12 Object-Oriented Programming

12.1 Overview

A different perspective on computing is provided by object-oriented programming. In this style of programming, certain objects are defined that respond to messages passed to them. Figuratively, we can think of an object as a computer dedicated to solving a particular type of problem. The input is the message passed to the object, the object does the computation, and the output is the value returned by the object. In this chapter, we see how such objects are defined, and we illustrate the use of objects to define such data structures as stacks and queues.

12.2 Boxes, Counters, Accumulators, and Gauges

In Chapters 3 and 5, the concept of data abstraction was discussed and illustrated. We saw there that we can write programs that are independent of the representation of the data and are based on certain predefined basic procedures, including the constructors and selectors used on the data type. The actual representation of the data was then used only in defining these basic procedures. We develop the idea of data abstraction further by defining certain objects that are combined with certain operations. It is not necessary for users to know how these objects and operators are implemented in order to use them. They only have to know the interface. An example of such an object is a stack that has associated with it such operations as `push!` and `pop!`. This offers a degree of security in the handling of data and makes it possible to change the internal representation of the object without the user's

being aware of any changes. Before looking at stacks and queues, we introduce the case expression, which makes it easier for us to define the various objects we shall study.

12.2.1 The Case Expression

Scheme provides a special form with keyword **case** that selects one of a sequence of clauses to evaluate based upon the value of an argument (or message) that it is passed. To see how **case** is used, let us first look at a procedure that tells us whether a letter is a vowel or a consonant. We can define it as

```
(define vowel-or-consonant
  (lambda (letter)
    (cond
      ((or (eq? letter 'a)
           (eq? letter 'e)
           (eq? letter 'i)
           (eq? letter 'o)
           (eq? letter 'u))
       'vowel)
      (else 'consonant))))
```

This procedure can also be defined using the special form **case** as follows:

```
(define vowel-or-consonant
  (lambda (letter)
    (case letter
      ((a e i o u) 'vowel)
      (else 'consonant))))
```

The value of **letter** is matched with each of the items (keys) in the list in the first clause of the case expression. If there is a match, the expression following the list of keys is evaluated and returned as the value of the case expression. Thus if **letter** evaluates to one of **a**, **e**, **i**, **o**, or **u**, **vowel** is returned. Otherwise, the next clause is evaluated, and since in this case it is the else clause, **consonant** is returned. In case **letter** evaluates to one of the five vowels, it is more convenient to use the case expression, which matches it with the possible key values rather than the cond expression, which must list a separate test for each possibility.

The syntax of **case** is

```
(case target
  (keys expr1 expr2 ...)
  ...
  (else expr1 expr2 ...))
```

where *target* is an expression that is evaluated and its value is compared with the keys. Each clause begins with *keys*, which is a list of items each of which is matched (using eqv?) with the value of *target* to decide which of the clauses will be selected for evaluation. When the first such match is found, the expressions *expr* ... following the *keys* are evaluated in order and the value of the last is returned (there is an implicit `begin` following each *keys*). If no match is found and the optional else clause is present, then the expressions *expr* ... in the else clause are evaluated. If no else clause is present, then some unspecified value is returned. It is good programming style always to include an else clause even if only for reporting an error.

Below are some additional simple examples demonstrating the use of `case`:

```
[1] (case 'b
      ((a) (display "a was selected: ") (cons 'a '()))
      ((b) (display "b was selected: ") (cons 'b '()))
      ((c) (display "c was selected: ") (cons 'c '()))
      (else (display "None were selected.")))
b was selected: (b)

[2] (case (remainder 35 10)
      ((2 4 6 8) "positive and even")
      ((1 3 5 7 9) "positive and odd")
      ((-2 -4 -6 -8) "negative and even")
      ((-1 -3 -5 -7 -9) "negative and odd")
      (else "zero"))
"positive and odd"
```

In the various objects we shall define in this chapter, we shall use internal representations of the data, which are not supposed to be apparent to the user. In order to secure the data structures used, we introduce, in Program 12.1, the procedure `for-effect-only`, which evaluates its operand to perform the side effects and then returns the string `"unspecified value"`. Following our usual convention, we will not display `"unspecified value"`.

Program 12.1 `for-effect-only`

```
(define for-effect-only
  (lambda (item-ignored)
    "unspecified value"))
```

12.2.2 Boxes

A *box* is a place in which a value can be stored until it is needed later. A new box containing a given initial value is created by the procedure **box-maker**. There are five operations that we shall perform on a box. We use one of these operations to put a value into the box and another to show the value in the box. The operation that puts a value into the box is called **update!** and the operation that shows the value in the box is called **show**. Another useful operation, called **swap!**, puts a new value into the box and returns the old contents of the box. The operation called **reset!** resets the value stored in the box to its initial value. The ability to perform a reset operation is somewhat unusual. The operation **type**, specified for all objects, tells what kind of object is being sent a message. In this case the type is **"box"**. In general, the operations that are performed on an object are called *methods*. In the case of a box, there are five methods: **update!**, **show**, **swap!**, **reset!**, and **type**.

The objects, such as boxes, are themselves procedures. To apply one of the methods to an object, we invoke the object on the (quoted) name of the method followed by any additional arguments appropriate for that method. We then say that we send the name of the method and any additional arguments as a *message* to the object. We can use the call structure:

(*object* '*method-name operand* ...)

where *object* is sent the message consisting of the quoted method name and zero or more operands. On the other hand, we find it more suggestive and, in fact, more flexible to introduce the procedure **send**, which is used to send the message to the object. When we use **send**, we use the call structure

(send *object* '*method-name operand* ...)

The following shows a typical construction of a box **box-a** that is initialized

with (+ 3 4) and a box box-b that is initialized with 5. We shall describe the actual mechanism for constructing boxes after looking at the example.

```
[1] (define box-a (box-maker (+ 3 4)))
[2] (define box-b (box-maker 5))
[3] (send box-a 'show)
7
[4] (send box-b 'show)
5
[5] (send box-a 'update! 3)
[6] (send box-a 'show)
3
[7] (send box-b 'update! (send box-a 'swap! (send box-b 'show)))
[8] (send box-a 'show)
5
[9] (send box-b 'show)
3
[10] (send box-a 'reset!)
[11] (send box-a 'show)
7
[12] (send box-a 'type)
"box"
[13] (send box-a 'update 27)
Error: Bad method name: update sent to object of box type.
```

In [3], in order to see what is stored in box-a, we send the message show to box-a, and in [5], in order to change the value stored in box-a, we send it the message update! and the new value 3. In [13], we forgot to include the exclamation mark on the word update, and an error was signaled. The sending of these quoted method names and arguments as messages to the objects leads to a style of programming referred to as *message-passing style*.

In Program 12.2, we define box-maker. It takes as its argument an initial value stored in the box. It returns a procedure that takes an arbitrary number of arguments and is hence defined using the unrestricted lambda whose parameter list is denoted by msg. Each invocation of box-maker returns an object that we refer to as a box. Thus, in our experiment presented above, box-a and box-b are examples (or *instances*) of boxes. Also, in [3], the message consists of the single item 'show, whereas in [5], the message consists of two items, the method name 'update! and the operand 3. In the code given below for box-maker, we use 1st and 2nd to denote car and cadr,

Program 12.2 box-maker

```
(define box-maker
  (lambda (init-value)
    (let ((contents init-value))
      (lambda msg
        (case (1st msg)
          ((type) "box")
          ((show) contents)
          ((update!) (for-effect-only (set! contents (2nd msg))))
          ((swap!) (let ((ans contents))
                     (set! contents (2nd msg))
                     ans))
          ((reset!) (for-effect-only (set! contents init-value)))
          (else (delegate base-object msg)))))))
```

Program 12.3 delegate

```
(define delegate
  (lambda (obj msg)
    (apply obj msg)))
```

respectively. We also use `msg` to denote the message. `delegate` is defined in Program 12.3.

In order to be able to reset the box to its initial value, `init-value`, it is necessary to preserve that value. Thus a local variable `contents` is introduced to hold the current value stored in the box. It is initialized with `init-value`. In the case clause that matches `swap!`, a let expression binds `ans` to the current contents of the box. Then `set!` puts the new value into the box, but the old value `ans` that was stored in the box is returned. Whenever the else clause is reached, no match was found for the method name, so the message is passed on (or *delegated*)[1] to another object, which attempts to respond to it. (We find that it is more suggestive to use the procedure name

[1] When an object cannot respond to a message, there are mechanisms other than delegation which have been developed. One common mechanism is *inheritance*. We have chosen to use delegation instead of inheritance; however, all programs expressible with inheritance are also expressible with delegation.

