes three parameters. The first two are both procedures that take one input. The third parameter is a value (that can be the input to the first procedure).

For example,

```
> (fog square cube 2)
64
> (fog (lambda (x) (+ x 1)) square 2)
9
```

In the second example the first parameter is a new procedure,

```
(lambda (x) (+ x 1))
```

This procedure takes a number as input and produces as output that number plus one. We define `inc` (short for increment) to name this procedure:

```
(define inc (lambda (x) (+ x 1)))
```

A more useful composition procedure would separate the input value, `x`, from the composition. It takes two procedures as inputs and produce as output a procedure that is their composition:

```
(define compose
   (lambda (f g) (lambda (x) (g (f x)))))
```

The body of the `compose` procedure itself makes a procedure! Hence, the result of applying `compose` to two procedures is not a simple value, but a procedure. To get a value, we need to apply the result of the application of `compose` to a value.

For example,

```
> (compose inc inc)
#<procedure>
> ((compose inc inc) 1)
3
> (define sixth-power (compose square cube))
> (sixth-power 3)
729
```

**Exercise 4.1.**

**a.** What does

```
(compose (lambda (x) (* x 2))
         (lambda (x) (/ x 2)))
```

evaluate to?

**b.** What does

```
((compose (lambda (x) (* x 2))
          (lambda (x) (/ x 2))) 150)
```

evaluate to?

**c.** What does

```
((compose (compose inc inc) inc) 2)
```

evaluate to?

■

**Exercise 4.2.** Suppose we define `self-compose` as a procedure that composes a procedure with itself:

```
(define (self-compose f) (compose f f))
```

Explain how

```
(((compose self-compose self-compose) inc) 1)
```

would be evaluated. ■

**Exercise 4.3.** Define a procedure `compose3` that takes three procedures as input, and produces as output a procedure that is the composition of the three input procedures. For example, `((compose3 abs inc square) -5)` should evaluate to 36. Try defining `compose3` first without using `compose`, and then define it using `compose`. ■

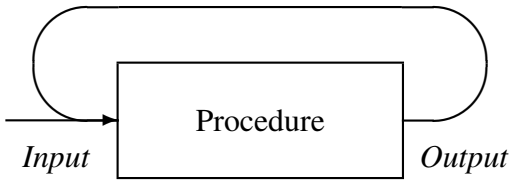


Figure 4.3: **Circular Composition.**

## 4.3 Recursive Problem Solving

In the previous section, we used functional composition to break a problem into two procedures that can be composed to produce the desired output. A particularly useful variation on this is when we can break a problem into two problems, where one of them is a smaller version of the original problem.

The goal is to be able to feed the output of one application of the procedure back into the same procedure as its input for the next application, as shown in Figure 4.3. Here's what this would look like in a Scheme procedure:

```
(define f (lambda (n) (f (- n 1))))
```

Of course, this doesn't work very well![2] Every time an application of `f` is evaluated, it results in another application of `f` to evaluate. We keep evaluating new applications, but never stop and can never produce an output.

What we need is a way of stopping. The circular applications could stop when the input to the procedure is simple enough that we can produce the output without needing another application of the procedure. This is called the *base case*, similarly to the grammar rules in Section 2.4. In our grammar examples, the base case usually involved replacing the nonterminal with nothing (e.g., $moredigits ::\Rightarrow \epsilon$). In problem solving, the base case will be some input for which the problem is so

[2]The curious should try entering this definition into a Scheme interpreter and evaluating an application of `(f 10)`. If you get tired of waiting for an output, in DrScheme you can click the **Stop** button in the upper right corner to interrupt the evaluation.

simple we already know the answer. When the input is a number, this is often (but not always) when the input is zero.

To define a recursive procedure, we need to use an if expression to test if the input matches the base case input. If it does, the consequent expression is the known answer for the base case. Otherwise, we enter the recursive case and apply the procedure again. Each time we apply the procedure we need to make progress towards reaching the base case. This means, the input has to change in a way that gets closer to the base case input. If the base case is a small number, and the input starts greater than that number, one way to get closer to the base case input is to subtract `1` from the input value with each recursive application.

Here is a procedure that does this:

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

Unlike the earlier circular `f` procedure, if we apply `g` to a positive number it will eventually produce an output.

