Part VI
Sequential Programming

The three big ideas in this part are *effect, sequence,* and *state.*

Until now, we've been doing functional programming, where the focus is on functions and their return values. Invoking a function is like asking a question: "What's two plus two?" In this part of the book we're going to talk about giving commands to the computer as well as asking it questions. That is, we'll invoke procedures that tell Scheme to *do* something, such as `wash-the-dishes`. (Unfortunately, the Scheme standard leaves out this primitive.) Instead of merely computing a value, such a procedure has an *effect,* an action that changes something.

Once we're thinking about actions, it's very natural to consider a *sequence* of actions. First cooking dinner, then eating, and then washing the dishes is one sequence. First eating, then washing the dishes, and then cooking is a much less sensible sequence.

Although these ideas of sequence and effect are coming near the end of our book, they're the ideas with which almost every introduction to programming begins. Most books compare a program to a recipe or a sequence of instructions, along the lines of

```
to go-to-work
  get-dressed
  eat-breakfast
  catch-the-bus
```

This sequential programming style is simple and natural, and it does a good job of modeling computations in which the problem concerns a sequence of events. If you're writing an airline reservation system, a sequential program with `reserve-seat` and `issue-ticket` commands makes sense. But if you want to know the acronym of a phrase, that's not inherently sequential, and a question-asking approach is best.

Some actions that Scheme can take affect the "outside" world, such as printing something on the computer screen. But Scheme can also carry out internal actions, invisible outside the computer, but changing the environment in which Scheme itself carries out computations. Defining a new variable with `define` is an example; before the definition, Scheme wouldn't understand what that name means, but once the definition has been made, the name can be used in evaluating later expressions. Scheme's knowledge about the leftover effects of past computations is called its *state*. The third big idea in this part of the book is that we can write programs that maintain state information and use it to determine their results.

Like sequence, the notion of state contradicts functional programming. Earlier in the book, we emphasized that every time a function is invoked with the same arguments, it must return the same value. But a procedure whose returned value depends on state—on the past history of the computation—might return a different value on each invocation, even with identical arguments.

We'll explore several situations in which effects, sequence, and state are useful:

- Interactive, question-and-answer programs that involve keyboard input while the computation is in progress;
- Programs that must read and write long-term data file storage;
- Computations that *model* an actual sequence of events in time and use the state of the program to model information about the state of the simulated events.

After introducing Scheme's mechanisms for sequential programming, we'll use those mechanisms to implement versions of two commonly used types of business computer applications, a spreadsheet and a database program.

20 Input and Output

In the tic-tac-toe project in Chapter 10, we didn't write a complete game program. We wrote a *function* that took a board position and `x` or `o` as arguments, returning the next move. We noted at the time that a complete game program would also need to carry on a *conversation* with the user. Instead of computing and returning one single value, a conversational program must carry out a sequence of events in time, reading information from the keyboard and displaying other information on the screen.

Before we complete the tic-tac-toe project, we'll start by exploring Scheme's mechanisms for interactive programming.

Printing

Up until now, we've never told Scheme to print anything. The programs we've written have computed values and returned them; we've relied on the read-eval-print loop to print these values.*

But let's say we want to write a program to print out all of the words to "99 Bottles of Beer on the Wall." We could implement a function to produce a humongous *list* of the lines of the song, like this:

```
(define (bottles n)
  (if (= n 0)
      '()
      (append (verse n)
              (bottles (- n 1)))))
```

* The only exception is that we've used `trace`, which prints messages about the progress of a computation.

```
(define (verse n)
  (list (cons n '(bottles of beer on the wall))
        (cons n '(bottles of beer))
        '(if one of those bottles should happen to fall)
        (cons (- n 1) '(bottles of beer on the wall))
        '()))
```

```
> (bottles 3)
((3 BOTTLES OF BEER ON THE WALL)
 (3 BOTTLES OF BEER)
 (IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
 (2 BOTTLES OF BEER ON THE WALL)
 ()
 (2 BOTTLES OF BEER ON THE WALL)
 (2 BOTTLES OF BEER)
 (IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
 (1 BOTTLES OF BEER ON THE WALL)
 ()
 (1 BOTTLES OF BEER ON THE WALL)
 (1 BOTTLES OF BEER)
 (IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
 (0 BOTTLES OF BEER ON THE WALL)
 ())
```