Program 12.4 `base-object`

```
(define base-object
  (lambda msg
    (case (1st msg)
      ((type) "base-object")
      (else invalid-method-name-indicator))))
```

Program 12.5 `send`

```
(define send
  (lambda args
    (let ((object (car args)) (message (cdr args)))
      (let ((try (apply object message)))
        (if (eq? invalid-method-name-indicator try)
            (error "Bad method name:" (car message)
                   "sent to object of"
                   (object 'type)
                   "type.")
            try)))))
```

delegate instead of *apply* to pass the message on to another object, although the two procedures `delegate` and `apply` behave the same by our simplification rule.) In this case, the object to which the message is delegated is the `base-object`, which returns `invalid-method-name-indicator`, which is bound to the string `"unknown"`.

```
(define invalid-method-name-indicator "unknown")
```

The procedure `send` then generates the appropriate invocation of `error`. The definitions of `base-object` and `send` are contained in Programs 12.4 and 12.5.

We shall define many different types of objects in this chapter using object makers similar to `box-maker`. These will each contain an else clause that must handle method names for which there is no match. One of the major advantages of using `send` is that all these else clauses will have exactly the same call structure

`(else (delegate` *object msg*`))`

and `send` takes the appropriate action. When writing the definitions of the object makers and when using the objects, we must remember that:

1. Every object should respond to the method name `type`.
2. When no match is found for a method name, the else clause should delegate the message to some object, which in some cases may be `base-object`.
3. Use `send` to pass messages to objects.

In this implementation of a box, the data structure used to store a value in the box is just a variable. The user is not concerned with this fact when using the box to store the value. We could have used a different data structure, such as a cons cell, in which to store the value. In the program below for `box-maker`, `init-value` is initially stored in the car position of a cons cell, which we denote by `cell`. The procedure `set-car!` is used to change the value stored in the box. This alternative version of `box-maker` is in Program 12.6.

Program 12.6 `box-maker` (Alternative)

```
(define box-maker
  (lambda (init-value)
    (let ((cell (cons init-value "any value")))
      (lambda msg
        (case (1st msg)
          ((type) "box")
          ((show) (car cell))
          ((update!) (for-effect-only (set-car! cell (2nd msg))))
          ((swap!) (let ((ans (car cell)))
                     (set-car! cell (2nd msg))
                     ans))
          ((reset!) (for-effect-only (set-car! cell init-value)))
          (else (delegate base-object msg)))))))
```

12.2.3 Counters

A *counter* is an object that stores an initial value and each time it is called, the stored value is changed according to some fixed rule. The counter has two arguments: the initial value stored and the procedure describing the action to be taken each time the counter is updated. For example, `(counter-maker 10 sub1)` is a counter with initial value 10 that decrements the counter by 1 when it is updated. The counter responds to the method names: `type`, `update!`, `show`, and `reset!`. The definition of `counter-maker` follows:

Program 12.7 `counter-maker` (Methods Disabled)

```
(define counter-maker
  (lambda (init-value unary-proc)
    (let ((total (box-maker init-value)))
      (lambda msg
        (case (1st msg)
          ((type) "counter")
          ((update!) (let ((result (unary-proc (send total 'show))))
                       (send total 'update! result)))
          ((swap!) (delegate base-object msg))
          (else (delegate total msg)))))))
```

The counter locally defines the box `total`, which contains the initial value stored in the counter. When the counter receives the message consisting of the method name `update!`, the unary update procedure `unary-proc` is applied to the value stored in the box `total` to obtain the new value which is then stored in `total`. For example, if we wanted the counter to count up by `1` each time it is updated, we can use `add1` as the unary update procedure. To create a counter with initial value `0` that increases the stored value by `5` each time it is updated, we write:

```
(counter-maker 0 (lambda (x) (+ 5 x)))
```

The counter is not supposed to respond to `swap!`. Thus if such messages are sent to a counter, they are delegated to `base-object` rather than to a box, which does respond to `swap!`. Since the counter responds to the messages `show` and `reset!` the same as the box `total`, the else clause merely passes these messages to `total`. Thus the message `show` displays the value currently stored in the counter, and `reset!` resets the counter to its initial value. The fact that the response of the counter to these messages can be found by passing them to the box is called *delegation*. The work of the counter is "delegated" to the behavior of the box.

The approach of catching the method names that are to be disabled, like `swap!`, is only one way of supporting the interface. Another alternative is to catch all the method names to be enabled. Thus, we can rewrite `counter-maker` using this view. As long as we delegate to the base object all the method names that are meaningless, we can use either approach. On one hand we are throwing the illegal method names out (i.e., *disabling* them), and on the other, we are delegating the legal ones (i.e., *enabling* them). In any

Program 12.8 counter-maker (Methods Enabled)

```
(define counter-maker
  (lambda (init-value unary-proc)
    (let ((total (box-maker init-value)))
      (lambda msg
        (case (1st msg)
          ((type) "counter")
          ((update!) (send total 'update!
                           (unary-proc (send total 'show))))
          ((show reset) (delegate total msg))
          (else (delegate base-object msg)))))))
```

event, both have the same effect, and each has aspects that recommend it. If we are delegating to an object with many legal method names, and only a few illegal ones, then we should disable illegal method names; otherwise we are free to choose to enable legal method names. A version of `counter-maker`, which enables legal method names, is presented in Program 12.8.

12.2.4 Accumulators

An *accumulator* is an object that has the initial value `init-value`. Each time it receives a message consisting of the method name `update!` and a value v, the binary update procedure `binary-proc` is applied to the value stored in the accumulator and v; the result is the new value stored in the accumulator. For example, if `acc` is an accumulator that initially stores the value 100 and has subtraction (-) as its binary update procedure, it is defined by

```
(define acc (accumulator-maker 100 -))
```

and

```
(send acc 'update! 10)
```

causes the number 90 to be stored in `acc`. If we then update `acc` with 25, we write

```
(send acc 'update! 25)
```

and the number 65 is stored in the accumulator.

Program 12.9 `accumulator-maker`

```
(define accumulator-maker
  (lambda (init-value binary-proc)
    (let ((total (box-maker init-value)))
      (lambda msg
        (case (1st msg)
          ((type) "accumulator")
          ((update!)
           (send total 'update!
             (binary-proc (send total 'show) (2nd msg))))
          ((swap!) (delegate base-object msg))
          (else (delegate total msg)))))))
```

The accumulator uses a box, called **total**, to store its values. In addition to responding to the message consisting of **update!** and a value, it uses delegation to pass such messages as **show** and **reset!** to the box **total**. Program 12.9 contains the code for **accumulator-maker**.

12.2.5 Gauges

A *gauge* is the last object to be defined in this section. A gauge is similar to a counter, but it has two unary update procedures, one to count up and the other to count down. The one to count up is called **unary-proc-up**, and the one to count down is called **unary-proc-down**. The gauge responds to two update messages **up!** and **down!**. It stores its values in a box called **total**. When the gauge receives the message **up!**, the update procedure **unary-proc-up** is invoked on the value stored in **total** to get the new value stored in **total**. Similarly, when the gauge receives the message **down!**, the update procedure **unary-proc-down** is invoked on the value stored in **total** to get the new value stored in **total**. The gauge also responds to the messages **show** and **reset!** by delegation from **total**. For example, to create a gauge **g** with initial value **10**, which either adds **1** or subtracts **1**, we write

```
(define g (gauge-maker 10 add1 sub1))
```

and

```
(send g 'up!)
```

causes the number **11** to be stored in **g**, while

Program 12.10 gauge-maker

```
(define gauge-maker
  (lambda (init-value unary-proc-up unary-proc-down)
    (let ((total (box-maker init-value)))
      (lambda msg
        (case (1st msg)
          ((type) "gauge")
          ((up!) (send total 'update!
                       (unary-proc-up (send total 'show))))
          ((down!) (send total 'update!
                         (unary-proc-down (send total 'show))))
          ((swap! update!) (delegate base-object msg))
          (else (delegate total msg)))))))
```

```
(send g 'down!)
```

returns the number stored in `g` to `10`. Program 12.10 contains the definition of `gauge-maker`.

Exercises

Exercise 12.1: `acc-max`
Define an accumulator `acc-max` that has initial value 0 and each time it is updated, it compares the value stored with a new value and stores the maximum of the two. Then test `acc-max` by updating it in succession with the numbers 3, 7, 2, 4, 10, 1, 5 and find the maximum by passing `acc-max` the `show` message.