For example, consider evaluating `(g 3)`. When we evaluate the first application, the value of the parameter `n` is 3, so the predicate expression `(= n 0)` evaluates to #f and the value of the procedure body is the value of the alternate expression, `(g (- n 1))`. The subexpression, `(- n 1)` evaluates to 2, so the result is the result of applying `g` to 2. As with the previous application, this leads to the application, `(g (- n 1))`, but this time the value of `n` is 2, so `(- n 1)` evaluates to 1. The next application leads to the application, `(g 0)`. This time, the predicate expression evaluates to #t and we have reached the base case. The consequent expression is just `1`, so no further applications of `g` are performed. The value 1 is the result of the application `(g 0)`. This is returned as the result of the `(g 1)` application in the previous recursive call, and then as the result of the `(g 2)` application, and finally as the output of the original `(g 3)`.

We can think of the recursive evaluation as winding until the base case is reached, and then unwinding the outputs back to the original application. For this procedure, the output is not very interesting: no matter what positive number we

apply `g` to, the eventual result is `1`. To solve interesting problems with recursive procedures, we need to accumulate results as the recursive applications wind or unwind. Examples 4.3 and 4.4 illustrate recursive procedures that accumulate the result during the unwinding process. Example **??** illustrates a recursive procedure that accumulates the result during the winding process.

**Example 4.1: Factorial.** How many different arrangements are there of a deck of 52 playing cards? The top card in the deck can be any of the 52 cards, so there are 52 possible choices for the top card. The second card can be any of the cards except for the card that is the top card, so there are 51 possible choices for the second card. The third card can be any of the 50 remaining cards, and so on, until the last card for which there is only one choice remaining. To determine the total number of possible arrangements we need to multiply the number of choices for each card:

$$52 * 51 * 50 * \cdots * 2 * 1$$

This is known as the *factorial* function (denoted in mathematics using the exclamation point, e.g., $52!$). It is defined as:

$$factorial(n) = \begin{cases} 1 & : \quad n = 0 \\ n * factorial(n-1) & : \quad n > 0 \end{cases}$$

The mathematical definition of factorial is recursive, so it is natural that we can define a recursive procedure that computes factorials:

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

We can now determine the number of deck arrangements:

```
> (factorial 52)
```
80658175170943878571660636856403766975289505440883277824000000000000

The `factorial` procedure has structure very similar to our earlier definition of the useless recursive `g` procedure. The only difference is the alternative expression for the if expression: with `g` we used `(g (- n 1))`; with `factorial` we added the outer application of `*`: `(* n (factorial (- n 1)))`. Instead

of just evaluating to the result of the recursive application, we are now combining the output of the recursive evaluation with the input `n` using a multiplication application.

**Exercise 4.4.** ($\star$) When Karl Gauss was in elementary school, his teacher assigned the class the task of summing the integers from 1 to 100 (e.g., $1 + 2 + 3 + \cdots + 100$) to keep them busy. Being the (future) "Prince of Mathematics", Gauss developed the formula for calculating this sum, that is now known as the *Gauss sum*. Had he been a computer scientist, however, and had access to a Scheme interpreter in the late 1700s, he could have instead defined a recursive procedure to solve the problem.

Define a recursive procedure, `gauss-sum`, that takes a number `n` as its input parameter, and evaluates to the sum of the integers from 1 to `n` as its output. For example, `(gauss-sum 100)` should evaluate to 5050. ■

## 4.4 Evaluating Recursive Applications

Evaluating an application of a recursive procedure follows the evaluation rules just like any other expression evaluation. It may be confusing, however, to see that this works because of the apparent circularity of the procedure definition. Here, we show in detail the evaluation steps for evaluating `(factorial 2)`. The evaluation and application rules refer to the rules summary in Section 3.8. We first show the complete evaluation following the substitution model evaluation rules in full gory detail, and later consider a subset containing the most revealing steps. The evaluation rule for an application expression does not specify the order in which the subexpressions are evaluated. A Scheme interpreter is free to evaluate them in any order. Here, we choose to evaluate the subexpressions in the order that is most readable.

