3

Modularity, Objects, and State

> Μεταβάλλον ἀναπαύεται
> (Even while it changes, it stands still.)
> —Heraclitus
>
> Plus ça change, plus c'est la même chose.
> —Alphonse Karr

The preceding chapters introduced the basic elements from which programs are made. We saw how primitive procedures and primitive data are combined to construct compound entities, and we learned that abstraction is vital in helping us to cope with the complexity of large systems. But these tools are not sufficient for designing programs. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they

will be *modular*, that is, so that they can be divided "naturally" into coherent parts that can be separately developed and maintained.

One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of our programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successful in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

To a large extent, then, the way we organize a large program is dictated by our perception of the system to be modeled. In this chapter we will investigate two prominent organizational strategies arising from two rather different "world views" of the structure of systems. The first organizational strategy concentrates on *objects*, viewing a large system as a collection of distinct objects whose behaviors may change over time. An alternative organizational strategy concentrates on the *streams* of information that flow in the system, much as an electrical engineer views a signal-processing system.

Both the object-based approach and the stream-processing approach raise significant linguistic issues in programming. With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation (Section 1.1.5) in favor of a more mechanistic but less theoretically tractable *environment model* of computation. The difficulties of dealing with objects, change, and identity are a fundamental con-

sequence of the need to grapple with time in our computational models. These difficulties become even greater when we allow the possibility of concurrent execution of programs. The stream approach can be most fully exploited when we decouple simulated time in our model from the order of the events that take place in the computer during evaluation. We will accomplish this using a technique known as *delayed evaluation*.

3.1 Assignment and Local State

We ordinarily view the world as populated by independent objects, each of which has a state that changes over time. An object is said to "have state" if its behavior is influenced by its history. A bank account, for example, has state in that the answer to the question "Can I withdraw $100?" depends upon the history of deposit and withdrawal transactions. We can characterize an object's state by one or more *state variables*, which among them maintain enough information about history to determine the object's current behavior. In a simple banking system, we could characterize the state of an account by a current balance rather than by remembering the entire history of account transactions.

In a system composed of many objects, the objects are rarely completely independent. Each may influence the states of others through interactions, which serve to couple the state variables of one object to those of other objects. Indeed, the view that a system is composed of separate objects is most useful when the state variables of the system can be grouped into closely coupled subsystems that are only loosely coupled to other subsystems.

This view of a system can be a powerful framework for organizing computational models of the system. For such a model to be modular, it should be decomposed into computational objects that model the actual

objects in the system. Each computational object must have its own *local state variables* describing the actual object's state. Since the states of objects in the system being modeled change over time, the state variables of the corresponding computational objects must also change. If we choose to model the flow of time in the system by the elapsed time in the computer, then we must have a way to construct computational objects whose behaviors change as our programs run. In particular, if we wish to model state variables by ordinary symbolic names in the programming language, then the language must provide an *assignment operator* to enable us to change the value associated with a name.

3.1.1 Local State Variables

To illustrate what we mean by having a computational object with time-varying state, let us model the situation of withdrawing money from a bank account. We will do this using a procedure `withdraw`, which takes as argument an `amount` to be withdrawn. If there is enough money in the account to accommodate the withdrawal, then `withdraw` should return the balance remaining after the withdrawal. Otherwise, `withdraw` should return the message *Insufficient funds*. For example, if we begin with $100 in the account, we should obtain the following sequence of responses using `withdraw`:

```
(withdraw 25)
75
(withdraw 25)
50
(withdraw 60)
"Insufficient funds"
(withdraw 15)
35
```

Observe that the expression `(withdraw 25)`, evaluated twice, yields different values. This is a new kind of behavior for a procedure. Until now, all our procedures could be viewed as specifications for computing mathematical functions. A call to a procedure computed the value of the function applied to the given arguments, and two calls to the same procedure with the same arguments always produced the same result.[1]

To implement `withdraw`, we can use a variable `balance` to indicate the balance of money in the account and define `withdraw` as a procedure that accesses `balance`. The `withdraw` procedure checks to see if `balance` is at least as large as the requested `amount`. If so, `withdraw` decrements `balance` by `amount` and returns the new value of `balance`. Otherwise, `withdraw` returns the *Insufficient funds* message. Here are the definitions of `balance` and `withdraw`:

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

Decrementing `balance` is accomplished by the expression

```
(set! balance (- balance amount))
```

This uses the `set!` special form, whose syntax is

```
(set! ⟨name⟩ ⟨new-value⟩)
```

[1] Actually, this is not quite true. One exception was the random-number generator in Section 1.2.6. Another exception involved the operation/type tables we introduced in Section 2.4.3, where the values of two calls to `get` with the same arguments depended on intervening calls to `put`. On the other hand, until we introduce assignment, we have no way to create such procedures ourselves.

Here ⟨*name*⟩ is a symbol and ⟨*new-value*⟩ is any expression. `set!` changes ⟨*name*⟩ so that its value is the result obtained by evaluating ⟨*new-value*⟩. In the case at hand, we are changing `balance` so that its new value will be the result of subtracting `amount` from the previous value of `balance`.[2]

`withdraw` also uses the `begin` special form to cause two expressions to be evaluated in the case where the `if` test is true: first decrementing `balance` and then returning the value of `balance`. In general, evaluating the expression

```
(begin ⟨exp1⟩ ⟨exp2⟩ ... ⟨expk⟩)
```

causes the expressions ⟨exp_1⟩ through ⟨exp_k⟩ to be evaluated in sequence and the value of the final expression ⟨exp_k⟩ to be returned as the value of the entire `begin` form.[3]

Although `withdraw` works as desired, the variable `balance` presents a problem. As specified above, `balance` is a name defined in the global environment and is freely accessible to be examined or modified by any procedure. It would be much better if we could somehow make `balance` internal to `withdraw`, so that `withdraw` would be the only procedure that could access `balance` directly and any other procedure could access `balance` only indirectly (through calls to `withdraw`). This would more accurately model the notion that `balance` is a local state variable used

[2] The value of a `set!` expression is implementation-dependent. `set!` should be used only for its effect, not for its value.

The name `set!` reflects a naming convention used in Scheme: Operations that change the values of variables (or that change data structures, as we will see in Section 3.3) are given names that end with an exclamation point. This is similar to the convention of designating predicates by names that end with a question mark.

[3] We have already used `begin` implicitly in our programs, because in Scheme the body of a procedure can be a sequence of expressions. Also, the ⟨*consequent*⟩ part of each clause in a `cond` expression can be a sequence of expressions rather than a single expression.

by withdraw to keep track of the state of the account.

We can make balance internal to withdraw by rewriting the definition as follows:

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

What we have done here is use let to establish an environment with a local variable balance, bound to the initial value 100. Within this local environment, we use lambda to create a procedure that takes amount as an argument and behaves like our previous withdraw procedure. This procedure—returned as the result of evaluating the let expression—is new-withdraw, which behaves in precisely the same way as withdraw but whose variable balance is not accessible by any other procedure.[4]

Combining set! with local variables is the general programming technique we will use for constructing computational objects with local state. Unfortunately, using this technique raises a serious problem: When we first introduced procedures, we also introduced the substitution model of evaluation (Section 1.1.5) to provide an interpretation of what procedure application means. We said that applying a procedure should be interpreted as evaluating the body of the procedure with the formal parameters replaced by their values. The trouble is that, as

[4] In programming-language jargon, the variable balance is said to be *encapsulated* within the new-withdraw procedure. Encapsulation reflects the general system-design principle known as the *hiding principle*: One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a "need to know."

soon as we introduce assignment into our language, substitution is no longer an adequate model of procedure application. (We will see why this is so in Section 3.1.3.) As a consequence, we technically have at this point no way to understand why the `new-withdraw` procedure behaves as claimed above. In order to really understand a procedure such as `new-withdraw`, we will need to develop a new model of procedure application. In Section 3.2 we will introduce such a model, together with an explanation of `set!` and local variables. First, however, we examine some variations on the theme established by `new-withdraw`.

The following procedure, `make-withdraw`, creates "withdrawal processors." The formal parameter `balance` in `make-withdraw` specifies the initial amount of money in the account.[5]

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

`make-withdraw` can be used as follows to create two objects `W1` and `W2`:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))

(W1 50)
50
(W2 70)
30
```

[5] In contrast with `new-withdraw` above, we do not have to use `let` to make `balance` a local variable, since formal parameters are already local. This will be clearer after the discussion of the environment model of evaluation in Section 3.2. (See also Exercise 3.10.)

```
(W2 40)
"Insufficient funds"
(W1 40)
10
```

Observe that `W1` and `W2` are completely independent objects, each with its own local state variable `balance`. Withdrawals from one do not affect the other.

We can also create objects that handle deposits as well as withdrawals, and thus we can represent simple bank accounts. Here is a procedure that returns a "bank-account object" with a specified initial balance:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Each call to `make-account` sets up an environment with a local state variable `balance`. Within this environment, `make-account` defines procedures `deposit` and `withdraw` that access `balance` and an additional procedure `dispatch` that takes a "message" as input and returns one of

the two local procedures. The `dispatch` procedure itself is returned as the value that represents the bank-account object. This is precisely the *message-passing* style of programming that we saw in Section 2.4.3, although here we are using it in conjunction with the ability to modify local variables.

`make-account` can be used as follows:

```
(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

Each call to `acc` returns the locally defined `deposit` or `withdraw` procedure, which is then applied to the specified `amount`. As was the case with `make-withdraw`, another call to `make-account`

```
(define acc2 (make-account 100))
```

will produce a completely separate account object, which maintains its own local `balance`.

> **Exercise 3.1:** An *accumulator* is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum; for example

```
(define A (make-accumulator 5))
(A 10)
15
(A 10)
25
```

Exercise 3.2: In software-testing applications, it is useful to be able to count the number of times a given procedure is called during the course of a computation. Write a procedure `make-monitored` that takes as input a procedure, `f`, that itself takes one input. The result returned by `make-monitored` is a third procedure, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the special symbol `how-many-calls?`, then `mf` returns the value of the counter. If the input is the special symbol `reset-count`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` procedure:

```
(define s (make-monitored sqrt))
(s 100)
10
(s 'how-many-calls?)
1
```

Exercise 3.3: Modify the `make-account` procedure so that it creates password-protected accounts. That is, `make-account` should take a symbol as an additional argument, as in

```
(define acc (make-account 100 'secret-password))
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
((acc 'secret-password 'withdraw) 40)
60
((acc 'some-other-password 'deposit) 50)
"Incorrect password"
```

Exercise 3.4: Modify the `make-account` procedure of Exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure `call-the-cops`.

3.1.2 The Benefits of Introducing Assignment

As we shall see, introducing assignment into our programming language leads us into a thicket of difficult conceptual issues. Nevertheless, viewing systems as collections of objects with local state is a powerful technique for maintaining a modular design. As a simple example, consider the design of a procedure `rand` that, whenever it is called, returns an integer chosen at random.

It is not at all clear what is meant by "chosen at random." What we presumably want is for successive calls to `rand` to produce a sequence of numbers that has statistical properties of uniform distribution. We will not discuss methods for generating suitable sequences here. Rather, let us assume that we have a procedure `rand-update` that has the property that if we start with a given number x_1 and form

x_2 = (rand-update x_1)
x_3 = (rand-update x_2)

then the sequence of values $x_1, x_2, x_3, \ldots$ will have the desired statistical properties.[6]

We can implement `rand` as a procedure with a local state variable `x` that is initialized to some fixed value `random-init`. Each call to `rand` computes `rand-update` of the current value of `x`, returns this as the random number, and also stores this as the new value of `x`.

```
(define rand (let ((x random-init))
               (lambda ()
                 (set! x (rand-update x))
                 x)))
```

Of course, we could generate the same sequence of random numbers without using assignment by simply calling `rand-update` directly. However, this would mean that any part of our program that used random numbers would have to explicitly remember the current value of `x` to be passed as an argument to `rand-update`. To realize what an annoyance this would be, consider using random numbers to implement a technique called *Monte Carlo simulation*.

The Monte Carlo method consists of choosing sample experiments at random from a large set and then making deductions on the basis of

[6] One common way to implement `rand-update` is to use the rule that x is updated to $ax + b$ modulo m, where a, b, and m are appropriately chosen integers. Chapter 3 of Knuth 1981 includes an extensive discussion of techniques for generating sequences of random numbers and establishing their statistical properties. Notice that the `rand-update` procedure computes a mathematical function: Given the same input twice, it produces the same output. Therefore, the number sequence produced by `rand-update` certainly is not "random," if by "random" we insist that each number in the sequence is unrelated to the preceding number. The relation between "real randomness" and so-called *pseudo-random* sequences, which are produced by well-determined computations and yet have suitable statistical properties, is a complex question involving difficult issues in mathematics and philosophy. Kolmogorov, Solomonoff, and Chaitin have made great progress in clarifying these issues; a discussion can be found in Chaitin 1975.

the probabilities estimated from tabulating the results of those experiments. For example, we can approximate π using the fact that $6/\pi^2$ is the probability that two integers chosen at random will have no factors in common; that is, that their greatest common divisor will be 1.[7] To obtain the approximation to π, we perform a large number of experiments. In each experiment we choose two integers at random and perform a test to see if their GCD is 1. The fraction of times that the test is passed gives us our estimate of $6/\pi^2$, and from this we obtain our approximation to π.

The heart of our program is a procedure `monte-carlo`, which takes as arguments the number of times to try an experiment, together with the experiment, represented as a no-argument procedure that will return either true or false each time it is run. `monte-carlo` runs the experiment for the designated number of trials and returns a number telling the fraction of the trials in which the experiment was found to be true.

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else
```

[7] This theorem is due to E. Cesàro. See section 4.5.2 of Knuth 1981 for a discussion and a proof.

```
          (iter (- trials-remaining 1)
                trials-passed))))
  (iter trials 0))
```

Now let us try the same computation using `rand-update` directly rather than `rand`, the way we would be forced to proceed if we did not use assignment to model local state:

```
(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))
(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2))
              (else
               (iter (- trials-remaining 1)
                     trials-passed
                     x2))))))
  (iter trials 0 initial-x))
```

While the program is still simple, it betrays some painful breaches of modularity. In our first version of the program, using `rand`, we can express the Monte Carlo method directly as a general `monte-carlo` procedure that takes as an argument an arbitrary `experiment` procedure. In our second version of the program, with no local state for the random-number generator, `random-gcd-test` must explicitly manipulate the random numbers `x1` and `x2` and recycle `x2` through the iterative loop as the new input to `rand-update`. This explicit handling of the random

numbers intertwines the structure of accumulating test results with the fact that our particular experiment uses two random numbers, whereas other Monte Carlo experiments might use one random number or three. Even the top-level procedure `estimate-pi` has to be concerned with supplying an initial random number. The fact that the random-number generator's insides are leaking out into other parts of the program makes it difficult for us to isolate the Monte Carlo idea so that it can be applied to other tasks. In the first version of the program, assignment encapsulates the state of the random-number generator within the `rand` procedure, so that the details of random-number generation remain independent of the rest of the program.

The general phenomenon illustrated by the Monte Carlo example is this: From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write computer programs whose structure reflects this decomposition, we make computational objects (such as bank accounts and random-number generators) whose behavior changes with time. We model state with local state variables, and we model the changes of state with assignments to those variables.

It is tempting to conclude this discussion by saying that, by introducing assignment and the technique of hiding state in local variables, we are able to structure systems in a more modular fashion than if all state had to be manipulated explicitly, by passing additional parameters. Unfortunately, as we shall see, the story is not so simple.

> **Exercise 3.5:** *Monte Carlo integration* is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate $P(x, y)$ that is true for points (x, y) in the region and false for points not in the region. For

example, the region contained within a circle of radius 3 centered at (5, 7) is described by the predicate that tests whether $(x - 5)^2 + (y - 7)^2 \leq 3^2$. To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at (2, 4) and (8, 10) contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing $P(x, y)$ for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

Implement Monte Carlo integration as a procedure `estimate-integral` that takes as arguments a predicate `P`, upper and lower bounds `x1`, `x2`, `y1`, and `y2` for the rectangle, and the number of trials to perform in order to produce the estimate. Your procedure should use the same `monte-carlo` procedure that was used above to estimate π. Use your `estimate-integral` to produce an estimate of π by measuring the area of a unit circle.

You will find it useful to have a procedure that returns a number chosen at random from a given range. The following `random-in-range` procedure implements this in terms of the `random` procedure used in Section 1.2.6, which re-

turns a nonnegative number less than its input.[8]

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```

Exercise 3.6: It is useful to be able to reset a random-number generator to produce a sequence starting from a given value. Design a new `rand` procedure that is called with an argument that is either the symbol `generate` or the symbol `reset` and behaves as follows: `(rand 'generate)` produces a new random number; `((rand 'reset)` ⟨*new-value*⟩`)` resets the internal state variable to the designated ⟨*new-value*⟩. Thus, by resetting the state, one can generate repeatable sequences. These are very handy to have when testing and debugging programs that use random numbers.

3.1.3 The Costs of Introducing Assignment

As we have seen, the `set!` operation enables us to model objects that have local state. However, this advantage comes at a price. Our programming language can no longer be interpreted in terms of the substitution model of procedure application that we introduced in Section 1.1.5. Moreover, no simple model with "nice" mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages.

So long as we do not use assignments, two evaluations of the same procedure with the same arguments will produce the same result, so

[8] MIT Scheme provides such a procedure. If `random` is given an exact integer (as in Section 1.2.6) it returns an exact integer, but if it is given a decimal value (as in this exercise) it returns a decimal value.

that procedures can be viewed as computing mathematical functions. Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as *functional programming*.

To understand how assignment complicates matters, consider a simplified version of the `make-withdraw` procedure of Section 3.1.1 that does not bother to check for an insufficient amount:

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

Compare this procedure with the following `make-decrementer` procedure, which does not use `set!`:

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))
```

`make-decrementer` returns a procedure that subtracts its input from a designated amount `balance`, but there is no accumulated effect over successive calls, as with `make-simplified-withdraw`:

```
(define D (make-decrementer 25))
(D 20)
5
(D 10)
15
```

We can use the substitution model to explain how `make-decrementer` works. For instance, let us analyze the evaluation of the expression

```
((make-decrementer 25) 20)
```

We first simplify the operator of the combination by substituting 25 for `balance` in the body of `make-decrementer`. This reduces the expression to

```
((lambda (amount) (- 25 amount)) 20)
```

Now we apply the operator by substituting 20 for `amount` in the body of the `lambda` expression:

```
(- 25 20)
```

The final answer is 5.

Observe, however, what happens if we attempt a similar substitution analysis with `make-simplified-withdraw`:

```
((make-simplified-withdraw 25) 20)
```

We first simplify the operator by substituting 25 for `balance` in the body of `make-simplified-withdraw`. This reduces the expression to[9]

```
((lambda (amount) (set! balance (- 25 amount)) 25) 20)
```

Now we apply the operator by substituting 20 for `amount` in the body of the `lambda` expression:

```
(set! balance (- 25 20)) 25
```

If we adhered to the substitution model, we would have to say that the meaning of the procedure application is to first set `balance` to 5 and then

[9] We don't substitute for the occurrence of `balance` in the `set!` expression because the ⟨*name*⟩ in a `set!` is not evaluated. If we did substitute for it, we would get `(set! 25 (- 25 amount))`, which makes no sense.

return 25 as the value of the expression. This gets the wrong answer. In order to get the correct answer, we would have to somehow distinguish the first occurrence of `balance` (before the effect of the `set!`) from the second occurrence of `balance` (after the effect of the `set!`), and the substitution model cannot do this.

The trouble here is that substitution is based ultimately on the notion that the symbols in our language are essentially names for values. But as soon as we introduce `set!` and the idea that the value of a variable can change, a variable can no longer be simply a name. Now a variable somehow refers to a place where a value can be stored, and the value stored at this place can change. In Section 3.2 we will see how environments play this role of "place" in our computational model.

Sameness and change

The issue surfacing here is more profound than the mere breakdown of a particular model of computation. As soon as we introduce change into our computational models, many notions that were previously straightforward become problematical. Consider the concept of two things being "the same."

Suppose we call `make-decrementer` twice with the same argument to create two procedures:

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
```

Are `D1` and `D2` the same? An acceptable answer is yes, because `D1` and `D2` have the same computational behavior—each is a procedure that subtracts its input from 25. In fact, `D1` could be substituted for `D2` in any computation without changing the result.

Contrast this with making two calls to `make-simplified-withdraw`:

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
```

Are `W1` and `W2` the same? Surely not, because calls to `W1` and `W2` have distinct effects, as shown by the following sequence of interactions:

```
(W1 20)
5
(W1 20)
-15
(W2 20)
5
```

Even though `W1` and `W2` are "equal" in the sense that they are both created by evaluating the same expression, `(make-simplified-withdraw 25)`, it is not true that `W1` could be substituted for `W2` in any expression without changing the result of evaluating the expression.

A language that supports the concept that "equals can be substituted for equals" in an expression without changing the value of the expression is said to be *referentially transparent*. Referential transparency is violated when we include `set!` in our computer language. This makes it tricky to determine when we can simplify expressions by substituting equivalent expressions. Consequently, reasoning about programs that use assignment becomes drastically more difficult.

Once we forgo referential transparency, the notion of what it means for computational objects to be "the same" becomes difficult to capture in a formal way. Indeed, the meaning of "same" in the real world that our programs model is hardly clear in itself. In general, we can determine that two apparently identical objects are indeed "the same one" only by modifying one object and then observing whether the other object has changed in the same way. But how can we tell if an object has "changed" other than by observing the "same" object twice and seeing whether

some property of the object differs from one observation to the next? Thus, we cannot determine "change" without some *a priori* notion of "sameness," and we cannot determine sameness without observing the effects of change.

As an example of how this issue arises in programming, consider the situation where Peter and Paul have a bank account with $100 in it. There is a substantial difference between modeling this as

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

and modeling it as

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

In the first situation, the two bank accounts are distinct. Transactions made by Peter will not affect Paul's account, and vice versa. In the second situation, however, we have defined `paul-acc` to be *the same thing* as `peter-acc`. In effect, Peter and Paul now have a joint bank account, and if Peter makes a withdrawal from `peter-acc` Paul will observe less money in `paul-acc`. These two similar but distinct situations can cause confusion in building computational models. With the shared account, in particular, it can be especially confusing that there is one object (the bank account) that has two different names (`peter-acc` and `paul-acc`); if we are searching for all the places in our program where `paul-acc` can be changed, we must remember to look also at things that change `peter-acc`.[10]

[10] The phenomenon of a single computational object being accessed by more than one name is known as *aliasing*. The joint bank account situation illustrates a very simple example of an alias. In Section 3.3 we will see much more complex examples, such as "distinct" compound data structures that share parts. Bugs can occur in our programs

With reference to the above remarks on "sameness" and "change," observe that if Peter and Paul could only examine their bank balances, and could not perform operations that changed the balance, then the issue of whether the two accounts are distinct would be moot. In general, so long as we never modify data objects, we can regard a compound data object to be precisely the totality of its pieces. For example, a rational number is determined by giving its numerator and its denominator. But this view is no longer valid in the presence of change, where a compound data object has an "identity" that is something different from the pieces of which it is composed. A bank account is still "the same" bank account even if we change the balance by making a withdrawal; conversely, we could have two different bank accounts with the same state information. This complication is a consequence, not of our programming language, but of our perception of a bank account as an object. We do not, for example, ordinarily regard a rational number as a changeable object with identity, such that we could change the numerator and still have "the same" rational number.

Pitfalls of imperative programming

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*. In addition to raising complications about computational models, programs written in imperative style are susceptible to bugs that cannot occur in functional programs. For example, recall the iterative factorial program

if we forget that a change to an object may also, as a "side effect," change a "different" object because the two "different" objects are actually a single object appearing under different aliases. These so-called *side-effect bugs* are so difficult to locate and to analyze that some people have proposed that programming languages be designed in such a way as to not allow side effects or aliasing (Lampson et al. 1981; Morris et al. 1980).

from Section 1.2.1:

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

Instead of passing arguments in the internal iterative loop, we could adopt a more imperative style by using explicit assignment to update the values of the variables `product` and `counter`:

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

This does not change the results produced by the program, but it does introduce a subtle trap. How do we decide the order of the assignments? As it happens, the program is correct as written. But writing the assignments in the opposite order

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

would have produced a different, incorrect result. In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct

version of the variables that have been changed. This issue simply does not arise in functional programs.[11]

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently. We will return to this in Section 3.4. First, however, we will address the issue of providing a computational model for expressions that involve assignment, and explore the uses of objects with local state in designing simulations.

> **Exercise 3.7:** Consider the bank account objects created by `make-account`, with the password modification described in Exercise 3.3. Suppose that our banking system requires the ability to make joint accounts. Define a procedure `make-joint` that accomplishes this. `make-joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make-joint` operation to proceed. The third argument is a new password. `make-joint` is to create an additional access to the original account using the new password. For example, if `peter-acc` is a bank account with password `open-sesame`, then
>
> ```
> (define paul-acc
>   (make-joint peter-acc 'open-sesame 'rosebud))
> ```

[11] In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call procedures must inherently be less efficient than programs that perform assignments. (Steele 1977 debunks this argument.) Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than procedure call. Whatever the reason, it often saddles beginning programmers with "should I set this variable before or after that one" concerns that can complicate programming and obscure the important ideas.

will allow one to make transactions on `peter-acc` using the name `paul-acc` and the password `rosebud`. You may wish to modify your solution to Exercise 3.3 to accommodate this new feature.

Exercise 3.8: When we defined the evaluation model in Section 1.1.3, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a difference to the result. Define a simple procedure `f` such that evaluating

```
(+ (f 0) (f 1))
```

will return 0 if the arguments to + are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

3.2 The Environment Model of Evaluation

When we introduced compound procedures in Chapter 1, we used the substitution model of evaluation (Section 1.1.5) to define what is meant by applying a procedure to arguments:

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

Once we admit assignment into our programming language, such a definition is no longer adequate. In particular, Section 3.1.3 argued that, in

the presence of assignment, a variable can no longer be considered to be merely a name for a value. Rather, a variable must somehow designate a "place" in which values can be stored. In our new model of evaluation, these places will be maintained in structures called *environments*.

An environment is a sequence of *frames*. Each frame is a table (possibly empty) of *bindings*, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its *enclosing environment*, unless, for the purposes of discussion, the frame is considered to be *global*. The *value of a variable* with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound* in the environment.

Figure 3.1 shows a simple environment structure consisting of three frames, labeled I, II, and III. In the diagram, A, B, C, and D are pointers to environments. C and D point to the same environment. The variables z and x are bound in frame II, while y and x are bound in frame I. The value of x in environment D is 3. The value of x with respect to environment B is also 3. This is determined as follows: We examine the first frame in the sequence (frame III) and do not find a binding for x, so we proceed to the enclosing environment D and find the binding in frame I. On the other hand, the value of x in environment A is 7, because the first frame in the sequence (frame II) contains a binding of x to 7. With respect to environment A, the binding of x to 7 in frame II is said to *shadow* the binding of x to 3 in frame I.