Exercise 12.2: `double-box-maker`
Define a procedure `double-box-maker` that takes two arguments, `item1` and `item2`, and stores these values in two boxes, the `left` and `right`, respectively. An instance of `double-box-maker` responds to the following messages: `show-left`, `show-right`, `update-left!`, `update-right!`, and `reset!`.

Exercise 12.3: `accumulator-maker`, `gauge-maker`
In the definitions of `accumulator-maker` and `gauge-maker` method names that are illegal have been disabled. Rewrite the last two lines of each of these

procedures so that instead of disabling illegal method names, we enable legal method names and disable all others.

Exercise 12.4: `restricted-counter-maker`

Our implementation of `counter-maker` places no restrictions on the possible values that can be stored in the counter. Define `restricted-counter-maker` to take an additional argument, a predicate `pred`. No value is stored in a restricted counter unless it satisfies the predicate. If a value fails to satisfy the predicate, then a reset occurs. For example, if the predicate is `(lambda (n) (and (> n 0) (< n 100)))` and we try to bring the restricted counter up to 105, it will reset to its initial value.

Exercise 12.5

Define the hour hand of a 12-hour clock as a restricted counter. (See the preceding exercise.)

Exercise 12.6

Define a 12-hour clock that has both a minute and an hour hand. This clock is to be constructed from two objects. One of them will be the 12-hour clock, which displays only its hour hand, and the other, the minute hand, will be built using a modified restricted counter. Such a counter is created using `modified-restricted-counter-maker`, which includes an additional argument. This new argument is a reset procedure that is invoked in place of the built-in reset in the `restricted-counter-maker`. When the minute hand of the clock is about to pass to 60 minutes, the reset procedure is used not only to reset the minute hand to 0 but also to update the hour hand. Do not forget to initialize the clock. The new clock is itself to be an object created by the procedure of one argument, `clock-maker`, that responds to two messages: `show` and `update!`. (See the preceding exercise.)

Exercise 12.7

As was done in Chapters 8 and 9, tag the objects by adding `object-tag` as `"object"`. Then define the simple procedures `object?` and `make-object`. Wrap `make-object` around `(lambda msg ...)` and redefine `send`.

Exercise 12.8

Is it possible to implement an accumulator with a `counter-maker` instead of a `box-maker`? Is it possible to implement a counter with an `accumulator-maker` instead of a `box-maker`?

As we saw in Chapter 11, a stack is an ordered collection of items into which new items may be inserted at one end and from which items may be removed from the same end. The end at which items may be inserted or removed is called the *top* of the stack. The image that is often conjured up when thinking of a stack is the rack of trays in a cafeteria, in which one takes the top one, and trays are added from the top. As the stack builds up, the item that was put on first is buried deeper and deeper, and as things are removed from the stack, the one that was put on first is the last one to be removed. The item that was added to the stack last is the first one to be removed. Thus a stack is referred to as a *last-in-first-out* data structure, or a *LIFO*.

The stack has several methods associated with it;

- `empty?`, which tests whether the stack is empty.
- `push!`, which adds an item to the top of the stack.
- `top`, which returns the item at the top of the stack.
- `pop!`, which removes an item from the top of stack.
- `size`, which returns the number of items on the stack.
- `print`, which prints the items on the stack.

An experiment with stacks is given in Figure 12.11. The two stacks, `r` and `s`, are created in `[1]` and `[2]`. In the definitions of `r` and `s`, we see that `stack-maker` is a thunk, that is, a procedure of no arguments. Its definition is given in Program 12.12.

In the code for `stack-maker`, we used a list as the internal representation of the stack. The user need never know how it is represented, for if we change the representation, we can alter the definitions of the methods so that when their names are passed as messages to the stack, the results seen by the user are the same as those produced by the above code. Even when the stack is printed, it does not show the internal representation of the stack.

Exercise

Exercise 12.9

In arithmetic, parentheses are used to form groupings of numbers and operators. For example, one writes $3*(4+2)$. In more complicated expressions, three different kinds of separators are used to form groupings: parentheses '(', ')', brackets '[',']', and braces '{','}'. Here is an expression that uses all three

```
[1] (define r (stack-maker))
[2] (define s (stack-maker))
[3] (send s 'print)
TOP:
[4] (send r 'print)
TOP:
[5] (send s 'empty?)
#t
[6] (send s 'push! 'a)
[7] (send s 'push! 'b)
[8] (send s 'push! 'c)
[9] (send s 'top)
c
[10] (send s 'print)
TOP: c b a
[11] (send s 'empty?)
#f
[12] (send r 'empty?)
#t
[13] (send r 'push! 'd)
[14] (send s 'size)
3
[15] (send s 'pop!)
[16] (send s 'pop!)
[17] (send s 'print)
TOP: a
[18] (send r 'print)
TOP: d
```

Figure 12.11 Using stack operations

kinds of grouping symbols:

$$13 + 5*\{[14 - 3*(12 - 7)] - 15\}$$

Write a program that will scan a mathematical expression made up of the four basic operations $+, -, *$, and $/$ and the three kinds of separators and test whether the separators are correctly nested. The examples $(3 - 4]$ and $(5 - [2 + 4) + 1]$ are not correctly nested. This is a natural problem for the use of a stack, for whenever a left-grouping symbol is encountered, it is pushed onto the stack, and whenever a right-grouping symbol is encountered, the stack is popped and the left symbol that comes off the stack is compared to the right symbol just encountered. If they are of different types, the nesting is not correct. You can model the arithmetic expression as a list of numbers,

Program 12.12 stack-maker

```
(define stack-maker
  (lambda ()
    (let ((stk '()))
      (lambda msg
        (case (1st msg)
          ((type) "stack")
          ((empty?) (null? stk))
          ((push!) (for-effect-only
                     (set! stk (cons (2nd msg) stk))))
          ((top) (if (null? stk)
                     (error "top: The stack is empty.")
                     (car stk)))
          ((pop!) (for-effect-only
                    (if (null? stk)
                        (error "pop!: The stack is empty.")
                        (set! stk (cdr stk)))))
          ((size) (length stk))
          ((print) (display "TOP: ")
                   (for-each
                     (lambda (x)
                       (display x)
                       (display " "))
                     stk)
                   (newline))
          (else (delegate base-object msg)))))))
```

operators, and grouping symbols. Since Scheme uses these symbols as special characters, one cannot use them as grouping symbols in the list modeling the arithmetic expression. Thus use the strings "(", ")", "[", "]", "{", and "}" in place of the grouping symbols. The above arithmetic expression, in this representation, looks like

```
(13 + 5 * "{" "[" 14 - 3 * "(" 12 - 7 ")" "]" - 15 "}")
```

Test your program on the examples given here and on several additional tests you devise, some correctly and others incorrectly nested.

12.4 Queues

A *queue* is an ordered collection of items into which items are inserted at one end, called the *rear*, and from which items are removed at the other end, called the *front*. People waiting in line for service normally form a queue in which new people join the line at the rear and people are served from the front. Similarly, processes waiting to be run on a computer are put into a queue to await their turn. Stacks are called *LIFO* lists because the last one in is the first one out. Queues are called *FIFO* lists because the first one in is the first one out. Adding an item to the rear of the queue is called *enqueuing* the item, and removing an item from the front of the queue is called *dequeuing*. We implement a queue as an object with the following methods:

- `empty?`, which tests whether the queue is empty.
- `enqueue!`, which adds an item to the rear of the queue.
- `front`, which returns the item at the front of the queue.
- `dequeue!`, which removes the item from the front of the queue.
- `size`, which returns the number of items in the queue.
- `print`, which prints the items in the queue.

Our first implementation of a queue will imitate the way we implemented a stack. The data structure we choose for the queue is a list, with the first element of the list the front of the queue. To dequeue an element, we essentially take the cdr of the list. To enqueue an element, we must put it at the end of the list, so we can make a list of the element and append that onto the end of the queue. The code for such an implementation is presented in Program 12.13.

The implementation using lists as the data structure for the queue produces the results we want, but it does it inefficiently. The trouble is that when we enqueue an item, we use `append!`, which must cdr down `q` until the last pair and then we attach the cdr pointer to the list containing the new item. The longer the queue, the more "expensive" it is to cdr down `q` to get to the last pair. It would be better to have an implementation that could attach the new item to the end of the queue without having to cdr down the whole queue. We accomplish this by introducing a second pointer called `rear`, which points to the last cons cell in the queue. When the queue is empty, the pointer `q` points to a cell formed by `(cons '() '())`, and `rear` also points to that cell. Only the `cdr` of `q` is used.