The problem is that we don't want a list. All we want is to print out the lines of the song; storing them in a data structure is unnecessary and inefficient. Also, some versions of Scheme would print the above list like this:

```
((3 BOTTLES OF BEER ON THE WALL) (3 BOTTLES OF BEER) (IF ONE OF
 THOSE BOTTLES SHOULD HAPPEN TO FALL) (2 BOTTLES OF BEER ON THE
 WALL) () (2 BOTTLES OF BEER ON THE WALL) (2 BOTTLES OF BEER) (IF
 ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL) (1 BOTTLES OF BEER ON
 THE WALL) () (1 BOTTLES OF BEER ON THE WALL) (1 BOTTLES OF BEER)
 (IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL) (0 BOTTLES OF BEER
 ON THE WALL) ())
```

or even all on one line. We can't rely on Scheme's mechanism for printing lists if we want to be sure of a particular arrangement on the screen.

Instead we'll write a program to *print* a verse, rather than return it in a list:

```
(define (bottles n)
  (if (= n 0)
      'burp
      (begin (verse n)
             (bottles (- n 1)))))
```

```
(define (verse n)
  (show (cons n '(bottles of beer on the wall)))
  (show (cons n '(bottles of beer)))
  (show '(if one of those bottles should happen to fall))
  (show (cons (- n 1) '(bottles of beer on the wall)))
  (show '()))
```

```
> (bottles 3)
(3 BOTTLES OF BEER ON THE WALL)
(3 BOTTLES OF BEER)
(IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
(2 BOTTLES OF BEER ON THE WALL)
()
(2 BOTTLES OF BEER ON THE WALL)
(2 BOTTLES OF BEER)
(IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
(1 BOTTLES OF BEER ON THE WALL)
()
(1 BOTTLES OF BEER ON THE WALL)
(1 BOTTLES OF BEER)
(IF ONE OF THOSE BOTTLES SHOULD HAPPEN TO FALL)
(0 BOTTLES OF BEER ON THE WALL)
()
BURP
```

Notice that Scheme doesn't print an outer set of parentheses. Each line was printed separately; there isn't one big list containing all of them.*

Why was "burp" printed at the end? Just because we're printing things explicitly doesn't mean that the read-eval-print loop stops functioning. We typed the expression `(bottles 3)`. In the course of evaluating that expression, Scheme printed several lines for us. But the *value* of the expression was the word `burp`, because that's what `bottles` returned.

Side Effects and Sequencing

How does our program work? There are two new ideas here: *side effects* and *sequencing*.

Until now, whenever we've invoked a procedure, our only goal has been to get a return value. The procedures we've used compute and return a value, and do nothing else. `Show` is different. Although every Scheme procedure returns a value, the Scheme

* We know that it's still not as beautiful as can be, because of the capital letters and parentheses, but we'll get to that later.

language standard doesn't specify what value the printing procedures should return.* Instead, we are interested in their side effects. In other words, we invoke `show` because we want it to *do* something, namely, print its argument on the screen.

What exactly do we mean by "side effect"? The kinds of procedures that we've used before this chapter can compute values, invoke helper procedures, provide arguments to the helper procedures, and return a value. There may be a lot of activity going on within the procedure, but the procedure affects the world outside of itself only by returning a value that some other procedure might use. `Show` affects the world outside of itself by putting something on the screen. After `show` has finished its work, someone who looks at the screen can tell that `show` was used.**

Here's an example to illustrate the difference between values and effects:

```
(define (effect x)
  (show x)
  'done)

(define (value x)
  x)

> (effect '(oh! darling))
(OH! DARLING)
DONE

> (value '(oh! darling))
(OH! DARLING)

> (bf (effect '(oh! darling)))
(OH! DARLING)
ONE
```

---

* Suppose `show` returns `#f` in your version of Scheme. Then you might see

```
> (show 7)
7
#F
```

But since the return value is unspecified, we try to write programs in such a way that we never use `show`'s return value as the return value from our procedures. That's why we return values like `burp`.