The environment is crucial to the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions in a programming language do

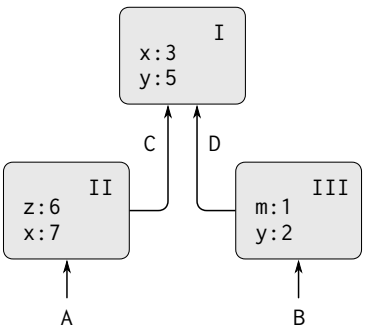

Figure 3.1: A simple environment structure.

not, in themselves, have any meaning. Rather, an expression acquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as `(+ 1 1)` depends on an understanding that one is operating in a context in which + is the symbol for addition. Thus, in our model of evaluation we will always speak of evaluating an expression with respect to some environment. To describe interactions with the interpreter, we will suppose that there is a global environment, consisting of a single frame (with no enclosing environment) that includes values for the symbols associated with the primitive procedures. For example, the idea that + is the symbol for addition is captured by saying that the symbol + is bound in the global environment to the primitive addition procedure.

3.2.1 The Rules for Evaluation

The overall specification of how the interpreter evaluates a combination remains the same as when we first introduced it in Section 1.1.3:

- To evaluate a combination:

1. Evaluate the subexpressions of the combination.[12]
2. Apply the value of the operator subexpression to the values of the operand subexpressions.

The environment model of evaluation replaces the substitution model in specifying what it means to apply a compound procedure to arguments.

In the environment model of evaluation, a procedure is always a pair consisting of some code and a pointer to an environment. Procedures are created in one way only: by evaluating a λ-expression. This produces a procedure whose code is obtained from the text of the λ-expression and whose environment is the environment in which the λ-expression was evaluated to produce the procedure. For example, consider the procedure definition

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

evaluated in the global environment. The procedure definition syntax is just syntactic sugar for an underlying implicit λ-expression. It would have been equivalent to have used

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

[12] Assignment introduces a subtlety into step 1 of the evaluation rule. As shown in Exercise 3.8, the presence of assignment allows us to write expressions that will produce different values depending on the order in which the subexpressions in a combination are evaluated. Thus, to be precise, we should specify an evaluation order in step 1 (e.g., left to right or right to left). However, this order should always be considered to be an implementation detail, and one should never write programs that depend on some particular order. For instance, a sophisticated compiler might optimize a program by varying the order in which subexpressions are evaluated.

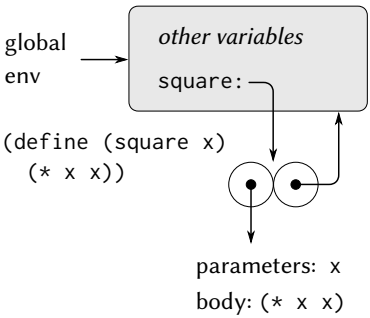

Figure 3.2: Environment structure produced by evaluating `(define (square x) (* x x))` in the global environment.

which evaluates `(lambda (x) (* x x))` and binds `square` to the resulting value, all in the global environment.

Figure 3.2 shows the result of evaluating this `define` expression. The procedure object is a pair whose code specifies that the procedure has one formal parameter, namely `x`, and a procedure body `(* x x)`. The environment part of the procedure is a pointer to the global environment, since that is the environment in which the λ-expression was evaluated to produce the procedure. A new binding, which associates the procedure object with the symbol `square`, has been added to the global frame. In general, `define` creates definitions by adding bindings to frames.

Now that we have seen how procedures are created, we can describe how procedures are applied. The environment model specifies: To apply a procedure to arguments, create a new environment containing a frame that binds the parameters to the values of the arguments. The enclosing environment of this frame is the environment specified by the

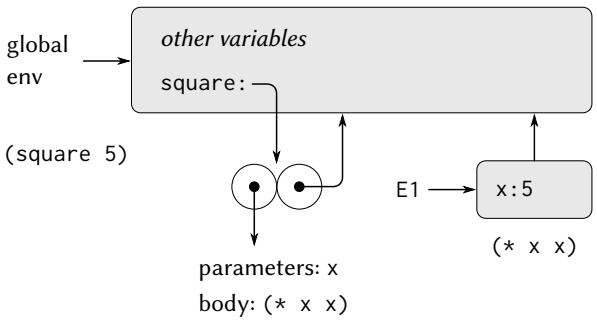

Figure 3.3: Environment created by evaluating `(square 5)` in the global environment.

procedure. Now, within this new environment, evaluate the procedure body.

To show how this rule is followed, Figure 3.3 illustrates the environment structure created by evaluating the expression `(square 5)` in the global environment, where `square` is the procedure generated in Figure 3.2. Applying the procedure results in the creation of a new environment, labeled E1 in the figure, that begins with a frame in which `x`, the formal parameter for the procedure, is bound to the argument 5. The pointer leading upward from this frame shows that the frame's enclosing environment is the global environment. The global environment is chosen here, because this is the environment that is indicated as part of the `square` procedure object. Within E1, we evaluate the body of the procedure, `(* x x)`. Since the value of `x` in E1 is 5, the result is `(* 5 5)`, or 25.

The environment model of procedure application can be summarized by two rules:

- A procedure object is applied to a set of arguments by constructing a frame, binding the formal parameters of the procedure to the arguments of the call, and then evaluating the body of the procedure in the context of the new environment constructed. The new frame has as its enclosing environment the environment part of the procedure object being applied.

- A procedure is created by evaluating a λ-expression relative to a given environment. The resulting procedure object is a pair consisting of the text of the λ-expression and a pointer to the environment in which the procedure was created.

We also specify that defining a symbol using `define` creates a binding in the current environment frame and assigns to the symbol the indicated value.[13] Finally, we specify the behavior of `set!`, the operation that forced us to introduce the environment model in the first place. Evaluating the expression `(set!` ⟨*variable*⟩ ⟨*value*⟩`)` in some environment locates the binding of the variable in the environment and changes that binding to indicate the new value. That is, one finds the first frame in the environment that contains a binding for the variable and modifies that frame. If the variable is unbound in the environment, then `set!` signals an error.

These evaluation rules, though considerably more complex than the substitution model, are still reasonably straightforward. Moreover, the evaluation model, though abstract, provides a correct description of

[13] If there is already a binding for the variable in the current frame, then the binding is changed. This is convenient because it allows redefinition of symbols; however, it also means that `define` can be used to change values, and this brings up the issues of assignment without explicitly using `set!`. Because of this, some people prefer redefinitions of existing symbols to signal errors or warnings.

how the interpreter evaluates expressions. In Chapter 4 we shall see how this model can serve as a blueprint for implementing a working interpreter. The following sections elaborate the details of the model by analyzing some illustrative programs.

3.2.2 Applying Simple Procedures

When we introduced the substitution model in Section 1.1.5 we showed how the combination `(f 5)` evaluates to 136, given the following procedure definitions:

```
(define (square x)
  (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

We can analyze the same example using the environment model. Figure 3.4 shows the three procedure objects created by evaluating the definitions of `f`, `square`, and `sum-of-squares` in the global environment. Each procedure object consists of some code, together with a pointer to the global environment.

In Figure 3.5 we see the environment structure created by evaluating the expression `(f 5)`. The call to `f` creates a new environment E1 beginning with a frame in which `a`, the formal parameter of `f`, is bound to the argument 5. In E1, we evaluate the body of `f`:

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

To evaluate this combination, we first evaluate the subexpressions. The first subexpression, `sum-of-squares`, has a value that is a procedure object. (Notice how this value is found: We first look in the first frame of

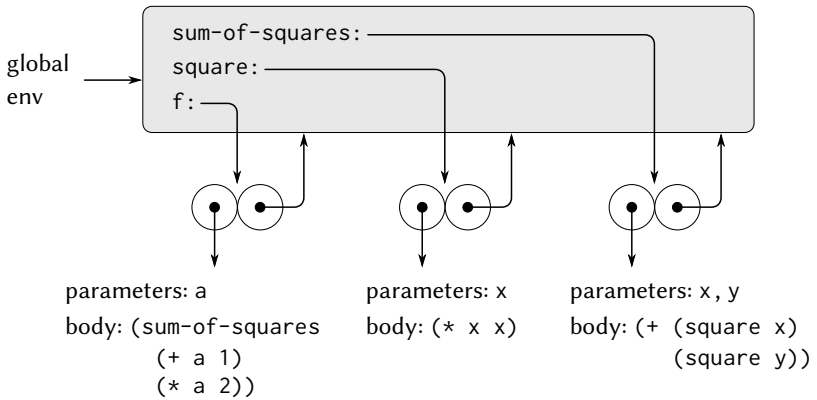

Figure 3.4: Procedure objects in the global frame.

E1, which contains no binding for `sum-of-squares`. Then we proceed to the enclosing environment, i.e. the global environment, and find the binding shown in Figure 3.4.) The other two subexpressions are evaluated by applying the primitive operations `+` and `*` to evaluate the two combinations `(+ a 1)` and `(* a 2)` to obtain 6 and 10, respectively.

Now we apply the procedure object `sum-of-squares` to the arguments 6 and 10. This results in a new environment E2 in which the formal parameters `x` and `y` are bound to the arguments. Within E2 we evaluate the combination `(+ (square x) (square y))`. This leads us to evaluate `(square x)`, where `square` is found in the global frame and `x` is 6. Once again, we set up a new environment, E3, in which `x` is bound to 6, and within this we evaluate the body of `square`, which is `(* x x)`. Also as part of applying `sum-of-squares`, we must evaluate the subexpression `(square y)`, where `y` is 10. This second call to `square` creates another environment, E4, in which `x`, the formal parameter of `square`, is bound to 10. And within E4 we must evaluate `(* x x)`.

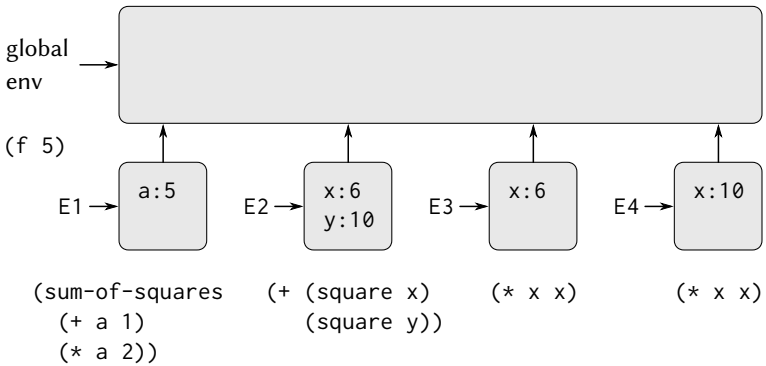

Figure 3.5: Environments created by evaluating `(f 5)` using the procedures in Figure 3.4.

The important point to observe is that each call to `square` creates a new environment containing a binding for `x`. We can see here how the different frames serve to keep separate the different local variables all named `x`. Notice that each frame created by `square` points to the global environment, since this is the environment indicated by the `square` procedure object.

After the subexpressions are evaluated, the results are returned. The values generated by the two calls to `square` are added by `sum-of-squares`, and this result is returned by `f`. Since our focus here is on the environment structures, we will not dwell on how these returned values are passed from call to call; however, this is also an important aspect of the evaluation process, and we will return to it in detail in Chapter 5.

Exercise 3.9: In Section 1.2.1 we used the substitution model to analyze two procedures for computing factorials, a recursive version

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

and an iterative version

```
(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)))
```

Show the environment structures created by evaluating `(factorial 6)` using each version of the `factorial` procedure.[14]

3.2.3 Frames as the Repository of Local State

We can turn to the environment model to see how procedures and assignment can be used to represent objects with local state. As an example, consider the "withdrawal processor" from Section 3.1.1 created by calling the procedure

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

[14] The environment model will not clarify our claim in Section 1.2.1 that the interpreter can execute a procedure such as `fact-iter` in a constant amount of space using tail recursion. We will discuss tail recursion when we deal with the control structure of the interpreter in Section 5.4.

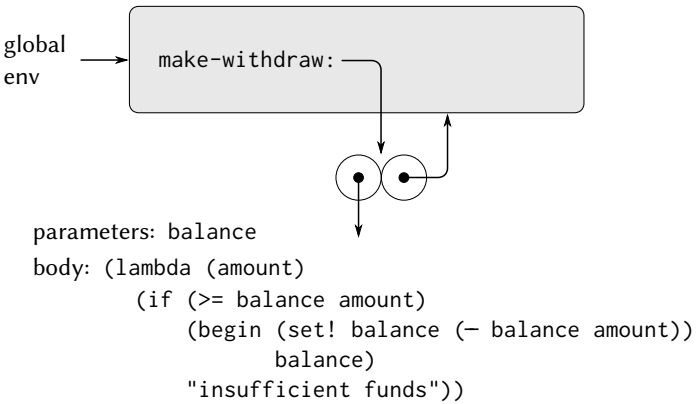

Figure 3.6: Result of defining `make-withdraw` in the global environment.

Let us describe the evaluation of

```
(define W1 (make-withdraw 100))
```

followed by

```
(W1 50)
50
```

Figure 3.6 shows the result of defining the `make-withdraw` procedure in the global environment. This produces a procedure object that contains a pointer to the global environment. So far, this is no different from the examples we have already seen, except that the body of the procedure is itself a λ-expression.

The interesting part of the computation happens when we apply the procedure `make-withdraw` to an argument:

```
(define W1 (make-withdraw 100))
```

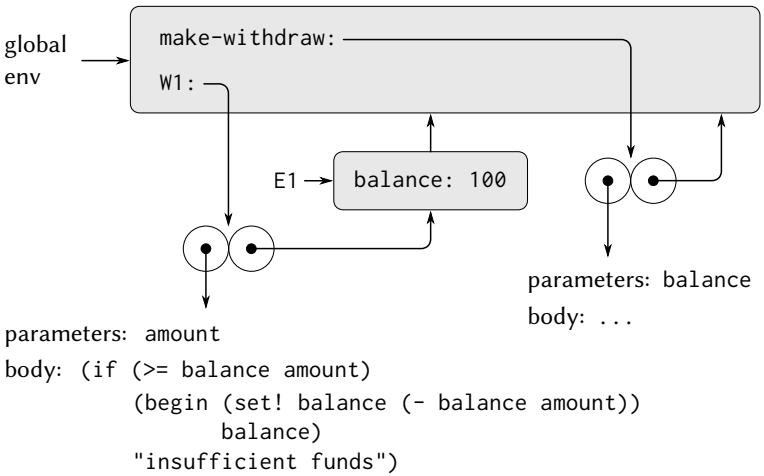

Figure 3.7: Result of evaluating `(define W1 (make-withdraw 100))`.

We begin, as usual, by setting up an environment E1 in which the formal parameter `balance` is bound to the argument 100. Within this environment, we evaluate the body of `make-withdraw`, namely the λ-expression. This constructs a new procedure object, whose code is as specified by the `lambda` and whose environment is E1, the environment in which the `lambda` was evaluated to produce the procedure. The resulting procedure object is the value returned by the call to `make-withdraw`. This is bound to `W1` in the global environment, since the `define` itself is being evaluated in the global environment. Figure 3.7 shows the resulting environment structure.

Now we can analyze what happens when `W1` is applied to an argument:

```
(W1 50)
50
```

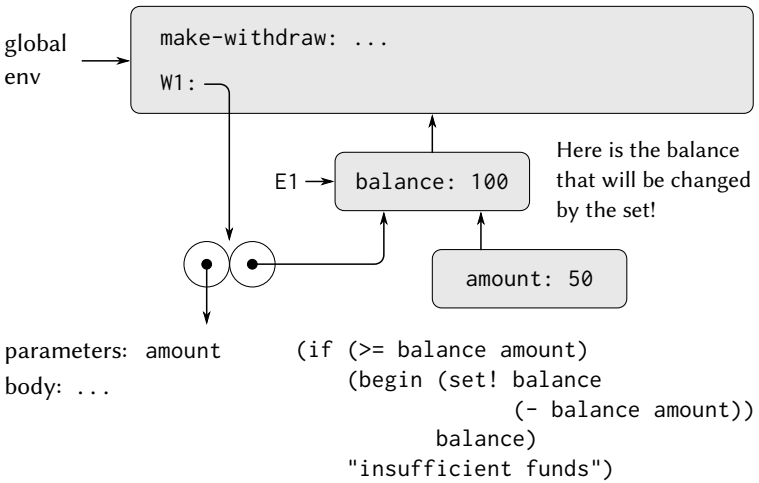

Figure 3.8: Environments created by applying the procedure object `W1`.

We begin by constructing a frame in which `amount`, the formal parameter of `W1`, is bound to the argument 50. The crucial point to observe is that this frame has as its enclosing environment not the global environment, but rather the environment E1, because this is the environment that is specified by the `W1` procedure object. Within this new environment, we evaluate the body of the procedure:

```
(if (>= balance amount)
    (begin (set! balance (- balance amount))
           balance)
    "Insufficient funds")
```

The resulting environment structure is shown in Figure 3.8. The expression being evaluated references both `amount` and `balance`. `amount` will be found in the first frame in the environment, while `balance` will be

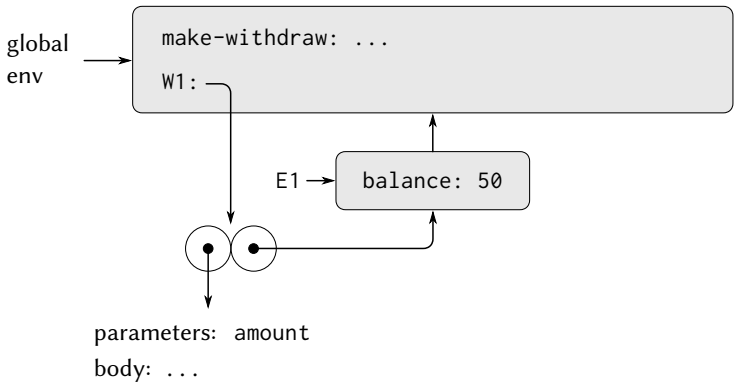

Figure 3.9: Environments after the call to `W1`.

found by following the enclosing-environment pointer to E1.

When the `set!` is executed, the binding of `balance` in E1 is changed. At the completion of the call to `W1`, `balance` is 50, and the frame that contains `balance` is still pointed to by the procedure object `W1`. The frame that binds `amount` (in which we executed the code that changed `balance`) is no longer relevant, since the procedure call that constructed it has terminated, and there are no pointers to that frame from other parts of the environment. The next time `W1` is called, this will build a new frame that binds `amount` and whose enclosing environment is E1. We see that E1 serves as the "place" that holds the local state variable for the procedure object `W1`. Figure 3.9 shows the situation after the call to `W1`.

Observe what happens when we create a second "withdraw" object by making another call to `make-withdraw`:

```
(define W2 (make-withdraw 100))
```

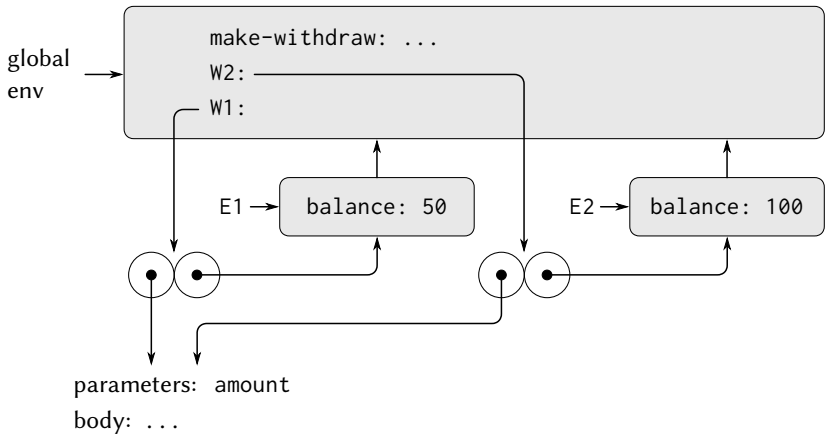

Figure 3.10: Using `(define W2 (make-withdraw 100))` to create a second object.

This produces the environment structure of Figure 3.10, which shows that `W2` is a procedure object, that is, a pair with some code and an environment. The environment E2 for `W2` was created by the call to `make-withdraw`. It contains a frame with its own local binding for `balance`. On the other hand, `W1` and `W2` have the same code: the code specified by the λ-expression in the body of `make-withdraw`.[15] We see here why `W1` and `W2` behave as independent objects. Calls to `W1` reference the state variable `balance` stored in E1, whereas calls to `W2` reference the `balance` stored in E2. Thus, changes to the local state of one object do not affect the other object.

[15] Whether `W1` and `W2` share the same physical code stored in the computer, or whether they each keep a copy of the code, is a detail of the implementation. For the interpreter we implement in Chapter 4, the code is in fact shared.

Exercise 3.10: In the `make-withdraw` procedure, the local variable `balance` is created as a parameter of `make-withdraw`. We could also create the local state variable explicitly, using `let`, as follows:

```
(define (make-withdraw initial-amount)
  (let ((balance initial-amount))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

Recall from Section 1.3.2 that `let` is simply syntactic sugar for a procedure call:

```
(let ((⟨var⟩ ⟨exp⟩)) ⟨body⟩)
```

is interpreted as an alternate syntax for

```
((lambda (⟨var⟩) ⟨body⟩) ⟨exp⟩)
```

Use the environment model to analyze this alternate version of `make-withdraw`, drawing figures like the ones above to illustrate the interactions

```
(define W1 (make-withdraw 100))
(W1 50)
(define W2 (make-withdraw 100))
```

Show that the two versions of `make-withdraw` create objects with the same behavior. How do the environment structures differ for the two versions?

3.2.4 Internal Definitions

Section 1.1.8 introduced the idea that procedures can have internal definitions, thus leading to a block structure as in the following procedure to compute square roots:

```
(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))
```

Now we can use the environment model to see why these internal definitions behave as desired. Figure 3.11 shows the point in the evaluation of the expression `(sqrt 2)` where the internal procedure `good-enough?` has been called for the first time with `guess` equal to 1.

Observe the structure of the environment. `sqrt` is a symbol in the global environment that is bound to a procedure object whose associated environment is the global environment. When `sqrt` was called, a new environment E1 was formed, subordinate to the global environment, in which the parameter `x` is bound to 2. The body of `sqrt` was then evaluated in E1. Since the first expression in the body of `sqrt` is

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

evaluating this expression defined the procedure `good-enough?` in the environment E1. To be more precise, the symbol `good-enough?` was added to the first frame of E1, bound to a procedure object whose asso-

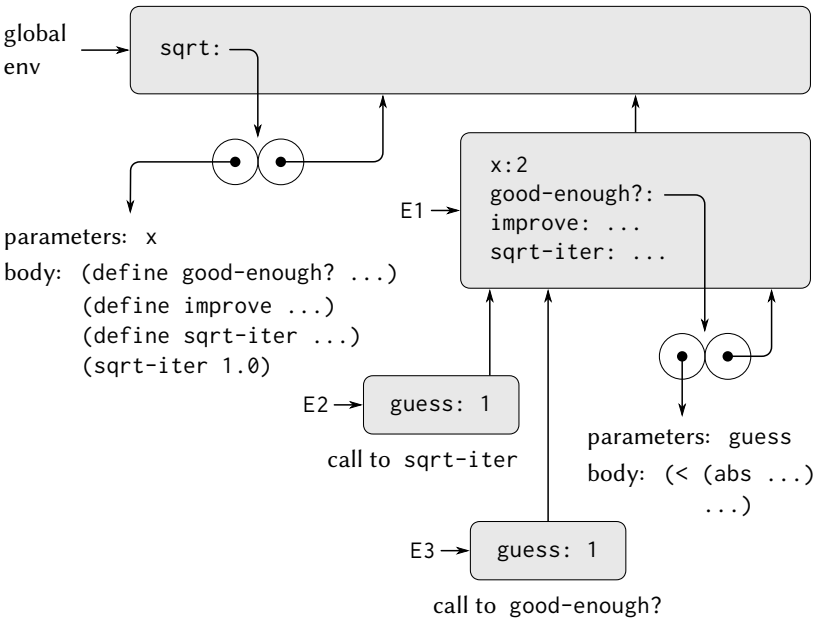

Figure 3.11: `sqrt` procedure with internal definitions.

ciated environment is E1. Similarly, `improve` and `sqrt-iter` were defined as procedures in E1. For conciseness, Figure 3.11 shows only the procedure object for `good-enough?`.

After the local procedures were defined, the expression `(sqrt-iter 1.0)` was evaluated, still in environment E1. So the procedure object bound to `sqrt-iter` in E1 was called with 1 as an argument. This created an environment E2 in which `guess`, the parameter of `sqrt-iter`, is bound to 1. `sqrt-iter` in turn called `good-enough?` with the value of `guess` (from E2) as the argument for `good-enough?`. This set up another

environment, E3, in which `guess` (the parameter of `good-enough?`) is bound to 1. Although `sqrt-iter` and `good-enough?` both have a parameter named `guess`, these are two distinct local variables located in different frames. Also, E2 and E3 both have E1 as their enclosing environment, because the `sqrt-iter` and `good-enough?` procedures both have E1 as their environment part. One consequence of this is that the symbol `x` that appears in the body of `good-enough?` will reference the binding of `x` that appears in E1, namely the value of `x` with which the original `sqrt` procedure was called.

The environment model thus explains the two key properties that make local procedure definitions a useful technique for modularizing programs:

- The names of the local procedures do not interfere with names external to the enclosing procedure, because the local procedure names will be bound in the frame that the procedure creates when it is run, rather than being bound in the global environment.

- The local procedures can access the arguments of the enclosing procedure, simply by using parameter names as free variables. This is because the body of the local procedure is evaluated in an environment that is subordinate to the evaluation environment for the enclosing procedure.

Exercise 3.11: In Section 3.2.3 we saw how the environment model described the behavior of procedures with local state. Now we have seen how internal definitions work. A typical message-passing procedure contains both of these aspects. Consider the bank account procedure of Section 3.1.1:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else
           (error "Unknown request: MAKE-ACCOUNT"
                  m))))
  dispatch)
```

Show the environment structure generated by the sequence of interactions

```
(define acc (make-account 50))
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

Where is the local state for `acc` kept? Suppose we define another account

```
(define acc2 (make-account 100))
```

How are the local states for the two accounts kept distinct? Which parts of the environment structure are shared between `acc` and `acc2`?

3.3 Modeling with Mutable Data

Chapter 2 dealt with compound data as a means for constructing computational objects that have several parts, in order to model real-world objects that have several aspects. In that chapter we introduced the discipline of data abstraction, according to which data structures are specified in terms of constructors, which create data objects, and selectors, which access the parts of compound data objects. But we now know that there is another aspect of data that Chapter 2 did not address. The desire to model systems composed of objects that have changing state leads us to the need to modify compound data objects, as well as to construct and select from them. In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called *mutators*, which modify data objects. For instance, modeling a banking system requires us to change account balances. Thus, a data structure for representing bank accounts might admit an operation

`(set-balance! ⟨account⟩ ⟨new-value⟩)`

that changes the balance of the designated account to the designated new value. Data objects for which mutators are defined are known as *mutable data objects*.

