10 Functional Languages

Previous chapters of this text have focused largely on imperative programming languages. In the current chapter and the next we emphasize functional and logic languages instead. While imperative languages are far more widely used, "industrial-strength" implementations exist for both functional and logic languages, and both models have commercially important applications. Lisp has traditionally been popular for the manipulation of symbolic data, particularly in the field of artificial intelligence. In recent years functional languages—statically typed ones in particular—have become increasingly popular for scientific and business applications as well. Logic languages are widely used for formal specifications and theorem proving and, less widely, for many other applications.

Of course, functional and logic languages have a great deal in common with their imperative cousins. Naming and scoping issues arise under every model. So do types, expressions, and the control-flow concepts of selection and recursion. All languages must be scanned, parsed, and analyzed semantically. In addition, functional languages make heavy use of subroutines—more so even than most von Neumann languages—and the notions of concurrency and nondeterminacy are as common in functional and logic languages as they are in the imperative case.

As noted in Chapter 1, the boundaries between language categories tend to be rather fuzzy. One can write in a largely functional style in many imperative languages, and many functional languages include imperative features (assignment and iteration). The most common logic language—Prolog—provides certain imperative features as well. Finally, it is easy to build a logic programming system in most functional programming languages.

Because of the overlap between imperative and functional concepts, we have had occasion several times in previous chapters to consider issues of particular importance to functional programming languages. Most such languages depend heavily on polymorphism (the implicit parametric kind—Sections 3.5.3 and ◎7.2.4). Most make heavy use of lists (Section 7.8). Several, historically, were dynamically scoped (Sections 3.3.6 and ◎3.4.2). All employ recursion (Section 6.6) for repetitive execution, with the result that program behavior and performance depend heavily on the evaluation rules for parameters

Programming Language Pragmatics. DOI: 10.1016/B978-0-12-374514-9.00021-5

(Section 6.6.2). All have a tendency to generate significant amounts of temporary data, which their implementations reclaim through garbage collection (Section 7.7.3).

Our chapter begins with a brief introduction to the historical origins of the imperative, functional, and logic programming models. We then enumerate fundamental concepts in functional programming and consider how these are realized in the Scheme dialect of Lisp. More briefly, we also consider Caml, Common Lisp, Erlang, Haskell, ML, Miranda, pH, Single Assignment C, and Sisal. We pay particular attention to issues of evaluation order and higher-order functions. For those with an interest in the theoretical foundations of functional programming, we provide (on the PLP CD) an introduction to functions, sets, and the lambda calculus. The formalism helps to clarify the notion of a "pure" functional language, and illuminates the differences between the pure notation and its realization in more practical programming languages.

10.1 Historical Origins

To understand the differences among programming models, it can be helpful to consider their theoretical roots, all of which predate the development of electronic computers. The imperative and functional models grew out of work undertaken by mathematicians Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, and others in the 1930s. Working largely independently, these individuals developed several very different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics. Over time, these various formalizations were shown to be equally powerful: anything that could be computed in one could be computed in the others. This result led Church to conjecture that *any* intuitively appealing model of computing would be equally powerful as well; this conjecture is known as *Church's thesis*.

Turing's model of computing was the *Turing machine*, an automaton reminiscent of a finite or pushdown automaton, but with the ability to access arbitrary cells of an unbounded storage "tape."[1] The Turing machine computes in an imperative way, by changing the values in cells of its tape, just as a high-level imperative program computes by changing the values of variables. Church's model of computing is called the *lambda calculus*. It is based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the

[1] Alan Turing (1912–1954), for whom the Turing Award is named, was a British mathematician, philosopher, and computer visionary. As intellectual leader of Britain's cryptanalytic group during World War II, he was instrumental in cracking the German "Enigma" code and turning the tide of the war. He also laid the theoretical foundations of modern computer science, conceived the general purpose electronic computer, and pioneered the field of Artificial Intelligence. Persecuted as a homosexual after the war, stripped of his security clearance, and sentenced to "treatment" with drugs, he committed suicide.

letter λ—hence the notation's name).[2] Lambda calculus was the inspiration for functional programming: one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions. The computing models of Kleene and Post are more abstract, and do not lend themselves directly to implementation as a programming language.

