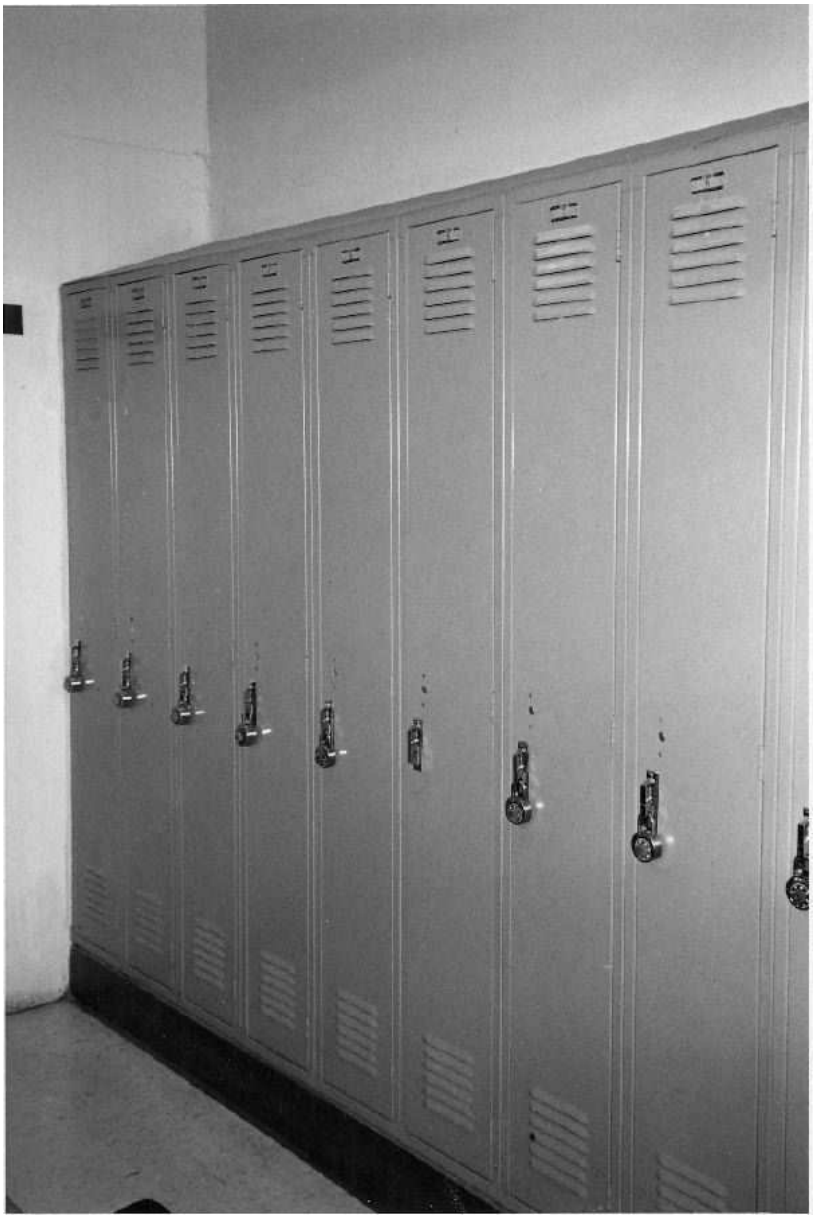

A row of boxes

23 Vectors

So far all the programs we've written in this book have had no memory of the past history of the computation. We invoke a function with certain arguments, and we get back a value that depends only on those arguments. Compare this with the operation of Scheme itself:

```
> (foo 3)
ERROR: FOO HAS NO VALUE

> (define (foo x)
    (word x x))

> (foo 3)
33
```

Scheme *remembers* that you have defined `foo`, so its response to the very same expression is different the second time. Scheme maintains a record of certain results of its past interaction with you; in particular, Scheme remembers the global variables that you have defined. This record is called its *state*.