Chapter 2 introduced pairs as a general-purpose "glue" for synthesizing compound data. We begin this section by defining basic mutators for pairs, so that pairs can serve as building blocks for constructing mutable data objects. These mutators greatly enhance the representational power of pairs, enabling us to build data structures other than the sequences and trees that we worked with in Section 2.2. We also present some examples of simulations in which complex systems are modeled as collections of objects with local state.

3.3.1 Mutable List Structure

The basic operations on pairs—`cons`, `car`, and `cdr`—can be used to construct list structure and to select parts from list structure, but they are incapable of modifying list structure. The same is true of the list operations we have used so far, such as `append` and `list`, since these can be defined in terms of `cons`, `car`, and `cdr`. To modify list structures we need new operations.

The primitive mutators for pairs are `set-car!` and `set-cdr!`. `set-car!` takes two arguments, the first of which must be a pair. It modifies this pair, replacing the `car` pointer by a pointer to the second argument of `set-car!`.[16]

As an example, suppose that `x` is bound to the list `((a b) c d)` and `y` to the list `(e f)` as illustrated in Figure 3.12. Evaluating the expression `(set-car! x y)` modifies the pair to which `x` is bound, replacing its `car` by the value of `y`. The result of the operation is shown in Figure 3.13. The structure `x` has been modified and would now be printed as `((e f) c d)`. The pairs representing the list `(a b)`, identified by the pointer that was replaced, are now detached from the original structure.[17]

Compare Figure 3.13 with Figure 3.14, which illustrates the result of executing `(define z (cons y (cdr x)))` with `x` and `y` bound to the original lists of Figure 3.12. The variable `z` is now bound to a new pair created by the `cons` operation; the list to which `x` is bound is unchanged.

The `set-cdr!` operation is similar to `set-car!`. The only difference is that the `cdr` pointer of the pair, rather than the `car` pointer, is replaced. The effect of executing `(set-cdr! x y)` on the lists of Figure 3.12 is

[16] `set-car!` and `set-cdr!` return implementation-dependent values. Like `set!`, they should be used only for their effect.

[17] We see from this that mutation operations on lists can create "garbage" that is not part of any accessible structure. We will see in Section 5.3.2 that Lisp memory-management systems include a *garbage collector*, which identifies and recycles the memory space used by unneeded pairs.

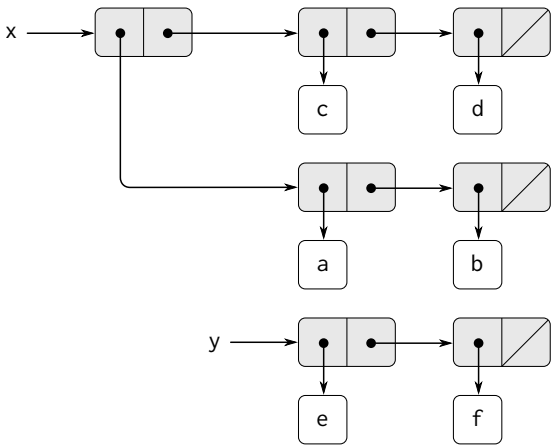

Figure 3.12: Lists `x`: `((a b) c d)` and `y`: `(e f)`.

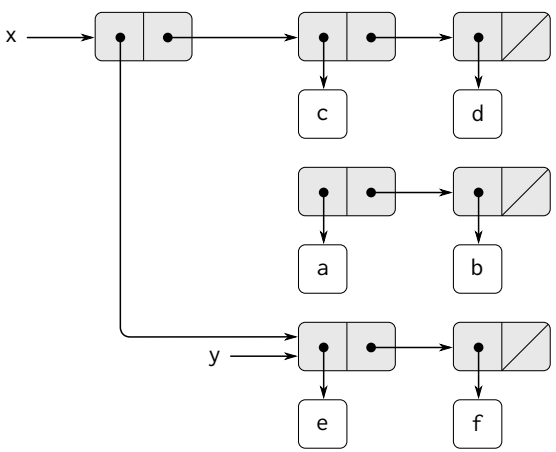

Figure 3.13: Effect of `(set-car! x y)` on the lists in Figure 3.12.

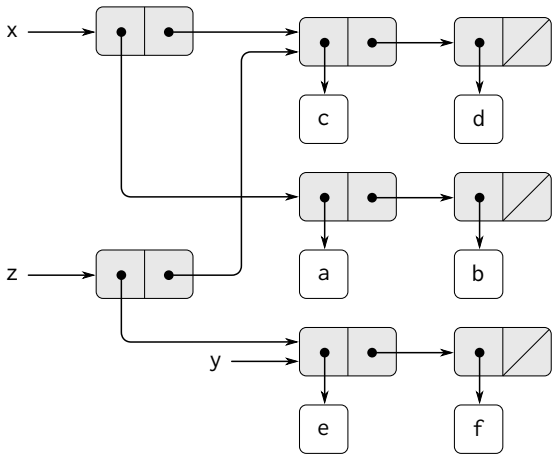

Figure 3.14: Effect of `(define z (cons y (cdr x)))` on the lists in Figure 3.12.

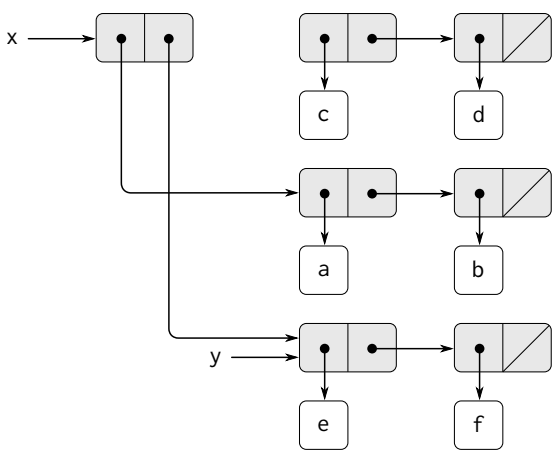

Figure 3.15: Effect of `(set-cdr! x y)` on the lists in Figure 3.12.

shown in Figure 3.15. Here the `cdr` pointer of `x` has been replaced by the pointer to `(e f)`. Also, the list `(c d)`, which used to be the `cdr` of `x`, is now detached from the structure.

`cons` builds new list structure by creating new pairs, while `set-car!` and `set-cdr!` modify existing pairs. Indeed, we could implement `cons` in terms of the two mutators, together with a procedure `get-new-pair`, which returns a new pair that is not part of any existing list structure. We obtain the new pair, set its `car` and `cdr` pointers to the designated objects, and return the new pair as the result of the `cons`.[18]

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

Exercise 3.12: The following procedure for appending lists was introduced in Section 2.2.1:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

`append` forms a new list by successively `cons`ing the elements of `x` onto `y`. The procedure `append!` is similar to `append`, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of `x` so that its `cdr` is now `y`. (It is an error to call `append!` with an empty `x`.)

[18] `get-new-pair` is one of the operations that must be implemented as part of the memory management required by a Lisp implementation. We will discuss this in Section 5.3.1.

```
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

Here `last-pair` is a procedure that returns the last pair in its argument:

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))))
```

Consider the interaction

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
z
(a b c d)
(cdr x)
⟨response⟩
(define w (append! x y))
w
(a b c d)
(cdr x)
⟨response⟩
```

What are the missing ⟨*response*⟩s? Draw box-and-pointer diagrams to explain your answer.

Exercise 3.13: Consider the following `make-cycle` procedure, which uses the `last-pair` procedure defined in Exercise 3.12:

```
(define (make-cycle x)
  (set-cdr! (last-pair x) x)
  x)
```

Draw a box-and-pointer diagram that shows the structure `z` created by

```
(define z (make-cycle (list 'a 'b 'c)))
```

What happens if we try to compute `(last-pair z)`?

Exercise 3.14: The following procedure is quite useful, although obscure:

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
          (set-cdr! x y)
          (loop temp x))))
  (loop x '()))
```

`loop` uses the "temporary" variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`. Explain what `mystery` does in general. Suppose `v` is defined by `(define v (list 'a 'b 'c 'd))`. Draw the box-and-pointer diagram that represents the list to which `v` is bound. Suppose that we now evaluate `(define w (mystery v))`. Draw box-and-pointer diagrams that show the structures `v` and `w` after evaluating this expression. What would be printed as the values of `v` and `w`?

Sharing and identity

We mentioned in Section 3.1.3 the theoretical issues of "sameness" and "change" raised by the introduction of assignment. These issues arise in

practice when individual pairs are *shared* among different data objects. For example, consider the structure formed by

```
(define x (list 'a 'b))
(define z1 (cons x x))
```

As shown in Figure 3.16, `z1` is a pair whose `car` and `cdr` both point to the same pair `x`. This sharing of `x` by the `car` and `cdr` of `z1` is a consequence of the straightforward way in which `cons` is implemented. In general, using `cons` to construct lists will result in an interlinked structure of pairs in which many individual pairs are shared by many different structures.

In contrast to Figure 3.16, Figure 3.17 shows the structure created by

```
(define z2 (cons (list 'a 'b) (list 'a 'b)))
```

In this structure, the pairs in the two `(a b)` lists are distinct, although the actual symbols are shared.[19]

When thought of as a list, `z1` and `z2` both represent "the same" list, `((a b) a b)`. In general, sharing is completely undetectable if we operate on lists using only `cons`, `car`, and `cdr`. However, if we allow mutators on list structure, sharing becomes significant. As an example of the difference that sharing can make, consider the following procedure, which modifies the `car` of the structure to which it is applied:

```
(define (set-to-wow! x) (set-car! (car x) 'wow) x)
```

[19] The two pairs are distinct because each call to `cons` returns a new pair. The symbols are shared; in Scheme there is a unique symbol with any given name. Since Scheme provides no way to mutate a symbol, this sharing is undetectable. Note also that the sharing is what enables us to compare symbols using `eq?`, which simply checks equality of pointers.

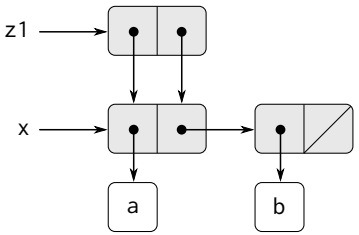

Figure 3.16: The list `z1` formed by `(cons x x)`.

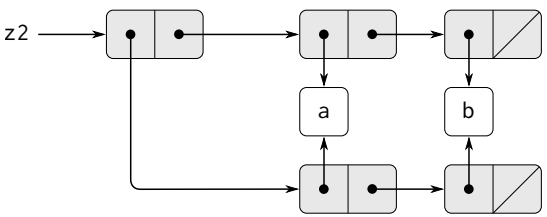

Figure 3.17: The list `z2` formed by `(cons (list 'a 'b) (list 'a 'b))`.

Even though `z1` and `z2` are "the same" structure, applying `set-to-wow!` to them yields different results. With `z1`, altering the `car` also changes the `cdr`, because in `z1` the `car` and the `cdr` are the same pair. With `z2`, the `car` and `cdr` are distinct, so `set-to-wow!` modifies only the `car`:

```
z1
((a b) a b)
(set-to-wow! z1)
((wow b) wow b)
z2
((a b) a b)
```

```
(set-to-wow! z2)
((wow b) a b)
```

One way to detect sharing in list structures is to use the predicate `eq?`, which we introduced in Section 2.3.1 as a way to test whether two symbols are equal. More generally, `(eq? x y)` tests whether `x` and `y` are the same object (that is, whether `x` and `y` are equal as pointers). Thus, with `z1` and `z2` as defined in Figure 3.16 and Figure 3.17, `(eq? (car z1) (cdr z1))` is true and `(eq? (car z2) (cdr z2))` is false.

As will be seen in the following sections, we can exploit sharing to greatly extend the repertoire of data structures that can be represented by pairs. On the other hand, sharing can also be dangerous, since modifications made to structures will also affect other structures that happen to share the modified parts. The mutation operations `set-car!` and `set-cdr!` should be used with care; unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.[20]

> **Exercise 3.15:** Draw box-and-pointer diagrams to explain the effect of `set-to-wow!` on the structures `z1` and `z2` above.

> **Exercise 3.16:** Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. "It's easy,"

[20] The subtleties of dealing with sharing of mutable data objects reflect the underlying issues of "sameness" and "change" that were raised in Section 3.1.3. We mentioned there that admitting change to our language requires that a compound object must have an "identity" that is something different from the pieces from which it is composed. In Lisp, we consider this "identity" to be the quality that is tested by `eq?`, i.e., by equality of pointers. Since in most Lisp implementations a pointer is essentially a memory address, we are "solving the problem" of defining the identity of objects by stipulating that a data object "itself" is the information stored in some particular set of memory locations in the computer. This suffices for simple Lisp programs, but is hardly a general way to resolve the issue of "sameness" in computational models.

he reasons. "The number of pairs in any structure is the number in the `car` plus the number in the `cdr` plus one more to count the current pair." So Ben writes the following procedure:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Show that this procedure is not correct. In particular, draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's procedure would return 3; return 4; return 7; never return at all.

Exercise 3.17: Devise a correct version of the `count-pairs` procedure of Exercise 3.16 that returns the number of distinct pairs in any structure. (Hint: Traverse the structure, maintaining an auxiliary data structure that is used to keep track of which pairs have already been counted.)

Exercise 3.18: Write a procedure that examines a list and determines whether it contains a cycle, that is, whether a program that tried to find the end of the list by taking successive `cdr`s would go into an infinite loop. Exercise 3.13 constructed such lists.

Exercise 3.19: Redo Exercise 3.18 using an algorithm that takes only a constant amount of space. (This requires a very clever idea.)

Mutation is just assignment

When we introduced compound data, we observed in Section 2.1.3 that pairs can be represented purely in terms of procedures:

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation: CONS" m))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
```

The same observation is true for mutable data. We can implement mutable data objects as procedures using assignment and local state. For instance, we can extend the above pair implementation to handle `set-car!` and `set-cdr!` in a manner analogous to the way we implemented bank accounts using `make-account` in Section 3.1.1:

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else
           (error "Undefined operation: CONS" m))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-value)
  ((z 'set-car!) new-value) z)
```

```
(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value) z)
```

Assignment is all that is needed, theoretically, to account for the behavior of mutable data. As soon as we admit `set!` to our language, we raise all the issues, not only of assignment, but of mutable data in general.[21]

> **Exercise 3.20:** Draw environment diagrams to illustrate the evaluation of the sequence of expressions
>
> ```
> (define x (cons 1 2))
> (define z (cons x x))
> (set-car! (cdr z) 17)
> (car x)
> 17
> ```
>
> using the procedural implementation of pairs given above. (Compare Exercise 3.11.)

3.3.2 Representing Queues

The mutators `set-car!` and `set-cdr!` enable us to use pairs to construct data structures that cannot be built with `cons`, `car`, and `cdr` alone. This section shows how to use pairs to represent a data structure called a queue. Section 3.3.3 will show how to represent data structures called tables.

A *queue* is a sequence in which items are inserted at one end (called the *rear* of the queue) and deleted from the other end (the *front*). Figure 3.18 shows an initially empty queue in which the items `a` and `b` are

[21] On the other hand, from the viewpoint of implementation, assignment requires us to modify the environment, which is itself a mutable data structure. Thus, assignment and mutation are equipotent: Each can be implemented in terms of the other.

Operation	Resulting Queue
`(define q (make-queue))`	
`(insert-queue! q 'a)`	`a`
`(insert-queue! q 'b)`	`a b`
`(delete-queue! q)`	`b`
`(insert-queue! q 'c)`	`b c`
`(insert-queue! q 'd)`	`b c d`
`(delete-queue! q)`	`c d`

Figure 3.18: Queue operations.

inserted. Then `a` is removed, `c` and `d` are inserted, and `b` is removed. Because items are always removed in the order in which they are inserted, a queue is sometimes called a *FIFO* (first in, first out) buffer.

In terms of data abstraction, we can regard a queue as defined by the following set of operations:

- a constructor: `(make-queue)` returns an empty queue (a queue containing no items).
- two selectors:

  `(empty-queue?` ⟨*queue*⟩`)` tests if the queue is empty.

  `(front-queue` ⟨*queue*⟩`)` returns the object at the front of the queue, signaling an error if the queue is empty; it does not modify the queue.
- two mutators:

  `(insert-queue!` ⟨*queue*⟩ ⟨*item*⟩`)` inserts the item at the rear of the queue and returns the modified queue as its value.

`(delete-queue!` ⟨*queue*⟩`)` removes the item at the front of the queue and returns the modified queue as its value, signaling an error if the queue is empty before the deletion.

Because a queue is a sequence of items, we could certainly represent it as an ordinary list; the front of the queue would be the `car` of the list, inserting an item in the queue would amount to appending a new element at the end of the list, and deleting an item from the queue would just be taking the `cdr` of the list. However, this representation is inefficient, because in order to insert an item we must scan the list until we reach the end. Since the only method we have for scanning a list is by successive `cdr` operations, this scanning requires $\Theta(n)$ steps for a list of n items. A simple modification to the list representation overcomes this disadvantage by allowing the queue operations to be implemented so that they require $\Theta(1)$ steps; that is, so that the number of steps needed is independent of the length of the queue.

The difficulty with the list representation arises from the need to scan to find the end of the list. The reason we need to scan is that, although the standard way of representing a list as a chain of pairs readily provides us with a pointer to the beginning of the list, it gives us no easily accessible pointer to the end. The modification that avoids the drawback is to represent the queue as a list, together with an additional pointer that indicates the final pair in the list. That way, when we go to insert an item, we can consult the rear pointer and so avoid scanning the list.

A queue is represented, then, as a pair of pointers, `front-ptr` and `rear-ptr`, which indicate, respectively, the first and last pairs in an ordinary list. Since we would like the queue to be an identifiable object, we can use `cons` to combine the two pointers. Thus, the queue itself will be the `cons` of the two pointers. Figure 3.19 illustrates this representation.

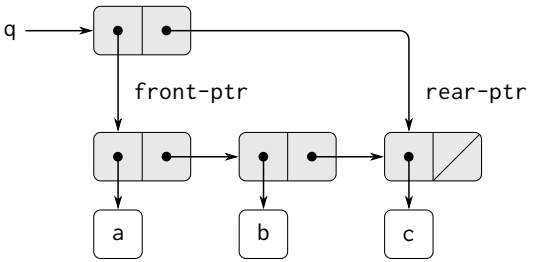

Figure 3.19: Implementation of a queue as a list with front and rear pointers.

To define the queue operations we use the following procedures, which enable us to select and to modify the front and rear pointers of a queue:

```
(define (front-ptr queue) (car queue))
(define (rear-ptr  queue) (cdr queue))
(define (set-front-ptr! queue item)
  (set-car! queue item))
(define (set-rear-ptr!  queue item)
  (set-cdr! queue item))
```

Now we can implement the actual queue operations. We will consider a queue to be empty if its front pointer is the empty list:

```
(define (empty-queue? queue)
  (null? (front-ptr queue)))
```

The `make-queue` constructor returns, as an initially empty queue, a pair whose `car` and `cdr` are both the empty list:

```
(define (make-queue) (cons '() '()))
```

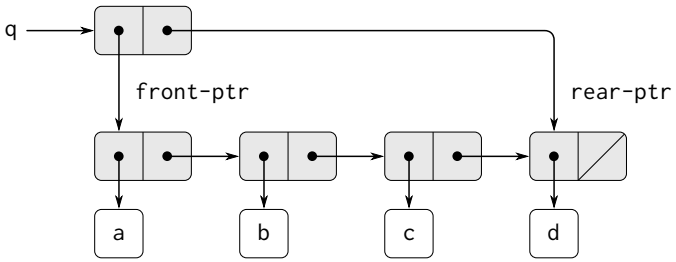

Figure 3.20: Result of using `(insert-queue! q 'd)` on the queue of Figure 3.19.

To select the item at the front of the queue, we return the `car` of the pair indicated by the front pointer:

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```

To insert an item in a queue, we follow the method whose result is indicated in Figure 3.20. We first create a new pair whose `car` is the item to be inserted and whose `cdr` is the empty list. If the queue was initially empty, we set the front and rear pointers of the queue to this new pair. Otherwise, we modify the final pair in the queue to point to the new pair, and also set the rear pointer to the new pair.

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
```

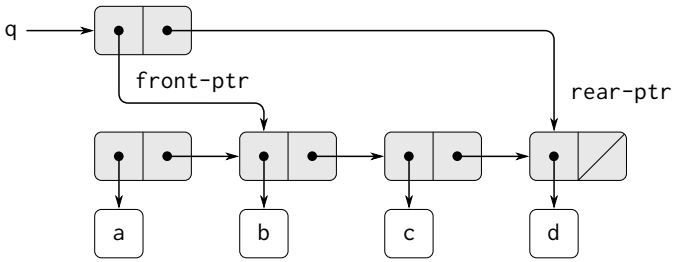

Figure 3.21: Result of using `(delete-queue! q)` on the queue of Figure 3.20.

```
    (else
     (set-cdr! (rear-ptr queue) new-pair)
     (set-rear-ptr! queue new-pair)
     queue))))
```

To delete the item at the front of the queue, we merely modify the front pointer so that it now points at the second item in the queue, which can be found by following the `cdr` pointer of the first item (see Figure 3.21):[22]

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else (set-front-ptr! queue (cdr (front-ptr queue)))
              queue)))
```

[22] If the first item is the final item in the queue, the front pointer will be the empty list after the deletion, which will mark the queue as empty; we needn't worry about updating the rear pointer, which will still point to the deleted item, because `empty-queue?` looks only at the front pointer.

Exercise 3.21: Ben Bitdiddle decides to test the queue implementation described above. He types in the procedures to the Lisp interpreter and proceeds to try them out:

```
(define q1 (make-queue))
(insert-queue! q1 'a)
((a) a)
(insert-queue! q1 'b)
((a b) b)
(delete-queue! q1)
((b) b)
(delete-queue! q1)
(() b)
```

"It's all wrong!" he complains. "The interpreter's response shows that the last item is inserted into the queue twice. And when I delete both items, the second `b` is still there, so the queue isn't empty, even though it's supposed to be." Eva Lu Ator suggests that Ben has misunderstood what is happening. "It's not that the items are going into the queue twice," she explains. "It's just that the standard Lisp printer doesn't know how to make sense of the queue representation. If you want to see the queue printed correctly, you'll have to define your own print procedure for queues." Explain what Eva Lu is talking about. In particular, show why Ben's examples produce the printed results that they do. Define a procedure `print-queue` that takes a queue as input and prints the sequence of items in the queue.

Exercise 3.22: Instead of representing a queue as a pair of pointers, we can build a queue as a procedure with local state. The local state will consist of pointers to the begin-

ning and the end of an ordinary list. Thus, the `make-queue` procedure will have the form

```
(define (make-queue)
  (let ((front-ptr ... )
        (rear-ptr ... ))
    ⟨definitions of internal procedures⟩
    (define (dispatch m) ...)
    dispatch))
```

Complete the definition of `make-queue` and provide implementations of the queue operations using this representation.

Exercise 3.23: A *deque* ("double-ended queue") is a sequence in which items can be inserted and deleted at either the front or the rear. Operations on deques are the constructor `make-deque`, the predicate `empty-deque?`, selectors `front-deque` and `rear-deque`, mutators `front-insert-deque!`, `rear-insert-deque!`, `front-delete-deque!`, and `rear-delete-deque!`. Show how to represent deques using pairs, and give implementations of the operations.[23] All operations should be accomplished in $\Theta(1)$ steps.

3.3.3 Representing Tables

When we studied various ways of representing sets in Chapter 2, we mentioned in Section 2.3.3 the task of maintaining a table of records indexed by identifying keys. In the implementation of data-directed programming in Section 2.4.3, we made extensive use of two-dimensional

[23] Be careful not to make the interpreter try to print a structure that contains cycles. (See Exercise 3.13.)

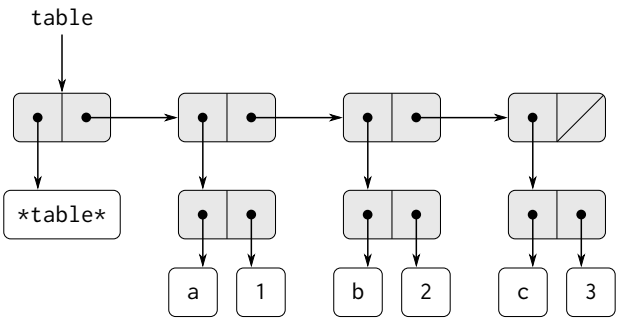

Figure 3.22: A table represented as a headed list.

tables, in which information is stored and retrieved using two keys. Here we see how to build tables as mutable list structures.

We first consider a one-dimensional table, in which each value is stored under a single key. We implement the table as a list of records, each of which is implemented as a pair consisting of a key and the associated value. The records are glued together to form a list by pairs whose `cars` point to successive records. These gluing pairs are called the *backbone* of the table. In order to have a place that we can change when we add a new record to the table, we build the table as a *headed list*. A headed list has a special backbone pair at the beginning, which holds a dummy "record"—in this case the arbitrarily chosen symbol `*table*`. Figure 3.22 shows the box-and-pointer diagram for the table

```
a:  1
b:  2
c:  3
```

To extract information from a table we use the `lookup` procedure, which takes a key as argument and returns the associated value (or false if

there is no value stored under that key). `lookup` is defined in terms of the `assoc` operation, which expects a key and a list of records as arguments. Note that `assoc` never sees the dummy record. `assoc` returns the record that has the given key as its `car`.[24] `lookup` then checks to see that the resulting record returned by `assoc` is not false, and returns the value (the `cdr`) of the record.

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```

To insert a value in a table under a specified key, we first use `assoc` to see if there is already a record in the table with this key. If not, we form a new record by `cons`ing the key with the value, and insert this at the head of the table's list of records, after the dummy record. If there already is a record with this key, we set the `cdr` of this record to the designated new value. The header of the table provides us with a fixed location to modify in order to insert the new record.[25]

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
```

[24] Because `assoc` uses `equal?`, it can recognize keys that are symbols, numbers, or list structure.

[25] Thus, the first backbone pair is the object that represents the table "itself"; that is, a pointer to the table is a pointer to this pair. This same backbone pair always starts the table. If we did not arrange things in this way, `insert!` would have to return a new value for the start of the table when it added a new record.

```
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value)
                        (cdr table)))))
  'ok)
```

To construct a new table, we simply create a list containing the symbol `*table*`:

```
(define (make-table)
  (list '*table*))
```

Two-dimensional tables

In a two-dimensional table, each value is indexed by two keys. We can construct such a table as a one-dimensional table in which each key identifies a subtable. Figure 3.23 shows the box-and-pointer diagram for the table

```
math:    +:  43        letters:    a:  97
         -:  45                    b:  98
         *:  42
```

which has two subtables. (The subtables don't need a special header symbol, since the key that identifies the subtable serves this purpose.)

When we look up an item, we use the first key to identify the correct subtable. Then we use the second key to identify the record within the subtable.