** The term *side* effect is based on the idea that a procedure may have a useful return value as its main purpose and may also have an effect "on the side." It's a misnomer to talk about the side effect of `show`, since the effect is its main purpose. But nobody ever says "side return value"!

```
> (bf (value '(oh! darling)))
(DARLING)

> (define (lots-of-effect x)
    (effect x)
    (effect x)
    (effect x))

> (define (lots-of-value x)
    (value x)
    (value x)
    (value x))

> (lots-of-effect '(oh! darling))
(OH! DARLING)
(OH! DARLING)
(OH! DARLING)
DONE

> (lots-of-value '(oh! darling))
(OH! DARLING)
```

This example also demonstrates the second new idea, sequencing: Each of `effect`, `lots-of-effect`, and `lots-of-value` contains more than one expression in its body. When you invoke such a procedure, Scheme evaluates all the expressions in the body, in order, and returns the value of the last one.* This also works in the body of a `let`, which is really the body of a procedure, and in each clause of a `cond`.**

---

* In Chapter 4, we said that the body of a procedure was always one single expression. We lied. But as long as you don't use any procedures with side effects, it doesn't do you any good to evaluate more than one expression in a body.

** For example:

```
> (cond ((< 4 0)
         (show '(how interesting))
         (show '(4 is less than zero?))
         #f)
        ((> 4 0)
         (show '(more reasonable))
         (show '(4 really is more than zero))
         'value)
        (else
         (show '(you mean 4=0?))
         #f))
(MORE REASONABLE)
(4 REALLY IS MORE THAN ZERO)
VALUE
```

When we invoked `lots-of-value`, Scheme invoked `value` three times; it discarded the values returned by the first two invocations, and returned the value from the third invocation. Similarly, when we invoked `lots-of-effect`, Scheme invoked `effect` three times and returned the value from the third invocation. But each invocation of `effect` caused its argument to be printed by invoking `show`.

The `Begin` Special Form

The `lots-of-effect` procedure accomplished sequencing by having more than one expression in its body. This works fine if the sequence of events that you want to perform is the entire body of a procedure. But in `bottles` we wanted to include a sequence as one of the alternatives in an `if` construction. We couldn't just say

```
(define (bottles n)                          ;; wrong
  (if (= n 0)
      '()
      (verse n)
      (bottles (- n 1))))
```

because `if` must have exactly three arguments. Otherwise, how would `if` know whether we meant `(verse n)` to be the second expression in the true case, or the first expression in the false case?

Instead, to turn the sequence of expressions into a single expression, we use the special form `begin`. It takes any number of arguments, evaluates them from left to right, and returns the value of the last one.

```
(define bottles n)
  (if (= n 0)
      'burp
      (begin (verse n)
             (bottles (- n 1)))))
```

(One way to think about sequences in procedure bodies is that every procedure body has an invisible `begin` surrounding it.)

This Isn't Functional Programming

Sequencing and side effects are radical departures from the idea of functional programming. In fact, we'd like to reserve the name *function* for something that computes and

returns one value, with no side effects. "Procedure" is the general term for the thing that `lambda` returns—an embodiment of an algorithm. If the algorithm is the kind that computes and returns a single value without side effects, then we say that the procedure implements a function.*

There is a certain kind of sequencing even in functional programming. If you say

```
(* (+ 3 4) (- 92 15))
```

it's clear that the addition has to happen before the multiplication, because the result of the addition provides one of the arguments to the multiplication. What's new in the sequential programming style is the *emphasis* on sequence, and the fact that the expressions in the sequence are *independent* instead of contributing values to each other. In this multiplication problem, for example, we don't care whether the addition happens before or after the subtraction. If the addition and subtraction were in a sequence, we'd be using them for independent purposes:

```
(begin
 (show (+ 3 4))
 (show (- 92 15)))
```

This is what we mean by being independent. Neither expression helps in computing the other. And the order matters because we can see the order in which the results are printed.

Not Moving to the Next Line

Each invocation of `show` prints a separate line. What if we want a program that prints several things on the same line, like this:

```
> (begin (show-addition 3 4)
         (show-addition 6 8)
         'done)
3+4=7
6+8=14
DONE
```

* Sometimes people sloppily say that the procedure *is* a function. In fact, you may hear people be *really* sloppy and call a non-functional procedure a function!

We use `display`, which doesn't move to the next line after printing its argument:

```
(define (show-addition x y)
  (display x)
  (display '+)
  (display y)
  (display '=)
  (show (+ x y)))
```

(The last one is a `show` because we *do* want to start a new line after it.)

What if you just want to print a blank line? You use `newline`:

```
(define (verse n)
  (show (cons n '(bottles of beer on the wall)))
  (show (cons n '(bottles of beer)))
  (show '(if one of those bottles should happen to fall))
  (show (cons (- n 1) '(bottles of beer on the wall)))
  (newline))                                  ; replaces (show '())
```

In fact, `show` isn't an official Scheme primitive; we wrote it in terms of `display` and `newline`.