Most of the programs that people use routinely are full of state; your text editor, for example, remembers all the characters in your file. In this chapter you will learn how to write programs with state.

The Indy 500

The Indianapolis 500 is an annual 500-mile automobile race, famous among people who like that sort of thing. It's held at the Indianapolis Motor Speedway, a racetrack in Indianapolis, Indiana. (Indiana is better known as the home of Dan Friedman, the

coauthor of some good books about Scheme.) The racetrack is $2\frac{1}{2}$ miles long, so, as you might imagine, the racers have to complete 200 laps in order to finish the race. This means that someone has to keep track of how many laps each car has completed so far.

Let's write a program to help this person keep count. Each car has a number, and the count person will invoke the procedure `lap` with that number as argument every time a car completes a lap. The procedure will return the number of laps that that car has completed altogether:

```
> (lap 87)
1

> (lap 64)
1

> (lap 17)
1

> (lap 64)
2

> (lap 64)
3
```

(Car 64 managed to complete three laps before the other cars completed two because the others had flat tires.) Note that we typed the expression `(lap 64)` three times and got three different answers. `Lap` isn't a function! A function has to return the same answer whenever it's invoked with the same arguments.

Vectors

The point of this chapter is to show how procedures like `lap` can be written. To accomplish this, we're going to use a data structure called a *vector*. (You may have seen something similar in other programming languages under the name "array.")

A vector is, in effect, a row of boxes into which values can be put. Each vector has a fixed number of boxes; when you create a vector, you have to say how many boxes you want. Once a vector is created, there are two things you can do with it: You can put a new value into a box (replacing any old value that might have been there), or you can examine the value in a box. The boxes are numbered, starting with zero.

```
> (define v (make-vector 5))
```

```
> (vector-set! v 0 'shoe)

> (vector-set! v 3 'bread)

> (vector-set! v 2 '(savoy truffle))

> (vector-ref v 3)
BREAD
```

There are several details to note here. When we invoke `make-vector` we give it one argument, the number of boxes we want the vector to have. (In this example, there are five boxes, numbered 0 through 4. There is no box 5.) When we create the vector, there is nothing in any of the boxes.*

We put things in boxes using the `vector-set!` procedure. The exclamation point in its name, indicates that this is a *mutator*—a procedure that changes the value of some previously created data structure. The exclamation point is pronounced "bang," as in "vector set bang." (Scheme actually has several such mutators, including mutators for lists, but this is the only one we'll use in this book. A procedure that modifies its argument is also called *destructive*.) The arguments to `vector-set!` are the vector, the number of the box (the *index*), and the desired new value. Like `define`, `vector-set!` returns an unspecified value.

We examine the contents of a box using `vector-ref`, which takes two arguments, the vector and an index. `Vector-ref` is similar to `list-ref`, except that it operates on vectors instead of lists.

We can change the contents of a box that already has something in it.

```
> (vector-set! v 3 'jewel)

> (vector-ref v 3)
JEWEL
```

The old value of box 3, `bread`, is no longer there. It's been replaced by the new value.

```
> (vector-set! v 1 741)
```

* The Scheme standard says that the initial contents of the boxes is "unspecified." That means that the result depends on the particular version of Scheme you're using. It's a bad idea to try to examine the contents of a box before putting something in it.

```
> (vector-set! v 4 #t)

> v
#(SHOE 741 (SAVOY TRUFFLE) JEWEL #T)
```

Once the vector is completely full, we can print its value. Scheme prints vectors in a format like that of lists, except that there is a number sign (`#`) before the open parenthesis. If you ever have need for a constant vector (one that you're not going to mutate), you can quote it using the same notation:

```
> (vector-ref '#(a b c d) 2)
C
```

Using Vectors in Programs

To implement our `lap` procedure, we'll keep its state information, the lap counts, in a vector. We'll use the car number as the index into the vector. It's not enough to create the vector; we have to make sure that each box has a zero as its initial value.

```
(define *lap-vector* (make-vector 100))

(define (initialize-lap-vector index)
  (if (< index 0)
      'done
      (begin (vector-set! *lap-vector* index 0)
             (initialize-lap-vector (- index 1)))))

> (initialize-lap-vector 99)
DONE
```

We've created a global variable whose value is the vector. We used a recursive procedure to put a zero into each box of the vector.* Note that the vector is of length 100, but its largest index is 99. Also, the base case of the recursion is that the index is less than zero, not equal to zero as in many earlier examples. That's because zero is a valid index.