Program 12.13 queue-maker

```
(define queue-maker
  (lambda ()
    (let ((q '()))
      (lambda msg
        (case (1st msg)
          ((type) "queue")
          ((empty?) (null? q))
          ((enqueue!) (for-effect-only
                        (let ((list-of-item (cons (2nd msg) '())))
                          (if (null? q)
                              (set! q list-of-item)
                              (append! q list-of-item)))))
          ((front) (if (null? q)
                       (error "front: The queue is empty.")
                       (car q)))
          ((dequeue!) (for-effect-only
                        (if (null? q)
                            (error "dequeue!: The queue is empty.")
                            (set! q (cdr q)))))
          ((size) (length q))
          ((print) (display "FRONT: ")
                   (for-each
                     (lambda (x) (display x) (display " "))
                     q)
                   (newline))
          (else (delegate base-object msg)))))))
```

Figure 12.14(a) shows a box-and-pointer representation of such a queue that has in it the numbers 1 and 2, with 1 at the front. Figure 12.14(b) shows how the new item 3 is added to the queue by setting the cdr of `rear` to be `(cons 3 '())` and then setting `rear` itself to point to the last cons cell in the list. Our new definition of `queue-maker` is given in Program 12.15. A sample session using a queue is given in Figure 12.16.

Exercises

Exercise 12.10

Add a message to the queue defined in Program 12.15 called `enqueue-list!`

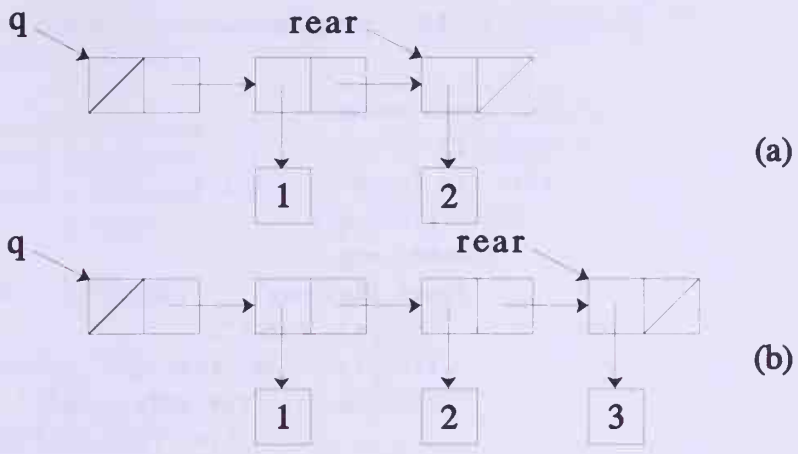

Figure 12.14 Box-and-pointer diagram for a queue

that takes as an argument a list `ls` and enqueues each of the elements of the list to the queue preserving their order. For example, if the queue `a` contains the elements `1`, `2`, `3`, with `1` at the front, and if `ls` is `(list 4 5 6)`, then after invocation of `(send a 'enqueue-list! ls)`, the queue `a` contains the elements `1`, `2`, `3`, `4`, `5`, `6` with `1` at the front. Do not use `append!`. Why?

Exercise 12.11

Revise the definition of `queue-maker` in Program 12.15 to include a message `enqueue-many!` that enqueues any number of items at one time. For example, `(send a 'enqueue-many! 'x 'y 'z)` has the same effect as

```
(begin
  (send a 'enqueue! 'x)
  (send a 'enqueue! 'y)
  (send a 'enqueue! 'z))
```

Exercise 12.12: `queue->list`

Define a procedure `queue->list` that takes as its argument a queue `q`, with `size` disabled, and returns a list of the elements in `q` without destroying the queue. In order to do this, one can first enqueue a unique element such as `(list '())`. Then cons the front of the queue onto the list, and also enqueue the front onto the queue. Now dequeue the queue, so that what was at the front is now at the rear of the queue. Repeat this operation of consing the front of the queue to the list, enqueuing the front of the queue so that it is at the rear, and then dequeuing the queue, until the unique element you enqueued

Program 12.15 queue-maker

```
(define queue-maker
  (lambda ()
    (let ((q (cons '() '())))
      (let ((rear q))
        (lambda msg
          (case (1st msg)
            ((type) "queue")
            ((empty?) (eq? rear q))
            ((enqueue!) (for-effect-only
                          (let ((list-of-item (cons (2nd msg) '())))
                            (set-cdr! rear list-of-item)
                            (set! rear list-of-item))))
            ((front) (if (eq? rear q)
                         (error "front: The queue is empty.")
                         (car (cdr q))))
            ((dequeue!) (for-effect-only
                          (if (eq? rear q)
                              (error "dequeue!: The queue is empty.")
                              (let ((front-cell (cdr q)))
                                (set-cdr! q (cdr front-cell))
                                (if (eq? front-cell rear)
                                    (set! rear q))))))
            ((size) (length (cdr q)))
            ((print) (display "FRONT: ")
                     (for-each
                       (lambda (x)
                         (display x)
                         (display " "))
                       (cdr q))
                     (newline))
            (else (delegate base-object msg)))))))
```

reaches the front. When it is dequeued, you have a list of the elements that are in the queue, and the queue is intact.

Exercise 12.13

Rework the previous problem with the method name `size` enabled.

```
[1] (define q (queue-maker))
[2] (send q 'empty?)
#t
[3] (send q 'enqueue! 1)
[4] (send q 'enqueue! 2)
[5] (send q 'enqueue! 3)
[6] (send q 'size)
3
[7] (send q 'front)
1
[8] (send q 'print)
FRONT: 1 2 3
[9] (send q 'empty?)
#f
[10] (send q 'dequeue!)
[11] (send q 'print)
FRONT: 2 3
```

Figure 12.16 Using queue operations

Exercise 12.14
In the first version of a queue given in this section, the message **enqueue!** contains the code **(append! q list-of-item)**. Discuss the correctness and the efficiency of the code for a queue if that line of code is replaced by **(append q list-of-item)** or by **(set! q (append q list-of-item))**.

12.5 Circular Lists

In the previous sections, we defined both the stack and the queue as objects. In the internal representation of these objects, we used lists. In the case of the queue, we used pointers to keep track of the front and the rear of the queue. There is another way of treating stacks and queues that is more elegant. It makes use of a data type known as a circular list. In this section, we first implement circular lists as objects and then use them to define both the stack and the queue, making use of delegation to take advantage of the properties of the circular list.

In an ordinary list, the cdr pointer of the last cons cell points to the empty list. This is denoted by placing a diagonal line in the right hand side of the last cons cell. If, instead, the cdr pointer of the last cons cell of the list points back to the first cons cell in the list, we say that the list is a *circular list*. The

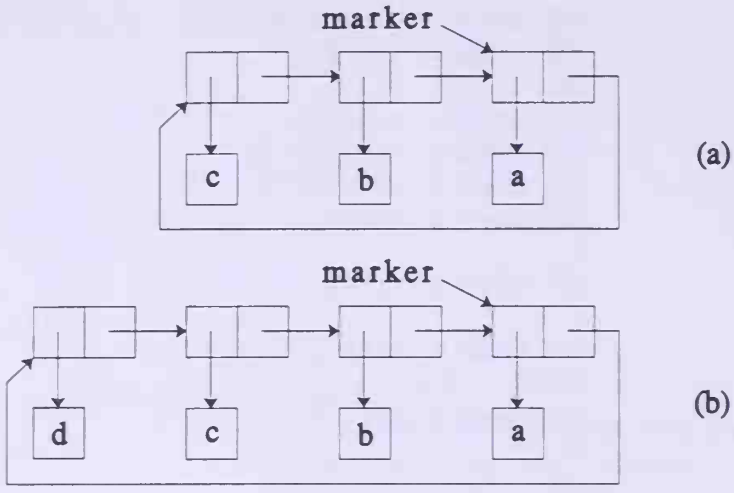

Figure 12.17 Box-and-pointer diagrams for a circular list

box and pointer diagram for a circular list containing the three items `c`, `b`, and `a` is shown in Figure 12.17a. Note that **`marker`** is a pointer to the cons cell whose **`car`** is `a`. Then to make the list circular, **`(cdr marker)`** points back to the cell whose **`car`** is `c`. To add an item `d` to this circular list, we cons `d` to **`(cdr marker)`** and then reset the **`cdr`** pointer of **`marker`** to point to the cons cell with `d` as its **`car`**. (See Figure 12.17b.) Thus inserting `d` into a nonempty circular list can be accomplished by invoking:

```
(set-cdr! marker (cons 'd (cdr marker)))
```

Similarly, to remove `d` from the resulting circular list, we note that **`(cdr (cdr marker))`** does not contain the item `d`, so we only have to write

```
(set-cdr! marker (cdr (cdr marker)))
```

to get back to the circular list in Figure 12.17a.