The goal of early work in computability was not to understand computers (aside from purely mechanical devices, computers did not exist) but rather to formalize the notion of an effective procedure. Over time, this work allowed mathematicians to formalize the distinction between a *constructive* proof (one that shows how to obtain a mathematical object with some desired property) and a *nonconstructive* proof (one that merely shows that such an object must exist, perhaps by contradiction, or counting arguments, or reduction to some other theorem whose proof is nonconstructive). In effect, a program can be seen as a constructive proof of the proposition that, given any appropriate inputs, there exist outputs that are related to the inputs in a particular, desired way. Euclid's algorithm, for example, can be thought of as a constructive proof of the proposition that every pair of non-negative integers has a greatest common divisor.

Logic programming is also intimately tied to the notion of constructive proofs, but at a more abstract level. Rather than write a general constructive proof that works for all appropriate inputs, the logic programmer writes a set of *axioms* that allow the *computer* to discover a constructive proof for each particular set of inputs. We will consider logic programming in more detail in Chapter 11.

10.2 Functional Programming Concepts

In a strict sense of the term, *functional programming* defines the outputs of a program as a mathematical function of the inputs, with no notion of internal state, and thus no side effects. Among the languages we consider here, Miranda, Haskell, pH, Sisal, and Single Assignment C are purely functional. Erlang is nearly so. Most others include imperative features. To make functional programming practical, functional languages provide a number of features that are often missing in imperative languages, including:

- First-class function values and higher-order functions
- Extensive polymorphism

[2] Alonzo Church (1903–1995) was a member of the mathematics faculty at Princeton University from 1929 to 1967, and at UCLA from 1967 to 1990. While at Princeton he supervised the doctoral theses of, among many others, Alan Turing, Stephen Kleene, Michael Rabin, and Dana Scott. His codiscovery, with Turing, of uncomputable problems was a major breakthrough in understanding the limits of mathematics.

- List types and operators
- Structured function returns
- Constructors (aggregates) for structured objects
- Garbage collection

In Section 3.6.2 we defined a first-class value as one that can be passed as a parameter, returned from a subroutine, or (in a language with side effects) assigned into a variable. Under a strict interpretation of the term, first-class status also requires the ability to create (compute) new values at run time. In the case of subroutines, this notion of first-class status requires nested lambda expressions that can capture values (with unlimited extent) defined in surrounding scopes. Subroutines are second-class values in most imperative languages, but first-class values (in the strict sense of the term) in all functional programming languages. A *higher-order function* takes a function as an argument, or returns a function as a result.

Polymorphism is important in functional languages because it allows a function to be used on as general a class of arguments as possible. As we have seen in Sections 7.1 and 7.2.4, Lisp and its dialects are dynamically typed, and thus inherently polymorphic, while ML and its relatives obtain polymorphism through the mechanism of type inference. Lists are important in functional languages because they have a natural recursive definition, and are easily manipulated by operating on their first element and (recursively) the remainder of the list. Recursion is important because in the absence of side effects it provides the only means of doing anything repeatedly.