* In some versions of Scheme, `make-vector` can take an optional argument specifying an initial value to put in every box. In those versions, we could just say

```
(define *lap-vector* (make-vector 100 0))
```

without having to use the initialization procedure.

Now that we have the vector, we can write `lap`.

```
(define (lap car-number)
  (vector-set! *lap-vector*
               car-number
               (+ (vector-ref *lap-vector* car-number) 1))
  (vector-ref *lap-vector* car-number))
```

Remember that a procedure body can include more than one expression. When the procedure is invoked, the expressions will be evaluated in order. The value returned by the procedure is the value of the last expression (in this case, the second one).

`Lap` has both a return value and a side effect. The job of the first expression is to carry out that side effect, that is, to add 1 to the lap count for the specified car. The second expression looks at the value we just put in a box to determine the return value.

Non-Functional Procedures and State

We remarked earlier that `lap` isn't a function because invoking it twice with the same argument doesn't return the same value both times.*

It's not a coincidence that `lap` also violates functional programming by maintaining state information. Any procedure whose return value is not a function of its arguments (that is, whose return value is not always the same for any particular arguments) must depend on knowledge of what has happened in the past. After all, computers don't pull results out of the air; if the result of a computation doesn't depend entirely on the arguments we give, then it must depend on some other information available to the program.

Suppose somebody asks you, "Car 54 has just completed a lap; how many has it completed in all?" You can't answer that question with only the information in the question itself; you have to remember earlier events in the race. By contrast, if someone asks you, "What's the plural of 'book'?" what has happened in the past doesn't matter at all.

The connection between non-functional procedures and state also applies to non-functional Scheme primitives. The `read` procedure, for example, returns different results when you invoke it repeatedly with the same argument because it remembers how

* That's what we mean by "non-functional," not that it doesn't work!

far it's gotten in the file. That's why the argument is a port instead of a file name: A port is an abstract data type that includes, among other things, this piece of state. (If you're reading from the keyboard, the state is in the person doing the typing.)

A more surprising example is the `random` procedure that you met in Chapter 2. `Random` isn't a function because it doesn't always return the same value when called with the same argument. How does `random` compute its result? Some versions of `random` compute a number that's based on the current time (in tiny units like milliseconds so you don't get the same answer from two calls in quick succession). How does your computer know the time? Every so often some procedure (or some hardware device) adds 1 to a remembered value, the number of milliseconds since midnight. That's state, and `random` relies on it.

The most commonly used algorithm for random numbers is a little trickier; each time you invoke `random`, the result is a function of *the result from the last time you invoked it.* (The procedure is pretty complicated; typically the old number is multiplied by some large, carefully chosen constant, and only the middle digits of the product are kept.) Each time you invoke `random`, the returned value is stashed away somehow so that the next invocation can remember it. That's state too.

Just because a procedure remembers something doesn't necessarily make it stateful. *Every* procedure remembers the arguments with which it was invoked, while it's running. Otherwise the arguments wouldn't be able to affect the computation. A procedure whose result depends only on its arguments (the ones used in the current invocation) is functional. The procedure is non-functional if it depends on something outside of its current arguments. It's that sort of "long-term" memory that we consider to be state.

In particular, a procedure that uses `let` isn't stateful merely because the body of the `let` remembers the values of the variables created by the `let`. Once `let` returns a value, the variables that it created no longer exist. You couldn't use `let`, for example, to carry out the kind of remembering that `random` needs. `Let` doesn't remember a value *between* invocations, just during a single invocation.