In general, we make an ordinary list circular by letting **`marker`** be a pointer to the end of the list. Then we set the cdr pointer of **`marker`** to point to the beginning of the list. The item to which the cdr pointer of **`marker`** points is referred to as the head of the circular list. As an object, a circular list responds to the following messages:

- **`empty?`**, which tests whether the circular list is empty.
- **`insert!`**, which adds an item to the circular list.

- **head**, which returns the head of the circular list, that is, the item that is just past the marker.
- **delete!**, which removes the head of the circular list.
- **move!**, which shifts the marker to point to the head of the circular list, thus making a new item the head.
- **size**, which returns the number of items in the circular list.
- **print**, which displays the circular list.

The code for **circular-list-maker** is given in Program 12.18. Initially, **marker** is locally defined to be the empty list, and when the method name **empty?** is received, it tests whether **marker** is the empty list. The message sent to insert an item into the circular list consists of two parts, the method name **insert!** and the item to be inserted. There are two cases to consider when inserting an item. If the list is empty, we first make a list consisting of the item to be inserted and then change **marker** to point to that list. Then we have to make the list circular, so we make the cdr pointer of **marker** point back to **marker** itself. We now have a circular list containing only the one item we inserted.

On the other hand, if the list is not empty, we use the fact that **(cdr marker)** points back to the head of the list when we cons the item to be inserted (that is, **(2nd msg)**) onto **(cdr marker)**. Once we have added the new item to the head of the list, we reset the cdr pointer of **marker** to point to the cell containing the new item, which becomes the new head of the list. We use the word *head* in spite of the fact that a circular list does not have a head or a tail. However, we may think of the cdr pointer of the cons cell to which **marker** points as pointing back to the head of the list to make the list circular. And we may think of **marker** itself as pointing to the last cell in the list.

If the list is empty when a **delete!** message is received, an error is signaled. If the list contains only one item (that is, if **(cdr marker)** points back to **marker** itself), then **marker** is set equal to the empty list. Otherwise, we again refer to the "head" of the list as the cons cell to which **(cdr marker)** points. Then we reset the cdr pointer of **marker** to point to **(cdr (cdr marker))**.

When we found the size of such objects as stacks and queues, we used the procedure **length** on their internal list representations. This requires cdring down the list while counting. We have given a more efficient way of doing this by keeping the size in a gauge and incrementing or decrementing it appropriately when we insert or delete something from the circular list.

We have to be careful in writing the code for a circular list that we do not get into an infinite loop, going around the circle of pointers indefinitely.

Program 12.18 circular-list-maker

```
(define circular-list-maker
  (lambda ()
    (let ((marker '())
          (size-gauge (gauge-maker 0 add1 sub1)))
      (lambda msg
        (case (1st msg)
          ((type) "circular list")
          ((empty?) (null? marker))
          ((insert!) (send size-gauge 'up!)
                     (for-effect-only
                       (if (null? marker)
                           (begin
                             (set! marker (cons (2nd msg) '()))
                             (set-cdr! marker marker))
                           (set-cdr! marker (cons (2nd msg) (cdr marker))))))
          ((head) (if (null? marker)
                      (error "head: The list is empty.")
                      (car (cdr marker))))
          ((delete!) (for-effect-only
                       (if (null? marker)
                           (error "delete!: The circular list is empty.")
                           (begin
                             (send size-gauge 'down!)
                             (if (eq? marker (cdr marker))
                                 (set! marker '())
                                 (set-cdr! marker (cdr (cdr marker))))))))
          ((move!) (for-effect-only
                     (if (null? marker)
                         (error "move!: The circular list is empty.")
                         (set! marker (cdr marker)))))
          ((size) (send size-gauge 'show))
          ((print) (if (not (null? marker))
                       (let ((next (cdr marker)))
                         (set-cdr! marker '())
                         (for-each (lambda (x) (display x) (display " "))
                                   next)
                         (set-cdr! marker next)))
                   (newline))
          (else (delegate base-object msg)))))))
```

Program 12.19 stack-maker

```
(define stack-maker
  (lambda ()
    (let ((c (circular-list-maker)))
      (lambda msg
        (case (1st msg)
          ((type) "stack")
          ((push!) (send c 'insert! (2nd msg)))
          ((pop!) (send c 'delete!))
          ((top) (send c 'head))
          ((print) (display "TOP: ") (send c 'print))
          ((insert! head delete! move!) (delegate base-object msg))
          (else (delegate c msg)))))))
```

In order to avoid this in the case of **print**, we use the trick of temporarily resetting the cdr pointer of **marker** to point to the empty list. Then the list is no longer circular, and we can use **for-each** without fear of looping indefinitely.

We are now ready to look at the definitions of stack and queue making use of a circular list. In implementing the stack, a circular list is used and the marker stays fixed. When the stack receives a **push!** message, it sends it to the circular list as an **insert!** message. Similarly, the **pop!** message is sent to the circular list as a **delete!** message. When the **print** message is received by the stack, the word **TOP:** is first printed, and then the message is sent to the circular list. The stack messages **size** and **empty?** are delegated to the circular list. The code for **stack-maker** using a circular list is in Program 12.19.

The **queue-maker** is similarly defined in terms of a circular list, but this time, the marker is moved each time an item is inserted, so that it points to the cell containing the new item. Again, most of the queue operations are delegated to the circular list. The code for **queue-maker** making use of a circular list is given in Program 12.20.

This is an elegant way of implementing both the **stack-maker** and the **queue-maker**. They take advantage of delegation by passing messages on to the circular list. The circular list was flexible enough because we were able to move the **marker** to keep track of certain cells. Notice that we have gained in efficiency by making use of the internal gauge in the circular list to keep the size of the stacks or queues. The circular list is, in general, a useful data structure.

Program 12.20 queue-maker

```
(define queue-maker
  (lambda ()
    (let ((c (circular-list-maker)))
      (lambda msg
        (case (1st msg)
          ((type) "queue")
          ((enqueue!) (send c 'insert! (2nd msg)) (send c 'move!))
          ((dequeue!) (send c 'delete!))
          ((front) (send c 'head))
          ((print) (display "FRONT: ") (send c 'print))
          ((insert! head delete! move!) (delegate base-object msg))
          (else (delegate c msg)))))))
```

Exercises

Exercise 12.15
Redefine the `stack-maker` and `queue-maker` procedures presented in Programs 12.19 and 12.20 so that, instead of the illegal method names being disabled, the legal method names are enabled.

Exercise 12.16
Draw the box-and-pointer diagrams for a stack implemented using a circular list. Start with the empty stack, push on the items `a`, `b`, `c`, and `d`, and then pop these four items. Show the box and pointer diagrams for the successive stages as the stack increases and decreases in size.

Exercise 12.17
Make the same sequence of box and pointer diagrams as in the previous exercise but this time for a queue.

Exercise 12.18
Redefine `circular-list-maker` in Program 12.18 keeping a local variable that is initialized to zero to keep the size of the circular list without using a gauge. Then do it without any local variables.

Exercise 12.19
When building a circular list, it is not necessary to build a circular structure. Instead, the method names, which rely on the circular structure, must be

redefined. For example, if before, the cdr of `marker` was a cell `c`, then using a simple list, it would be necessary to test `(null? (cdr marker))` and then return `c`. This approach has a cost because there is an additional local variable to maintain, which requires setting and testing. However, the benefit is that no structures are built that can unintentionally enter infinite loops. Redefine `circular-list-maker` without actually using an explicitly circular structure.

Exercise 12.20

Add a method `reverse` to the `circular-list-maker` that reverses the circular list in such a way that the cdr pointer of each cons cell is changed to point to the previous cell in the list instead of the next cell. The diagram in Figure 12.21 shows a circular list containing four items before and after reversing. As in the diagram, be sure your method moves the marker.