Several of the items in our list of functional language features (recursion, structured function returns, constructors, garbage collection) can be found in some but not all imperative languages. Fortran 77 has no recursion, nor does it allow structured types (i.e., arrays) to be returned from functions. Pascal and early versions of Modula-2 allow only simple and pointer types to be returned from functions. As we saw in Section 7.1.5, several imperative languages, including Ada, C, and Fortran 90, provide aggregate constructs that allow a structured value to be specified in-line. In most imperative languages, however, such constructs are lacking or incomplete. C# 3.0 and several scripting languages—Python and Ruby among them—provide aggregates capable of representing an (unnamed) functional value (a *lambda expression*), but few imperative languages are so expressive. A pure functional language must provide completely general aggregates: because there is no way to update existing objects, newly created ones must be initialized "all at once." Finally, though garbage collection is increasingly common in imperative languages, it is by no means universal, nor does it usually apply to the local variables of subroutines, which are typically allocated in the stack. Because of the desire to provide unlimited extent for first-class functions and other objects, functional languages tend to employ a (garbage-collected) heap for *all* dynamically allocated data (or at least for all data for which the compiler is unable to prove that stack allocation is safe).

Because Lisp was the original functional language, and is probably still the most widely used, several characteristics of Lisp are commonly, though inaccurately, described as though they pertained to functional programming in general. We will examine these characteristics (in the context of Scheme) in Section 10.3. They include:

- Homogeneity of programs and data: A program in Lisp is itself a list, and can be manipulated with the same mechanisms used to manipulate data.
- Self-definition: The operational semantics of Lisp can be defined elegantly in terms of an interpreter written in Lisp.
- Interaction with the user through a "read-eval-print" loop.

Many programmers—probably most—who have written significant amounts of software in both imperative and functional styles find the latter more aesthetically appealing. Moreover experience with a variety of large commercial projects (see the Bibliographic Notes at the end of the chapter) suggests that the absence of side effects makes functional programs significantly easier to write, debug, and maintain than their imperative counterparts. When passed a given set of arguments, a pure function can always be counted on to return the same results. Issues of undocumented side effects, misordered updates, and dangling or (in most cases) uninitialized references simply don't occur. At the same time, most implementations of functional languages still fall short in terms of portability, richness of library packages, interfaces to other languages, and debugging and profiling tools. We will return to the tradeoffs between functional and imperative programming in Section 10.7.

10.3 A Review/Overview of Scheme

EXAMPLE 10.1

The read-eval-print loop

Most Scheme implementations employ an interpreter that runs a "read-eval-print" loop. The interpreter repeatedly reads an expression from standard input (generally typed by the user), evaluates that expression, and prints the resulting value. If the user types

```
(+ 3 4)
```

the interpreter will print

```
7
```

If the user types

```
7
```

the interpreter will also print

```
7
```

(The number 7 is already fully evaluated.) To save the programmer the need to type an entire program verbatim at the keyboard, most Scheme implementations provide a `load` function that reads (and evaluates) input from a file:

```
(load "my_Scheme_program")
```

EXAMPLE 10.2
Significance of parentheses

As we noted in Section 6.1, Scheme (like all Lisp dialects) uses *Cambridge Polish* notation for expressions. Parentheses indicate a function application (or in some cases the use of a macro). The first expression inside the left parenthesis indicates the function; the remaining expressions are its arguments. Suppose the user types

```
((+ 3 4))
```

When it sees the inner set of parentheses, the interpreter will call the function `+`, passing `3` and `4` as arguments. Because of the outer set of parentheses, it will then attempt to call `7` as a zero-argument function—a run-time error:

```
eval: 7 is not a procedure
```

Unlike the situation in almost all other programming languages, extra parentheses change the semantics of Lisp/Scheme programs.

```
(+ 3 4)     ⟹ 7
((+ 3 4))   ⟹ error
```

Here the ⟹ means "evaluates to." This symbol is not a part of the syntax of Scheme itself.