Shuffling a Deck

One of the advantages of the vector data structure is that it allows elements to be rearranged. As an example, we'll create and shuffle a deck of cards.

We'll start with a procedure `card-list` that returns a list of all the cards, in standard order:

```
(define (card-list)
  (reduce append
          (map (lambda (suit) (map (lambda (rank) (word suit rank))
                                   '(a 2 3 4 5 6 7 8 9 10 j q k)))
               '(h s d c))))

> (card-list)
(HA H2 H3 H4 H5 H6 H7 H8 H9 H10 HJ HQ HK
 SA S2 S3 S4 S5 S6 S7 S8 S9 S10 SJ SQ SK
 DA D2 D3 D4 D5 D6 D7 D8 D9 D10 DJ DQ DK
 CA C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK)
```

In writing `card-list`, we need `reduce append` because the result from the outer invocation of `map` is a list of lists: `((HA H2 ...) (SA ...) ...)`.*

Each time we want a new deck of cards, we start with this list of 52 cards, copy the list into a vector, and shuffle that vector. We'll use the Scheme primitive `list->vector`, which takes a list as argument and returns a vector of the same length, with the boxes initialized to the corresponding elements of the list. (There is also a procedure `vector->list` that does the reverse. The characters `->` in these function names are

---

* We could get around this problem in a different way:

```
(define (card-list)
  (every (lambda (suit) (every (lambda (rank) (word suit rank))
                               '(a 2 3 4 5 6 7 8 9 10 j q k)))
         '(h s d c)))
```

In this version, we're taking advantage of the fact that our sentence data type was defined in a way that prevents the creation of sublists. A sentence of cards is a good representation for the deck. However, with this approach we are mixing up the list and sentence data types, because later we're going to invoke `list->vector` with this deck of cards as its argument. If we use sentence tools such as `every` to create the deck, then the procedure `card-list` should really be called `card-sentence`.

What difference does it make? The `every` version works fine, as long as sentences are implemented as lists, so that `list->vector` can be applied to a sentence. But the point about abstract data types such as sentences is to avoid making assumptions about their implementation. If for some reason we decided to change the internal representation of sentences, then `list->vector` could no longer be applied to a sentence. Strictly speaking, if we're going to use this trick, we need a separate conversion procedure `sentence->vector`.

Of course, if you don't mind a little typing, you can avoid this whole issue by having a quoted list of all 52 cards built into the definition of `card-list`.

meant to look like an arrow (→); this is a Scheme convention for functions that convert information from one data type to another.)

```
(define (make-deck)
  (shuffle! (list->vector (card-list)) 51))

(define (shuffle! deck index)
  (if (< index 0)
      deck
      (begin (vector-swap! deck index (random (+ index 1)))
             (shuffle! deck (- index 1)))))

(define (vector-swap! vector index1 index2)
  (let ((temp (vector-ref vector index1)))
    (vector-set! vector index1 (vector-ref vector index2))
    (vector-set! vector index2 temp)))
```

Now, each time we call `make-deck`, we get a randomly shuffled vector of cards:

```
> (make-deck)
#(C4 SA C7 DA S4 D9 SQ H4 C10 D5 H9 S10 D6
  S9 CA C9 S2 H7 S5 H6 D7 HK S7 C3 C2 C6
  HJ SK CQ CJ D4 SJ D8 S8 HA C5 DK D3 HQ
  D10 H8 DJ C8 H2 H5 H3 CK S3 DQ S6 D2 H10)

> (make-deck)
#(CQ H7 D10 D5 S8 C7 H10 SQ H4 H3 D8 C9 S7
  SK DK S6 DA D4 C6 HQ D6 S2 H5 CA H2 HJ
  CK D7 H6 HA CJ C4 SJ HK SA C2 D2 S4 DQ
  S5 C10 H9 D9 C5 D3 DJ C3 S9 S3 C8 S10 H8)
```

How does the shuffling algorithm work? Conceptually it's not complicated, but there are some implementation details that make the actual procedures a little tricky. The general idea is this: We want all the cards shuffled into a random order. So we choose any card at random, and make it the first card. We're then left with a one-card-smaller deck to shuffle, and we do that by recursion. (This algorithm is similar to selection sort from Chapter 15, except that we select a random card each time instead of selecting the smallest value.)