```
(define (lookup key-1 key-2 table)
  (let ((subtable
         (assoc key-1 (cdr table))))
```

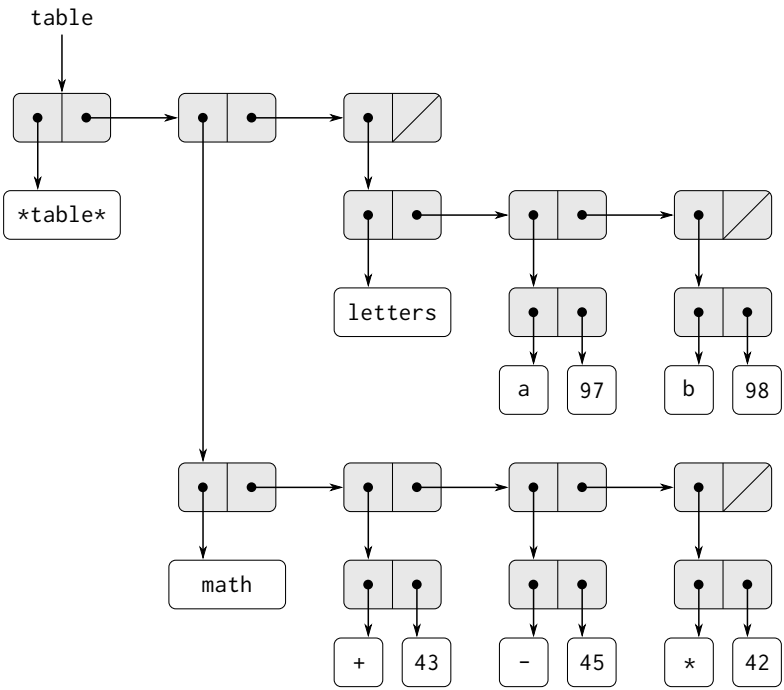

Figure 3.23: A two-dimensional table.

```
(if subtable
    (let ((record
           (assoc key-2 (cdr subtable))))
      (if record
          (cdr record)
          false))
    false)))
```

To insert a new item under a pair of keys, we use `assoc` to see if there is a subtable stored under the first key. If not, we build a new subtable containing the single record (`key-2`, `value`) and insert it into the table under the first key. If a subtable already exists for the first key, we insert the new record into this subtable, using the insertion method for one-dimensional tables described above:

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)
```

Creating local tables

The `lookup` and `insert!` operations defined above take the table as an argument. This enables us to use programs that access more than one table. Another way to deal with multiple tables is to have separate `lookup` and `insert!` procedures for each table. We can do this by representing a table procedurally, as an object that maintains an internal table as part of its local state. When sent an appropriate message, this "table object" supplies the procedure with which to operate on the internal table. Here is a generator for two-dimensional tables represented in this fashion:

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                   (assoc key-2 (cdr subtable))))
              (if record (cdr record) false))
            false)))
    (define (insert! key-1 key-2 value)
      (let ((subtable
             (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record
                   (assoc key-2 (cdr subtable))))
              (if record
                  (set-cdr! record value)
                  (set-cdr! subtable
                            (cons (cons key-2 value)
                                  (cdr subtable)))))
            (set-cdr! local-table
                      (cons (list key-1 (cons key-2 value))
                            (cdr local-table)))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation: TABLE" m))))
    dispatch))
```

Using `make-table`, we could implement the `get` and `put` operations used in Section 2.4.3 for data-directed programming, as follows:

```
(define operation-table (make-table))
(define get (operation-table 'lookup-proc))
(define put (operation-table 'insert-proc!))
```

`get` takes as arguments two keys, and `put` takes as arguments two keys and a value. Both operations access the same local table, which is encapsulated within the object created by the call to `make-table`.

> **Exercise 3.24:** In the table implementations above, the keys are tested for equality using `equal?` (called by `assoc`). This is not always the appropriate test. For instance, we might have a table with numeric keys in which we don't need an exact match to the number we're looking up, but only a number within some tolerance of it. Design a table constructor `make-table` that takes as an argument a `same-key?` procedure that will be used to test "equality" of keys. `make-table` should return a `dispatch` procedure that can be used to access appropriate `lookup` and `insert!` procedures for a local table.

> **Exercise 3.25:** Generalizing one- and two-dimensional tables, show how to implement a table in which values are stored under an arbitrary number of keys and different values may be stored under different numbers of keys. The `lookup` and `insert!` procedures should take as input a list of keys used to access the table.

> **Exercise 3.26:** To search a table as implemented above, one needs to scan through the list of records. This is basically the unordered list representation of Section 2.3.3. For large tables, it may be more efficient to structure the table in a different manner. Describe a table implementation where the

(key, value) records are organized using a binary tree, assuming that keys can be ordered in some way (e.g., numerically or alphabetically). (Compare Exercise 2.66 of Chapter 2.)

Exercise 3.27: *Memoization* (also called *tabulation*) is a technique that enables a procedure to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized procedure maintains a table in which values of previous calls are stored using as keys the arguments that produced the values. When the memoized procedure is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall from Section 1.2.2 the exponential process for computing Fibonacci numbers:

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

The memoized version of the same procedure is

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

where the memoizer is defined as

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result
             (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

Draw an environment diagram to analyze the computation of `(memo-fib 3)`. Explain why `memo-fib` computes the n^{th} Fibonacci number in a number of steps proportional to n. Would the scheme still work if we had simply defined `memo-fib` to be `(memoize fib)`?

3.3.4 A Simulator for Digital Circuits

Designing complex digital systems, such as computers, is an important engineering activity. Digital systems are constructed by interconnecting simple elements. Although the behavior of these individual elements is simple, networks of them can have very complex behavior. Computer simulation of proposed circuit designs is an important tool used by digital systems engineers. In this section we design a system for performing digital logic simulations. This system typifies a kind of program called an *event-driven simulation*, in which actions ("events") trigger further events that happen at a later time, which in turn trigger more events, and so on.

Our computational model of a circuit will be composed of objects that correspond to the elementary components from which the circuit

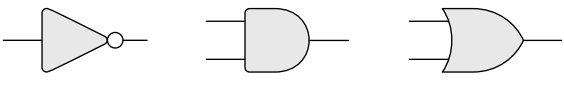

Figure 3.24: Primitive functions in the digital logic simulator.

is constructed. There are *wires*, which carry *digital signals*. A digital signal may at any moment have only one of two possible values, 0 and 1. There are also various types of digital *function boxes*, which connect wires carrying input signals to other output wires. Such boxes produce output signals computed from their input signals. The output signal is delayed by a time that depends on the type of the function box. For example, an *inverter* is a primitive function box that inverts its input. If the input signal to an inverter changes to 0, then one inverter-delay later the inverter will change its output signal to 1. If the input signal to an inverter changes to 1, then one inverter-delay later the inverter will change its output signal to 0. We draw an inverter symbolically as in Figure 3.24. An *and-gate*, also shown in Figure 3.24, is a primitive function box with two inputs and one output. It drives its output signal to a value that is the *logical and* of the inputs. That is, if both of its input signals become 1, then one and-gate-delay time later the and-gate will force its output signal to be 1; otherwise the output will be 0. An *or-gate* is a similar two-input primitive function box that drives its output signal to a value that is the *logical or* of the inputs. That is, the output will become 1 if at least one of the input signals is 1; otherwise the output will become 0.

We can connect primitive functions together to construct more complex functions. To accomplish this we wire the outputs of some function boxes to the inputs of other function boxes. For example, the *half-adder*

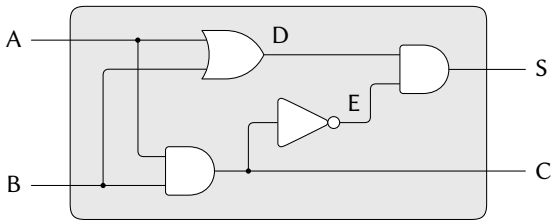

Figure 3.25: A half-adder circuit.

circuit shown in Figure 3.25 consists of an or-gate, two and-gates, and an inverter. It takes two input signals, A and B, and has two output signals, S and C. S will become 1 whenever precisely one of A and B is 1, and C will become 1 whenever A and B are both 1. We can see from the figure that, because of the delays involved, the outputs may be generated at different times. Many of the difficulties in the design of digital circuits arise from this fact.

We will now build a program for modeling the digital logic circuits we wish to study. The program will construct computational objects modeling the wires, which will "hold" the signals. Function boxes will be modeled by procedures that enforce the correct relationships among the signals.

One basic element of our simulation will be a procedure `make-wire`, which constructs wires. For example, we can construct six wires as follows:

```
(define a (make-wire))
(define b (make-wire))
(define c (make-wire))
(define d (make-wire))
(define e (make-wire))
(define s (make-wire))
```

We attach a function box to a set of wires by calling a procedure that constructs that kind of box. The arguments to the constructor procedure are the wires to be attached to the box. For example, given that we can construct and-gates, or-gates, and inverters, we can wire together the half-adder shown in Figure 3.25:

```
(or-gate a b d)
ok
(and-gate a b c)
ok
(inverter c e)
ok
(and-gate d e s)
ok
```

Better yet, we can explicitly name this operation by defining a procedure `half-adder` that constructs this circuit, given the four external wires to be attached to the half-adder:

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

The advantage of making this definition is that we can use `half-adder` itself as a building block in creating more complex circuits. Figure 3.26, for example, shows a *full-adder* composed of two half-adders and an or-gate.[26] We can construct a full-adder as follows:

[26] A full-adder is a basic circuit element used in adding two binary numbers. Here A and B are the bits at corresponding positions in the two numbers to be added, and

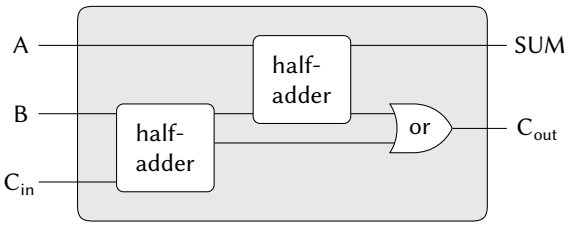

Figure 3.26: A full-adder circuit.

```
(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire)) (c1 (make-wire)) (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```

Having defined `full-adder` as a procedure, we can now use it as a building block for creating still more complex circuits. (For example, see Exercise 3.30.)

In essence, our simulator provides us with the tools to construct a language of circuits. If we adopt the general perspective on languages with which we approached the study of Lisp in Section 1.1, we can say that the primitive function boxes form the primitive elements of the language, that wiring boxes together provides a means of combination, and that specifying wiring patterns as procedures serves as a means of abstraction.

C_{in} is the carry bit from the addition one place to the right. The circuit generates SUM, which is the sum bit in the corresponding position, and C_{out}, which is the carry bit to be propagated to the left.

Primitive function boxes

The primitive function boxes implement the "forces" by which a change in the signal on one wire influences the signals on other wires. To build function boxes, we use the following operations on wires:

- `(get-signal` ⟨*wire*⟩`)`

  returns the current value of the signal on the wire.

- `(set-signal!` ⟨*wire*⟩ ⟨*new value*⟩`)`

  changes the value of the signal on the wire to the new value.

- `(add-action!` ⟨*wire*⟩ ⟨*procedure of no arguments*⟩`)`

  asserts that the designated procedure should be run whenever the signal on the wire changes value. Such procedures are the vehicles by which changes in the signal value on the wire are communicated to other wires.

In addition, we will make use of a procedure `after-delay` that takes a time delay and a procedure to be run and executes the given procedure after the given delay.

Using these procedures, we can define the primitive digital logic functions. To connect an input to an output through an inverter, we use `add-action!` to associate with the input wire a procedure that will be run whenever the signal on the input wire changes value. The procedure computes the `logical-not` of the input signal, and then, after one `inverter-delay`, sets the output signal to be this new value:

```
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
```

```
      (after-delay inverter-delay
                   (lambda () (set-signal! output new-value)))))
  (add-action! input invert-input) 'ok)
(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))
```

An and-gate is a little more complex. The action procedure must be run if either of the inputs to the gate changes. It computes the `logical-and` (using a procedure analogous to `logical-not`) of the values of the signals on the input wires and sets up a change to the new value to occur on the output wire after one `and-gate-delay`.

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay
       and-gate-delay
       (lambda () (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)
```

Exercise 3.28: Define an or-gate as a primitive function box. Your `or-gate` constructor should be similar to `and-gate`.

Exercise 3.29: Another way to construct an or-gate is as a compound digital logic device, built from and-gates and inverters. Define a procedure `or-gate` that accomplishes

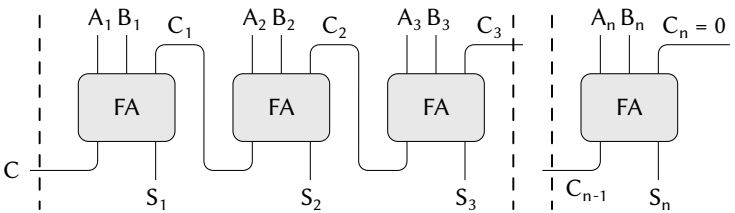

Figure 3.27: A ripple-carry adder for n-bit numbers.

this. What is the delay time of the or-gate in terms of `and-gate-delay` and `inverter-delay`?

Exercise 3.30: Figure 3.27 shows a *ripple-carry adder* formed by stringing together n full-adders. This is the simplest form of parallel adder for adding two n-bit binary numbers. The inputs A_1, A_2, A_3, ..., A_n and B_1, B_2, B_3, ..., B_n are the two binary numbers to be added (each A_k and B_k is a 0 or a 1). The circuit generates S_1, S_2, S_3, ..., S_n, the n bits of the sum, and C, the carry from the addition. Write a procedure `ripple-carry-adder` that generates this circuit. The procedure should take as arguments three lists of n wires each—the A_k, the B_k, and the S_k—and also another wire C. The major drawback of the ripple-carry adder is the need to wait for the carry signals to propagate. What is the delay needed to obtain the complete output from an n-bit ripple-carry adder, expressed in terms of the delays for and-gates, or-gates, and inverters?

Representing wires

A wire in our simulation will be a computational object with two local state variables: a `signal-value` (initially taken to be 0) and a collection of `action-procedures` to be run when the signal changes value. We implement the wire, using message-passing style, as a collection of local procedures together with a `dispatch` procedure that selects the appropriate local operation, just as we did with the simple bank-account object in Section 3.1.1:

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '()))
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure! proc)
      (set! action-procedures
            (cons proc action-procedures))
      (proc))
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure!)
            (else (error "Unknown operation: WIRE" m))))
    dispatch))
```

The local procedure `set-my-signal!` tests whether the new signal value changes the signal on the wire. If so, it runs each of the action procedures, using the following procedure `call-each`, which calls each of the items in a list of no-argument procedures:

```
(define (call-each procedures)
```

```
  (if (null? procedures)
      'done
      (begin ((car procedures))
             (call-each (cdr procedures)))))
```

The local procedure `accept-action-procedure!` adds the given procedure to the list of procedures to be run, and then runs the new procedure once. (See Exercise 3.31.)

With the local `dispatch` procedure set up as specified, we can provide the following procedures to access the local operations on wires:[27]

```
(define (get-signal wire) (wire 'get-signal))
(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))
(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

Wires, which have time-varying signals and may be incrementally attached to devices, are typical of mutable objects. We have modeled them as procedures with local state variables that are modified by assignment. When a new wire is created, a new set of state variables is allocated (by the `let` expression in `make-wire`) and a new `dispatch` procedure is constructed and returned, capturing the environment with the new state variables.

[27] These procedures are simply syntactic sugar that allow us to use ordinary procedural syntax to access the local procedures of objects. It is striking that we can interchange the role of "procedures" and "data" in such a simple way. For example, if we write `(wire 'get-signal)` we think of `wire` as a procedure that is called with the message `get-signal` as input. Alternatively, writing `(get-signal wire)` encourages us to think of `wire` as a data object that is the input to a procedure `get-signal`. The truth of the matter is that, in a language in which we can deal with procedures as objects, there is no fundamental difference between "procedures" and "data," and we can choose our syntactic sugar to allow us to program in whatever style we choose.

The wires are shared among the various devices that have been connected to them. Thus, a change made by an interaction with one device will affect all the other devices attached to the wire. The wire communicates the change to its neighbors by calling the action procedures provided to it when the connections were established.

The agenda

The only thing needed to complete the simulator is `after-delay`. The idea here is that we maintain a data structure, called an *agenda*, that contains a schedule of things to do. The following operations are defined for agendas:

- `(make-agenda)` returns a new empty agenda.
- `(empty-agenda?` ⟨*agenda*⟩`)` is true if the specified agenda is empty.
- `(first-agenda-item` ⟨*agenda*⟩`)` returns the first item on the agenda.
- `(remove-first-agenda-item!` ⟨*agenda*⟩`)` modifies the agenda by removing the first item.
- `(add-to-agenda!` ⟨*time*⟩ ⟨*action*⟩ ⟨*agenda*⟩`)` modifies the agenda by adding the given action procedure to be run at the specified time.
- `(current-time` ⟨*agenda*⟩`)` returns the current simulation time.

The particular agenda that we use is denoted by `the-agenda`. The procedure `after-delay` adds new elements to `the-agenda`:

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda))
```

The simulation is driven by the procedure `propagate`, which operates on `the-agenda`, executing each procedure on the agenda in sequence. In general, as the simulation runs, new items will be added to the agenda, and `propagate` will continue the simulation as long as there are items on the agenda:

```
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

A sample simulation

The following procedure, which places a "probe" on a wire, shows the simulator in action. The probe tells the wire that, whenever its signal changes value, it should print the new signal value, together with the current time and a name that identifies the wire:

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name) (display " ")
                 (display (current-time the-agenda))
                 (display "  New-value = ")
                 (display (get-signal wire)))))
```

We begin by initializing the agenda and specifying delays for the primitive function boxes:

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

Now we define four wires, placing probes on two of them:

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

(probe 'sum sum)
sum 0 New-value = 0

(probe 'carry carry)
carry 0 New-value = 0
```

Next we connect the wires in a half-adder circuit (as in Figure 3.25), set the signal on `input-1` to 1, and run the simulation:

```
(half-adder input-1 input-2 sum carry)
ok

(set-signal! input-1 1)
done

(propagate)
sum 8 New-value = 1
done
```

The `sum` signal changes to 1 at time 8. We are now eight time units from the beginning of the simulation. At this point, we can set the signal on `input-2` to 1 and allow the values to propagate:

```
(set-signal! input-2 1)
done

(propagate)
carry 11 New-value = 1
sum 16 New-value = 0
done
```

The carry changes to 1 at time 11 and the sum changes to 0 at time 16.

> **Exercise 3.31:** The internal procedure accept-action-procedure! defined in make-wire specifies that when a new action procedure is added to a wire, the procedure is immediately run. Explain why this initialization is necessary. In particular, trace through the half-adder example in the paragraphs above and say how the system's response would differ if we had defined accept-action-procedure! as
>
> ```
> (define (accept-action-procedure! proc)
>   (set! action-procedures
>         (cons proc action-procedures)))
> ```

Implementing the agenda

Finally, we give details of the agenda data structure, which holds the procedures that are scheduled for future execution.

The agenda is made up of *time segments*. Each time segment is a pair consisting of a number (the time) and a queue (see Exercise 3.32) that holds the procedures that are scheduled to be run during that time segment.

```
(define (make-time-segment time queue)
  (cons time queue))
```

```
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

We will operate on the time-segment queues using the queue operations described in Section 3.3.2.

The agenda itself is a one-dimensional table of time segments. It differs from the tables described in Section 3.3.3 in that the segments will be sorted in order of increasing time. In addition, we store the *current time* (i.e., the time of the last action that was processed) at the head of the agenda. A newly constructed agenda has no time segments and has a current time of 0:[28]

```
(define (make-agenda) (list 0))
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time)
  (set-car! agenda time))
(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))
```

An agenda is empty if it has no time segments:

```
(define (empty-agenda? agenda)
  (null? (segments agenda)))
```

To add an action to an agenda, we first check if the agenda is empty. If so, we create a time segment for the action and install this in the agenda. Otherwise, we scan the agenda, examining the time of each segment. If we find a segment for our appointed time, we add the action to the

[28] The agenda is a headed list, like the tables in Section 3.3.3, but since the list is headed by the time, we do not need an additional dummy header (such as the `*table*` symbol used with tables).

associated queue. If we reach a time later than the one to which we are appointed, we insert a new time segment into the agenda just before it. If we reach the end of the agenda, we must create a new time segment at the end.

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                       action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments!
         agenda
         (cons (make-new-time-segment time action)
               segments))
        (add-to-segments! segments))))
```

The procedure that removes the first item from the agenda deletes the item at the front of the queue in the first time segment. If this deletion

makes the time segment empty, we remove it from the list of segments:[29]

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

The first agenda item is found at the head of the queue in the first time segment. Whenever we extract an item, we also update the current time:[30]

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty: FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda
                           (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

> **Exercise 3.32:** The procedures to be run during each time segment of the agenda are kept in a queue. Thus, the procedures for each segment are called in the order in which they were added to the agenda (first in, first out). Explain why this order must be used. In particular, trace the behavior of an and-gate whose inputs change from 0, 1 to 1, 0

[29] Observe that the `if` expression in this procedure has no ⟨*alternative*⟩ expression. Such a "one-armed `if` statement" is used to decide whether to do something, rather than to select between two expressions. An `if` expression returns an unspecified value if the predicate is false and there is no ⟨*alternative*⟩.

[30] In this way, the current time will always be the time of the action most recently processed. Storing this time at the head of the agenda ensures that it will still be available even if the associated time segment has been deleted.

in the same segment and say how the behavior would differ if we stored a segment's procedures in an ordinary list, adding and removing procedures only at the front (last in, first out).

3.3.5 Propagation of Constraints

Computer programs are traditionally organized as one-directional computations, which perform operations on prespecified arguments to produce desired outputs. On the other hand, we often model systems in terms of relations among quantities. For example, a mathematical model of a mechanical structure might include the information that the deflection d of a metal rod is related to the force F on the rod, the length L of the rod, the cross-sectional area A, and the elastic modulus E via the equation

$$dAE = FL.$$

Such an equation is not one-directional. Given any four of the quantities, we can use it to compute the fifth. Yet translating the equation into a traditional computer language would force us to choose one of the quantities to be computed in terms of the other four. Thus, a procedure for computing the area A could not be used to compute the deflection d, even though the computations of A and d arise from the same equation.[31]

[31] Constraint propagation first appeared in the incredibly forward-looking SKETCHPAD system of Ivan Sutherland (1963). A beautiful constraint-propagation system based on the Smalltalk language was developed by Alan Borning (1977) at Xerox Palo Alto Research Center. Sussman, Stallman, and Steele applied constraint propagation to electrical circuit analysis (Sussman and Stallman 1975; Sussman and Steele 1980). TK!Solver (Konopasek and Jayaraman 1984) is an extensive modeling environment based on constraints.

In this section, we sketch the design of a language that enables us to work in terms of relations themselves. The primitive elements of the language are *primitive constraints*, which state that certain relations hold between quantities. For example, `(adder a b c)` specifies that the quantities a, b, and c must be related by the equation $a + b = c$, `(multiplier x y z)` expresses the constraint $xy = z$, and `(constant 3.14 x)` says that the value of x must be 3.14.

Our language provides a means of combining primitive constraints in order to express more complex relations. We combine constraints by constructing *constraint networks*, in which constraints are joined by *connectors*. A connector is an object that "holds" a value that may participate in one or more constraints. For example, we know that the relationship between Fahrenheit and Celsius temperatures is

$$9C = 5(F - 32).$$

Such a constraint can be thought of as a network consisting of primitive adder, multiplier, and constant constraints (Figure 3.28). In the figure, we see on the left a multiplier box with three terminals, labeled $m1$, $m2$, and p. These connect the multiplier to the rest of the network as follows: The $m1$ terminal is linked to a connector C, which will hold the Celsius temperature. The $m2$ terminal is linked to a connector w, which is also linked to a constant box that holds 9. The p terminal, which the multiplier box constrains to be the product of $m1$ and $m2$, is linked to the p terminal of another multiplier box, whose $m2$ is connected to a constant 5 and whose $m1$ is connected to one of the terms in a sum.

Computation by such a network proceeds as follows: When a connector is given a value (by the user or by a constraint box to which it is linked), it awakens all of its associated constraints (except for the constraint that just awakened it) to inform them that it has a value.

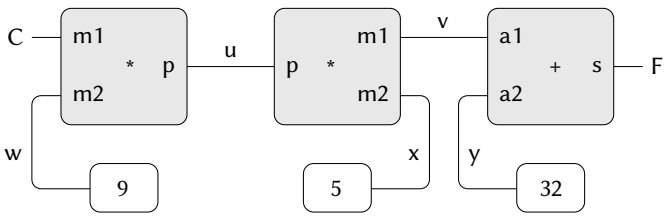

Figure 3.28: The relation $9C = 5(F - 32)$ expressed as a constraint network.

Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector, which then awakens all of its associated constraints, and so on. For instance, in conversion between Celsius and Fahrenheit, w, x, and y are immediately set by the constant boxes to 9, 5, and 32, respectively. The connectors awaken the multipliers and the adder, which determine that there is not enough information to proceed. If the user (or some other part of the network) sets C to a value (say 25), the leftmost multiplier will be awakened, and it will set u to $25 \cdot 9 = 225$. Then u awakens the second multiplier, which sets v to 45, and v awakens the adder, which sets f to 77.

Using the constraint system

To use the constraint system to carry out the temperature computation outlined above, we first create two connectors, `C` and `F`, by calling the constructor `make-connector`, and link `C` and `F` in an appropriate network:

```
(define C (make-connector))
(define F (make-connector))
```

```
(celsius-fahrenheit-converter C F)
ok
```

The procedure that creates the network is defined as follows:

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

This procedure creates the internal connectors `u`, `v`, `w`, `x`, and `y`, and links them as shown in Figure 3.28 using the primitive constraint constructors `adder`, `multiplier`, and `constant`. Just as with the digital-circuit simulator of Section 3.3.4, expressing these combinations of primitive elements in terms of procedures automatically provides our language with a means of abstraction for compound objects.

To watch the network in action, we can place probes on the connectors `C` and `F`, using a `probe` procedure similar to the one we used to monitor wires in Section 3.3.4. Placing a probe on a connector will cause a message to be printed whenever the connector is given a value:

```
(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)
```

Next we set the value of `C` to 25. (The third argument to `set-value!` tells `C` that this directive comes from the `user`.)