EXAMPLE 10.3
Quoting

One can prevent the Scheme interpreter from evaluating a parenthesized expression by *quoting* it:

```
(quote (+ 3 4))  ⟹ (+ 3 4)
```

Here the result is a three-element list. More commonly, quoting is specified with a special shorthand notation consisting of a leading single quote mark:

```
'(+ 3 4)  ⟹ (+ 3 4)
```

EXAMPLE 10.4
Dynamic typing

Though every expression has a type in Scheme, that type is generally not determined until run time. Most predefined functions check dynamically to make sure that their arguments are of appropriate types. The expression

```
(if (> a 0) (+ 2 3) (+ 2 "foo"))
```

will evaluate to `5` if `a` is positive, but will produce a run-time type clash error if `a` is negative or zero. More significantly, as noted in Section 3.5.3, functions that make sense for arguments of multiple types are implicitly polymorphic:

```
(define min (lambda (a b) (if (< a b) a b)))
```

The expression `(min 123 456)` will evaluate to `123`; `(min 3.14159 2.71828)` will evaluate to `2.71828`. ■

EXAMPLE 10.5

Type predicates

User-defined functions can implement their own type checks using predefined *type predicate* functions:

```
(boolean? x)     ; is x a Boolean?
(char? x)        ; is x a character?
(string? x)      ; is x a string?
(symbol? x)      ; is x a symbol?
(number? x)      ; is x a number?
(pair? x)        ; is x a (not necessarily proper) pair?
(list? x)        ; is x a (proper) list?
```

(This is not an exhaustive list.) ■

EXAMPLE 10.6

Liberal syntax for symbols

A *symbol* in Scheme is comparable to what other languages call an identifier. The lexical rules for identifiers vary among Scheme implementations, but are in general much looser than they are in other languages. In particular, identifiers are permitted to contain a wide variety of punctuation marks:

```
(symbol? 'x$_%:&=*!)  ⟹ #t
```

The symbol `#t` represents the Boolean value true. False is represented by `#f`. Note the use here of `quote` (`'`); the symbol begins with `x`. ■

EXAMPLE 10.7

Lambda expressions

To create a function in Scheme one evaluates a *lambda expression*:[3]

```
(lambda (x) (* x x))  ⟹ function
```

The first "argument" to `lambda` is a list of formal parameters for the function (in this case the single parameter `x`). The remaining "arguments" (again just one in this case) constitute the body of the function. As we shall see in Section 10.4, Scheme differentiates between functions and so-called *special forms* (`lambda` among them), which resemble functions but have special evaluation rules. Strictly speaking, only functions have arguments, but we will also use the term informally to refer to the subexpressions that look like arguments in a special form. ■

A `lambda` expression does not give its function a name; this can be done using `let` or `define` (to be introduced in the next subsection). In this sense, a `lambda`

[3] A word of caution for readers familiar with Common Lisp: A `lambda` expression in Scheme *evaluates to* a function. A `lambda` expression in Common Lisp *is* a function (or, more accurately, is automatically coerced to be a function, without evaluation). The distinction becomes important whenever `lambda` expressions are passed as parameters or returned from functions: they must be quoted in Common Lisp (with `function` or `#'`) to prevent evaluation. Common Lisp also distinguishes between a symbol's *value* and its meaning as a function; Scheme does not: if a symbol represents a function, then the function is the symbol's value.

expression is like the aggregates that we used in Section 7.1.5 to specify array or record values.

EXAMPLE 10.8
Function evaluation

When a function is called, the language implementation restores the referencing environment that was in effect when the `lambda` expression was evaluated (like all languages with static scope and first-class, nested subroutines, Scheme employs deep binding). It then augments this environment with bindings for the formal parameters and evaluates the expressions of the function body in order. The value of the last such expression (most often there *is* only one) becomes the value returned by the function:

```
((lambda (x) (* x x)) 3)  ⟹ 9
```

EXAMPLE 10.9
`If` expressions

Simple conditional expressions can be written using `if`:

```
(if (< 2 3) 4 5)  ⟹ 4
(if #f 2 3)       ⟹ 3
```

In general, Scheme expressions are evaluated in applicative order, as described in Section 6.6.2. Special forms such as `lambda` and `if` are exceptions to this rule. The implementation of `if` checks to see whether the first argument evaluates to `#t`. If so, it returns the value of the second argument, without evaluating the third argument. Otherwise it returns the value of the third argument, without evaluating the second. We will return to the issue of evaluation order in Section 10.4.

