[Go to first, previous, next page; contents; index]

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 We will do this using a procedure `withdraw`, which takes as argument an `amount` to be withdrawn.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>)
```

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 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 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
(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 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
           (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 returns 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 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 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 expresssion 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]

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 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))
```

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.

---

[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.

[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.

[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.''

[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.)

[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.

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

[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.

[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.

[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 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, Schmidt, and Wadler 1980).

[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.

[Go to first, previous, next page; contents; index]