```
(set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
```

The probe on `C` awakens and reports the value. `C` also propagates its value through the network as described above. This sets `F` to 77, which is reported by the probe on `F`.

Now we can try to set `F` to a new value, say 212:

```
(set-value! F 212 'user)
Error! Contradiction (77 212)
```

The connector complains that it has sensed a contradiction: Its value is 77, and someone is trying to set it to 212. If we really want to reuse the network with new values, we can tell `C` to forget its old value:

```
(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
```

`C` finds that the `user`, who set its value originally, is now retracting that value, so `C` agrees to lose its value, as shown by the probe, and informs the rest of the network of this fact. This information eventually propagates to `F`, which now finds that it has no reason for continuing to believe that its own value is 77. Thus, `F` also gives up its value, as shown by the probe.

Now that `F` has no value, we are free to set it to 212:

```
(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
done
```

This new value, when propagated through the network, forces `C` to have a value of 100, and this is registered by the probe on `C`. Notice that the very same network is being used to compute `C` given `F` and to compute `F` given `C`. This nondirectionality of computation is the distinguishing feature of constraint-based systems.

Implementing the constraint system

The constraint system is implemented via procedural objects with local state, in a manner very similar to the digital-circuit simulator of Section 3.3.4. Although the primitive objects of the constraint system are somewhat more complex, the overall system is simpler, since there is no concern about agendas and logic delays.

The basic operations on connectors are the following:

- `(has-value?` ⟨*connector*⟩`)` tells whether the connector has a value.
- `(get-value` ⟨*connector*⟩`)` returns the connector's current value.
- `(set-value!` ⟨*connector*⟩ ⟨*new-value*⟩ ⟨*informant*⟩`)` indicates that the informant is requesting the connector to set its value to the new value.
- `(forget-value!` ⟨*connector*⟩ ⟨*retractor*⟩`)` tells the connector that the retractor is requesting it to forget its value.
- `(connect` ⟨*connector*⟩ ⟨*new-constraint*⟩`)` tells the connector to participate in the new constraint.

The connectors communicate with the constraints by means of the procedures `inform-about-value`, which tells the given constraint that the

connector has a value, and `inform-about-no-value`, which tells the constraint that the connector has lost its value.

`adder` constructs an adder constraint among summand connectors `a1` and `a2` and a `sum` connector. An adder is implemented as a procedure with local state (the procedure `me` below):

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
                       (- (get-value sum) (get-value a1))
                       me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1
                       (- (get-value sum) (get-value a2))
                       me))))
  (define (process-forget-value)
    (forget-value! sum me)
    (forget-value! a1 me)
    (forget-value! a2 me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)  (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request: ADDER" request))))
  (connect a1 me)
  (connect a2 me)
  (connect sum me)
  me)
```

`adder` connects the new adder to the designated connectors and returns it as its value. The procedure `me`, which represents the adder, acts as a dispatch to the local procedures. The following "syntax interfaces" (see Footnote 27 in Section 3.3.4) are used in conjunction with the dispatch:

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

The adder's local procedure `process-new-value` is called when the adder is informed that one of its connectors has a value. The adder first checks to see if both `a1` and `a2` have values. If so, it tells `sum` to set its value to the sum of the two addends. The `informant` argument to `set-value!` is `me`, which is the adder object itself. If `a1` and `a2` do not both have values, then the adder checks to see if perhaps `a1` and `sum` have values. If so, it sets `a2` to the difference of these two. Finally, if `a2` and `sum` have values, this gives the adder enough information to set `a1`. If the adder is told that one of its connectors has lost a value, it requests that all of its connectors now lose their values. (Only those values that were set by this adder are actually lost.) Then it runs `process-new-value`. The reason for this last step is that one or more connectors may still have a value (that is, a connector may have had a value that was not originally set by the adder), and these values may need to be propagated back through the adder.

A multiplier is very similar to an adder. It will set its `product` to 0 if either of the factors is 0, even if the other factor is not known.

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
```

```
          (set-value! product 0 me))
         ((and (has-value? m1) (has-value? m2))
          (set-value! product
                      (* (get-value m1) (get-value m2))
                      me))
         ((and (has-value? product) (has-value? m1))
          (set-value! m2
                      (/ (get-value product)
                         (get-value m1))
                      me))
         ((and (has-value? product) (has-value? m2))
          (set-value! m1
                      (/ (get-value product)
                         (get-value m2))
                      me))))
  (define (process-forget-value)
    (forget-value! product me)
    (forget-value! m1 me)
    (forget-value! m2 me)
    (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)  (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request: MULTIPLIER"
                       request))))
  (connect m1 me)
  (connect m2 me)
  (connect product me)
  me)
```

A `constant` constructor simply sets the value of the designated connector. Any `I-have-a-value` or `I-lost-my-value` message sent to the constant box will produce an error.

```
(define (constant value connector)
  (define (me request)
    (error "Unknown request: CONSTANT" request))
  (connect connector me)
  (set-value! connector value me)
  me)
```

Finally, a probe prints a message about the setting or unsetting of the designated connector:

```
(define (probe name connector)
  (define (print-probe value)
    (newline) (display "Probe: ") (display name)
    (display " = ") (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value) (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)  (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request: PROBE" request))))
  (connect connector me)
  me)
```

Representing connectors

A connector is represented as a procedural object with local state variables `value`, the current value of the connector; `informant`, the object that set the connector's value; and `constraints`, a list of the constraints in which the connector participates.

```
(define (make-connector)
  (let ((value false) (informant false) (constraints '()))
    (define (set-my-value newval setter)
```

```
  (cond ((not (has-value? me))
         (set! value newval)
         (set! informant setter)
         (for-each-except setter
                          inform-about-value
                          constraints))
        ((not (= value newval))
         (error "Contradiction" (list value newval)))
        (else 'ignored)))
(define (forget-my-value retractor)
  (if (eq? retractor informant)
      (begin (set! informant false)
             (for-each-except retractor
                              inform-about-no-value
                              constraints))
      'ignored))
(define (connect new-constraint)
  (if (not (memq new-constraint constraints))
      (set! constraints
            (cons new-constraint constraints)))
  (if (has-value? me)
      (inform-about-value new-constraint))
  'done)
(define (me request)
  (cond ((eq? request 'has-value?)
         (if informant true false))
        ((eq? request 'value) value)
        ((eq? request 'set-value!) set-my-value)
        ((eq? request 'forget) forget-my-value)
        ((eq? request 'connect) connect)
        (else (error "Unknown operation: CONNECTOR"
                     request))))
me))
```

The connector's local procedure `set-my-value` is called when there is a request to set the connector's value. If the connector does not currently have a value, it will set its value and remember as `informant` the constraint that requested the value to be set.[32] Then the connector will notify all of its participating constraints except the constraint that requested the value to be set. This is accomplished using the following iterator, which applies a designated procedure to all items in a list except a given one:

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items)))))
  (loop list))
```

If a connector is asked to forget its value, it runs the local procedure `forget-my-value`, which first checks to make sure that the request is coming from the same object that set the value originally. If so, the connector informs its associated constraints about the loss of the value.

The local procedure `connect` adds the designated new constraint to the list of constraints if it is not already in that list. Then, if the connector has a value, it informs the new constraint of this fact.

The connector's procedure `me` serves as a dispatch to the other internal procedures and also represents the connector as an object. The following procedures provide a syntax interface for the dispatch:

```
(define (has-value? connector)
  (connector 'has-value?))
```

[32] The `setter` might not be a constraint. In our temperature example, we used `user` as the `setter`.

```
(define (get-value connector)
  (connector 'value))
(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))
(define (forget-value! connector retractor)
  ((connector 'forget) retractor))
(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

Exercise 3.33: Using primitive multiplier, adder, and constant constraints, define a procedure `averager` that takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that the value of `c` is the average of the values of `a` and `b`.

Exercise 3.34: Louis Reasoner wants to build a squarer, a constraint device with two terminals such that the value of connector `b` on the second terminal will always be the square of the value `a` on the first terminal. He proposes the following simple device made from a multiplier:

```
(define (squarer a b)
  (multiplier a a b))
```

There is a serious flaw in this idea. Explain.

Exercise 3.35: Ben Bitdiddle tells Louis that one way to avoid the trouble in Exercise 3.34 is to define a squarer as a new primitive constraint. Fill in the missing portions in Ben's outline for a procedure to implement such a constraint:

```
(define (squarer a b)
  (define (process-new-value)
    (if (has-value? b)
        (if (< (get-value b) 0)
            (error "square less than 0: SQUARER"
                   (get-value b))
            ⟨alternative1⟩)
        ⟨alternative2⟩))
  (define (process-forget-value) ⟨body1⟩)
  (define (me request) ⟨body2⟩)
  ⟨rest of definition⟩
  me)
```

Exercise 3.36: Suppose we evaluate the following sequence of expressions in the global environment:

```
(define a (make-connector))
(define b (make-connector))
(set-value! a 10 'user)
```

At some time during evaluation of the `set-value!`, the following expression from the connector's local procedure is evaluated:

```
(for-each-except
  setter inform-about-value constraints)
```

Draw an environment diagram showing the environment in which the above expression is evaluated.

Exercise 3.37: The `celsius-fahrenheit-converter` procedure is cumbersome when compared with a more expression-oriented style of definition, such as

```
(define (celsius-fahrenheit-converter x)
  (c+ (c* (c/ (cv 9) (cv 5))
          x)
      (cv 32)))
(define C (make-connector))
(define F (celsius-fahrenheit-converter C))
```

Here `c+`, `c*`, etc. are the "constraint" versions of the arithmetic operations. For example, `c+` takes two connectors as arguments and returns a connector that is related to these by an adder constraint:

```
(define (c+ x y)
  (let ((z (make-connector)))
    (adder x y z)
    z))
```

Define analogous procedures `c-`, `c*`, `c/`, and `cv` (constant value) that enable us to define compound constraints as in the converter example above.[33]

[33] The expression-oriented format is convenient because it avoids the need to name the intermediate expressions in a computation. Our original formulation of the constraint language is cumbersome in the same way that many languages are cumbersome when dealing with operations on compound data. For example, if we wanted to compute the product $(a+b)\cdot(c+d)$, where the variables represent vectors, we could work in "imperative style," using procedures that set the values of designated vector arguments but do not themselves return vectors as values:

```
(v-sum a b temp1)
(v-sum c d temp2)
(v-prod temp1 temp2 answer)
```

Alternatively, we could deal with expressions, using procedures that return vectors as values, and thus avoid explicitly mentioning `temp1` and `temp2`:

```
(define answer (v-prod (v-sum a b) (v-sum c d)))
```

3.4 Concurrency: Time Is of the Essence

We've seen the power of computational objects with local state as tools for modeling. Yet, as Section 3.1.3 warned, this power extracts a price: the loss of referential transparency, giving rise to a thicket of questions about sameness and change, and the need to abandon the substitution model of evaluation in favor of the more intricate environment model.

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit *time* into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value. In contrast, recall the example of modeling withdrawals from a bank account and returning the resulting balance, introduced at the beginning of Section 3.1.1:

```
(withdraw 25)
75
(withdraw 25)
50
```

Since Lisp allows us to return compound objects as values of procedures, we can transform our imperative-style constraint language into an expression-oriented style as shown in this exercise. In languages that are impoverished in handling compound objects, such as Algol, Basic, and Pascal (unless one explicitly uses Pascal pointer variables), one is usually stuck with the imperative style when manipulating compound objects. Given the advantage of the expression-oriented format, one might ask if there is any reason to have implemented the system in imperative style, as we did in this section. One reason is that the non-expression-oriented constraint language provides a handle on constraint objects (e.g., the value of the `adder` procedure) as well as on connector objects. This is useful if we wish to extend the system with new operations that communicate with constraints directly rather than only indirectly via operations on connectors. Although it is easy to implement the expression-oriented style in terms of the imperative implementation, it is very difficult to do the converse.

Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in this case, assignments to the variable `balance`) delineates *moments in time* when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

We can go further in structuring computational models to match our perception of the physical world. Objects in the world do not change one at a time in sequence. Rather we perceive them as acting *concurrently*—all at once. So it is often natural to model systems as collections of computational processes that execute concurrently. Just as we can make our programs modular by organizing models in terms of objects with separate local state, it is often appropriate to divide computational models into parts that evolve separately and concurrently. Even if the programs are to be executed on a sequential computer, the practice of writing programs as if they were to be executed concurrently forces the programmer to avoid inessential timing constraints and thus makes programs more modular.

In addition to making programs more modular, concurrent computation can provide a speed advantage over sequential computation. Sequential computers execute only one operation at a time, so the amount of time it takes to perform a task is proportional to the total number of operations performed.[34] However, if it is possible to decompose a

[34]Most real processors actually execute a few operations at a time, following a strategy called *pipelining*. Although this technique greatly improves the effective utilization of the hardware, it is used only to speed up the execution of a sequential instruction stream, while retaining the behavior of the sequential program.

problem into pieces that are relatively independent and need to communicate only rarely, it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

Unfortunately, the complexities introduced by assignment become even more problematic in the presence of concurrency. The fact of concurrent execution, either because the world operates in parallel or because our computers do, entails additional complexity in our understanding of time.

3.4.1 The Nature of Time in Concurrent Systems

On the surface, time seems straightforward. It is an ordering imposed on events.[35] For any events A and B, either A occurs before B, A and B are simultaneous, or A occurs after B. For instance, returning to the bank account example, suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either \$100 → \$90 → \$65 or \$100 → \$75 → \$65 . In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to a variable `balance`.

In complex situations, however, such a view can be problematic. Suppose that Peter and Paul, and other people besides, are accessing the same bank account through a network of banking machines distributed all over the world. The actual sequence of balances in the account will depend critically on the detailed timing of the accesses and the details of the communication among the machines.

[35] To quote some graffiti seen on a Cambridge building wall: "Time is a device that was invented to keep everything from happening at once."

This indeterminacy in the order of events can pose serious problems in the design of concurrent systems. For instance, suppose that the withdrawals made by Peter and Paul are implemented as two separate processes sharing a common variable `balance`, each process specified by the procedure given in Section 3.1.1:

```
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance (- balance amount))
        balance)
      "Insufficient funds"))
```

If the two processes operate independently, then Peter might test the balance and attempt to withdraw a legitimate amount. However, Paul might withdraw some funds in between the time that Peter checks the balance and the time Peter completes the withdrawal, thus invalidating Peter's test.

Things can be worse still. Consider the expression

```
(set! balance (- balance amount))
```

executed as part of each withdrawal process. This consists of three steps: (1) accessing the value of the `balance` variable; (2) computing the new balance; (3) setting `balance` to this new value. If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access `balance` and set it to the new value.

The timing diagram in Figure 3.29 depicts an order of events where `balance` starts at 100, Peter withdraws 10, Paul withdraws 25, and yet the final value of `balance` is 75. As shown in the diagram, the reason for this anomaly is that Paul's assignment of 75 to `balance` is made under the assumption that the value of `balance` to be decremented is 100. That

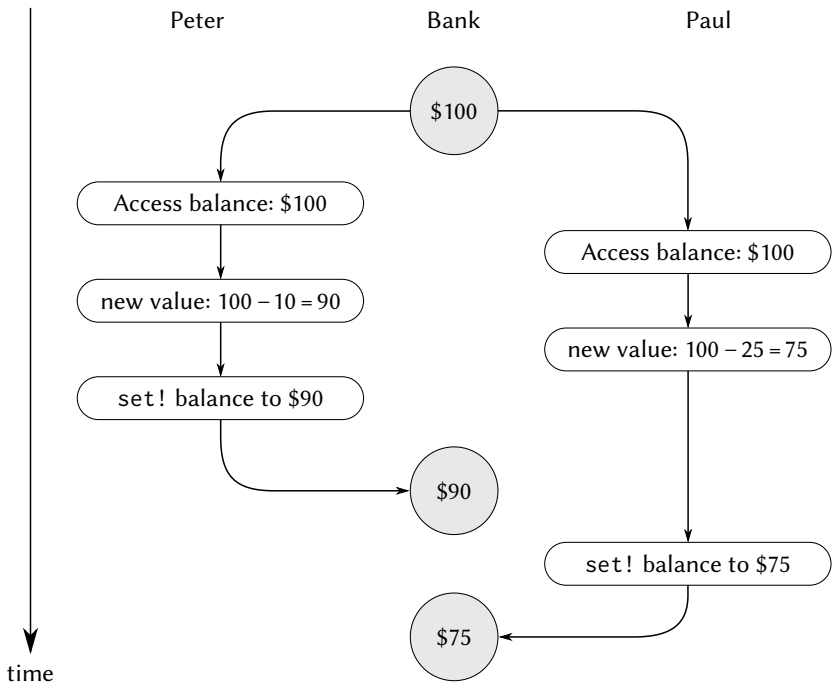

Figure 3.29: Timing diagram showing how interleaving the order of events in two banking withdrawals can lead to an incorrect final balance.

assumption, however, became invalid when Peter changed `balance` to 90. This is a catastrophic failure for the banking system, because the total amount of money in the system is not conserved. Before the transactions, the total amount of money was $100. Afterwards, Peter has $10, Paul has $25, and the bank has $75.[36]

The general phenomenon illustrated here is that several processes may share a common state variable. What makes this complicated is that more than one process may be trying to manipulate the shared state at the same time. For the bank account example, during each transaction, each customer should be able to act as if the other customers did not exist. When a customer changes the balance in a way that depends on the balance, he must be able to assume that, just before the moment of change, the balance is still what he thought it was.

Correct behavior of concurrent programs

The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use `set!`, because the results of a computation depend on the order in which the

[36] An even worse failure for this system could occur if the two `set!` operations attempt to change the balance simultaneously, in which case the actual data appearing in memory might end up being a random combination of the information being written by the two processes. Most computers have interlocks on the primitive memory-write operations, which protect against such simultaneous access. Even this seemingly simple kind of protection, however, raises implementation challenges in the design of multiprocessing computers, where elaborate *cache-coherence* protocols are required to ensure that the various processors will maintain a consistent view of memory contents, despite the fact that data may be replicated ("cached") among the different processors to increase the speed of memory access.

assignments occur.[37] With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

One possible restriction on concurrency would stipulate that no two operations that change any shared state variables can occur at the same time. This is an extremely stringent requirement. For distributed banking, it would require the system designer to ensure that only one transaction could proceed at a time. This would be both inefficient and overly conservative. Figure 3.30 shows Peter and Paul sharing a bank account, where Paul has a private account as well. The diagram illustrates two withdrawals from the shared account (one by Peter and one by Paul) and a deposit to Paul's private account.[38] The two withdrawals from the shared account must not be concurrent (since both access and update the same account), and Paul's deposit and withdrawal must not be concurrent (since both access and update the amount in Paul's wallet). But there should be no problem permitting Paul's deposit to his private account to proceed concurrently with Peter's withdrawal from the shared account.

A less stringent restriction on concurrency would ensure that a con-

[37] The factorial program in Section 3.1.3 illustrates this for a single sequential process.

[38] The columns show the contents of Peter's wallet, the joint account (in Bank1), Paul's wallet, and Paul's private account (in Bank2), before and after each withdrawal (W) and deposit (D). Peter withdraws $10 from Bank1; Paul deposits $5 in Bank2, then withdraws $25 from Bank1.

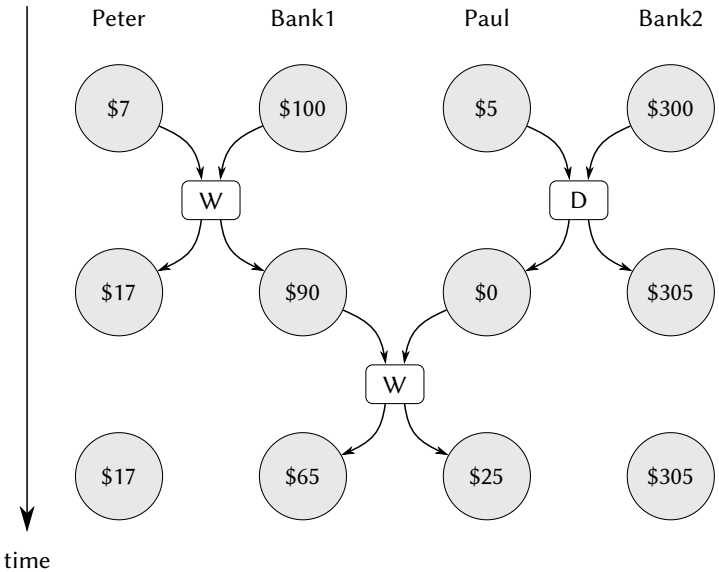

Figure 3.30: Concurrent deposits and withdrawals from a joint account in Bank1 and a private account in Bank2.

current system produces the same result as if the processes had run sequentially in some order. There are two important aspects to this requirement. First, it does not require the processes to actually run sequentially, but only to produce results that are the same *as if* they had run sequentially. For the example in Figure 3.30, the designer of the bank account system can safely allow Paul's deposit and Peter's withdrawal to happen concurrently, because the net result will be the same as if the two operations had happened sequentially. Second, there may be more than one possible "correct" result produced by a concurrent program, because we require only that the result be the same as for

some sequential order. For example, suppose that Peter and Paul's joint account starts out with $100, and Peter deposits $40 while Paul concurrently withdraws half the money in the account. Then sequential execution could result in the account balance being either $70 or $90 (see Exercise 3.38).[39]

There are still weaker requirements for correct execution of concurrent programs. A program for simulating diffusion (say, the flow of heat in an object) might consist of a large number of processes, each one representing a small volume of space, that update their values concurrently. Each process repeatedly changes its value to the average of its own value and its neighbors' values. This algorithm converges to the right answer independent of the order in which the operations are done; there is no need for any restrictions on concurrent use of the shared values.

> **Exercise 3.38:** Suppose that Peter, Paul, and Mary share a joint bank account that initially contains $100. Concurrently, Peter deposits $10, Paul withdraws $20, and Mary withdraws half the money in the account, by executing the following commands:
>
> ```
> Peter: (set! balance (+ balance 10))
> Paul:  (set! balance (- balance 20))
> Mary:  (set! balance (- balance (/ balance 2)))
> ```
>
> a. List all the different possible values for `balance` after these three transactions have been completed, assum-

[39] A more formal way to express this idea is to say that concurrent programs are inherently *nondeterministic*. That is, they are described not by single-valued functions, but by functions whose results are sets of possible values. In Section 4.3 we will study a language for expressing nondeterministic computations.

ing that the banking system forces the three processes to run sequentially in some order.

b. What are some other values that could be produced if the system allows the processes to be interleaved? Draw timing diagrams like the one in Figure 3.29 to explain how these values can occur.

3.4.2 Mechanisms for Controlling Concurrency

We've seen that the difficulty in dealing with concurrent processes is rooted in the need to consider the interleaving of the order of events in the different processes. For example, suppose we have two processes, one with three ordered events (a, b, c) and one with three ordered events (x, y, z). If the two processes run concurrently, with no constraints on how their execution is interleaved, then there are 20 different possible orderings for the events that are consistent with the individual orderings for the two processes:

```
(a,b,c,x,y,z)  (a,x,b,y,c,z)  (x,a,b,c,y,z)  (x,a,y,z,b,c)
(a,b,x,c,y,z)  (a,x,b,y,z,c)  (x,a,b,y,c,z)  (x,y,a,b,c,z)
(a,b,x,y,c,z)  (a,x,y,b,c,z)  (x,a,b,y,z,c)  (x,y,a,b,z,c)
(a,b,x,y,z,c)  (a,x,y,b,z,c)  (x,a,y,b,c,z)  (x,y,a,z,b,c)
(a,x,b,c,y,z)  (a,x,y,z,b,c)  (x,a,y,b,z,c)  (x,y,z,a,b,c)
```

As programmers designing this system, we would have to consider the effects of each of these 20 orderings and check that each behavior is acceptable. Such an approach rapidly becomes unwieldy as the numbers of processes and events increase.

A more practical approach to the design of concurrent systems is to devise general mechanisms that allow us to constrain the interleaving of concurrent processes so that we can be sure that the program behavior

is correct. Many mechanisms have been developed for this purpose. In this section, we describe one of them, the *serializer*.

Serializing access to shared state

Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of procedures that cannot be executed concurrently. More precisely, serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time. If some procedure in the set is being executed, then a process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.

We can use serialization to control access to shared variables. For example, if we want to update a shared variable based on the previous value of that variable, we put the access to the previous value of the variable and the assignment of the new value to the variable in the same procedure. We then ensure that no other procedure that assigns to the variable can run concurrently with this procedure by serializing all of these procedures with the same serializer. This guarantees that the value of the variable cannot be changed between an access and the corresponding assignment.

Serializers in Scheme

To make the above mechanism more concrete, suppose that we have extended Scheme to include a procedure called `parallel-execute`:

`(parallel-execute` $\langle p_1 \rangle$ $\langle p_2 \rangle$ `...` $\langle p_k \rangle$`)`

Each $\langle p \rangle$ must be a procedure of no arguments. `parallel-execute` creates a separate process for each $\langle p \rangle$, which applies $\langle p \rangle$ (to no arguments).

These processes all run concurrently.[40]

As an example of how this is used, consider

```
(define x 10)
(parallel-execute
 (lambda () (set! x (* x x)))
 (lambda () (set! x (+ x 1))))
```

This creates two concurrent processes—P_1, which sets `x` to `x` times `x`, and P_2, which increments `x`. After execution is complete, `x` will be left with one of five possible values, depending on the interleaving of the events of P_1 and P_2:

`101`: P_1 sets `x` to 100 and then P_2 increments `x` to 101.
`121`: P_2 increments `x` to 11 and then P_1 sets `x` to `x * x`.
`110`: P_2 changes `x` from 10 to 11 between the two times that P_1 accesses the value of `x` during the evaluation of `(* x x)`.
`11`: P_2 accesses `x`, then P_1 sets `x` to 100, then P_2 sets `x`.
`100`: P_1 accesses `x` (twice), then P_2 sets `x` to 11, then P_1 sets `x`.

We can constrain the concurrency by using serialized procedures, which are created by *serializers*. Serializers are constructed by `make-serializer`, whose implementation is given below. A serializer takes a procedure as argument and returns a serialized procedure that behaves like the original procedure. All calls to a given serializer return serialized procedures in the same set.

Thus, in contrast to the example above, executing

```
(define x 10)
```

[40] `parallel-execute` is not part of standard Scheme, but it can be implemented in MIT Scheme. In our implementation, the new concurrent processes also run concurrently with the original Scheme process. Also, in our implementation, the value returned by `parallel-execute` is a special control object that can be used to halt the newly created processes.

```
(define s (make-serializer))
(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

can produce only two possible values for `x`, 101 or 121. The other possibilities are eliminated, because the execution of P_1 and P_2 cannot be interleaved.