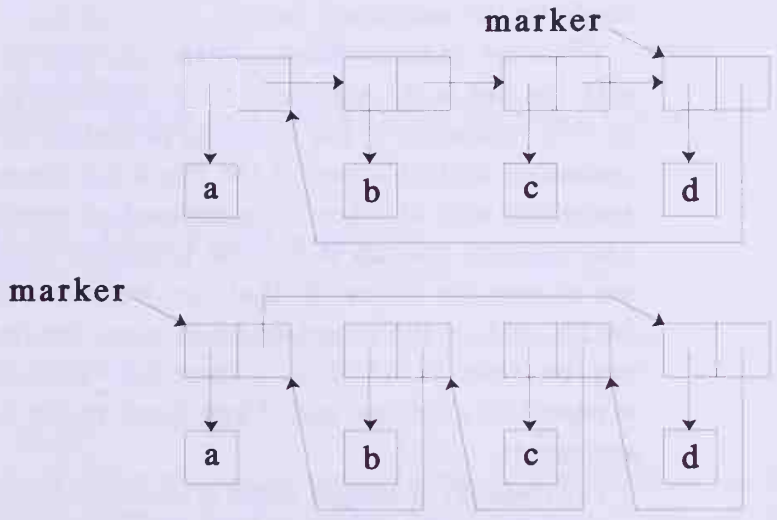

Figure 12.21 Reversing a circular list

12.6 Buckets and Hash Tables

In Chapter 11, we used a table to store the values computed by procedures by memoizing those procedures. The values were retrieved from the table by calling a procedure `lookup`. In this section, we construct objects that have the properties of tables. These objects are called *buckets*. We also present a second way of storing data using *hash tables*, which are vectors in which the

entry for each index is a bucket. In this way, large amounts of data can be stored in relatively small vectors.

Buckets respond to two messages:

- **update!**, which adds (or alters) a bucket entry.
- **lookup**, which retrieves a bucket entry.

A *bucket* is a structure like a stack or queue whose internal representation can be thought of as a flat list. Unlike a stack or queue, the order in which things are entered into a bucket is unimportant, and a bucket can only get bigger. An entry in a bucket (much the same as in a table) consists of two parts: the *key* and its *associated value*. When we memoize the Fibonacci procedure, each table entry consists of the procedure's argument and the value of the procedure when called with that argument. In our bucket, the procedure's argument would be the key, and the value of the procedure for that argument would be the associated value.

When we update a bucket, if the key is present, then the value associated with the key is the argument to an updating procedure. The value returned by this invocation of the updating procedure determines the new value to be associated with this key. If the key is not present, then the new value to be associated with this key is determined by invoking an initializing procedure. The message **lookup** is like the procedure **lookup** introduced in the previous chapter for tables. In that use, we invoke **(lookup key table success fail)**, and in the object-oriented view, we invoke **(send bucket 'lookup key success fail)**. Thus if there is a value associated with **key**, that value is passed to **success**, and if **key** is not in the table, **fail** is invoked on zero arguments.

For **update!** messages there is some similarity with **lookup** because there are separate responses to the existence or nonexistence of the key in the table. The call structure for **update!** is **(send bucket 'update! key proc-if-present proc-if-absent)**. Again a search of the bucket for the key occurs. If **key** exists with associated value, *val*, that value is replaced with the result of evaluating **(proc-if-present** *val***)**. If **key** does not exist, it is added with the associated value **(proc-if-absent key)**. A typical session with a bucket is given as an example in Figure 12.22. Program 12.23 is an implementation of a **bucket-maker**.

Recall that we defined memoize in the previous chapter as a mechanism for improving the efficiency of any single-argument procedure **proc**. We can use the bucket mechanism to obtain another version of **memoize** (Program 12.24). The key will be the argument, **n**, and its associated value will be the value of **(proc n)**.

```
[1] (define b (bucket-maker))
[2] (send b 'lookup 'a (lambda (x) x) (lambda () 'no))
no
[3] (send b 'update! 'a (lambda (x) (add1 x)) (lambda (x) 0))
[4] (send b 'lookup 'a (lambda (x) x) (lambda () 'no))
0
[5] (send b 'update! 'a (lambda (x) (add1 x)) (lambda (x) 0))
[6] (send b 'lookup 'a (lambda (x) x) (lambda () 'no))
1
[7] (send b 'update! 'q (lambda (x) (+ 2 x)) (lambda (x) 1000))
[8] (send b 'lookup 'q (lambda (x) x) (lambda () 'no))
1000
[9] (send b 'update! 'q (lambda (x) (+ 2 x)) (lambda (x) 1000))
[10] (send b 'lookup 'q (lambda (x) x) (lambda () 'no))
1002
[11] (send b 'update! 'q
       (lambda (x)
         (send b 'lookup 'a (lambda (y) (- x y)) (lambda () 'no)))
       (lambda (y) 'no))
[12] (send b 'lookup 'q (lambda (x) x) (lambda () 'no))
1001
```

Figure 12.22 Using bucket operations

Exercise

Exercise 12.21

The two invocations of **send** in **memoize** can be simplified to one by adding a new method name to **bucket-maker** (see Programs 12.23 and 12.24) that combines the update and lookup into one operation and thus avoids one of the two searches. Rewrite **bucket-maker** to run the definition of **memoize** below.

```
(define memoize
  (lambda (proc)
    (let ((bucket (bucket-maker)))
      (lambda (arg)
        (send bucket 'update!-lookup arg (lambda (val) val) proc)))))
```

Requiring no upper bound on the size of a bucket has its own cost. As the bucket gets bigger, we discover that the search for updating and looking information up in the bucket gets more and more expensive. Let us consider another program that uses a bucket. Suppose we have a list of strings, like

Program 12.23 `bucket-maker`

```
(define bucket-maker
  (lambda ()
    (let ((table '()))
      (lambda msg
        (case (1st msg)
          ((type) "bucket")
          ((lookup)
           (let ((key (2nd msg)) (succ (3rd msg)) (fail (4th msg)))
             (lookup key table (lambda (pr) (succ (cdr pr))) fail)))
          ((update!)
           (for-effect-only
             (let ((key (2nd msg))
                   (updater (3rd msg))
                   (initializer (4th msg)))
               (lookup key table
                 (lambda (pr)
                   (set-cdr! pr (updater (cdr pr))))
                 (lambda ()
                   (let ((pr (cons key (initializer key))))
                     (set! table (cons pr table))))))))
          (else (delegate base-object msg)))))))
```

Program 12.24 `memoize`

```
(define memoize
  (lambda (proc)
    (let ((bucket (bucket-maker)))
      (lambda (arg)
        (send bucket 'update! arg (lambda (val) val) proc)
        (send bucket 'lookup arg
          (lambda (val) val) (lambda () #f))))))
```

the contents of a book. We would like to find the word count frequency of the articles *a*, *an*, and *the* and possibly some others. We could solve this as follows:

```
(define word-frequency
  (lambda (string-list)
    (let ((b (bucket-maker)))
      (for-each
        (lambda (s) (send b 'update! s add1 (lambda (s) 1)))
        string-list)
      b)))
```

This defines the procedure `word-frequency`, which, when passed a text (a list of strings), returns a bucket that has each of the different strings in the text as a key and the number of times that string appears in the text as its associated value. Now suppose that the variable `string-list` is bound to some text; for example, the text might start with `("four" "score" "and" "seven" "years" "ago" "our" "fathers" ...)`. By writing

```
(define word-frequency-bucket (word-frequency string-list))
```

we define a bucket, called `word-frequency-bucket`, that contains each of the different words in our text as keys and the frequency of that word as its associated value. To see how many times the three strings `"a"`, `"an"`, and `"the"` appear in the text, we write:

```
(map
  (lambda (s)
    (cons s (send word-frequency-bucket 'lookup s
                  (lambda (v) v)
                  (lambda () 0))))
  '("a" "an" "the"))
```

This returns a list of the form `(("a" . 7) ("an" . 0) ("the" . 10))`.

If we were maintaining a frequency count for a book with 1,000 different words, then the bucket would be a list of 1,000 items, and searching it would be expensive. We next show how to avoid this problem.

We have now seen two ways of handling the building of tables for such purposes as memoizing. The first method was to use lists or buckets, which has the disadvantage that when the table gets long, lookup becomes a costly operation. The other method was to use a large vector so that each entry can be stored with a unique index and can be accessed randomly. This has the disadvantage that the vector has a predetermined fixed length and can hold a limited number of entries. We are now ready to look at a surprisingly simple solution to avoid the long searches and to allow for an unlimited number of

entries. We create a vector that holds one bucket per index. This way we can partition the pairs by placing individual keys and their associated values in a bucket as a function of what the key is.