Strings

Throughout the book we've occasionally used strings, that is, words enclosed in double-quote marks so that Scheme will permit the use of punctuation or other unusual characters. Strings also preserve the case of letters, so they can be used to beautify our song even more. Since *any* character can be in a string, including spaces, the easiest thing to do in this case is to treat all the letters, spaces, and punctuation characters of each line of the song as one long word. (If we wanted to be able to compute functions of the individual words in each line, that wouldn't be such a good idea.)

```
(define (verse n)
  (display n)
  (show " bottles of beer on the wall,")
  (display n)
  (show " bottles of beer.")
  (show "If one of those bottles should happen to fall,")
  (display (- n 1))
  (show " bottles of beer on the wall.")
  (newline))
```

```
> (verse 6)
6 bottles of beer on the wall,
6 bottles of beer.
If one of those bottles should happen to fall,
5 bottles of beer on the wall.

#F                                ; or whatever is returned by (newline)
```

It's strange to think of "`bottles of beer on the wall,`" as a single word. But the rule is that anything inside double quotes counts as a single word. It doesn't have to be an English word.

A Higher-Order Procedure for Sequencing

Sometimes we want to print each element of a list separately:

```
(define (show-list lst)
  (if (null? lst)
      'done
      (begin (show (car lst))
             (show-list (cdr lst)))))
```

```
> (show-list '((dig a pony) (doctor robert) (for you blue)))
(DIG A PONY)
(DOCTOR ROBERT)
(FOR YOU BLUE)
DONE
```

Like other patterns of computation involving lists, this one can be abstracted into a higher-order procedure. (We can't call it a "higher-order function" because this one is for computations with side effects.) The procedure `for-each` is part of standard Scheme:

```
> (for-each show '((mean mr mustard) (no reply) (tell me why)))
(MEAN MR MUSTARD)
(NO REPLY)
(TELL ME WHY)
```

The value returned by `for-each` is unspecified.

Why couldn't we just use `map` for this purpose? There are two reasons. One is just an efficiency issue: `Map` constructs a list containing the values returned by each of its sub-computations; in this example, it would be a list of three instances of the unspecified value returned by `show`. But we aren't going to use that list for anything, so there's no point in constructing it. The second reason is more serious. In functional programming, the order of evaluation of subexpressions is unspecified. For example, when we evaluate the expression

```
(- (+ 4 5) (* 6 7))
```

we don't know whether the addition or the multiplication happens first. Similarly, the order in which `map` computes the results for each element is unspecified. That's okay as long as the ultimately returned list of results is in the right order. But when we are using side effects, we *do* care about the order of evaluation. In this case, we want to make sure that the elements of the argument list are printed from left to right. `For-each` guarantees this ordering.

Tic-Tac-Toe Revisited

We're working up toward playing a game of tic-tac-toe against the computer. But as a first step, let's have the computer play against itself. What we already have is `ttt`, a *strategy* function: one that takes a board position as argument (and also a letter `x` or `o`) and returns the chosen next move. In order to play a game of tic-tac-toe, we need two players; to make it more interesting, each should have its own strategy. So we'll write another one, quickly, that just moves in the first empty square it sees:

```
(define (stupid-ttt position letter)
  (location '_ position))

(define (location letter word)
  (if (equal? letter (first word))
      1
      (+ 1 (location letter (bf word)))))
```

Now we can write a program that takes two strategies as arguments and actually plays a game between them.

```
(define (play-ttt x-strat o-strat)
  (play-ttt-helper x-strat o-strat '_________ 'x))

(define (play-ttt-helper x-strat o-strat position whose-turn)
  (cond ((already-won? position (opponent whose-turn))
         (list (opponent whose-turn) 'wins!))
        ((tie-game? position) '(tie game))
        (else (let ((square (if (equal? whose-turn 'x)
                                (x-strat position 'x)
                                (o-strat position 'o))))
                (play-ttt-helper x-strat
                                 o-strat
                                 (add-move square whose-turn position)
                                 (opponent whose-turn))))))
```

We use a helper procedure because we need to keep track of two pieces of information besides the strategy procedures: the current board position and whose turn it is (`x` or `o`). The helper procedure is invoked recursively for each move. First it checks whether the game is already over (won or tied).* If not, the helper procedure invokes the current player's strategy procedure, which returns the square number for the next move. For the recursive call, the arguments are the same two strategies, the new position after the move, and the letter for the other player.