Here is a version of the `make-account` procedure from Section 3.1.1, where the deposits and withdrawals have been serialized:

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Unknown request: MAKE-ACCOUNT"
                         m))))
    dispatch))
```

With this implementation, two processes cannot be withdrawing from or depositing into a single account concurrently. This eliminates the source of the error illustrated in Figure 3.29, where Peter changes the account balance between the times when Paul accesses the balance to compute the new value and when Paul actually performs the assign-

ment. On the other hand, each account has its own serializer, so that deposits and withdrawals for different accounts can proceed concurrently.

Exercise 3.39: Which of the five possibilities in the parallel execution shown above remain if we instead serialize execution as follows:

```
(define x 10)
(define s (make-serializer))
(parallel-execute
 (lambda () (set! x ((s (lambda () (* x x))))))
 (s (lambda () (set! x (+ x 1)))))
```

Exercise 3.40: Give all possible values of x that can result from executing

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (* x x x))))
```

Which of these possibilities remain if we instead use serialized procedures:

```
(define x 10)
(define s (make-serializer))
(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (* x x x)))))
```

Exercise 3.41: Ben Bitdiddle worries that it would be better to implement the bank account as follows (where the commented line has been changed):

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
                     (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance)
             ((protected
               (lambda () balance)))) ; serialized
            (else
             (error "Unknown request: MAKE-ACCOUNT"
                    m))))
    dispatch))
```

because allowing unserialized access to the bank balance can result in anomalous behavior. Do you agree? Is there any scenario that demonstrates Ben's concern?

Exercise 3.42: Ben Bitdiddle suggests that it's a waste of time to create a new serialized procedure in response to every `withdraw` and `deposit` message. He says that `make-account` could be changed so that the calls to `protected` are done outside the `dispatch` procedure. That is, an account would return the same serialized procedure (which

was created at the same time as the account) each time it is asked for a withdrawal procedure.

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (let ((protected-withdraw (protected withdraw))
          (protected-deposit (protected deposit)))
      (define (dispatch m)
        (cond ((eq? m 'withdraw) protected-withdraw)
              ((eq? m 'deposit) protected-deposit)
              ((eq? m 'balance) balance)
              (else
               (error "Unknown request: MAKE-ACCOUNT"
                      m))))
      dispatch)))
```

Is this a safe change to make? In particular, is there any difference in what concurrency is allowed by these two versions of `make-account`?

Complexity of using multiple shared resources

Serializers provide a powerful abstraction that helps isolate the complexities of concurrent programs so that they can be dealt with carefully and (hopefully) correctly. However, while using serializers is relatively

straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources.

To illustrate one of the difficulties that can arise, suppose we wish to swap the balances in two bank accounts. We access each account to find the balance, compute the difference between the balances, withdraw this difference from one account, and deposit it in the other account. We could implement this as follows:[41]

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

This procedure works well when only a single process is trying to do the exchange. Suppose, however, that Peter and Paul both have access to accounts $a1$, $a2$, and $a3$, and that Peter exchanges $a1$ and $a2$ while Paul concurrently exchanges $a1$ and $a3$. Even with account deposits and withdrawals serialized for individual accounts (as in the `make-account` procedure shown above in this section), `exchange` can still produce incorrect results. For example, Peter might compute the difference in the balances for $a1$ and $a2$, but then Paul might change the balance in $a1$ before Peter is able to complete the exchange.[42] For correct behavior, we must arrange for the `exchange` procedure to lock out any other concurrent accesses to the accounts during the entire time of the exchange.

[41] We have simplified `exchange` by exploiting the fact that our `deposit` message accepts negative amounts. (This is a serious bug in our banking system!)

[42] If the account balances start out as $10, $20, and $30, then after any number of concurrent exchanges, the balances should still be $10, $20, and $30 in some order. Serializing the deposits to individual accounts is not sufficient to guarantee this. See Exercise 3.43.

One way we can accomplish this is by using both accounts' serializers to serialize the entire `exchange` procedure. To do this, we will arrange for access to an account's serializer. Note that we are deliberately breaking the modularity of the bank-account object by exposing the serializer. The following version of `make-account` is identical to the original version given in Section 3.1.1, except that a serializer is provided to protect the balance variable, and the serializer is exported via message passing:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Unknown request: MAKE-ACCOUNT" m))))
    dispatch))
```

We can use this to do serialized deposits and withdrawals. However, unlike our earlier serialized account, it is now the responsibility of each user of bank-account objects to explicitly manage the serialization, for example as follows:[43]

[43] Exercise 3.45 investigates why deposits and withdrawals are no longer automatically serialized by the account.

```
(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))
```

Exporting the serializer in this way gives us enough flexibility to implement a serialized exchange program. We simply serialize the original `exchange` procedure with the serializers for both accounts:

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))
```

Exercise 3.43: Suppose that the balances in three accounts start out as $10, $20, and $30, and that multiple processes run, exchanging the balances in the accounts. Argue that if the processes are run sequentially, after any number of concurrent exchanges, the account balances should be $10, $20, and $30 in some order. Draw a timing diagram like the one in Figure 3.29 to show how this condition can be violated if the exchanges are implemented using the first version of the account-exchange program in this section. On the other hand, argue that even with this `exchange` program, the sum of the balances in the accounts will be preserved. Draw a timing diagram to show how even this condition would be violated if we did not serialize the transactions on individual accounts.

Exercise 3.44: Consider the problem of transferring an amount from one account to another. Ben Bitdiddle claims that this

can be accomplished with the following procedure, even if there are multiple people concurrently transferring money among multiple accounts, using any account mechanism that serializes deposit and withdrawal transactions, for example, the version of `make-account` in the text above.

```
(define (transfer from-account to-account amount)
  ((from-account 'withdraw) amount)
  ((to-account 'deposit) amount))
```

Louis Reasoner claims that there is a problem here, and that we need to use a more sophisticated method, such as the one required for dealing with the exchange problem. Is Louis right? If not, what is the essential difference between the transfer problem and the exchange problem? (You should assume that the balance in `from-account` is at least `amount`.)

Exercise 3.45: Louis Reasoner thinks our bank-account system is unnecessarily complex and error-prone now that deposits and withdrawals aren't automatically serialized. He suggests that `make-account-and-serializer` should have exported the serializer (for use by such procedures as `serialized-exchange`) in addition to (rather than instead of) using it to serialize accounts and deposits as `make-account` did. He proposes to redefine accounts as follows:

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
```

```
(let ((balance-serializer (make-serializer)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) (balance-serializer withdraw))
          ((eq? m 'deposit) (balance-serializer deposit))
          ((eq? m 'balance) balance)
          ((eq? m 'serializer) balance-serializer)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch))
```

Then deposits are handled as with the original `make-account`:

```
(define (deposit account amount)
  ((account 'deposit) amount))
```

Explain what is wrong with Louis's reasoning. In particular, consider what happens when `serialized-exchange` is called.

Implementing serializers

We implement serializers in terms of a more primitive synchronization mechanism called a *mutex*. A mutex is an object that supports two operations—the mutex can be *acquired*, and the mutex can be *released*. Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.[44] In our implementa-

[44]The term "mutex" is an abbreviation for *mutual exclusion*. The general problem of arranging a mechanism that permits concurrent processes to safely share resources is called the mutual exclusion problem. Our mutex is a simple variant of the *semaphore* mechanism (see Exercise 3.47), which was introduced in the "THE" Multiprogramming System developed at the Technological University of Eindhoven and named for the university's initials in Dutch (Dijkstra 1968a). The acquire and release operations were originally called P and V, from the Dutch words *passeren* (to pass) and *vrijgeven* (to release), in reference to the semaphores used on railroad systems. Dijkstra's classic exposition (Dijkstra 1968b) was one of the first to clearly present the issues of concur-

tion, each serializer has an associated mutex. Given a procedure p, the serializer returns a procedure that acquires the mutex, runs p, and then releases the mutex. This ensures that only one of the procedures produced by the serializer can be running at once, which is precisely the serialization property that we need to guarantee.

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

The mutex is a mutable object (here we'll use a one-element list, which we'll refer to as a *cell*) that can hold the value true or false. When the value is false, the mutex is available to be acquired. When the value is true, the mutex is unavailable, and any process that attempts to acquire the mutex must wait.

Our mutex constructor `make-mutex` begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available.[45] To release the mutex, we set the cell

rency control, and showed how to use semaphores to handle a variety of concurrency problems.

[45] In most time-shared operating systems, processes that are blocked by a mutex do not waste time "busy-waiting" as above. Instead, the system schedules another process to run while the first is waiting, and the blocked process is awakened when the mutex becomes available.

contents to false.

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))
(define (clear! cell) (set-car! cell false))
```

`test-and-set!` tests the cell and returns the result of the test. In addition, if the test was false, `test-and-set!` sets the cell contents to true before returning false. We can express this behavior as the following procedure:

```
(define (test-and-set! cell)
  (if (car cell) true (begin (set-car! cell true) false)))
```

However, this implementation of `test-and-set!` does not suffice as it stands. There is a crucial subtlety here, which is the essential place where concurrency control enters the system: The `test-and-set!` operation must be performed *atomically*. That is, we must guarantee that, once a process has tested the cell and found it to be false, the cell contents will actually be set to true before any other process can test the cell. If we do not make this guarantee, then the mutex can fail in a way similar to the bank-account failure in Figure 3.29. (See Exercise 3.46.)

The actual implementation of `test-and-set!` depends on the details of how our system runs concurrent processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and mov-

ing on to the next process. In that case, `test-and-set!` can work by disabling time slicing during the testing and setting.[46] Alternatively, multiprocessing computers provide instructions that support atomic operations directly in hardware.[47]

> **Exercise 3.46:** Suppose that we implement `test-and-set!` using an ordinary procedure as shown in the text, without attempting to make the operation atomic. Draw a timing

[46] In MIT Scheme for a single processor, which uses a time-slicing model, `test-and-set!` can be implemented as follows:

```
(define (test-and-set! cell)
  (without-interrupts
   (lambda ()
     (if (car cell)
         true
         (begin (set-car! cell true)
                false)))))
```

`without-interrupts` disables time-slicing interrupts while its procedure argument is being executed.

[47] There are many variants of such instructions—including test-and-set, test-and-clear, swap, compare-and-exchange, load-reserve, and store-conditional—whose design must be carefully matched to the machine's processor-memory interface. One issue that arises here is to determine what happens if two processes attempt to acquire the same resource at exactly the same time by using such an instruction. This requires some mechanism for making a decision about which process gets control. Such a mechanism is called an *arbiter*. Arbiters usually boil down to some sort of hardware device. Unfortunately, it is possible to prove that one cannot physically construct a fair arbiter that works 100% of the time unless one allows the arbiter an arbitrarily long time to make its decision. The fundamental phenomenon here was originally observed by the fourteenth-century French philosopher Jean Buridan in his commentary on Aristotle's *De caelo*. Buridan argued that a perfectly rational dog placed between two equally attractive sources of food will starve to death, because it is incapable of deciding which to go to first.

diagram like the one in Figure 3.29 to demonstrate how the mutex implementation can fail by allowing two processes to acquire the mutex at the same time.

Exercise 3.47: A semaphore (of size n) is a generalization of a mutex. Like a mutex, a semaphore supports acquire and release operations, but it is more general in that up to n processes can acquire it concurrently. Additional processes that attempt to acquire the semaphore must wait for release operations. Give implementations of semaphores

a. in terms of mutexes
b. in terms of atomic `test-and-set!` operations.

Deadlock

Now that we have seen how to implement serializers, we can see that account exchanging still has a problem, even with the `serialized-exchange` procedure above. Imagine that Peter attempts to exchange $a1$ with $a2$ while Paul concurrently attempts to exchange $a2$ with $a1$. Suppose that Peter's process reaches the point where it has entered a serialized procedure protecting $a1$ and, just after that, Paul's process enters a serialized procedure protecting $a2$. Now Peter cannot proceed (to enter a serialized procedure protecting $a2$) until Paul exits the serialized procedure protecting $a2$. Similarly, Paul cannot proceed until Peter exits the serialized procedure protecting $a1$. Each process is stalled forever, waiting for the other. This situation is called a *deadlock*. Deadlock is always a danger in systems that provide concurrent access to multiple shared resources.

One way to avoid the deadlock in this situation is to give each account a unique identification number and rewrite `serialized-exchange`

so that a process will always attempt to enter a procedure protecting the lowest-numbered account first. Although this method works well for the exchange problem, there are other situations that require more sophisticated deadlock-avoidance techniques, or where deadlock cannot be avoided at all. (See Exercise 3.48 and Exercise 3.49.)[48]

> **Exercise 3.48:** Explain in detail why the deadlock-avoidance method described above, (i.e., the accounts are numbered, and each process attempts to acquire the smaller-numbered account first) avoids deadlock in the exchange problem. Rewrite `serialized-exchange` to incorporate this idea. (You will also need to modify `make-account` so that each account is created with a number, which can be accessed by sending an appropriate message.)

> **Exercise 3.49:** Give a scenario where the deadlock-avoidance mechanism described above does not work. (Hint: In the exchange problem, each process knows in advance which accounts it will need to get access to. Consider a situation where a process must get access to some shared resources before it can know which additional shared resources it will require.)

[48] The general technique for avoiding deadlock by numbering the shared resources and acquiring them in order is due to Havender (1968). Situations where deadlock cannot be avoided require *deadlock-recovery* methods, which entail having processes "back out" of the deadlocked state and try again. Deadlock-recovery mechanisms are widely used in database management systems, a topic that is treated in detail in Gray and Reuter 1993.

Concurrency, time, and communication

We've seen how programming concurrent systems requires controlling the ordering of events when different processes access shared state, and we've seen how to achieve this control through judicious use of serializers. But the problems of concurrency lie deeper than this, because, from a fundamental point of view, it's not always clear what is meant by "shared state."

Mechanisms such as `test-and-set!` require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.[49]

The problematic aspects of shared state also arise in large, distributed systems. For instance, imagine a distributed banking system where individual branch banks maintain local values for bank balances and periodically compare these with values maintained by other branches. In such a system the value of "the account balance" would be undetermined, except right after synchronization. If Peter deposits money in an account he holds jointly with Paul, when should we say that the account balance has changed—when the balance in the local branch changes, or not until after the synchronization? And if Paul accesses the account

[49] One such alternative to serialization is called *barrier synchronization*. The programmer permits concurrent processes to execute as they please, but establishes certain synchronization points ("barriers") through which no process can proceed until all the processes have reached the barrier. Modern processors provide machine instructions that permit programmers to establish synchronization points at places where consistency is required. The PowerPC, for example, includes for this purpose two instructions called SYNC and EIEIO (Enforced In-order Execution of Input/Output).

from a different branch, what are the reasonable constraints to place on the banking system such that the behavior is "correct"? The only thing that might matter for correctness is the behavior observed by Peter and Paul individually and the "state" of the account immediately after synchronization. Questions about the "real" account balance or the order of events between synchronizations may be irrelevant or meaningless.[50]

The basic phenomenon here is that synchronizing different processes, establishing shared state, or imposing an order on events requires communication among the processes. In essence, any notion of time in concurrency control must be intimately tied to communication.[51] It is intriguing that a similar connection between time and communication also arises in the Theory of Relativity, where the speed of light (the fastest signal that can be used to synchronize events) is a fundamental constant relating time and space. The complexities we encounter in dealing with time and state in our computational models may in fact mirror a fundamental complexity of the physical universe.

3.5 Streams

We've gained a good understanding of assignment as a tool in modeling, as well as an appreciation of the complex problems that assignment raises. It is time to ask whether we could have gone about things in a different way, so as to avoid some of these problems. In this section,

[50] This may seem like a strange point of view, but there are systems that work this way. International charges to credit-card accounts, for example, are normally cleared on a per-country basis, and the charges made in different countries are periodically reconciled. Thus the account balance may be different in different countries.

[51] For distributed systems, this perspective was pursued by Lamport (1978), who showed how to use communication to establish "global clocks" that can be used to establish orderings on events in distributed systems.

we explore an alternative approach to modeling state, based on data structures called *streams*. As we shall see, streams can mitigate some of the complexity of modeling state.

Let's step back and review where this complexity comes from. In an attempt to model real-world phenomena, we made some apparently reasonable decisions: We modeled real-world objects with local state by computational objects with local variables. We identified time variation in the real world with time variation in the computer. We implemented the time variation of the states of the model objects in the computer with assignments to the local variables of the model objects.

Is there another approach? Can we avoid identifying time in the computer with time in the modeled world? Must we make the model change with time in order to model phenomena in a changing world? Think about the issue in terms of mathematical functions. We can describe the time-varying behavior of a quantity x as a function of time $x(t)$. If we concentrate on x instant by instant, we think of it as a changing quantity. Yet if we concentrate on the entire time history of values, we do not emphasize change—the function itself does not change.[52]

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence. In this section, we will see how to model change in terms of sequences that represent the time histories of the systems being modeled. To accomplish this, we introduce new data structures called *streams*. From an abstract point of view, a stream is simply a sequence. However, we will find that the straightforward implementation of streams as lists (as in Section 2.2.1) doesn't fully re-

[52]Physicists sometimes adopt this view by introducing the "world lines" of particles as a device for reasoning about motion. We've also already mentioned (Section 2.2.3) that this is the natural way to think about signal-processing systems. We will explore applications of streams to signal processing in Section 3.5.3.

veal the power of stream processing. As an alternative, we introduce the technique of *delayed evaluation*, which enables us to represent very large (even infinite) sequences as streams.

Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment. On the other hand, the stream framework raises difficulties of its own, and the question of which modeling technique leads to more modular and more easily maintained systems remains open.

3.5.1 Streams Are Delayed Lists

As we saw in Section 2.2.3, sequences can serve as standard interfaces for combining program modules. We formulated powerful abstractions for manipulating sequences, such as `map`, `filter`, and `accumulate`, that capture a wide variety of operations in a manner that is both succinct and elegant.

Unfortunately, if we represent sequences as lists, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations. When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

To see why this is true, let us compare two programs for computing the sum of all the prime numbers in an interval. The first program is written in standard iterative style:[53]

[53] Assume that we have a predicate `prime?` (e.g., as in Section 1.2.6) that tests for primality.

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
             (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

The second program performs the same computation using the sequence operations of Section 2.2.3:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

In carrying out the computation, the first program needs to store only the sum being accumulated. In contrast, the filter in the second program cannot do any testing until `enumerate-interval` has constructed a complete list of the numbers in the interval. The filter generates another list, which in turn is passed to `accumulate` before being collapsed to form a sum. Such large intermediate storage is not needed by the first program, which we can think of as enumerating the interval incrementally, adding each prime to the sum as it is generated.

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

This expression does find the second prime, but the computational overhead is outrageous. We construct a list of almost a million integers, filter

this list by testing each element for primality, and then ignore almost all of the result. In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.

Streams are a clever idea that allows one to use sequence manipulations without incurring the costs of manipulating sequences as lists. With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation. The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists. In other words, although we will write programs as if we were processing complete sequences, we design our stream implementation to automatically and transparently interleave the construction of the stream with its use.

On the surface, streams are just lists with different names for the procedures that manipulate them. There is a constructor, `cons-stream`, and two selectors, `stream-car` and `stream-cdr`, which satisfy the constraints

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

There is a distinguishable object, `the-empty-stream`, which cannot be the result of any `cons-stream` operation, and which can be identified with the predicate `stream-null?`.[54] Thus we can make and use streams,

[54] In the MIT implementation, `the-empty-stream` is the same as the empty list `'()`, and `stream-null?` is the same as `null?`.

in just the same way as we can make and use lists, to represent aggregate data arranged in a sequence. In particular, we can build stream analogs of the list operations from Chapter 2, such as `list-ref`, `map`, and `for-each`:[55]

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))
(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

`stream-for-each` is useful for viewing streams:

```
(define (display-stream s)
  (stream-for-each display-line s))
(define (display-line x) (newline) (display x))
```

To make the stream implementation automatically and transparently interleave the construction of a stream with its use, we will arrange for

[55] This should bother you. The fact that we are defining such similar procedures for streams and lists indicates that we are missing some underlying abstraction. Unfortunately, in order to exploit this abstraction, we will need to exert finer control over the process of evaluation than we can at present. We will discuss this point further at the end of Section 3.5.4. In Section 4.2, we'll develop a framework that unifies lists and streams.

the cdr of a stream to be evaluated when it is accessed by the stream-cdr procedure rather than when the stream is constructed by cons-stream. This implementation choice is reminiscent of our discussion of rational numbers in Section 2.1.2, where we saw that we can choose to implement rational numbers so that the reduction of numerator and denominator to lowest terms is performed either at construction time or at selection time. The two rational-number implementations produce the same data abstraction, but the choice has an effect on efficiency. There is a similar relationship between streams and ordinary lists. As a data abstraction, streams are the same as lists. The difference is the time at which the elements are evaluated. With ordinary lists, both the car and the cdr are evaluated at construction time. With streams, the cdr is evaluated at selection time.

Our implementation of streams will be based on a special form called delay. Evaluating (delay ⟨*exp*⟩) does not evaluate the expression ⟨*exp*⟩, but rather returns a so-called *delayed object*, which we can think of as a "promise" to evaluate ⟨*exp*⟩ at some future time. As a companion to delay, there is a procedure called force that takes a delayed object as argument and performs the evaluation—in effect, forcing the delay to fulfill its promise. We will see below how delay and force can be implemented, but first let us use these to construct streams.

cons-stream is a special form defined so that

```
(cons-stream ⟨a⟩ ⟨b⟩)
```

is equivalent to

```
(cons ⟨a⟩ (delay ⟨b⟩))
```

What this means is that we will construct streams using pairs. However, rather than placing the value of the rest of the stream into the cdr of the pair we will put there a promise to compute the rest if it is ever

requested. `stream-car` and `stream-cdr` can now be defined as procedures:

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

`stream-car` selects the `car` of the pair; `stream-cdr` selects the `cdr` of the pair and evaluates the delayed expression found there to obtain the rest of the stream.[56]

The stream implementation in action

To see how this implementation behaves, let us analyze the "outrageous" prime computation we saw above, reformulated in terms of streams:

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                 (stream-enumerate-interval
                  10000 1000000))))
```

We will see that it does indeed work efficiently.

We begin by calling `stream-enumerate-interval` with the arguments 10,000 and 1,000,000. `stream-enumerate-interval` is the stream analog of `enumerate-interval` (Section 2.2.3):

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
```

[56] Although `stream-car` and `stream-cdr` can be defined as procedures, `cons-stream` must be a special form. If `cons-stream` were a procedure, then, according to our model of evaluation, evaluating `(cons-stream` $\langle a \rangle$ $\langle b \rangle$`)` would automatically cause $\langle b \rangle$ to be evaluated, which is precisely what we do not want to happen. For the same reason, `delay` must be a special form, though `force` can be an ordinary procedure.

```
      low
      (stream-enumerate-interval (+ low 1) high))))
```

and thus the result returned by `stream-enumerate-interval`, formed by the `cons-stream`, is[57]

```
(cons 10000
      (delay (stream-enumerate-interval 10001 1000000)))
```

That is, `stream-enumerate-interval` returns a stream represented as a pair whose `car` is 10,000 and whose `cdr` is a promise to enumerate more of the interval if so requested. This stream is now filtered for primes, using the stream analog of the `filter` procedure (Section 2.2.3):

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter
                       pred
                       (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

`stream-filter` tests the `stream-car` of the stream (the `car` of the pair, which is 10,000). Since this is not prime, `stream-filter` examines the `stream-cdr` of its input stream. The call to `stream-cdr` forces evaluation of the delayed `stream-enumerate-interval`, which now returns

```
(cons 10001
      (delay (stream-enumerate-interval 10002 1000000)))
```

[57] The numbers shown here do not really appear in the delayed expression. What actually appears is the original expression, in an environment in which the variables are bound to the appropriate numbers. For example, `(+ low 1)` with `low` bound to 10,000 actually appears where `10001` is shown.

`stream-filter` now looks at the `stream-car` of this stream, 10,001, sees that this is not prime either, forces another `stream-cdr`, and so on, until `stream-enumerate-interval` yields the prime 10,007, whereupon `stream-filter`, according to its definition, returns

```
(cons-stream (stream-car stream)
             (stream-filter pred (stream-cdr stream)))
```

which in this case is

```
(cons 10007
      (delay (stream-filter
              prime?
              (cons 10008
                    (delay (stream-enumerate-interval
                            10009
                            1000000))))))
```

This result is now passed to `stream-cdr` in our original expression. This forces the delayed `stream-filter`, which in turn keeps forcing the delayed `stream-enumerate-interval` until it finds the next prime, which is 10,009. Finally, the result passed to `stream-car` in our original expression is

```
(cons 10009
      (delay (stream-filter
              prime?
              (cons 10010
                    (delay (stream-enumerate-interval
                            10011
                            1000000))))))
```

`stream-car` returns 10,009, and the computation is complete. Only as many integers were tested for primality as were necessary to find the

second prime, and the interval was enumerated only as far as was necessary to feed the prime filter.

In general, we can think of delayed evaluation as "demand-driven" programming, whereby each stage in the stream process is activated only enough to satisfy the next stage. What we have done is to decouple the actual order of events in the computation from the apparent structure of our procedures. We write procedures as if the streams existed "all at once" when, in reality, the computation is performed incrementally, as in traditional programming styles.

Implementing `delay` and `force`

Although `delay` and `force` may seem like mysterious operations, their implementation is really quite straightforward. `delay` must package an expression so that it can be evaluated later on demand, and we can accomplish this simply by treating the expression as the body of a procedure. `delay` can be a special form such that

```
(delay ⟨exp⟩)
```

is syntactic sugar for

```
(lambda () ⟨exp⟩)
```

`force` simply calls the procedure (of no arguments) produced by `delay`, so we can implement `force` as a procedure:

```
(define (force delayed-object) (delayed-object))
```

This implementation suffices for `delay` and `force` to work as advertised, but there is an important optimization that we can include. In many applications, we end up forcing the same delayed object many times. This can lead to serious inefficiency in recursive programs involving streams. (See Exercise 3.57.) The solution is to build delayed objects so that the

first time they are forced, they store the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement `delay` as a special-purpose memoized procedure similar to the one described in Exercise 3.27. One way to accomplish this is to use the following procedure, which takes as argument a procedure (of no arguments) and returns a memoized version of the procedure. The first time the memoized procedure is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

`delay` is then defined so that `(delay` ⟨*exp*⟩`)` is equivalent to

```
(memo-proc (lambda () ⟨exp⟩))
```

and `force` is as defined previously.[58]

[58] There are many possible implementations of streams other than the one described in this section. Delayed evaluation, which is the key to making streams practical, was inherent in Algol 60's *call-by-name* parameter-passing method. The use of this mechanism to implement streams was first described by Landin (1965). Delayed evaluation for streams was introduced into Lisp by Friedman and Wise (1976). In their implementation, `cons` always delays evaluating its arguments, so that lists automatically behave as streams. The memoizing optimization is also known as *call-by-need*. The Algol community would refer to our original delayed objects as *call-by-name thunks* and to the optimized versions as *call-by-need thunks*.