A rather naive solution to the word frequency problem would be to associate a bucket with each letter. This would create 26 buckets, and if we were lucky, the average length of each bucket would be 1000/26 (approximately 40). Then we could use the first or last letter of a string to determine which bucket to update. Of course, words in English being what they are, the z-bucket will not carry its load. The choice of function and the length of the vector vary with the nature of the data being stored. The function must take a key and replace it by some *nonnegative integer that can reference the vector.* This function is called a *hash function* because it hashes up the data and turns them into integers, which are then used to access the vector. The important point here is that we want the hash function to spread the data evenly in the buckets. For the Fibonacci numbers example, a reasonable hash function is the remainder with the size of the vector. Here is how to create hash tables.

Program 12.25 `hash-table-maker`

```
(define hash-table-maker
  (lambda (size hash-fn)
    (let ((v ((vector-generator (lambda (i) (bucket-maker))) size)))
      (lambda msg
        (case (1st msg)
          ((type) "hash table")
          (else
            (delegate (vector-ref v (hash-fn (2nd msg))) msg)))))))
```

An empty bucket is placed at each index of the vector, v. Then, using the key, an index is determined by applying the `hash-fn` to the key. The value at that index is a bucket that responds to the same messages as hash tables. By delegating to the bucket the original message, the same information is forwarded to the bucket. We now write the new definition of `memoize` using a hash table in Program 12.26. In order to write this new memoize we only need to supply arguments to `hash-table-maker`. Everything else remains unchanged. This version of `memoize` is restricted to numerical data since its associated hash function invokes remainder on its argument. The hash function can be as general as the problem for which it is being used demands.

Program 12.26 memoize

```
(define memoize
  (let ((hashf (lambda (x) (remainder x 1000))))
    (let ((h (hash-table-maker 1000 hashf)))
      (lambda (proc)
        (lambda (arg)
          (send h 'update! arg (lambda (v) v) proc)
          (send h 'lookup arg (lambda (v) v) (lambda () #f)))))))
```

Similarly we can rewrite `word-frequency` by including a hash table where we earlier had a bucket. For this, we need a way of converting the first letter of each string into an integer. The code that assigns to each keyboard character a unique integer (see Appendix A1) provides us with just the help we need. Scheme has a procedure `string-ref` that takes a string and an integer as arguments and returns the character in the string having that integer as its index. Scheme also has the procedure `char->integer`, which takes a character as its argument and returns the integer associated with that character. We shall study the character data type more fully in Chapter 15. We use these two procedures now to define the hash function for the procedure `word-frequency`:

```
(define word-frequency
  (let ((naive-hash-function
         (lambda (s)
           (remainder (char->integer (string-ref s 0)) 26))))
    (let ((h (hash-table-maker 26 naive-hash-function)))
      (lambda (string-list)
        (for-each
          (lambda (s) (send h 'update! s add1 (lambda (s) 1)))
          string-list)
        h))))
```

A popular hash function for strings is one that sums the `(char->integer (string-ref s i))` for `i = 0` to `(sub1 (string-length s))` and finds the remainder with the length of the vector. The important aspect of the choice of hash function is that it must spread the data randomly into the buckets so that each bucket carries its load.

The advantage of hash tables is that when order is not important, a table can be stored in a vector so that retrieval and updating are far more efficient than in simple linear search. The disadvantage is that we rely on a

hash function that cannot know in advance what the data will look like. To demonstrate this, consider the following definition of `new-bucket-maker`:

```
(define new-bucket-maker
  (lambda ()
    (hash-table-maker 1 (lambda (d) 0))))
```

This hash table is as inefficient as a bucket. We chose the vector too small, and we chose the hash function too naively. Of course, this would never be done. In practice, most systems discourage the user from worrying about the size of the hash table or the nature of the hash function.

Exercises

Exercise 12.22
Construct a list of strings from some paragraph in this section, and run `word-frequency` over that list. Determine how many of each of the articles *a*, *an*, and *the* were used.

Exercise 12.23
Using the list of strings from the previous exercise, introduce a hash function that uses a large prime number for the vector length and uses the *sum of integers corresponding to characters* hash function as described in this section.

Exercise 12.24
Include a message `re-initialize!` in the definition of `bucket-maker` and `hash-table-maker`. In both cases, this method returns the object to its initial state.

Exercise 12.25
Lists that can only grow can get expensive.

a. Include a `remove!` message in `bucket-maker` that removes the key and its associated value from a bucket. The operation guarantees that if `b` is a bucket, then the following expression is always false.

```
(begin
  (send b 'remove! key)
  (send b 'lookup key (lambda (v) #t) (lambda () #f)))
```

is always false.

b. Include a `remove!` message in `hash-table-maker` that removes the key and its associated value from the hash table. If `b` is a hash table, then the expression above is always false.

Exercise 12.26: `store!`

Define a procedure `store!` that takes a hash table (or bucket), a key, and a value and is defined so that if `b` is a hash table (or bucket), then

```
(begin
  (store! b key value)
  (send b 'lookup key (lambda (v) (equal? value v)) (lambda () #f)))
```

is always true. Do this without adding any new messages to `hash-table-maker` (or `bucket-maker`).

Exercise 12.27

Include an `image` message in `bucket-maker` whose value is a list of the key-value pairs. Design it so that in the event of a subsequent update to an existing key, that update will not mutate the list previously returned by the `image` message. If `b` is a bucket and `(send b 'lookup key number? (lambda () #f))` is true then

```
(let ((prs (send b 'image)))
  (send b 'update! key add1 (lambda (k) 0))
  (= (cdr (assoc key prs)) (cdr (assoc key (send b 'image)))))
```

is always false.

Exercise 12.28

Using the previous exercise, include an `image` message in `hash-table-maker` whose value is a list of key-value pairs. Design it so that in the event of a subsequent update to an existing key, that update will not mutate the list previously returned by the image method. If `b` is a hash table and `(send b 'lookup key number? (lambda () #f))` is true, then the equation of the previous exercise holds. *Hint*: You may be tempted to use `append`, but here is an example where if you defined `bucket-maker` correctly, you should be able to use `append!`.

The next four problems are related. Work them in order and you will discover an interesting generalization of delegation.

Exercise 12.29: `theater-maker`

Consider the definition of `theater-maker` below. When entering a theater, there is usually a line to purchase tickets. Sometimes what is showing at the theater attracts a massive audience. When that happens, the doors to the theater may close while there is still a line to purchase tickets. By using a gauge for modeling the flow of patrons into the loge and a ticket queue where each patron waits, we can model these facets of a theater. What are the advantages of using `delegate` in the else clause of `theater-maker`? What are the disadvantages?

```
(define theater-maker
  (lambda (capacity)
    (let ((ticket-line (queue-maker))
          (vacancies (gauge-maker capacity add1 sub1)))
      (lambda msg
        (case (1st msg)
          ((type) "theater")
          ((enter!) (if (zero? (send vacancies 'show))
                        (display "doors closed")
                        (begin
                          (send ticket-line 'dequeue!)
                          (send vacancies 'down!))))
          ((leave!) (if (< (send vacancies 'show) capacity)
                        (send vacancies 'up!)
                        (error "leave!: The theater is empty.")))
          (else (delegate ticket-line msg)))))))
```

Exercise 12.30

In `theater-maker`, suppose we would like to know how many seats are vacant for the next showing. We cannot find this out without introducing a message, say `show`, in the definition of `theater-maker`. See the code below. Why must we include the extra message?

```
(define theater-maker
  (lambda (capacity)
    (let ((ticket-line (queue-maker))
          (vacancies (gauge-maker capacity add1 sub1)))
      (lambda msg
        (case (1st msg)
          ((type) "theater")
```

```
((enter!) (if (zero? (send vacancies 'show))
              (display "doors closed")
              (begin
                (send ticket-line 'dequeue!)
                (send vacancies 'down!))))
((leave!) (if (< (send vacancies 'show) capacity)
              (send vacancies 'up!)
              (error "leave!: The theater is empty.")))
((show) (send vacancies 'show))
(else (delegate ticket-line msg)))))))
```

We have two active objects: `ticket-line` and `vacancies`. The default line has `(delegate ticket-line msg)`. This means that we do not have `(delegate vacancies msg)`. We are only allowing one default. With *double delegation*, we can have two defaults. If the message is not applicable to the first default, it tries the second. So far, we have only seen objects with single delegation. In this exercise, we build objects with *multiple delegation*.