10.3.1 Bindings

EXAMPLE 10.10
Nested scopes with `let`

Names can be bound to values by introducing a nested scope:

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
  (sqrt (plus (square a) (square b))))  ⟹ 5.0
```

The special form `let` takes two or more arguments. The first of these is a list of pairs. In each pair, the first element is a name and the second is the value that the name is to represent within the remaining arguments to `let`. Remaining arguments are then evaluated in order; the value of the construct as a whole is the value of the final argument.

The scope of the bindings produced by `let` is `let`'s second argument only:

```
(let ((a 3))
  (let ((a 4)
        (b a))
    (+ a b)))  ⟹ 7
```

Here b takes the value of the *outer* a. The way in which names become visible "all at once" at the end of the declaration list precludes the definition of recursive functions. For these one employs letrec:

```
(letrec ((fact
          (lambda (n)
            (if (= n 1) 1
              (* n (fact (- n 1)))))))
    (fact 5))                              ⟹ 120
```

There is also a let* construct in which names become visible "one at a time" so that later ones can make use of earlier ones, but not vice versa.

EXAMPLE 10.11

Global bindings with define

As noted in Section 3.3, Scheme is statically scoped. (Common Lisp is also statically scoped. Most other Lisp dialects are dynamically scoped.) While let and letrec allow the user to create nested scopes, they do not affect the meaning of global names (names known at the outermost level of the Scheme interpreter). For these Scheme provides a special form called define that has the side effect of creating a global binding for a name:

```
(define hypot
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))
(hypot 3 4)                                ⟹ 5
```

10.3.2 Lists and Numbers

EXAMPLE 10.12

Basic list operations

Like all Lisp dialects, Scheme provides a wealth of functions to manipulate lists. We saw many of these in Section 7.8; we do not repeat them all here. The three most important are car, which returns the head of a list, cdr ("coulder"), which returns the rest of the list (everything after the head), and cons, which joins a head to the rest of a list:

```
(car '(2 3 4))    ⟹ 2
(cdr '(2 3 4))    ⟹ (3 4)
(cons 2 '(3 4))   ⟹ (2 3 4)
```

Also useful is the null? predicate, which determines whether its argument is the empty list. Recall that the notation '(2 3 4) indicates a *proper* list, in which the final element is the empty list:

```
(cdr '(2))    ⟹ ()
(cons 2 3)    ⟹ (2 . 3)   ; an improper list
```

For fast access to arbitrary elements of a sequence, Scheme provides a vector type that is indexed by integers, like an array, and may have elements of heterogeneous types, like a record. Interested readers are referred to the Scheme manual [SDF$^+$07] for further information.

Scheme also provides a wealth of numeric and logical (Boolean) functions and special forms. The language manual describes a hierarchy of five numeric types: `integer`, `rational`, `real`, `complex`, and `number`. The last two levels are optional: implementations may choose not to provide any numbers that are not real. Most but not all implementations employ arbitrary-precision representations of both integers and rationals, with the latter stored internally as (numerator, denominator) pairs.

10.3.3 Equality Testing and Searching

Scheme provides several different equality-testing functions. For numerical comparisons, `=` performs type conversions where necessary (e.g., to compare an integer and a floating-point number). For general-purpose use, `eqv?` performs a *shallow* comparison, while `equal?` performs a *deep* (recursive) comparison, using `eqv?` at the leaves. The `eq?` function also performs a shallow comparison, and may be cheaper than `eqv?` in certain circumstances (in particular, `eq?` is not required to detect the equality of discrete values stored in different locations, though it may in some implementations). Further details were presented in Section 7.10.

EXAMPLE 10.13
List search functions

To search for elements in lists, Scheme provides two sets of functions, each of which has variants corresponding to the three general-purpose equality predicates. The functions `memq`, `memv`, and `member` take an element and a list as argument, and return the longest suffix of the list (if any) beginning with the element:

```
(memq 'z '(x y z w))          ⟹ (z w)
(memv '(z) '(x y (z) w))      ⟹ #f       ; (eq? '(z) '(z))     ⟹ #f
(member '(z) '(x y (z) w))    ⟹ ((z) w)  ; (equal? '(z) '(z))  ⟹ #t
```

The `memq`, `memv`, and `member` functions perform their comparisons using `eq?`, `eqv?`, and `equal?`, respectively. They return `#f` if the desired element is not found. It turns out that Scheme's conditional expressions (e.g., `if`) treat anything other than `#f` as true.[4] One therefore often sees expressions of the form

```
(if (memq desired-element list-that-might-contain-it) ...
```

EXAMPLE 10.14
Searching association lists

The functions `assq`, `assv`, and `assoc` search for values in *association lists* (otherwise known as *A-lists*). A-lists were introduced in Section ◎3.4.2 in the context of name lookup for languages with dynamic scoping. An A-list is a dictionary

[4] One of the more confusing differences between Scheme and Common Lisp is that Common Lisp uses the empty list `()` for false, while most implementations of Scheme (including all that conform to the version 5 standard) treat it as true.

implemented as a list of pairs.[5] The first element of each pair is a key of some sort; the second element is information corresponding to that key. Assq, assv, and assoc take a key and an A-list as argument, and return the first pair in the list, if there is one, whose first element is eq?, eqv?, or equal?, respectively, to the key. If there is no matching pair, #f is returned.

10.3.4 Control Flow and Assignment

EXAMPLE 10.15
Multiway conditional expressions

We have already seen the special form if. It has a cousin named cond that resembles a more general if... elsif... else:

```
(cond
  ((< 3 2) 1)
  ((< 4 3) 2)
  (else 3))    ⟹ 3
```

The arguments to cond are pairs. They are considered in order from first to last. The value of the overall expression is the value of the second element of the first pair in which the first element evaluates to #t. If none of the first elements evaluates to #t, then the overall value is #f. The symbol else is permitted only as the first element of the last pair of the construct, where it serves as syntactic sugar for #t.

Recursion, of course, is the principal means of doing things repeatedly in Scheme. Many issues related to recursion were discussed in Section 6.6; we do not repeat that discussion here.

EXAMPLE 10.16
Assignment

For programmers who wish to make use of side effects, Scheme provides assignment, sequencing, and iteration constructs. Assignment employs the special form set! and the functions set-car! and set-cdr!:

```
(let ((x 2)                        ; initialize x to 2
      (l '(a b)))                  ; initialize l to (a b)
    (set! x 3)                     ; assign x the value 3
    (set-car! l '(c d))            ; assign head of l the value (c d)
    (set-cdr! l '(e))              ; assign rest of l the value (e)
    ... x                ⟹ 3
    ... l                ⟹ ((c d) e)
```

The return values of the various varieties of set! are implementation-dependent.

EXAMPLE 10.17
Sequencing

Sequencing uses the special form begin:

```
(begin
  (display "hi ")
  (display "mom"))
```

[5] For clarity, the figures in Section ©3.4.2 elided the internal structure of the pairs.

EXAMPLE 10.18
Iteration

Iteration uses the special form do and the function for-each:

```
(define iter-fib (lambda (n)
  ; print the first n+1 Fibonacci numbers
  (do ((i 0 (+ i 1))        ; initially 0, inc'ed in each iteration
       (a 0 b)              ; initially 0, set to b in each iteration
       (b 1 (+ a b)))       ; initially 1, set to sum of a and b
      ((= i n) b)           ; termination test and final value
    (display b)             ; body of loop
    (display " "))))        ; body of loop