The details that complicate this algorithm have to do with the fact that we're using a vector, in which it's easy to change the value in one particular position, but it's not easy to do what would otherwise be the most natural thing: If you had a handful of actual cards and wanted to move one of them to the front, you'd slide the other cards over to make

room. There's no "sliding over" in a vector. Instead we use a trick; we happen to have an empty slot, the one from which we removed the randomly chosen card, so instead of moving several cards, we just move the one card that was originally at the front into that slot. In other words, we exchange two cards, the randomly chosen one and the one that used to be in front.

Second, there's nothing comparable to `cdr` to provide a one-card-smaller vector to the recursive invocation. Instead, we must use the entire vector and also provide an additional `index` argument, a number that keeps track of how many cards remain to be shuffled. It's simplest if each recursive invocation is responsible for the range of cards from position `0` to position `index` of the vector, and therefore the program actually moves each randomly selected card to the *end* of the remaining portion of the deck.

More Vector Tools

If you want to make a vector with only a few boxes, and you know in advance what values you want in those boxes, you can use the constructor `vector`. Like `list`, it takes any number of arguments and returns a vector containing those arguments as elements:

```
> (define beatles (vector 'john 'paul 'george 'pete))

> (vector-set! beatles 3 'ringo)

> beatles
#(JOHN PAUL GEORGE RINGO)
```

The procedure `vector-length` takes a vector as argument and returns the number of boxes in the vector.

```
> (vector-length beatles)
4
```

The predicate `equal?`, which we've used with words and lists, also accepts vectors as arguments. Two vectors are equal if they are the same size and all their corresponding elements are equal. (A list and a vector are never equal, even if their elements are equal.)

Finally, the predicate `vector?` takes anything as argument and returns `#t` if and only if its argument is a vector.

The Vector Pattern of Recursion

Here are two procedures that you've seen earlier in this chapter, which do something to each element of a vector:

```
(define (initialize-lap-vector index)
  (if (< index 0)
      'done
      (begin (vector-set! *lap-vector* index 0)
             (initialize-lap-vector (- index 1)))))

(define (shuffle! deck index)
  (if (< index 0)
      deck
      (begin (vector-swap! deck index (random (+ index 1)))
             (shuffle! deck (- index 1)))))
```

These procedures have a similar structure, like the similarities we found in other recursive patterns. Both of these procedures take an index as an argument, and both have

```
(< index 0)
```

as their base case. Also, both have, as their recursive case, a `begin` in which the first action does something to the vector element selected by the current index, and the second action is a recursive call with the index decreased by one. These procedures are initially called with the largest possible index value.

In some cases it's more convenient to count the index upward from zero:

```
(define (list->vector lst)
  (l->v-helper (make-vector (length lst)) lst 0))

(define (l->v-helper vec lst index)
  (if (= index (vector-length vec))
      vec
      (begin (vector-set! vec index (car lst))
             (l->v-helper vec (cdr lst) (+ index 1)))))
```

Since lists are naturally processed from left to right (using `car` and `cdr`), this program must process the vector from left to right also.

Vectors versus Lists

Since we introduced vectors to provide mutability, you may have the impression that mutability is the main difference between vectors and lists. Actually, lists are mutable too, although the issues are more complicated; that's why we haven't used list mutation in this book.

The most important difference between lists and vectors is that each kind of aggregate lends itself to a different style of programming, because some operations are faster than others in each. List programming is characterized by two operations: dividing a list into its first element and all the rest, and sticking one new element onto the front of a list. Vector programming is characterized by selecting elements in any order, from a collection whose size is set permanently when the vector is created.