We still need `add-move`, the procedure that takes a square and an old position as arguments and returns the new position.

```
(define (add-move square letter position)
  (if (= square 1)
      (word letter (bf position))
      (word (first position)
            (add-move (- square 1) letter (bf position)))))

> (play-ttt ttt stupid-ttt)
(X WINS!)

> (play-ttt stupid-ttt ttt)
(O WINS!)
```

Accepting User Input

The work we did in the last section was purely functional. We didn't print anything (except the ultimate return value, as always) and we didn't have to read information from a human player, because there wasn't one.

You might expect that the structure of an *interactive* game program would be very different, with a top-level procedure full of sequential operations. But the fact is that we hardly have to change anything to turn this into an interactive game. All we need is a

* You wrote the procedures `already-won?` and `tie-game?` in Exercises 10.1 and 10.2:

```
(define (already-won? position who)
  (member? (word who who who)  (find-triples position)))

(define (tie-game? position)
  (not (member? '_ position)))
```

new “strategy” procedure that asks the user where to move, instead of computing a move based on built-in rules.

```
(define (ask-user position letter)
  (print-position position)
  (display letter)
  (display "'s move: ")
  (read))

(define (print-position position)            ;; first version
  (show position))
```

(Ultimately we're going to want a beautiful two-dimensional display of the current position, but we don't want to get distracted by that just now. That's why we've written a trivial temporary version.)

```
> (play-ttt ttt ask-user)
____X____
O'S MOVE: 1
O___XX___
O'S MOVE: 4
O__OXXX__
O'S MOVE: 3
OXOOXXX__
O'S MOVE: 8
(TIE GAME)
```

What the user typed is just the single digits shown in boldface at the ends of the lines.

What's new here is that we invoke the procedure `read`. It waits for you to type a Scheme expression, and returns that expression. Don't be confused: `Read` does *not* evaluate what you type. It returns exactly the same expression that you type:

```
(define (echo)
  (display "What? ")
  (let ((expr (read)))
    (if (equal? expr 'stop)
        'okay
        (begin
         (show expr)
         (echo)))))
```

```
> (echo)
What? hello
HELLO
What? (+ 2 3)
(+ 2 3)
What? (first (glass onion))
(FIRST (GLASS ONION))
What? stop
OKAY
```

Aesthetic Board Display

Here's our beautiful position printer:

```
(define (print-position position)
  (print-row (subword position 1 3))
  (show "-+-+-")
  (print-row (subword position 4 6))
  (show "-+-+-")
  (print-row (subword position 7 9))
  (newline))

(define (print-row row)
  (maybe-display (first row))
  (display "|")
  (maybe-display (first (bf row)))
  (display "|")
  (maybe-display (last row))
  (newline))

(define (maybe-display letter)
  (if (not (equal? letter '_))
      (display letter)
      (display " ")))

(define (subword wd start end)
  ((repeated bf (- start 1))
   ((repeated bl (- (count wd) end))
    wd)))*
```

* Alternate version:

```
(define (subword wd start end)
  (cond ((> start 1) (subword (bf wd) (- start 1) (- end 1)))
        ((< end (count wd)) (subword (bl wd) start end))
        (else wd)))
```

You can take your choice, depending on which you think is easier, recursion or higher-order functions.

Here's how it works:

```
> (print-position '_x_oo__xx)
 |X|
-+-+-
O|O|
-+-+-
 |X|X
```

Reading and Writing Normal Text

The `read` procedure works fine as long as what you type looks like a Lisp program. That is, it reads one expression at a time. In the tic-tac-toe program the user types a single number, which is a Scheme expression, so `read` works fine. But what if we want to read more than one word?

```
(define (music-critic)                          ;; first version
  (show "What's your favorite Beatles song?")
  (let ((song (read)))
    (show (se "I like" song "too."))))

> (music-critic)
What's your favorite Beatles song?
She Loves You
(I like SHE too.)
```

If the user had typed the song title in parentheses, then it would have been a single Scheme expression and `read` would have accepted it. But we don't want the users of our program to have to be typing parentheses all the time.