(for-each (lambda (a b) (display (* a b)) (newline))
  '(2 4 6)
  '(3 5 7))
```

The first argument to do is a list of triples, each of which specifies a new variable, an initial value for that variable, and an expression to be evaluated and placed in a fresh instance of the variable at the end of each iteration. The second argument to do is a pair that specifies the termination condition and the expression to be returned. At the end of each iteration all new values of loop variables (e.g., a and b) are computed using the current values. Only after all new values are computed are the new variable instances created.

The function for-each takes as argument a function and a sequence of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length. For-each calls its function argument repeatedly, passing successive sets of arguments from the lists. In the example shown here, the unnamed function produced by the lambda expression will be called on the arguments 2 and 3, 4 and 5, and 6 and 7. The interpreter will print

```
6
20
42
()
```

The last line is the return value of for-each, assumed here to be the empty list. The language definition allows this value to be implementation-dependent; the construct is executed for its side effects. ■

DESIGN & IMPLEMENTATION

Iteration in functional programs

It is important to distinguish between iteration as a notation for repeated execution and iteration as a means of orchestrating side effects. One can in fact define iteration as syntactic sugar for tail recursion, and Val, Sisal, and pH do precisely that (with special syntax to facilitate the passing of values from one iteration to the next). Such a notation may still be entirely side-effect free, that is, entirely functional. In Scheme, assignment and I/O are the truly imperative features. We think of iteration as imperative because most Scheme programs that use it have assignments or I/O in their loops.

Two other control-flow constructs—`delay` and `force`—have been mentioned in previous chapters. `Delay` and `force` (Section 6.6.2) permit the lazy evaluation of expressions. `Call-with-current-continuation` (`call/cc`; Section 6.2.2) allows the current program counter and referencing environment to be saved in the form of a closure, and passed to a specified subroutine. We will discuss `delay` and `force` further in Section 10.4.

10.3.5 Programs as Lists

As should be clear by now, a program in Scheme takes the form of a list. In technical terms, we say that Lisp and Scheme are *homoiconic*—self-representing. A parenthesized string of symbols (in which parentheses are balanced) is called an *S-expression* regardless of whether we think of it as a program or as a list. In fact, an unevaluated program *is* a list, and can be constructed, deconstructed, and otherwise manipulated with all the usual list functions.

EXAMPLE 10.19
Evaluating data as code

Just as `quote` can be used to inhibit the evaluation of a list that appears as an argument in a function call, Scheme provides an `eval` function that can be used to evaluate a list that has been created as a data structure:

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))
((compose car cdr) '(1 2 3))  ⟹ 2

(define compose2
  (lambda (f g)
    (eval (list 'lambda '(x) (list f (list g 'x)))
          (scheme-report-environment 5))))
((compose2 car cdr) '(1 2 3))                     ⟹ 2
```

In the first of these declarations, `compose` takes as arguments a pair of functions `f` and `g`. It returns as result a function that takes as parameter a value `x`, applies `g` to it, then applies `f`, and finally returns the result. In the second declaration, `compose2` performs the same function, but in a different way. The function `list` returns a list consisting of its (evaluated) arguments. In the body of `compose2`, this list is the *unevaluated* expression `(lambda (x) (f (g x)))`. When passed to `eval`, this list evaluates to the desired function. The second argument of `eval` specifies the referencing environment in which the expression is to be evaluated. In our example we have specified the environment defined by the Scheme version 5 report [ADH+98].

`Eval` *and* `Apply`

The original description of Lisp [MAE+65] included a *self-definition* of the language: code for a Lisp interpreter, written in Lisp. Though Scheme differs in a number of ways from this early Lisp (most notably in its use of lexical scoping), such a *metacircular* interpreter can still be written easily [AS96, Chap. 4]. The code is based on the functions `eval` and `apply`. The first of these we have just seen. The second, `apply`, takes two arguments: a function and a list. It achieves the effect of calling the function, with the elements of the list as arguments.