1. `(factorial 2)`
   Evaluation Rule 3(a): Application subexpressions
2. `(factorial 2)`
   Evaluation Rule 2: Name
3. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`
   Evaluation Rule 4: Lambda

4. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`
   Evaluation Rule 1: Primitive
5. `((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 2)`
   Evaluation Rule 3(b): Application, Application Rule 2
6. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`
   Evaluation Rule 5(a): If predicate
7. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`
   Evaluation Rule 3(a): Application subexpressions
8. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`
   Evaluation Rule 1: Primitive
9. `(if (= 2 0) 1 (* 2 (factorial (- 2 1))))`
   Evaluation Rule 3(b): Application, Application Rule 1
10. `(if #f 1 (* 2 (factorial (- 2 1))))`
    Evaluation Rule 5(b): If alternate
11. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 3(a): Application subexpressions
12. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 1: Primitive
13. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 3(a): Application subexpressions
14. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 3(a): Application subexpressions
15. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 1: Primitive
16. `(* 2 (factorial (- 2 1)))`
    Evaluation Rule 3(b): Application, Application Rule 1
17. `(* 2 (factorial 1))`
    Continuing Evaluation Rule 3(a); Evaluation Rule 2: Name
18. `(* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))`
    Evaluation Rule 4: Lambda
19. `(* 2 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 1))`
    Evaluation Rule 3(b): Application, Application Rule 2
20. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`
    Evaluation Rule 5(a): If predicate
21. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`
    Evaluation Rule 3(a): Application subexpressions
22. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 1: Primitives

23. `(* 2 (if (= 1 0) 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 3(b): Application Rule 1

24. `(* 2 (if #f 1 (* 1 (factorial (- 1 1)))))`

Evaluation Rule 5(b): If alternate

25. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 3(a): Application

26. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 1: Primitives

27. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 3(a): Application

28. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 3(a): Application

29. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 1: Primitives

30. `(* 2 (* 1 (factorial (- 1 1))))`

Evaluation Rule 3(b): Application, Application Rule 1

31. `(* 2 (* 1 (factorial 0)))`

Evaluation Rule 2: Name

32. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`

Evaluation Rule 4, Lambda

33. `(* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (factorial (- n 1))))) 0)))`

Evaluation Rule 3(b), Application Rule 2

34. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`

Evaluation Rule 5(a): If predicate

35. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`

Evaluation Rule 3(a): Application subexpressions

36. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`

Evaluation Rule 1: Primitives

37. `(* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))`

Evaluation Rule 3(b): Application, Application Rule 1

38. `(* 2 (* 1 (if #t 1 (* 0 (factorial (- 0 1))))))`

Evaluation Rule 5(b): If consequent

39. `(* 2 (* 1 1))`

Evaluation Rule 1: Primitives

40. `(* 2 (* 1 1))`

Evaluation Rule 3(b): Application, Application Rule 1

41. `(* 2 1)`
    Evaluation Rule 3(b): Application, Application Rule 1
42. **3**
    Value reached

The key to evaluating applications of recursive procedures is the evaluation rule for the if special form. If the if expression were evaluated like a regular application all subexpressions would be evaluated, and the alternative expression with the recursive call would never finish evaluating! Since the evaluation rule for if requires that the predicate expression is evaluated and the alternative expression is *not* evaluated when the predicate expression is true, the circularity in the definition ends when the predicate expression evaluates to true. This is the base case of the recursion, when `(= n 0)` evaluates to true. This occurs when `(factorial 0)` is evaluated, and instead of producing another recursive call it evaluates to the value of the consequent expression, 1.

**Exercise 4.5.** These exercises test your understanding of the `(factorial 2)` evaluation. Try to answer them yourself before reading the remainder of this section.

**a.** In step 5, the second part of the application evaluation rule, Rule 3(b), is used. In which step does this evaluation rule complete?

**b.** In step 11, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

**c.** In step 25, the first part of the application evaluation rule, Rule 3(a), is used. In which step is the following use of Rule 3(b) started?

**d.** How many times is Evaluation Rule 5 (If) used in evaluating `(factorial 2)`?

**e.** In order to evaluate `(factorial 3)` how many times would Evaluation Rule 2 be used to evaluate the name `factorial`?

■

**Exercise 4.6.** For what input values *n* with an evaluation of `(factorial n)` reach a value? For values where the evaluation is guaranteed to finish, make a

convincing argument why it must finish. For values where the evaluation does not finish, explain why. ■

**Example 4.2: Maximum.** In Chapter 3, we saw a procedure, `max`, that takes two inputs and evaluates to the greater input. Here, we consider the problem of defining a procedure that takes as its input a procedure, a low value, and a high value, and outputs the maximum value the input procedure produces when applied to an integer value between the low value and high value input. We will name the inputs `f`, `low`, and `high`. To find the maximum, the `find-maximum` procedure should evaluate the input procedure `f` at every integer value between the `low` and `high`, and produce as output the greatest value found.