Scheme also lets you read one character at a time. This allows you to read any text, with no constraints on its format. The disadvantage is that you find yourself putting a lot of effort into minor details. We've provided a procedure `read-line` that reads one line of input and returns a sentence. The words in that sentence will contain any punctuation characters that appear on the line, including parentheses, which are not interpreted as sublist delimiters by `read-line`. `Read-line` also preserves the case of letters.

```
(define (music-critic)                          ;; second version
  (read-line)   ; See explanation on next page.
  (show "What's your favorite Beatles song?")
  (let ((song (read-line)))
    (show (se "I like" song "too."))))
```

```
> (music-critic)
What's your favorite Beatles song?
She Loves You
(I like She Loves You too.)
```

Why do we call `read-line` and ignore its result at the beginning of `music-critic`? It has to do with the interaction between `read-line` and `read`. `Read` treats what you type as a sequence of Scheme expressions; each invocation of `read` reads one of them. `Read` pays no attention to formatting details, such as several consecutive spaces or line breaks. If, for example, you type several expressions on the same line, it will take several invocations of `read` to read them all.

By contrast, `read-line` treats what you type as a sequence of lines, reading one line per invocation, so it does pay attention to line breaks.

Either of these ways to read input is sensible in itself, but if you mix the two, by invoking `read` sometimes and `read-line` sometimes in the same program, the results can be confusing. Suppose you type a line containing an expression and your program invokes `read` to read it. Since there might have been another expression on the line, `read` doesn't advance to the next line until you ask for the next expression. So if you now invoke `read-line`, thinking that it will read another line from the keyboard, it will instead return an empty list, because what it sees is an empty line—what's left after `read` uses up the expression you typed.

You may be thinking, "But `music-critic` doesn't call `read`!" That's true, but Scheme itself used `read` to read the expression that you used to invoke `music-critic`. So the first invocation of `read-line` is needed to skip over the spurious empty line.

Our solution works only if `music-critic` is invoked directly at a Scheme prompt. If `music-critic` were a subprocedure of some larger program that has already called `read-line` before calling `music-critic`, the extra `read-line` in `music-critic` would really read and ignore a useful line of text.

If you write a procedure using `read-line` that will sometimes be called directly and sometimes be used as a subprocedure, you can't include an extra `read-line` call in it. Instead, when you call your procedure directly from the Scheme prompt, you must say

```
> (begin (read-line) (my-procedure))
```

Another technical detail about `read-line` is that since it preserves the capitalization of words, its result may include strings, which will be shown in quotation marks if you return the value rather than `show`ing it:

```
(define (music-critic-return)
  (read-line)
  (show "What's your favorite Beatles song?")
  (let ((song (read-line)))
    (se "I like" song "too.")))

> (music-critic-return)
What's your favorite Beatles song?
She Loves You
("I like" "She" "Loves" "You" "too.")
```

We have also provided `show-line,` which takes a sentence as argument. It prints the sentence without surrounding parentheses, followed by a newline. (Actually, it takes any list as argument; it prints all the parentheses except for the outer ones.)

```
(define (music-critic)
  (read-line)
  (show "What's your favorite Beatles song?")
  (let ((song (read-line)))
    (show-line (se "I like" song "too."))))

> (music-critic)
What's your favorite Beatles song?
She Loves You
I like She Loves You too.
```

The difference between `show` and `show-line` isn't crucial. It's just a matter of a pair of parentheses. The point is that `read-line` and `show-line` go together. `Read-line` reads a bunch of disconnected words and combines them into a sentence. `Show-line` takes a sentence and prints it as if it were a bunch of disconnected words. Later, when we read and write files in Chapter 22, this ability to print in the same form in which we read will be important.

Formatted Text

We've been concentrating on the use of sequential programming with explicit printing instructions for the sake of conversational programs. Another common application of sequential printing is to display tabular information, such as columns of numbers. The difficulty is to get the numbers to line up so that corresponding digits are in the same position, even when the numbers have very widely separated values. The `align` function

can be used to convert a number to a printable word with a fixed number of positions before and after the decimal point:

```
(define (square-root-table nums)
  (if (null? nums)
      'done
      (begin (display (align (car nums) 7 1))
             (show (align (sqrt (car nums)) 10 5))
             (square-root-table (cdr nums)))))

> (square-root-table '(7 8 9 10 20 98 99 100 101 1234 56789))
    7.0   2.64575
    8.0   2.82843
    9.0   3.00000
   10.0   3.16228
   20.0   4.47214
   98.0   9.89949
   99.0   9.94987
  100.0  10.00000
  101.0  10.04988
 1234.0  35.12834
56789.0 238.30443
DONE
```

`Align` takes three arguments. The first is the value to be displayed. The second is the width of the column in which it will be displayed; the returned value will be a word with that many characters in it. The third argument is the number of digits that should be displayed to the right of the decimal point. (If this number is zero, then no decimal point will be displayed.) The width must be great enough to include all the digits, as well as the decimal point and minus sign, if any.