Exercise 3.50: Complete the following definition, which generalizes `stream-map` to allow procedures that take multiple arguments, analogous to `map` in Section 2.2.1, Footnote 12.

```
(define (stream-map proc . argstreams)
  (if (⟨??⟩ (car argstreams))
      the-empty-stream
      (⟨??⟩
       (apply proc (map ⟨??⟩ argstreams))
       (apply stream-map
              (cons proc (map ⟨??⟩ argstreams))))))
```

Exercise 3.51: In order to take a closer look at delayed evaluation, we will use the following procedure, which simply returns its argument after printing it:

```
(define (show x)
  (display-line x)
  x)
```

What does the interpreter print in response to evaluating each expression in the following sequence?[59]

```
(define x
```

[59] Exercises such as Exercise 3.51 and Exercise 3.52 are valuable for testing our understanding of how `delay` works. On the other hand, intermixing delayed evaluation with printing—and, even worse, with assignment—is extremely confusing, and instructors of courses on computer languages have traditionally tormented their students with examination questions such as the ones in this section. Needless to say, writing programs that depend on such subtleties is odious programming style. Part of the power of stream processing is that it lets us ignore the order in which events actually happen in our programs. Unfortunately, this is precisely what we cannot afford to do in the presence of assignment, which forces us to be concerned with time and change.

```
  (stream-map show
              (stream-enumerate-interval 0 10)))
(stream-ref x 5)
(stream-ref x 7)
```

Exercise 3.52: Consider the sequence of expressions

```
(define sum 0)
(define (accum x) (set! sum (+ x sum)) sum)
(define seq
  (stream-map accum
              (stream-enumerate-interval 1 20)))
(define y (stream-filter even? seq))
(define z
  (stream-filter (lambda (x) (= (remainder x 5) 0))
                 seq))
(stream-ref y 7)
(display-stream z)
```

What is the value of `sum` after each of the above expressions is evaluated? What is the printed response to evaluating the `stream-ref` and `display-stream` expressions? Would these responses differ if we had implemented `(delay` ⟨*exp*⟩`)` simply as `(lambda () `⟨*exp*⟩`)` without using the optimization provided by `memo-proc`? Explain.

3.5.2 Infinite Streams

We have seen how to support the illusion of manipulating streams as complete entities even though, in actuality, we compute only as much of the stream as we need to access. We can exploit this technique to represent sequences efficiently as streams, even if the sequences are very

long. What is more striking, we can use streams to represent sequences that are infinitely long. For instance, consider the following definition of the stream of positive integers:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))
```

This makes sense because `integers` will be a pair whose `car` is 1 and whose `cdr` is a promise to produce the integers beginning with 2. This is an infinitely long stream, but in any given time we can examine only a finite portion of it. Thus, our programs will never know that the entire infinite stream is not there.

Using `integers` we can define other infinite streams, such as the stream of integers that are not divisible by 7:

```
(define (divisible? x y) (= (remainder x y) 0))
(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

Then we can find integers not divisible by 7 simply by accessing elements of this stream:

```
(stream-ref no-sevens 100)
117
```

In analogy with `integers`, we can define the infinite stream of Fibonacci numbers:

```
(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

`fibs` is a pair whose `car` is 0 and whose `cdr` is a promise to evaluate `(fibgen 1 1)`. When we evaluate this delayed `(fibgen 1 1)`, it will

produce a pair whose `car` is 1 and whose `cdr` is a promise to evaluate `(fibgen 1 2)`, and so on.

For a look at a more exciting infinite stream, we can generalize the `no-sevens` example to construct the infinite stream of prime numbers, using a method known as the *sieve of Eratosthenes*.[60] We start with the integers beginning with 2, which is the first prime. To get the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of this stream. This leaves a stream beginning with 5, which is the next prime, and so on. In other words, we construct the primes by a sieving process, described as follows: To sieve a stream `S`, form a stream whose first element is the first element of `S` and the rest of which is obtained by filtering all multiples of the first element of `S` out of the rest of `S` and sieving the result. This process is readily described in terms of stream operations:

```
(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))
(define primes (sieve (integers-starting-from 2)))
```

[60] Eratosthenes, a third-century B.C. Alexandrian Greek philosopher, is famous for giving the first accurate estimate of the circumference of the Earth, which he computed by observing shadows cast at noon on the day of the summer solstice. Eratosthenes's sieve method, although ancient, has formed the basis for special-purpose hardware "sieves" that, until recently, were the most powerful tools in existence for locating large primes. Since the 70s, however, these methods have been superseded by outgrowths of the probabilistic techniques discussed in Section 1.2.6.

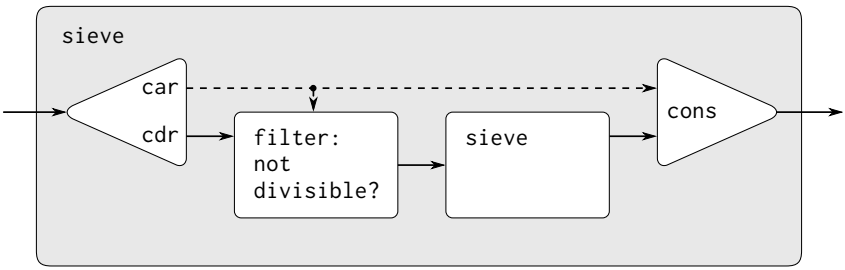

Figure 3.31: The prime sieve viewed as a signal-processing system.

Now to find a particular prime we need only ask for it:

```
(stream-ref primes 50)
233
```

It is interesting to contemplate the signal-processing system set up by `sieve`, shown in the "Henderson diagram" in Figure 3.31.[61] The input stream feeds into an "unconser" that separates the first element of the stream from the rest of the stream. The first element is used to construct a divisibility filter, through which the rest is passed, and the output of the filter is fed to another sieve box. Then the original first element is consed onto the output of the internal sieve to form the output stream. Thus, not only is the stream infinite, but the signal processor is also infinite, because the sieve contains a sieve within it.

[61] We have named these figures after Peter Henderson, who was the first person to show us diagrams of this sort as a way of thinking about stream processing. Each solid line represents a stream of values being transmitted. The dashed line from the `car` to the `cons` and the `filter` indicates that this is a single value rather than a stream.

Defining streams implicitly

The `integers` and `fibs` streams above were defined by specifying "generating" procedures that explicitly compute the stream elements one by one. An alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly. For example, the following expression defines the stream `ones` to be an infinite stream of ones:

```
(define ones (cons-stream 1 ones))
```

This works much like the definition of a recursive procedure: `ones` is a pair whose `car` is 1 and whose `cdr` is a promise to evaluate `ones`. Evaluating the `cdr` gives us again a 1 and a promise to evaluate `ones`, and so on.

We can do more interesting things by manipulating streams with operations such as `add-streams`, which produces the elementwise sum of two given streams:[62]

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

Now we can define the integers as follows:

```
(define integers
  (cons-stream 1 (add-streams ones integers)))
```

This defines `integers` to be a stream whose first element is 1 and the rest of which is the sum of `ones` and `integers`. Thus, the second element of `integers` is 1 plus the first element of `integers`, or 2; the third element of `integers` is 1 plus the second element of `integers`, or 3; and so on. This definition works because, at any point, enough of the `integers` stream has been generated so that we can feed it back into the definition to produce the next integer.

We can define the Fibonacci numbers in the same style:

[62] This uses the generalized version of `stream-map` from Exercise 3.50.

```
(define fibs
  (cons-stream
   0
   (cons-stream 1 (add-streams (stream-cdr fibs) fibs))))
```

This definition says that `fibs` is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding `fibs` to itself shifted by one place:

```
      1  1  2  3  5  8   13  21  ...  =  (stream-cdr fibs)
      0  1  1  2  3  5   8   13  ...  =  fibs
0  1  1  2  3  5  8  13  21  34  ...  =  fibs
```

`scale-stream` is another useful procedure in formulating such stream definitions. This multiplies each item in a stream by a given constant:

```
(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor))
              stream))
```

For example,

```
(define double (cons-stream 1 (scale-stream double 2)))
```

produces the stream of powers of 2: 1, 2, 4, 8, 16, 32, ...

An alternate definition of the stream of primes can be given by starting with the integers and filtering them by testing for primality. We will need the first prime, 2, to get started:

```
(define primes
  (cons-stream
   2
   (stream-filter prime? (integers-starting-from 3))))
```

This definition is not so straightforward as it appears, because we will test whether a number n is prime by checking whether n is divisible by a prime (not by just any integer) less than or equal to $\sqrt{n}$:

```
(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) true)
          ((divisible? n (stream-car ps)) false)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

This is a recursive definition, since `primes` is defined in terms of the `prime?` predicate, which itself uses the `primes` stream. The reason this procedure works is that, at any point, enough of the `primes` stream has been generated to test the primality of the numbers we need to check next. That is, for every n we test for primality, either n is not prime (in which case there is a prime already generated that divides it) or n is prime (in which case there is a prime already generated—i.e., a prime less than n—that is greater than $\sqrt{n}$).[63]

> **Exercise 3.53:** Without running the program, describe the elements of the stream defined by
>
> ```
> (define s (cons-stream 1 (add-streams s s)))
> ```

> **Exercise 3.54:** Define a procedure `mul-streams`, analogous to `add-streams`, that produces the elementwise product of its two input streams. Use this together with the stream of `integers` to complete the following definition of the stream whose n^{th} element (counting from 0) is $n + 1$ factorial:

[63] This last point is very subtle and relies on the fact that $p_{n+1} \leq p_n^2$. (Here, p_k denotes the k^{th} prime.) Estimates such as these are very difficult to establish. The ancient proof by Euclid that there are an infinite number of primes shows that $p_{n+1} \leq p_1 p_2 \cdots p_n + 1$, and no substantially better result was proved until 1851, when the Russian mathematician P. L. Chebyshev established that $p_{n+1} \leq 2p_n$ for all n. This result, originally conjectured in 1845, is known as *Bertrand's hypothesis*. A proof can be found in section 22.3 of Hardy and Wright 1960.

```
(define factorials
  (cons-stream 1 (mul-streams ⟨??⟩ ⟨??⟩)))
```

Exercise 3.55: Define a procedure `partial-sums` that takes as argument a stream S and returns the stream whose elements are S_0, S_0+S_1, $S_0+S_1+S_2$, ... For example, `(partial-sums integers)` should be the stream 1, 3, 6, 10, 15, ...

Exercise 3.56: A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, let us call the required stream of numbers `S` and notice the following facts about it.

- `S` begins with 1.
- The elements of `(scale-stream S 2)` are also elements of `S`.
- The same is true for `(scale-stream S 3)` and `(scale-stream 5 S)`.
- These are all the elements of `S`.

Now all we have to do is combine elements from these sources. For this we define a procedure `merge` that combines two ordered streams into one ordered result stream, eliminating repetitions:

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (let ((s1car (stream-car s1))
               (s2car (stream-car s2)))
           (cond ((< s1car s2car)
                  (cons-stream
                   s1car
                   (merge (stream-cdr s1) s2)))
                 ((> s1car s2car)
                  (cons-stream
                   s2car
                   (merge s1 (stream-cdr s2))))
                 (else
                  (cons-stream
                   s1car
                   (merge (stream-cdr s1)
                          (stream-cdr s2)))))))))
```

Then the required stream may be constructed with `merge`, as follows:

```
(define S (cons-stream 1 (merge ⟨??⟩ ⟨??⟩)))
```

Fill in the missing expressions in the places marked ⟨??⟩ above.

Exercise 3.57: How many additions are performed when we compute the n^{th} Fibonacci number using the definition of `fibs` based on the `add-streams` procedure? Show that the number of additions would be exponentially greater if we had implemented `(delay` ⟨*exp*⟩`)` simply as `(lambda`

`()` ⟨*exp*⟩), without using the optimization provided by the `memo-proc` procedure described in Section 3.5.1.[64]

Exercise 3.58: Give an interpretation of the stream computed by the following procedure:

```
(define (expand num den radix)
  (cons-stream
   (quotient (* num radix) den)
   (expand (remainder (* num radix) den) den radix)))
```

(`quotient` is a primitive that returns the integer quotient of two integers.) What are the successive elements produced by `(expand 1 7 10)`? What is produced by `(expand 3 8 10)`?

Exercise 3.59: In Section 2.5.3 we saw how to implement a polynomial arithmetic system representing polynomials as lists of terms. In a similar way, we can work with *power series*, such as

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} + \dots,$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4 \cdot 3 \cdot 2} - \dots,$$

$$\sin x = x - \frac{x^3}{3 \cdot 2} + \frac{x^5}{5 \cdot 4 \cdot 3 \cdot 2} - \dots$$

[64] This exercise shows how call-by-need is closely related to ordinary memoization as described in Exercise 3.27. In that exercise, we used assignment to explicitly construct a local table. Our call-by-need stream optimization effectively constructs such a table automatically, storing values in the previously forced parts of the stream.

represented as infinite streams. We will represent the series $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$ as the stream whose elements are the coefficients $a_0, a_1, a_2, a_3, \ldots$.

a. The integral of the series $a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots$ is the series

$$c + a_0x + \frac{1}{2}a_1x^2 + \frac{1}{3}a_2x^3 + \frac{1}{4}a_3x^4 + \ldots,$$

where c is any constant. Define a procedure `integrate-series` that takes as input a stream $a_0, a_1, a_2, \ldots$ representing a power series and returns the stream $a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \ldots$ of coefficients of the non-constant terms of the integral of the series. (Since the result has no constant term, it doesn't represent a power series; when we use `integrate-series`, we will `cons` on the appropriate constant.)

b. The function $x \mapsto e^x$ is its own derivative. This implies that e^x and the integral of e^x are the same series, except for the constant term, which is $e^0 = 1$. Accordingly, we can generate the series for e^x as

```
(define exp-series
  (cons-stream 1 (integrate-series exp-series)))
```

Show how to generate the series for sine and cosine, starting from the facts that the derivative of sine is cosine and the derivative of cosine is the negative of sine:

```
(define cosine-series (cons-stream 1 ⟨??⟩))
(define sine-series (cons-stream 0 ⟨??⟩))
```

Exercise 3.60: With power series represented as streams of coefficients as in Exercise 3.59, adding series is implemented by `add-streams`. Complete the definition of the following procedure for multiplying series:

```
(define (mul-series s1 s2)
  (cons-stream ⟨??⟩ (add-streams ⟨??⟩ ⟨??⟩)))
```

You can test your procedure by verifying that $\sin^2 x + \cos^2 x = 1$, using the series from Exercise 3.59.

Exercise 3.61: Let S be a power series (Exercise 3.59) whose constant term is 1. Suppose we want to find the power series $1/S$, that is, the series X such that $SX = 1$. Write $S = 1 + S_R$ where S_R is the part of S after the constant term. Then we can solve for X as follows:

$$\begin{aligned} S \cdot X &= 1, \\ (1 + S_R) \cdot X &= 1, \\ X + S_R \cdot X &= 1, \\ X &= 1 - S_R \cdot X. \end{aligned}$$

In other words, X is the power series whose constant term is 1 and whose higher-order terms are given by the negative of S_R times X. Use this idea to write a procedure `invert-unit-series` that computes $1/S$ for a power series S with constant term 1. You will need to use `mul-series` from Exercise 3.60.

Exercise 3.62: Use the results of Exercise 3.60 and Exercise 3.61 to define a procedure `div-series` that divides two power series. `div-series` should work for any two series,

provided that the denominator series begins with a nonzero constant term. (If the denominator has a zero constant term, then `div-series` should signal an error.) Show how to use `div-series` together with the result of Exercise 3.59 to generate the power series for tangent.

3.5.3 Exploiting the Stream Paradigm

Streams with delayed evaluation can be a powerful modeling tool, providing many of the benefits of local state and assignment. Moreover, they avoid some of the theoretical tangles that accompany the introduction of assignment into a programming language.

The stream approach can be illuminating because it allows us to build systems with different module boundaries than systems organized around assignment to state variables. For example, we can think of an entire time series (or signal) as a focus of interest, rather than the values of the state variables at individual moments. This makes it convenient to combine and compare components of state from different moments.

Formulating iterations as stream processes

In Section 1.2.1, we introduced iterative processes, which proceed by updating state variables. We know now that we can represent state as a "timeless" stream of values rather than as a set of variables to be updated. Let's adopt this perspective in revisiting the square-root procedure from Section 1.1.7. Recall that the idea is to generate a sequence of better and better guesses for the square root of x by applying over and over again the procedure that improves guesses:

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

In our original `sqrt` procedure, we made these guesses be the successive values of a state variable. Instead we can generate the infinite stream of guesses, starting with an initial guess of 1:[65]

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream
     1.0
     (stream-map (lambda (guess) (sqrt-improve guess x))
                 guesses)))
  guesses)

(display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
...
```

We can generate more and more terms of the stream to get better and better guesses. If we like, we can write a procedure that keeps generating terms until the answer is good enough. (See Exercise 3.64.)

Another iteration that we can treat in the same way is to generate an approximation to π, based upon the alternating series that we saw in Section 1.3.1:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots .$$

We first generate the stream of summands of the series (the reciprocals of the odd integers, with alternating signs). Then we take the stream of

[65] We can't use `let` to bind the local variable `guesses`, because the value of `guesses` depends on `guesses` itself. Exercise 3.63 addresses why we want a local variable here.

sums of more and more terms (using the `partial-sums` procedure of Exercise 3.55) and scale the result by 4:

```
(define (pi-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (pi-summands (+ n 2)))))
(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

(display-stream pi-stream)
4.
2.666666666666667
3.466666666666667
2.8952380952380956
3.3396825396825403
2.9760461760461765
3.2837384837384844
3.017071817071818
...
```

This gives us a stream of better and better approximations to π, although the approximations converge rather slowly. Eight terms of the sequence bound the value of π between 3.284 and 3.017.

So far, our use of the stream of states approach is not much different from updating state variables. But streams give us an opportunity to do some interesting tricks. For example, we can transform a stream with a *sequence accelerator* that converts a sequence of approximations to a new sequence that converges to the same value as the original, only faster.

One such accelerator, due to the eighteenth-century Swiss mathematician Leonhard Euler, works well with sequences that are partial sums of alternating series (series of terms with alternating signs). In Euler's technique, if S_n is the n^{th} term of the original sum sequence, then

the accelerated sequence has terms

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}.$$

Thus, if the original sequence is represented as a stream of values, the transformed sequence is given by

```
(define (euler-transform s)
  (let ((s0 (stream-ref s 0))      ; S_{n-1}
        (s1 (stream-ref s 1))      ; S_n
        (s2 (stream-ref s 2)))     ; S_{n+1}
    (cons-stream (- s2 (/ (square (- s2 s1))
                          (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

We can demonstrate Euler acceleration with our sequence of approximations to π:

```
(display-stream (euler-transform pi-stream))
3.166666666666667
3.1333333333333337
3.1452380952380956
3.13968253968254
3.1427128427128435
3.1408813408813416
3.142071817071818
3.1412548236077655
...
```

Even better, we can accelerate the accelerated sequence, and recursively accelerate that, and so on. Namely, we create a stream of streams (a structure we'll call a *tableau*) in which each stream is the transform of the preceding one:

```
(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))
```

The tableau has the form

$$\begin{array}{ccccccc} s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \ldots \\ & s_{10} & s_{11} & s_{12} & s_{13} & \ldots \\ & & s_{20} & s_{21} & s_{22} & \ldots \\ & & & \ldots & & \end{array}$$

Finally, we form a sequence by taking the first term in each row of the tableau:

```
(define (accelerated-sequence transform s)
  (stream-map stream-car (make-tableau transform s)))
```

We can demonstrate this kind of "super-acceleration" of the π sequence:

```
(display-stream
 (accelerated-sequence euler-transform pi-stream))
4.
3.166666666666667
3.142105263157895
3.141599357319005
3.1415927140337785
3.1415926539752927
3.1415926535911765
3.141592653589778
...
```

The result is impressive. Taking eight terms of the sequence yields the correct value of π to 14 decimal places. If we had used only the original π sequence, we would need to compute on the order of 10^{13} terms (i.e., expanding the series far enough so that the individual terms are less than 10^{-13}) to get that much accuracy!

We could have implemented these acceleration techniques without using streams. But the stream formulation is particularly elegant and convenient because the entire sequence of states is available to us as a data structure that can be manipulated with a uniform set of operations.

Exercise 3.63: Louis Reasoner asks why the `sqrt-stream` procedure was not written in the following more straightforward way, without the local variable `guesses`:

```
(define (sqrt-stream x)
  (cons-stream 1.0 (stream-map
                    (lambda (guess)
                      (sqrt-improve guess x))
                    (sqrt-stream x))))
```

Alyssa P. Hacker replies that this version of the procedure is considerably less efficient because it performs redundant computation. Explain Alyssa's answer. Would the two versions still differ in efficiency if our implementation of `delay` used only `(lambda () ⟨exp⟩)` without using the optimization provided by `memo-proc` (Section 3.5.1)?

Exercise 3.64: Write a procedure `stream-limit` that takes as arguments a stream and a number (the tolerance). It should examine the stream until it finds two successive elements that differ in absolute value by less than the tolerance, and return the second of the two elements. Using this, we could compute square roots up to a given tolerance by

```
(define (sqrt x tolerance)
  (stream-limit (sqrt-stream x) tolerance))
```

Exercise 3.65: Use the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for π. How rapidly do these sequences converge?

Infinite streams of pairs

In Section 2.2.3, we saw how the sequence paradigm handles traditional nested loops as processes defined on sequences of pairs. If we generalize this technique to infinite streams, then we can write programs that are not easily represented as loops, because the "looping" must range over an infinite set.

For example, suppose we want to generalize the `prime-sum-pairs` procedure of Section 2.2.3 to produce the stream of pairs of *all* integers (i, j) with $i \le j$ such that $i + j$ is prime. If `int-pairs` is the sequence of all pairs of integers (i, j) with $i \le j$, then our required stream is simply[66]

```
(stream-filter
 (lambda (pair) (prime? (+ (car pair) (cadr pair))))
 int-pairs)
```

Our problem, then, is to produce the stream `int-pairs`. More generally, suppose we have two streams $S = (S_i)$ and $T = (T_j)$, and imagine the infinite rectangular array

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ (S_1, T_0) & (S_1, T_1) & (S_1, T_2) & \ldots \\ (S_2, T_0) & (S_2, T_1) & (S_2, T_2) & \ldots \\ \ldots & & & \end{array}$$

[66] As in Section 2.2.3, we represent a pair of integers as a list rather than a Lisp pair.

We wish to generate a stream that contains all the pairs in the array that lie on or above the diagonal, i.e., the pairs

$$\begin{array}{cccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ & (S_1, T_1) & (S_1, T_2) & \ldots \\ & & (S_2, T_2) & \ldots \\ & & & \ldots \end{array}$$

(If we take both S and T to be the stream of integers, then this will be our desired stream `int-pairs`.)

Call the general stream of pairs `(pairs S T)`, and consider it to be composed of three parts: the pair (S_0, T_0), the rest of the pairs in the first row, and the remaining pairs:[67]

$$\begin{array}{c|ccc} (S_0, T_0) & (S_0, T_1) & (S_0, T_2) & \ldots \\ \hline & (S_1, T_1) & (S_1, T_2) & \ldots \\ & & (S_2, T_2) & \ldots \\ & & & \ldots \end{array}$$

Observe that the third piece in this decomposition (pairs that are not in the first row) is (recursively) the pairs formed from `(stream-cdr S)` and `(stream-cdr T)`. Also note that the second piece (the rest of the first row) is

```
(stream-map (lambda (x) (list (stream-car s) x))
            (stream-cdr t))
```

Thus we can form our stream of pairs as follows:

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
```

[67] See Exercise 3.68 for some insight into why we chose this decomposition.

```
  (⟨combine-in-some-way⟩
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

In order to complete the procedure, we must choose some way to combine the two inner streams. One idea is to use the stream analog of the append procedure from Section 2.2.1:

```
(define (stream-append s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (stream-append (stream-cdr s1) s2))))
```

This is unsuitable for infinite streams, however, because it takes all the elements from the first stream before incorporating the second stream. In particular, if we try to generate all pairs of positive integers using

```
(pairs integers integers)
```

our stream of results will first try to run through all pairs with the first integer equal to 1, and hence will never produce pairs with any other value of the first integer.

To handle infinite streams, we need to devise an order of combination that ensures that every element will eventually be reached if we let our program run long enough. An elegant way to accomplish this is with the following `interleave` procedure:[68]

[68] The precise statement of the required property on the order of combination is as follows: There should be a function f of two arguments such that the pair corresponding to element i of the first stream and element j of the second stream will appear as element number $f(i, j)$ of the output stream. The trick of using `interleave` to accomplish this was shown to us by David Turner, who employed it in the language KRC (Turner 1981).

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

Since `interleave` takes elements alternately from the two streams, every element of the second stream will eventually find its way into the interleaved stream, even if the first stream is infinite.

We can thus generate the required stream of pairs as

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

Exercise 3.66: Examine the stream `(pairs integers integers)`. Can you make any general comments about the order in which the pairs are placed into the stream? For example, approximately how many pairs precede the pair (1, 100)? the pair (99, 100)? the pair (100, 100)? (If you can make precise mathematical statements here, all the better. But feel free to give more qualitative answers if you find yourself getting bogged down.)

Exercise 3.67: Modify the `pairs` procedure so that `(pairs integers integers)` will produce the stream of *all* pairs of integers (i, j) (without the condition $i \leq j$). Hint: You will need to mix in an additional stream.

Exercise 3.68: Louis Reasoner thinks that building a stream of pairs from three parts is unnecessarily complicated. Instead of separating the pair (S_0, T_0) from the rest of the pairs in the first row, he proposes to work with the whole first row, as follows:

```
(define (pairs s t)
  (interleave
   (stream-map (lambda (x) (list (stream-car s) x))
               t)
   (pairs (stream-cdr s) (stream-cdr t))))
```

Does this work? Consider what happens if we evaluate `(pairs integers integers)` using Louis's definition of `pairs`.

Exercise 3.69: Write a procedure `triples` that takes three infinite streams, S, T, and U, and produces the stream of triples (S_i, T_j, U_k) such that $i \leq j \leq k$. Use `triples` to generate the stream of all Pythagorean triples of positive integers, i.e., the triples (i, j, k) such that $i \leq j$ and $i^2 + j^2 = k^2$.

Exercise 3.70: It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. We can use a technique similar to the `merge` procedure of Exercise 3.56, if we define a way to say that one pair of integers is "less than" another. One way to do this is to define a "weighting function" $W(i, j)$ and stipulate that (i_1, j_1) is less than (i_2, j_2) if $W(i_1, j_1) < W(i_2, j_2)$. Write a procedure `merge-weighted` that is like `merge`, except that `merge-weighted` takes an additional argument `weight`, which is a procedure that computes the weight of a pair, and is used

to determine the order in which elements should appear in the resulting merged stream.[69] Using this, generalize `pairs` to a procedure `weighted-pairs` that takes two streams, together with a procedure that computes a weighting function, and generates the stream of pairs, ordered according to weight. Use your procedure to generate

a. the stream of all pairs of positive integers (i, j) with $i \leq j$ ordered according to the sum $i + j$,
b. the stream of all pairs of positive integers (i, j) with $i \leq j$, where neither i nor j is divisible by 2, 3, or 5, and the pairs are ordered according to the sum $2i + 3j + 5ij$.

Exercise 3.71: Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.[70] Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$

[69] We will require that the weighting function be such that the weight of a pair increases as we move out along a row or down along a column of the array of pairs.

[70] To quote from G. H. Hardy's obituary of Ramanujan (Hardy 1921): "It was Mr. Littlewood (I believe) who remarked that 'every positive integer was one of his friends.' I remember once going to see him when he was lying ill at Putney. I had ridden in taxicab No. 1729, and remarked that the number seemed to me a rather dull one, and that I hoped it was not an unfavorable omen. 'No,' he replied, 'it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.' " The trick of using weighted pairs to generate the Ramanujan numbers was shown to us by Charles Leiserson.

(see Exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a procedure to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

Exercise 3.72: In a similar way to Exercise 3.71 generate a stream of all numbers that can be written as the sum of two squares in three different ways (showing how they can be so written).

Streams as signals

We began our discussion of streams by describing them as computational analogs of the "signals" in signal-processing systems. In fact, we can use streams to model signal-processing systems in a very direct way, representing the values of a signal at successive time intervals as consecutive elements of a stream. For instance, we can implement an *integrator* or *summer* that, for an input stream $x = (x_i)$, an initial value C, and a small increment dt, accumulates the sum

$$S_i = C + \sum_{j=1}^{i} x_j \, dt$$

and returns the stream of values $S = (S_i)$. The following `integral` procedure is reminiscent of the "implicit style" definition of the stream of integers (Section 3.5.2):