To make these rather vague descriptions more concrete, here are two procedures, one of which squares every number in a list, and the other of which squares every number in a vector:

```
(define (list-square numbers)
  (if (null? numbers)
      '()
      (cons (square (car numbers))
            (list-square (cdr numbers)))))

(define (vector-square numbers)
  (vec-sq-helper (make-vector (vector-length numbers))
                 numbers
                 (- (vector-length numbers) 1)))

(define (vec-sq-helper new old index)
  (if (< index 0)
      new
      (begin (vector-set! new index (square (vector-ref old index)))
             (vec-sq-helper new old (- index 1)))))
```

In the list version, the intermediate stages of the algorithm deal with lists that are smaller than the original argument. Each recursive invocation "strips off" one element of its argument and "glues on" one extra element in its return value. In the vector version, the returned vector is created, at full size, as the first step in the algorithm; its component parts are filled in as the program proceeds.

This example can plausibly be done with either vectors or lists, so we've used it to compare the two techniques. But some algorithms fit most naturally with one kind of

aggregate and would be awkward and slow using the other kind. The swapping of pairs of elements in the shuffling algorithm would be much harder using lists, while mergesort would be harder using vectors.

The best way to understand these differences in style is to know the operations that are most efficient for each kind of aggregate. In each case, there are certain operations that can be done in one small unit of time, regardless of the number of elements in the aggregate, while other operations take more time for more elements. The *constant time* operations for lists are `cons`, `car`, `cdr`, and `null?`; the ones for vectors are `vector-ref`, `vector-set!`, and `vector-length`.* And if you reread the squaring programs, you'll find that these are precisely the operations they use.

We might have used `list-ref` in the list version, but we didn't, and Scheme programmers usually don't, because we know that it would be slower. Similarly, we could implement something like `cdr` for vectors, but that would be slow, too, since it would have to make a one-smaller vector and copy the elements one at a time. There are two possible morals to this story, and they're both true: First, programmers invent and learn the algorithms that make sense for whatever data structure is available. Thus we have well-known programming patterns, such as the `filter` pattern, appropriate for lists, and different patterns appropriate for vectors. Second, programmers choose which data structure to use depending on what algorithms they need. If you want to shuffle cards, use a vector, but if you want to split the deck into a bunch of variable-size piles, lists might be more appropriate. In general, vectors are good at selecting elements in arbitrary order from a fixed-size collection; lists are good only at selecting elements strictly from left to right, but they can vary in size.

In this book, despite what we're saying here about efficiency, we've generally tried to present algorithms in the way that's easiest to understand, even when we know that there's a faster way. For example, we've shown several recursive procedures in which the base case test was

```
(= (count sent) 1)
```

If we were writing the program for practical use, rather than for a book, we would have written

```
(empty? (butfirst sent))
```

---

* Where did this information come from? Just take our word for it. In later courses you'll study how vectors and lists are implemented, and then there will be reasons.

because we know that `empty?` and `butfirst` are both constant time operations (because for sentences they're implemented as `null?` and `cdr`), while `count` takes a long time for large sentences. But the version using `count` makes the intent clearer.*

State, Sequence, and Effects

Effects, sequence, and state are three sides of the same coin.**

In Chapter 20 we explained the connection between effect (printing something on the screen) and sequence: It matters what you print first. We also noted that there's no benefit to a sequence of expressions unless those expressions produce an effect, since the values returned by all but the last expression are discarded.

In this chapter we've seen another connection. The way our vector programs maintain state information is by carrying out effects, namely, `vector-set!` invocations. Actually, *every* effect changes some kind of state; if not in Scheme's memory, then on the computer screen or in a file.

The final connection to be made is between state and sequence. Once a program maintains state, it matters whether some computation is carried out before or after another computation that changes the state. The example at the beginning of this chapter in which an expression had different results before and after defining a variable illustrates this point. As another example, if we evaluate `(lap 1)` 200 times and `(lap 2)` 200 times, the program's determination of the winner of the race depends on whether the last evaluation of `(lap 1)` comes before or after the last invocation of `(lap 2)`.