As the program example above indicates, `align` does not print anything. It's a function that returns a value suitable for printing with `display` or `show`.

What if the number is too big to fit in the available space?

```
> (align 12345679 4 0)
"123+"
```

`Align` returns a word containing the first few digits, as many as fit, ending with a plus sign to indicate that part of the value is missing.

`Align` can also be used to include non-numeric text in columns. If the first argument is not a number, then only two arguments are needed; the second is the column width.

In this case `align` returns a word with extra spaces at the right, if necessary, so that the argument word will appear at the left in its column:

```
(define (name-table names)
  (if (null? names)
      'done
      (begin (display (align (cadar names) 11))
             (show (caar names))
             (name-table (cdr names)))))

> (name-table '((john lennon) (paul mccartney)
                (george harrison) (ringo starr)))
LENNON     JOHN
MCCARTNEY  PAUL
HARRISON   GEORGE
STARR      RINGO
DONE
```

As with numbers, if a non-numeric word won't fit in the allowed space, `align` returns a partial word ending with a plus sign.

This `align` function is not part of standard Scheme. Most programming languages, including some versions of Scheme, offer much more elaborate formatting capabilities with many alternate ways to represent both numbers and general text. Our version is a minimal capability to show the flavor and to meet the needs of projects in this book.

Sequential Programming and Order of Evaluation

Our expanded tic-tac-toe program includes both functional and sequential parts. The program computes its strategy functionally but uses sequences of commands to control the *user interface* by alternately printing information to the screen and reading information from the keyboard.

By adding sequential programming to our toolkit, we've increased our ability to write interactive programs. But there is a cost that goes along with this benefit: We now have to pay more attention to the order of events than we did in purely functional programs.

The obvious concern about order of events is that sequences of `show` expressions must come in the order in which we want them to appear, and `read` expressions must fit into the sequence properly so that the user is asked for the right information at the right time.

But there is another, less obvious issue about order of events. When the evaluation of expressions can have side effects in addition to returning values, the order of evaluation of argument subexpressions becomes important. Here's an example to show what we mean. Suppose we type the expression

```
(list (+ 3 4) (- 10 2))
```

The answer, of course, is `(7 8)`. It doesn't matter whether Scheme computes the seven first (left to right) or the eight first (right to left). But here's a similar example in which it *does* matter:

```
(define (show-and-return x)
  (show x)
  x)

> (list (show-and-return (+ 3 4)) (show-and-return (- 10 2)))
8
7
(7 8)
```

The value that's ultimately returned, in this example, is the same as before. But the two numeric values that go into the list are also printed separately, so we can see which is computed first. (We've shown the case of right-to-left computation; your Scheme might be different.)

Suppose you want to make sure that the seven prints first, regardless of which order your Scheme uses. You could do this:

```
> (let ((left (show-and-return (+ 3 4))))
    (list left (show-and-return (- 10 2))))
7
8
(7 8)
```

The expression in the body of a `let` can't be evaluated until the `let` variables (such as `left`) have had their values computed.

It's hard to imagine a practical use for the artificial `show-and-return` procedure, but a similar situation arises whenever we use `read`. Suppose we want to write a procedure to ask a person for his or her full name, returning a two-element list containing the first and last name. A natural mistake to make would be to write this procedure:

```
(define (ask-for-name)                          ;; wrong
  (show "Please type your first name, then your last name:")
  (list (read) (read)))

> (ask-for-name)
Please type your first name, then your last name:
John
Lennon
(LENNON JOHN)
```

What went wrong? We happen to be using a version of Scheme that evaluates argument subexpressions from right to left. Therefore, the word `John` was read by the rightmost call to `read`, which provided the second argument to `list`. The best solution is to use `let` as we did above:

```
(define (ask-for-name)
  (show "Please type your first name, then your last name:")
  (let ((first-name (read)))
    (list first-name (read))))
```

Even this example looks artificially simple, because of the two invocations of `read` that are visibly right next to each other in the erroneous version. But look at `play-ttt-helper`. The word `read` doesn't appear in its body at all. But when we invoke it using `ask-user` as the strategy procedure for `x`, the expression

```
(x-strat position 'x)
```

hides an invocation of `read`. The structure of `play-ttt-helper` includes a `let` that controls the timing of that `read`. (As it turns out, in this particular case we could have gotten away with writing the program without `let`. The hidden invocation of `read` is the only subexpression with a side effect, so there aren't two effects that might get out of order. But we had to think carefully about the program to be sure of that.)