```
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int)
```

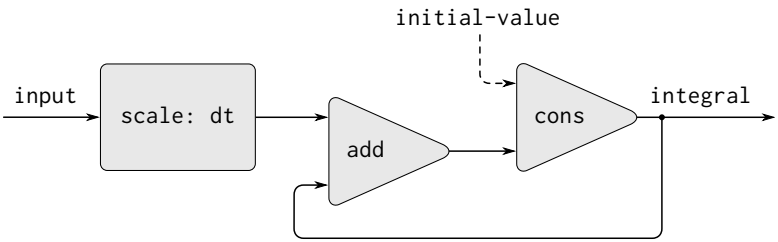

Figure 3.32: The `integral` procedure viewed as a signal-processing system.

Figure 3.32 is a picture of a signal-processing system that corresponds to the `integral` procedure. The input stream is scaled by *dt* and passed through an adder, whose output is passed back through the same adder. The self-reference in the definition of `int` is reflected in the figure by the feedback loop that connects the output of the adder to one of the inputs.

> **Exercise 3.73:** We can model electrical circuits using streams to represent the values of currents or voltages at a sequence of times. For instance, suppose we have an *RC circuit* consisting of a resistor of resistance R and a capacitor of capacitance C in series. The voltage response v of the circuit to an injected current i is determined by the formula in Figure 3.33, whose structure is shown by the accompanying signal-flow diagram.
>
> Write a procedure `RC` that models this circuit. `RC` should take as inputs the values of R, C, and dt and should return a procedure that takes as inputs a stream representing the current i and an initial value for the capacitor voltage v_0

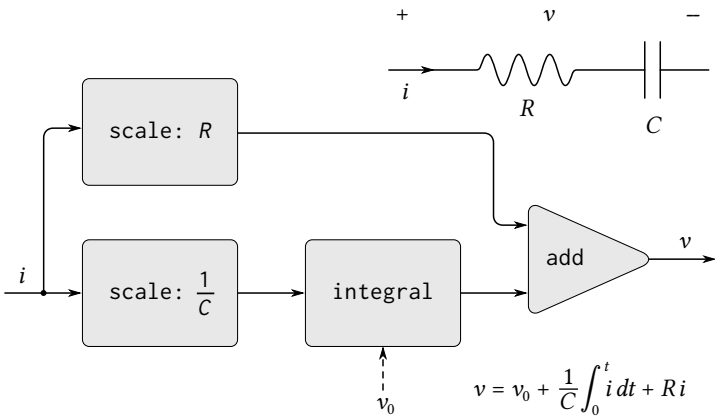

Figure 3.33: An RC circuit and the associated signal-flow diagram.

and produces as output the stream of voltages v. For example, you should be able to use `RC` to model an RC circuit with $R = 5$ ohms, $C = 1$ farad, and a 0.5-second time step by evaluating `(define RC1 (RC 5 1 0.5))`. This defines `RC1` as a procedure that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages.

Exercise 3.74: Alyssa P. Hacker is designing a system to process signals coming from physical sensors. One important feature she wishes to produce is a signal that describes the *zero crossings* of the input signal. That is, the resulting signal should be +1 whenever the input signal changes from negative to positive, −1 whenever the input signal changes from positive to negative, and 0 otherwise. (Assume that the sign of a 0 input is positive.) For example, a typical in-

put signal with its associated zero-crossing signal would be

```
... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...
... 0 0  0  0  0   -1   0  0  0   0   1  0 0 ...
```

In Alyssa's system, the signal from the sensor is represented as a stream `sense-data` and the stream `zero-crossings` is the corresponding stream of zero crossings. Alyssa first writes a procedure `sign-change-detector` that takes two values as arguments and compares the signs of the values to produce an appropriate 0, 1, or - 1. She then constructs her zero-crossing stream as follows:

```
(define (make-zero-crossings input-stream last-value)
  (cons-stream
   (sign-change-detector
    (stream-car input-stream)
    last-value)
   (make-zero-crossings
    (stream-cdr input-stream)
    (stream-car input-stream))))
(define zero-crossings
  (make-zero-crossings sense-data 0))
```

Alyssa's boss, Eva Lu Ator, walks by and suggests that this program is approximately equivalent to the following one, which uses the generalized version of `stream-map` from Exercise 3.50:

```
(define zero-crossings
  (stream-map sign-change-detector
              sense-data
              ⟨expression⟩))
```

Complete the program by supplying the indicated ⟨*expression*⟩.

Exercise 3.75: Unfortunately, Alyssa's zero-crossing detector in Exercise 3.74 proves to be insufficient, because the noisy signal from the sensor leads to spurious zero crossings. Lem E. Tweakit, a hardware specialist, suggests that Alyssa smooth the signal to filter out the noise before extracting the zero crossings. Alyssa takes his advice and decides to extract the zero crossings from the signal constructed by averaging each value of the sense data with the previous value. She explains the problem to her assistant, Louis Reasoner, who attempts to implement the idea, altering Alyssa's program as follows:

```
(define (make-zero-crossings input-stream last-value)
  (let ((avpt (/ (+ (stream-car input-stream)
                    last-value)
                 2)))
    (cons-stream
     (sign-change-detector avpt last-value)
     (make-zero-crossings
      (stream-cdr input-stream) avpt))))
```

This does not correctly implement Alyssa's plan. Find the bug that Louis has installed and fix it without changing the structure of the program. (Hint: You will need to increase the number of arguments to `make-zero-crossings`.)

Exercise 3.76: Eva Lu Ator has a criticism of Louis's approach in Exercise 3.75. The program he wrote is not modular, because it intermixes the operation of smoothing with the zero-crossing extraction. For example, the extractor should

not have to be changed if Alyssa finds a better way to condition her input signal. Help Louis by writing a procedure `smooth` that takes a stream as input and produces a stream in which each element is the average of two successive input stream elements. Then use `smooth` as a component to implement the zero-crossing detector in a more modular style.

3.5.4 Streams and Delayed Evaluation

The `integral` procedure at the end of the preceding section shows how we can use streams to model signal-processing systems that contain feedback loops. The feedback loop for the adder shown in Figure 3.32 is modeled by the fact that `integral`'s internal stream `int` is defined in terms of itself:

```
(define int
  (cons-stream
   initial-value
   (add-streams (scale-stream integrand dt)
                int)))
```

The interpreter's ability to deal with such an implicit definition depends on the `delay` that is incorporated into `cons-stream`. Without this `delay`, the interpreter could not construct `int` before evaluating both arguments to `cons-stream`, which would require that `int` already be defined. In general, `delay` is crucial for using streams to model signal-processing systems that contain loops. Without `delay`, our models would have to be formulated so that the inputs to any signal-processing component would be fully evaluated before the output could be produced. This would outlaw loops.

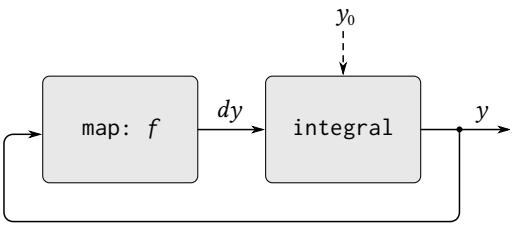

Figure 3.34: An "analog computer circuit" that solves the equation $dy/dt = f(y)$.

Unfortunately, stream models of systems with loops may require uses of `delay` beyond the "hidden" `delay` supplied by `cons-stream`. For instance, Figure 3.34 shows a signal-processing system for solving the differential equation $dy/dt = f(y)$ where f is a given function. The figure shows a mapping component, which applies f to its input signal, linked in a feedback loop to an integrator in a manner very similar to that of the analog computer circuits that are actually used to solve such equations.

Assuming we are given an initial value y_0 for y, we could try to model this system using the procedure

```
(define (solve f y0 dt)
  (define y (integral dy y0 dt))
  (define dy (stream-map f y))
  y)
```

This procedure does not work, because in the first line of `solve` the call to `integral` requires that the input `dy` be defined, which does not happen until the second line of `solve`.

On the other hand, the intent of our definition does make sense, because we can, in principle, begin to generate the `y` stream without

knowing dy. Indeed, integral and many other stream operations have properties similar to those of cons-stream, in that we can generate part of the answer given only partial information about the arguments. For integral, the first element of the output stream is the specified initial-value. Thus, we can generate the first element of the output stream without evaluating the integrand dy. Once we know the first element of y, the stream-map in the second line of solve can begin working to generate the first element of dy, which will produce the next element of y, and so on.

To take advantage of this idea, we will redefine integral to expect the integrand stream to be a *delayed argument*. integral will force the integrand to be evaluated only when it is required to generate more than the first element of the output stream:

```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream
     initial-value
     (let ((integrand (force delayed-integrand)))
       (add-streams (scale-stream integrand dt) int))))
  int)
```

Now we can implement our solve procedure by delaying the evaluation of dy in the definition of y:[71]

```
(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)
```

[71] This procedure is not guaranteed to work in all Scheme implementations, although for any implementation there is a simple variation that will work. The problem has to do with subtle differences in the ways that Scheme implementations handle internal definitions. (See Section 4.1.6.)

In general, every caller of `integral` must now `delay` the integrand argument. We can demonstrate that the `solve` procedure works by approximating $e \approx 2.718$ by computing the value at $y = 1$ of the solution to the differential equation $dy/dt = y$ with initial condition $y(0) = 1$:

```
(stream-ref (solve (lambda (y) y)
                   1
                   0.001)
            1000)
2.716924
```

Exercise 3.77: The `integral` procedure used above was analogous to the "implicit" definition of the infinite stream of integers in Section 3.5.2. Alternatively, we can give a definition of `integral` that is more like `integers-starting-from` (also in Section 3.5.2):

```
(define (integral integrand initial-value dt)
  (cons-stream
   initial-value
   (if (stream-null? integrand)
       the-empty-stream
       (integral (stream-cdr integrand)
                 (+ (* dt (stream-car integrand))
                    initial-value)
                 dt))))
```

When used in systems with loops, this procedure has the same problem as does our original version of `integral`. Modify the procedure so that it expects the `integrand` as a delayed argument and hence can be used in the `solve` procedure shown above.

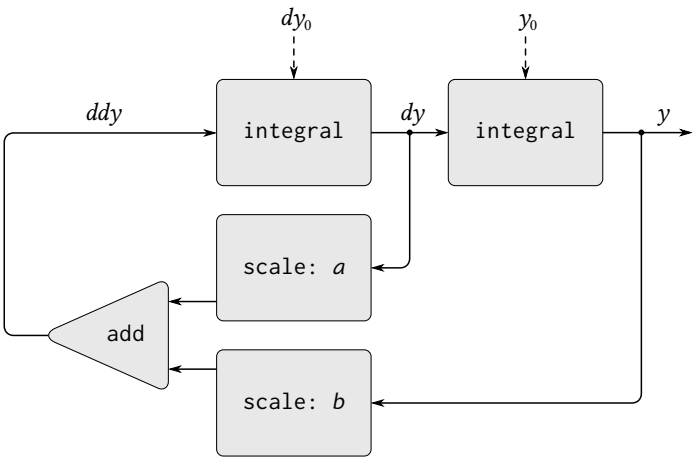

Figure 3.35: Signal-flow diagram for the solution to a second-order linear differential equation.

Exercise 3.78: Consider the problem of designing a signal-processing system to study the homogeneous second-order linear differential equation

$$\frac{d^2y}{dt^2} - a\frac{dy}{dt} - by = 0.$$

The output stream, modeling y, is generated by a network that contains a loop. This is because the value of d^2y/dt^2 depends upon the values of y and dy/dt and both of these are determined by integrating d^2y/dt^2. The diagram we would like to encode is shown in Figure 3.35. Write a procedure `solve-2nd` that takes as arguments the constants a, b, and dt and the initial values y_0 and dy_0 for y and dy/dt and gen-

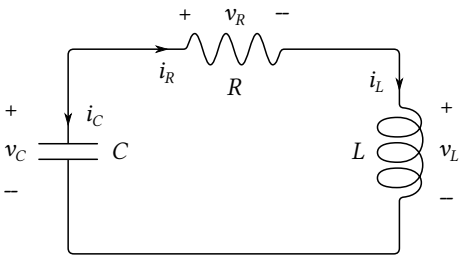

Figure 3.36: A series RLC circuit.

erates the stream of successive values of y.

Exercise 3.79: Generalize the `solve-2nd` procedure of Exercise 3.78 so that it can be used to solve general second-order differential equations $d^2y/dt^2 = f(dy/dt, y)$.

Exercise 3.80: A *series RLC circuit* consists of a resistor, a capacitor, and an inductor connected in series, as shown in Figure 3.36. If R, L, and C are the resistance, inductance, and capacitance, then the relations between voltage (v) and current (i) for the three components are described by the equations

$$v_R = i_R R, \qquad v_L = L\frac{di_L}{dt}, \qquad i_C = C\frac{dv_C}{dt},$$

and the circuit connections dictate the relations

$$i_R = i_L = -i_C, \qquad v_C = v_L + v_R.$$

Combining these equations shows that the state of the circuit (summarized by v_C, the voltage across the capacitor,

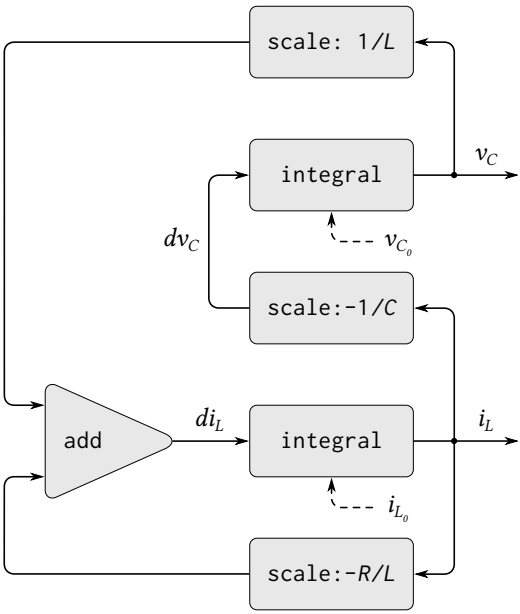

Figure 3.37: A signal-flow diagram for the solution to a series RLC circuit.

and i_L, the current in the inductor) is described by the pair of differential equations

$$\frac{dv_C}{dt} = -\frac{i_L}{C}\,, \qquad \frac{di_L}{dt} = \frac{1}{L}v_C - \frac{R}{L}i_L\,.$$

The signal-flow diagram representing this system of differential equations is shown in Figure 3.37.

Write a procedure `RLC` that takes as arguments the parameters R, L, and C of the circuit and the time increment dt.

In a manner similar to that of the `RC` procedure of Exercise 3.73, `RLC` should produce a procedure that takes the initial values of the state variables, v_{C_0} and i_{L_0}, and produces a pair (using `cons`) of the streams of states v_C and i_L. Using `RLC`, generate the pair of streams that models the behavior of a series RLC circuit with $R = 1$ ohm, $C = 0.2$ farad, $L = 1$ henry, $dt = 0.1$ second, and initial values $i_{L_0} = 0$ amps and $v_{C_0} = 10$ volts.

Normal-order evaluation

The examples in this section illustrate how the explicit use of `delay` and `force` provides great programming flexibility, but the same examples also show how this can make our programs more complex. Our new `integral` procedure, for instance, gives us the power to model systems with loops, but we must now remember that `integral` should be called with a delayed integrand, and every procedure that uses `integral` must be aware of this. In effect, we have created two classes of procedures: ordinary procedures and procedures that take delayed arguments. In general, creating separate classes of procedures forces us to create separate classes of higher-order procedures as well.[72]

[72] This is a small reflection, in Lisp, of the difficulties that conventional strongly typed languages such as Pascal have in coping with higher-order procedures. In such languages, the programmer must specify the data types of the arguments and the result of each procedure: number, logical value, sequence, and so on. Consequently, we could not express an abstraction such as "map a given procedure `proc` over all the elements in a sequence" by a single higher-order procedure such as `stream-map`. Rather, we would need a different mapping procedure for each different combination of argument and result data types that might be specified for a `proc`. Maintaining a practical notion of "data type" in the presence of higher-order procedures raises many difficult issues. One way of dealing with this problem is illustrated by the language ML (Gordon et al. 1979),

One way to avoid the need for two different classes of procedures is to make all procedures take delayed arguments. We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed (for example, when they are required by a primitive operation). This would transform our language to use normal-order evaluation, which we first described when we introduced the substitution model for evaluation in Section 1.1.5. Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation, and this would be a natural strategy to adopt if we were concerned only with stream processing. In Section 4.2, after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including delays in procedure calls wreaks havoc with our ability to design programs that depend on the order of events, such as programs that use assignment, mutate data, or perform input or output. Even the single `delay` in `cons-stream` can cause great confusion, as illustrated by Exercise 3.51 and Exercise 3.52. As far as anyone knows, mutability and delayed evaluation do not mix well in programming languages, and devising ways to deal with both of these at once is an active area of research.

---

whose "polymorphic data types" include templates for higher-order transformations between data types. Moreover, data types for most procedures in ML are never explicitly declared by the programmer. Instead, ML includes a *type-inferencing* mechanism that uses information in the environment to deduce the data types for newly defined procedures.

3.5.5 Modularity of Functional Programs and Modularity of Objects

As we saw in Section 3.1.2, one of the major benefits of introducing assignment is that we can increase the modularity of our systems by encapsulating, or "hiding," parts of the state of a large system within local variables. Stream models can provide an equivalent modularity without the use of assignment. As an illustration, we can reimplement the Monte Carlo estimation of π, which we examined in Section 3.1.2, from a stream-processing point of view.

The key modularity issue was that we wished to hide the internal state of a random-number generator from programs that used random numbers. We began with a procedure `rand-update`, whose successive values furnished our supply of random numbers, and used this to produce a random-number generator:

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

In the stream formulation there is no random-number generator *per se*, just a stream of random numbers produced by successive calls to `rand-update`:

```
(define random-numbers
  (cons-stream
   random-init
   (stream-map rand-update random-numbers)))
```

We use this to construct the stream of outcomes of the Cesàro experiment performed on consecutive pairs in the `random-numbers` stream:

```
(define cesaro-stream
  (map-successive-pairs
   (lambda (r1 r2) (= (gcd r1 r2) 1))
   random-numbers))
(define (map-successive-pairs f s)
  (cons-stream
   (f (stream-car s) (stream-car (stream-cdr s)))
   (map-successive-pairs f (stream-cdr (stream-cdr s)))))
```

The `cesaro-stream` is now fed to a `monte-carlo` procedure, which produces a stream of estimates of probabilities. The results are then converted into a stream of estimates of π. This version of the program doesn't need a parameter telling how many trials to perform. Better estimates of π (from performing more experiments) are obtained by looking farther into the `pi` stream:

```
(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
     (/ passed (+ passed failed))
     (monte-carlo
      (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1))))
(define pi
  (stream-map
   (lambda (p) (sqrt (/ 6 p)))
   (monte-carlo cesaro-stream 0 0)))
```

There is considerable modularity in this approach, because we still can formulate a general `monte-carlo` procedure that can deal with arbitrary experiments. Yet there is no assignment or local state.

Exercise 3.81: Exercise 3.6 discussed generalizing the random-number generator to allow one to reset the random-number sequence so as to produce repeatable sequences of "random" numbers. Produce a stream formulation of this same generator that operates on an input stream of requests to `generate` a new random number or to `reset` the sequence to a specified value and that produces the desired stream of random numbers. Don't use assignment in your solution.

Exercise 3.82: Redo Exercise 3.5 on Monte Carlo integration in terms of streams. The stream version of `estimate-integral` will not have an argument telling how many trials to perform. Instead, it will produce a stream of estimates based on successively more trials.

A functional-programming view of time

Let us now return to the issues of objects and state that were raised at the beginning of this chapter and examine them in a new light. We introduced assignment and mutable objects to provide a mechanism for modular construction of programs that model systems with state. We constructed computational objects with local state variables and used assignment to modify these variables. We modeled the temporal behavior of the objects in the world by the temporal behavior of the corresponding computational objects.

Now we have seen that streams provide an alternative way to model objects with local state. We can model a changing quantity, such as the local state of some object, using a stream that represents the time history of successive states. In essence, we represent time explicitly, using streams, so that we decouple time in our simulated world from the se-

quence of events that take place during evaluation. Indeed, because of the presence of `delay` there may be little relation between simulated time in the model and the order of events during the evaluation.

In order to contrast these two approaches to modeling, let us reconsider the implementation of a "withdrawal processor" that monitors the balance in a bank account. In Section 3.1.3 we implemented a simplified version of such a processor:

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

Calls to `make-simplified-withdraw` produce computational objects, each with a local state variable `balance` that is decremented by successive calls to the object. The object takes an `amount` as an argument and returns the new balance. We can imagine the user of a bank account typing a sequence of inputs to such an object and observing the sequence of returned values shown on a display screen.

Alternatively, we can model a withdrawal processor as a procedure that takes as input a balance and a stream of amounts to withdraw and produces the stream of successive balances in the account:

```
(define (stream-withdraw balance amount-stream)
  (cons-stream
   balance
   (stream-withdraw (- balance (stream-car amount-stream))
                    (stream-cdr amount-stream))))
```

`stream-withdraw` implements a well-defined mathematical function whose output is fully determined by its input. Suppose, however, that the input `amount-stream` is the stream of successive values typed by the user and that the resulting stream of balances is displayed. Then, from the

perspective of the user who is typing values and watching results, the stream process has the same behavior as the object created by `make-simplified-withdraw`. However, with the stream version, there is no assignment, no local state variable, and consequently none of the theoretical difficulties that we encountered in Section 3.1.3. Yet the system has state!

This is really remarkable. Even though `stream-withdraw` implements a well-defined mathematical function whose behavior does not change, the user's perception here is one of interacting with a system that has a changing state. One way to resolve this paradox is to realize that it is the user's temporal existence that imposes state on the system. If the user could step back from the interaction and think in terms of streams of balances rather than individual transactions, the system would appear stateless.[73]

From the point of view of one part of a complex process, the other parts appear to change with time. They have hidden time-varying local state. If we wish to write programs that model this kind of natural decomposition in our world (as we see it from our viewpoint as a part of that world) with structures in our computer, we make computational objects that are not functional—they must change with time. We model state with local state variables, and we model the changes of state with assignments to those variables. By doing this we make the time of execution of a computation model time in the world that we are part of, and thus we get "objects" in our computer.

Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are

[73] Similarly in physics, when we observe a moving particle, we say that the position (state) of the particle is changing. However, from the perspective of the particle's world line in space-time there is no change involved.

part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of *functional programming languages*, which do not include any provision for assignment or mutable data. In such a language, all procedures implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems.[74]

On the other hand, if we look closely, we can see time-related problems creeping into functional models as well. One particularly troublesome area arises when we wish to design interactive systems, especially ones that model interactions between independent entities. For instance, consider once more the implementation a banking system that permits joint bank accounts. In a conventional system using assignment and objects, we would model the fact that Peter and Paul share an account by having both Peter and Paul send their transaction requests to the same bank-account object, as we saw in Section 3.1.3. From the stream point of view, where there are no "objects" *per se*, we have already indicated that a bank account can be modeled as a process that operates on a stream of transaction requests to produce a stream of responses. Accordingly, we could model the fact that Peter and Paul have a joint bank account by merging Peter's stream of transaction requests with Paul's stream of requests and feeding the result to the bank-account stream process, as shown in Figure 3.38.

[74]John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington et al. 1982.

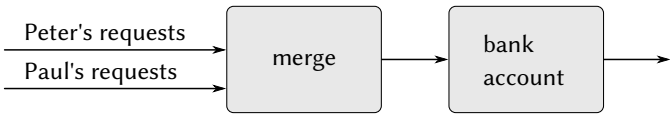

Figure 3.38: A joint bank account, modeled by merging two streams of transaction requests.

The trouble with this formulation is in the notion of *merge*. It will not do to merge the two streams by simply taking alternately one request from Peter and one request from Paul. Suppose Paul accesses the account only very rarely. We could hardly force Peter to wait for Paul to access the account before he could issue a second transaction. However such a merge is implemented, it must interleave the two transaction streams in some way that is constrained by "real time" as perceived by Peter and Paul, in the sense that, if Peter and Paul meet, they can agree that certain transactions were processed before the meeting, and other transactions were processed after the meeting.[75] This is precisely the same constraint that we had to deal with in Section 3.4.1, where we found the need to introduce explicit synchronization to ensure a "correct" order of events in concurrent processing of objects with state. Thus, in an attempt to support the functional style, the need to merge inputs from different agents reintroduces the same problems that the functional style was meant to eliminate.

[75] Observe that, for any two streams, there is in general more than one acceptable order of interleaving. Thus, technically, "merge" is a relation rather than a function—the answer is not a deterministic function of the inputs. We already mentioned (Footnote 39) that nondeterminism is essential when dealing with concurrency. The merge relation illustrates the same essential nondeterminism, from the functional perspective. In Section 4.3, we will look at nondeterminism from yet another point of view.

We began this chapter with the goal of building computational models whose structure matches our perception of the real world we are trying to model. We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is completely satisfactory. A grand unification has yet to emerge.[76]

[76]The object model approximates the world by dividing it into separate pieces. The functional model does not modularize along object boundaries. The object model is useful when the unshared state of the "objects" is much larger than the state that they share. An example of a place where the object viewpoint fails is quantum mechanics, where thinking of things as individual particles leads to paradoxes and confusions. Unifying the object view with the functional view may have little to do with programming, but rather with fundamental epistemological issues.