Because these three ideas are so closely connected, the names *sequential programming* (emphasizing sequence) and *imperative programming* (emphasizing effect) are both used to refer to a style of programming that uses all three. This style is in contrast with functional programming, which, as you know, uses none of them.

Although functional and sequential programming are, in a sense, opposites, it's perfectly possible to use both styles within one program, as we pointed out in the tic-tac-toe program of Chapter 20. We'll show more such hybrid programs in the following chapters.

---

* For words, it turns out, the `count` version is faster, because words behave more like vectors than like lists.

** . . . to coin a phrase.

Pitfalls

⇒ Don't forget that the first element of a vector is number zero, and there is no element whose index number is equal to the length of the vector. (Although these points are equally true for lists, it doesn't often matter, because we rarely select the elements of a list by number.) In particular, in a vector recursion, if zero is the base case, then there's probably still one element left to process.

⇒ Try the following experiment:

```
> (define dessert (vector 'chocolate 'sundae))
> (define two-desserts (list dessert dessert))
> (vector-set! (car two-desserts) 1 'shake)
> two-desserts
(#(CHOCOLATE SHAKE) #(CHOCOLATE SHAKE))
```

You might have expected that after asking to change one word in `two-desserts`, the result would be

```
(#(CHOCOLATE SHAKE) #(CHOCOLATE SUNDAE))
```

However, because of the way we created `two-desserts`, both of its elements are the *same* vector. If you think of a list as a collection of things, it's strange to imagine the very same thing in two different places, but that's the situation. If you want to have two separate vectors that happen to have the same values in their elements, but are individually mutable, you'd have to say

```
> (define two-desserts (list (vector 'chocolate 'sundae)
                             (vector 'chocolate 'sundae)))
> (vector-set! (car two-desserts) 1 'shake)
> two-desserts
(#(CHOCOLATE SHAKE) #(CHOCOLATE SUNDAE))
```

Each invocation of `vector` or `make-vector` creates a new, independent vector.

Exercises

Do not solve any of the following exercises by converting a vector to a list, using list procedures, and then converting the result back to a vector.

23.1 Write a procedure `sum-vector` that takes a vector full of numbers as its argument and returns the sum of all the numbers:

```
> (sum-vector '#(6 7 8))
21
```

23.2 Some versions of Scheme provide a procedure `vector-fill!` that takes a vector and anything as its two arguments. It replaces every element of the vector with the second argument, like this:

```
> (define vec (vector 'one 'two 'three 'four))

> vec
#(one two three four)

> (vector-fill! vec 'yeah)

> vec
#(yeah yeah yeah yeah)
```

Write `vector-fill!`. (It doesn't matter what value it returns.)

23.3 Write a function `vector-append` that works just like regular `append`, but for vectors:

```
> (vector-append '#(not a) '#(second time))
#(not a second time)
```

23.4 Write `vector->list`.

23.5 Write a procedure `vector-map` that takes two arguments, a function and a vector, and returns a new vector in which each box contains the result of applying the function to the corresponding element of the argument vector.

23.6 Write a procedure `vector-map!` that takes two arguments, a function and a vector, and modifies the argument vector by replacing each element with the result of applying the function to that element. Your procedure should return the same vector.

23.7 Could you write `vector-filter`? How about `vector-filter!`? Explain the issues involved.

23.8 Modify the `lap` procedure to print "Car 34 wins!" when car 34 completes its 200th lap. (A harder but more correct modification is to print the message only if no other car has completed 200 laps.)

23.9 Write a procedure `leader` that says which car is in the lead right now.

23.10 Why doesn't this solution to Exercise 23.9 work?

```
(define (leader)
  (leader-helper 0 1))

(define (leader-helper leader index)
  (cond ((= index 100) leader)
        ((> (lap index) (lap leader))
         (leader-helper index (+ index 1)))
        (else (leader-helper leader (+ index 1)))))
```

23.11 In some restaurants, the servers use computer terminals to keep track of what each table has ordered. Every time you order more food, the server enters your order into the computer. When you're ready for the check, the computer prints your bill.