Pitfalls

⇒ It's easy to get confused about what is printed explicitly by your program and what is printed by Scheme's read-eval-print loop. Until now, *all* printing was of the second kind. Here's an example that doesn't do anything very interesting but will help make the point clear:

```
(define (name)
  (display "MATT ")
  'wright)

> (name)
MATT WRIGHT
```

At first glance it looks as if putting the word "Matt" inside a call to `display` is unnecessary. After all, the word `wright` is printed even without using `display`. But watch this:

```
> (bf (name))
MATT RIGHT
```

Every time you invoke `name`, whether or not as the entire expression used at a Scheme prompt, the word `MATT` is printed. But the word `wright` is *returned,* and may or may not be printed depending on the context in which `name` is invoked.

⇒ A sequence of expressions returns the value of the *last* expression. If that isn't what you want, you must remember the value you want to return using `let`:

```
(let ((result (compute-this-first)))
  (begin
   (compute-this-second)
   (compute-this-third)
   result))
```

⇒ Don't forget that the first call to `read-line`, or any call to `read-line` after a call to `read`, will probably read the empty line that `read` left behind.

⇒ Sometimes you want to use what the user typed more than once in your program. But don't forget that `read` has an effect as well as a return value. Don't try to read the same expression twice:

```
(define (ask-question question)              ;; wrong
  (show question)
  (cond ((equal? (read) 'yes) #t)
        ((equal? (read) 'no) #f)
        (else (show "Please answer yes or no.")
              (ask-question question))))
```

If the answer is `yes`, this procedure will work fine. But if not, the second invocation of `read` will read a second expression, not test the same expression again as intended. To

avoid this problem, invoke `read` only once for each expression you want to read, and use `let` to remember the result:

```
(define (ask-question question)
  (show question)
  (let ((answer (read)))
    (cond ((equal? answer 'yes) #t)
          ((equal? answer 'no) #f)
          (else (show "Please answer yes or no.")
                (ask-question question)))))
```

Boring Exercises

20.1 What happens when we evaluate the following expression? What is printed, and what is the return value? Try to figure it out in your head before you try it on the computer.

```
(cond ((= 2 3) (show '(lady madonna)) '(i call your name))
      ((< 2 3) (show '(the night before)) '(hello little girl))
      (else '(p.s. i love you)))
```

20.2 What does `newline` return in your version of Scheme?

20.3 Define `show` in terms of `newline` and `display`.

Real Exercises

20.4 Write a program that carries on a conversation like the following example. What the user types is in boldface.

```
> (converse)
Hello, I'm the computer.  What's your name? Brian Harvey
Hi, Brian.  How are you? I'm fine.
Glad to hear it.
```

20.5 Our `name-table` procedure uses a fixed width for the column containing the last names of the people in the argument list. Suppose that instead of liking British-invasion music you are into late romantic Russian composers:

```
> (name-table '((piotr tchaikovsky) (nicolay rimsky-korsakov)
                (sergei rachmaninov) (modest musorgsky)))
```

Alternatively, perhaps you like jazz:

```
> (name-table '((bill evans) (paul motian) (scott lefaro)))
```

Modify `name-table` so that it figures out the longest last name in its argument list, adds two for spaces, and uses that number as the width of the first column.

20.6 The procedure `ask-user` isn't robust. What happens if you type something that isn't a number, or isn't between 1 and 9? Modify it to check that what the user types is a number between 1 and 9. If not, it should print a message and ask the user to try again.

20.7 Another problem with `ask-user` is that it allows a user to request a square that isn't free. If the user does this, what happens? Fix `ask-user` to ensure that this can't happen.

20.8 At the end of the game, if the computer wins or ties, you never find out which square it chose for its final move. Modify the program to correct this. (Notice that this exercise requires you to make `play-ttt-helper` non-functional.)

20.9 The way we invoke the game program isn't very user-friendly. Write a procedure `game` that asks you whether you wish to play `x` or `o`, then starts a game. (By definition, `x` plays first.) Then write a procedure `games` that allows you to keep playing repeatedly. It can ask "do you want to play again?" after each game. (Make sure that the outcome of each game is still reported, and that the user can choose whether to play `x` or `o` before each game.)