To define the procedure, we need to think about this in a way that allows us to combine results from simpler problems to find the result. For the base case, we need to identify a case so simple we already know the answer. Consider the case when `low` and `high` are equal. Then, there is only one value to use, and we know the value of the maximum is `(f low)`. So, the base case is

```
(if (= low high) (f low) Alternate Expression)
```

How to we make progress towards the base case? Suppose the value of `high` is equal to the value of `low` plus 1. Then, the maximum value is either the value of `(f low)` or the value of `(f (+ low 1))`. We could select it using the `max` procedure:

```
(max (f low) (f (+ low 1)))
```

Of course, we can extend this to the case where `high` is equal to the value of `low` plus 2:

```
(max (f low) (max (f (+ low 1)) (f (+ low 2))))
```

The second operand for the outer `max` evaluation is the maximum value of the input procedure between the low value plus one and the high value input. If we name the procedure we are defining `find-maximum`, then this is the result of `(find-maximum f (+ low 1) high)`. This works whether `high` is equal to `low` plus 1, or `low` plus 2, or any other value greater than `high`! Putting things together, we have our recursive definition of `find-maximum`:

```
(define (find-maximum f low high)
    (if (= low high)
        (f low)
        (max (f low)
             (find-maximum f (+ low 1) high)))))
```

Here are a few examples:

```
> (find-maximum (lambda (x) x) 1 20)
20
> (find-maximum (lambda (x) (- 10 x)) 1 20)
9
> (find-maximum (lambda (x) (* x (- 10 x))) 1 20)
25
```

**Exercise 4.7.** (◇) To find the maximum of a continuous function, we need to evaluate at all numbers in the range, not just the integers. There are infinitely many numbers between any two numbers, however, so this is impossible. We can approximate this, however, by evaluating the function at many numbers in the range.

Define a procedure `find-maximum-continuous` that takes as input a function `f`, a low range value `low`, a high range value `high`, and an increment `inc`, and produces as output the maximum value of `f` in the range between `low` and `high` where `f` is evaluated at `low`, `(+ low inc)`, `(+ low inc inc)`, `(+ low inc inc inc)`, ..., `high`.

Here are some examples:

```
> (find-maximum-continuous
      (lambda (x) (* x (- 5.5 x)))
      1 10 1)
7.5
> (find-maximum-continuous
      (lambda (x) (* x (- 5.5 x)))
      1 10 0.0001)
7.5625
```

As the value of increment decreases, we expect to find a more accurate maximum value. ■

**Exercise 4.8.** (⋆) The `find-maximum` procedure we defined evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure that finds the input in the range that produces the maximum output value. ■

**Exercise 4.9.** (⋆) Define a `find-area` procedure that takes as input a function `f`, a low range value `low`, a high range value `high`, and an increment `inc`, and produces as output an estimate for the area under the curve produced by the function `f` between `low` and `high` using the `inc` value to determine how many points to evaluate. ■

### 4.4.1 The Evaluation Stack

We can see the structure of the evaluation process better if we focus just on the most revealing stems. Here are the evaluation steps that involve applications of the `factorial` procedure extracted from the full evaluation and indented to line up the steps from each application evaluation:

```
1.  (factorial 2)
17.     (* 2 (factorial 1))
31.         (* 2 (* 1 (factorial 0)))
40.         (* 2 (* 1 1))
41.     (* 2 1)
42. 3
```

In Step 1, we start to evaluate `(factorial 2)`. The final result of this evaluation is found in Step 42. To evaluate `(factorial 2)`, we follow the evaluation rules, eventually reaching the body expression of the if expression in the factorial definition. This is Step 17: `(* 2 (factorial 1))`. To evaluate this expression, we need to evaluate the `(factorial 1)` subexpression.

We can think of each of these application evaluations as a stack. A stack has the property that the first tray pushed on the stack will be the last tray removed—all

the trays pushed on top of this one must be removed before this tray can be removed. For application evaluations, the elements on the stack are expressions to evaluate instead of trays. To finish evaluating the first expression, all of its component subexpressions must be evaluated. Hence, the first application evaluation started will be the last one to finish.