We introduce a binary function, `combine`, that, like `compose`, takes two procedures (in this case, objects) as parameters and returns a procedure as a value.

Program 12.27 `combine`

```
(define combine
  (lambda (f g)
    (lambda msg
      (let ((f-try (delegate f msg)))
        (if (eq? invalid-method-name-indicator f-try)
            (delegate g msg)
            f-try)))))
```

The returned procedure will delegate a message, in order, to the two objects until it finds one that does not return `invalid-method-name-indicator`. If `f` is not such an object, it invokes `g`. The procedure `combine` takes only two arguments. Rewrite `combine` to take two or more arguments. Below we have changed `theater-maker` to use `combine` so that the vacancies messages will be delegated too. The result will be multiple delegation, which will delegate `show` messages without the additional line in `theater-maker`.

```
(define theater-maker
  (lambda (capacity)
    (let ((ticket-line (queue-maker))
          (vacancies (gauge-maker capacity add1 sub1)))
      (lambda msg
        (case (1st msg)
          ((type) "theater")
          ((enter!) (if (zero? (send vacancies 'show))
                        (display "doors closed")
                        (begin
                          (send ticket-line 'dequeue!)
                          (send vacancies 'down!))))
          ((leave!) (if (< (send vacancies 'show) capacity)
                        (send vacancies 'up!)
                        (error "leave! The theater is empty.")))
          (else (delegate (combine ticket-line vacancies) msg)))))))
```

Exercise 12.31

The multiple delegation used with `combine` in the previous exercise is sometimes dangerous because method names are symbols. What would happen if `show` were used instead of `front` as the message for looking at the first element in a queue? Consider both expressions:

```
(delegate (combine ticket-line vacancies) msg)
```

and

```
(delegate (combine vacancies ticket-line) msg)
```

Exercise 12.32

In the interests of security, we would like to disable some operations from delegation. For example, we would like to keep anyone from resetting the gauge. This would correspond to yelling "fire," clearing the loge before the showing, and then allowing just the current contents of the ticket line to enter the loge. That would not be fair. The patrons who were already in the loge would have paid without receiving any entertainment. Or perhaps an `update!` message might be sent so that everyone might think the loge was full. Such skullduggery is possible with the current configuration of `theater-maker`. However, if we form a list of those "unfriendly" messages and disable them, we can keep these theaters from allowing such nefarious acts. Below is a partial solution where we have disabled `reset!` and `update!`. Rewrite the definition of `theater-maker` below to include all of the messages that should be disabled:

```
(define theater-maker
  (lambda (capacity)
    (let ((ticket-line (queue-maker))
          (vacancies (gauge-maker capacity add1 sub1)))
      (lambda msg
        (case (1st msg)
          ((type) "theater")
          ((enter!) (if (zero? (send vacancies 'show))
                        (display "doors closed")
                        (begin
                          (send ticket-line 'dequeue!)
                          (send vacancies 'down!))))
          ((leave!) (if (< (send vacancies 'show) capacity)
                        (send vacancies 'up!)
                        (error "leave!: The theater is empty.")))
          ((reset! update!) (delegate base-object msg))
          (else (delegate (combine ticket-line vacancies) msg)))))))
```

The next six problems are related. Work them in order, and you will discover some interesting generalizations of objects as we have defined them in this chapter.

Exercise 12.33
Consider a new definition of `send`.

Program 12.28 `send`

```
(define send
  (lambda args
    (let ((try (apply (car args) args)))
      (if (eq? invalid-method-name-indicator try)
          (let ((object (car args)) (message (cdr args)))
            (error "Bad method name:" (car message)
                   "sent to object of"
                   (object object 'type)
                   "type."))
          try))))
```

According to this definition, uses of `send` are the same as before, but each message includes the receiver of the message as the first element of the message. Here is an example that uses this `send` to build `counter-maker`:

```
(define counter-maker
  (lambda (init-value unary-proc)
    (let ((total (box-maker init-value)))
      (lambda message
        (let ((self (car message)) (msg (cdr message)))
          (case (1st msg)
            ((type) "counter")
            ((update!) (let ((result (unary-proc (send total 'show))))
                         (send total 'update! result)))
            ((swap!) (delegate base-object message))
            (else (delegate total message)))))))))
```

The variable **message** contains the receiver as its car and the original **msg** as its cdr. When we delegate, we use the whole **message**. Rewrite **box-maker** so that this definition of **counter-maker** works. Be sure to redefine **base-object**.

Exercise 12.34
Below is the definition of **cartesian-point-maker**:

```
(define cartesian-point-maker
  (lambda (x-coord y-coord)
    (lambda message
      (let ((self (car message)) (msg (cdr message)))
        (case (1st msg)
          ((type) "Cartesian point")
          ((distance) (sqrt (+ (square x-coord) (square y-coord))))
          ((closer?) (< (send self 'distance) (send (2nd msg) 'distance)))
          (else (delegate base-object message)))))))
```

Fill in the gaps in the experiment below:

```
[1] (define cp1 (cartesian-point-maker 3.0 4.0))
[2] (send cp1 'distance)
?__________
[3] (define cp2 (cartesian-point-maker 1.0 6.0))
[4] (send cp2 'distance)
?__________
[5] (send cp1 'closer? cp2)
?__________
[6] (send cp2 'closer? cp1)
?__________
```

Exercise 12.35

Using the definitions of the previous exercise, we add a new kind of point. In the Cartesian point, we found the distance to the origin as a straight line. In this definition, we determine the distance as the sum of two straight lines: the distance to the x-axis and the distance to the y-axis. This type of point is called a Manhattan point because it is reminiscent of distances traveled in cities. Below is the definition of `manhattan-point-maker`:

```
(define manhattan-point-maker
  (lambda (x-coord y-coord)
    (let ((p (cartesian-point-maker x-coord y-coord)))
      (lambda message
        (let ((self (car message)) (msg (cdr message)))
          (case (1st msg)
            ((type) "Manhattan point")
            ((distance) (+ x-coord y-coord))
            (else (delegate p message))))))))
```

With this definition, we have refined `cartesian-point-maker` by determining the distance differently. The determination of which of two points is closer to the origin stays the same, but if the point is a Manhattan point, it determines its distance to the origin by summing instead of finding the square root of the sum of squares. Fill in the gaps in the experiment below:

```
[7] (define mp1 (manhattan-point-maker 6.0 1.0))
[8] (send mp1 'distance)
?__________
[9] (send cp2 'closer? mp1)
?__________
[10] (send mp1 'closer? cp2)
?__________
```

Exercise 12.36

Suppose that we always create points at the origin (0,0). Then we could add a method name `moveto!` that would take an x-coordinate and a y-coordinate as arguments. In addition, we want a list of the two current coordinates. Add the method names `current-coordinates` and `moveto!` to the definition of `cartesian-origin-maker` below. We have left the type as `"Cartesian point"` because it still is a point as one would find in the plane.

```
(define cartesian-origin-maker
  (lambda ()
    (let ((x-coord 0) (y-coord 0))
      (lambda message
        (let ((self (car message)) (msg (cdr message)))
          (case (car msg)
            ((type) "Cartesian point")
            ((distance) (sqrt (+ (square x-coord) (square y-coord))))
            ((closer?) (< (send self 'distance) (send (2nd msg) 'distance)))
            (else (delegate base-object message))))))))
```

Exercise 12.37

Using the definition of your solution to `cartesian-origin-maker` from the previous exercise, fill in the gaps in the experiment below.

```
[1] (define cp1 (cartesian-origin-maker))
[2] (send cp1 'distance)
?__________
[3] (send cp1 'current-coordinates)
?__________
[4] (send cp1 'moveto! 6.0 1.0)
?__________
[5] (send cp1 'distance)
?__________
[6] (send cp1 'current-coordinates)
?__________
```

Exercise 12.38

Fill in the rest of the definition of `manhattan-origin-maker` below, but do *not* use `p`. Test `closer?` on a Manhattan point and a Cartesian point that have both been moved to (6.0,1.0).

```
(define manhattan-origin-maker
  (lambda ()
    (let ((p (cartesian-origin-maker)))
      (lambda message
        (let ((self (car message)) (msg (cdr message)))
          (case (1st msg)
            ((type) "Manhattan point")
            ((distance) ?__________  )
            (else (delegate p message))))))))
```