You're going to write two procedures, `order` and `bill`. `Order` takes a table number and an item as arguments and adds the cost of that item to that table's bill. `Bill` takes a table number as its argument, returns the amount owed by that table, and resets the table for the next customers. (Your `order` procedure can examine a global variable `*menu*` to find the price of each item.)

```
> (order 3 'potstickers)

> (order 3 'wor-won-ton)

> (order 5 'egg-rolls)

> (order 3 'shin-shin-special-prawns)

> (bill 3)
13.85

> (bill 5)
2.75
```

23.12 Rewrite selection sort (from Chapter 15) to sort a vector. This can be done in a way similar to the procedure for shuffling a deck: Find the smallest element of the vector and exchange it (using `vector-swap!`) with the value in the first box. Then find the smallest element not including the first box, and exchange that with the second box, and so on. For example, suppose we have a vector of numbers:

```
#(23 4 18 7 95 60)
```

Your program should transform the vector through these intermediate stages:

```
#(4 23 18 7 95 60)   ; exchange 4 with 23
#(4 7 18 23 95 60)   ; exchange 7 with 23
#(4 7 18 23 95 60)   ; exchange 18 with itself
#(4 7 18 23 95 60)   ; exchange 23 with itself
#(4 7 18 23 60 95)   ; exchange 60 with 95
```

23.13 Why doesn't this work?

```
(define (vector-swap! vector index1 index2)
  (vector-set! vector index1 (vector-ref vector index2))
  (vector-set! vector index2 (vector-ref vector index1)))
```

23.14 Implement a two-dimensional version of vectors. (We'll call one of these structures a *matrix*.) The implementation will use a vector of vectors. For example, a three-by-five matrix will be a three-element vector, in which each of the elements is a five-element vector. Here's how it should work:

```
> (define m (make-matrix 3 5))

> (matrix-set! m 2 1 '(her majesty))

> (matrix-ref m 2 1)
(HER MAJESTY)
```

23.15 Generalize Exercise 23.14 by implementing an *array* structure that can have any number of dimensions. Instead of taking two numbers as index arguments, as the matrix procedures do, the array procedures will take one argument, a *list* of numbers. The number of numbers is the number of dimensions, and it will be constant for any particular array. For example, here is a three-dimensional array ($4 \times 5 \times 6$):

```
> (define a1 (make-array '(4 5 6)))

> (array-set! a1 '(3 2 3) '(the end))
```

23.16 We want to reimplement sentences as vectors instead of lists.

(a) Write versions of `sentence`, `empty?`, `first`, `butfirst`, `last`, and `butlast` that use vectors. Your selectors need only work for sentences, not for words.

```
> (sentence 'a 'b 'c)
#(A B C)

> (butfirst (sentence 'a 'b 'c))
#(B C)
```

(You don't have to make these procedures work on lists as well as vectors!)

(b) Does the following program still work with the new implementation of sentences? If not, fix the program.

```
(define (praise stuff)
  (sentence stuff '(is good)))
```

(c) Does the following program still work with the new implementation of sentences? If not, fix the program.

```
(define (praise stuff)
  (sentence stuff 'rules!))
```

(d) Does the following program still work with the new implementation of sentences? If not, fix the program. If so, is there some optional rewriting that would improve its performance?

```
(define (item n sent)
  (if (= n 1)
      (first sent)
      (item (- n 1) (butfirst sent))))
```

(e) Does the following program still work with the new implementation of sentences? If not, fix the program. If so, is there some optional rewriting that would improve its performance?

```
(define (every fn sent)
  (if (empty? sent)
      sent
      (sentence (fn (first sent))
                (every fn (butfirst sent)))))
```

(f) In what ways does using vectors to implement sentences affect the speed of the selectors and constructor? Why do you think we chose to use lists?