At Step 17, the first evaluation is in progress, but to complete it we need the value resulting from the second evaluation. The second evaluation results in the body expression, `(* 1 (factorial 0))`, shown for Step 31. At this point, the evaluation of `(factorial 2)` is stuck in Evaluation Rule 3, waiting for the value of `(factorial 1)` subexpression. The evaluation of the `(factorial 1)` application leads to the `(factorial 0)` subexpression, which must be evaluated before the `(factorial 1)` evaluation can complete. In Step 40, the `(factorial 0)` subexpression evaluation has completed and produced the value 1. Now, the `(factorial 1)` evaluation can complete, producing 1 as shown in Step 41. Once the `(factorial 1)` evaluation completes, all the subexpressions needed to evaluate the expression in Step 17 are now evaluated, and the evaluation completes in Step 42.

**Example 4.3: Euclid's Algorithm.** In Book 7 of the *Elements*, Euclid describes an algorithm for finding the greatest common divisor of two non-zero integers. The greatest common divisor is the greatest integer that divides both of the input numbers without leaving any remainder. For example, the greatest common divisor of 150 and 200 is 50 since `(/ 150 50)` evaluates to 3 and `(/ 200 50)` evaluates to 4, and there is no number greater than 50 which can divide both 150 and 200 without leaving a remainder.

The `modulo` primitive procedure takes two integers as its inputs and evaluates to the remainder when the first input is divided by the second input. For example, `(modulo 6 3)` evaluates to 0 and `(modulo 7 3)` evaluates to 1.

Euclid's algorithm stems from two properties of integers:

1. If `(modulo a b)` evaluates to 0 then $b$ is the greatest common divisor of $a$ and $b$.

2. If `(modulo a b)` evaluates to a non-zero integer $r$, then the greatest common divisor of $a$ and $b$ is the greatest common divisor of $b$ and $r$.

We can define a recursive procedure for finding the greatest common divisor closely following Euclid's algorithm:

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

The structure of the definition is similar to the `factorial` definition: the procedure body is an if expression and the predicate tests for the base case. For the `gcd` procedure, the base case corresponds to the first property above. It occurs when `b` divides `a` evenly, and the consequent expression is `b`. The alternate expression, `(gcd b (modulo a b))`, is the recursive application. It differs from the alternate expression in the `factorial` definition in that there is no outer application expression (the $\star$ application). We do not need to combine the result of the recursive application with some other value as was done in the `factorial` definition, the result of the recursive application is the final result. Unlike the `factorial` and `find-maximum` examples, the `gcd` procedure produces the result in the base case, and no further computation is necessary to produce the final result. When no further evaluation is necessary to get from the result of the recursive application to the final result, a recursive definition is said to be *tail recursive*.

**Exercise 4.10.** Show the structure of the applications of `gcd` in evaluating

```
(gcd 6 9)
```

■

**Exercise 4.11.** ($\star$) Provide a convincing argument why the evaluation of `(gcd` *`a b`*`)` will always finish when the inputs are both positive integers. ■

**Exercise 4.12.** ($\star$) Provide an alternate definition of `factorial` that is tail recursive. To be tail recursive, the expression containing the recursive application cannot be part of another application expression.

Hint: define a `factorial-helper` procedure that takes an extra parameter, and then define `factorial` as:

```
(define (factorial n) (factorial-helper n 1))
```

■

**Exercise 4.13.** (⋆⋆) Provide an alternate definition of `find-max` that is tail recursive. ■

**Exercise 4.14.** (⋆⋆⋆) Provide a convincing argument why it is always possible to transform a recursive procedure into an equivalent procedure that is tail recursive. ■

## 4.5 Summary

By breaking problems down into simpler problems we can develop solutions to complex problems. Many problems can be solved by combining instances of the same problem on simpler inputs. When we define a procedure to solve a problem this way, it needs to have a predicate expression to determine when the base case has been reached, a consequent expression that provides the value for the base case, and an alternate expression that defines the solution to the given input as an expression using a solution to a slightly smaller input.

Our general problem solving strategy is:

1. Be optimistic! Assume you can solve it.
2. Think of the simplest version of the problem, something you can already solve. This is the base case.
3. Consider how you would solve a big version of the problem by using the result for a slightly smaller version of the problem. This is the recursive case.
4. Combine the base case and the recursive case to solve the problem.

For problems involving numbers, the base case will often be when the input value is 0 (but not always, as we saw in the `find-maximum` value, where the base

case is reached when the different between two of the input values is 0). The way the problem size is reduced is by subtracting some value (usually 1) from one of the inputs. In the next chapter, we introduce more complex data structures. For problems involving complex data, the same strategy will often work, but we will have to find other ways to identify a base case and to shrink the problem size.