The functions `eval` and `apply` can be defined as mutually recursive. When passed a number or a string, `eval` simply returns that number or string. When passed a symbol, it looks that symbol up in the specified environment and returns the value to which it is bound. When passed a list it checks to see whether the first element of the list is one of a small number of symbols that name so-called *primitive* special forms, built into the language implementation. For each of these special forms (`lambda`, `if`, `define`, `set!`, `quote`, etc.) `eval` provides a direct implementation. For other lists, `eval` calls itself recursively on each element and then calls `apply`, passing as arguments the value of the first element (which must be a function) and a list of the values of the remaining elements. Finally, `eval` returns what `apply` returned.

When passed a function f and a list of arguments l, `apply` inspects the internal representation of f to see whether it is primitive. If so it invokes the built-in implementation. Otherwise it retrieves (from the representation of f) the referencing environment in which f's lambda expression was originally evaluated. To this environment it adds the names of f's parameters, with values taken from l. Call this resulting environment e. Next `apply` retrieves the list of expressions that make up the body of f. It passes these expressions, together with e, one at a time to `eval`. Finally, `apply` returns what the `eval` of the last expression in the body of f returned.

Formalizing Self-Definition

The idea of self-definition—a Scheme interpreter written in Scheme—may seem a bit confusing unless one keeps in mind the distinction between the Scheme code that constitutes the interpreter and the Scheme code that the interpreter is interpreting. In particular, the interpreter is not running itself, though it could run a *copy* of itself. What we really mean by "self-definition" is that for all expressions E, we get the same result by evaluating E under the interpreter I that we get by evaluating E directly.

EXAMPLE 10.20
Denotational semantics of Scheme

Suppose now that we wish to formalize the semantics of Scheme as some as-yet-unknown mathematical function $\mathcal{M}$ that takes a Scheme expression as an argument and returns the expression's value. (This value may be a number, a list, a function, or a member of any of a small number of other domains.) How might we go about this task? For certain simple strings of symbols we can define a value directly: strings of digits, for example, map onto the natural numbers. For more complex expressions, we note that

$$\forall E[\mathcal{M}(E) = (\mathcal{M}(I))(E)]$$

Put another way,

$$\mathcal{M}(I) = \mathcal{M}$$

Suppose now that we let $H(\mathcal{F}) = \mathcal{F}(I)$ where $\mathcal{F}$ can be any function that takes a Scheme expression as its argument. Clearly

$$H(\mathcal{M}) = \mathcal{M}$$

Our desired function $\mathcal{M}$ is said to be a *fixed point* of H. Because H is well defined (it simply applies its argument to I), we can use it to obtain a rigorous definition of $\mathcal{M}$. The tools to do so come from the field of denotational semantics, a subject beyond the scope of this book.[6]

10.3.6 Extended Example: DFA Simulation

EXAMPLE 10.21
Simulating a DFA in Scheme

To conclude our introduction to Scheme, we present a complete program to simulate the execution of a DFA (deterministic finite automaton). The code appears in Figure 10.1. Finite automata details can be found in Sections 2.2 and ◎2.4.1. Here we represent a DFA as a list of three items: the start state, the transition function, and a list of final states. The transition function in turn is represented by a list of pairs. The first element of each pair is another pair, whose first element is a state and whose second element is an input symbol. If the current state and next input symbol match the first element of a pair, then the finite automaton enters the state given by the second element of the pair.

To make this concrete, consider the DFA of Figure 10.2. It accepts all strings of zeros and ones in which each digit appears an even number of times. To simulate this machine, we pass it to the function `simulate` along with an input string. As it runs, the automaton accumulates as a list a trace of the states through which it has traveled, ending with the symbol `accept` or `reject`. For example, if we type

```
(simulate
 zero-one-even-dfa   ; machine description
 '(0 1 1 0 1))       ; input string
```

then the Scheme interpreter will print

```
(q0 q2 q3 q2 q0 q1 reject)
```

[6] Actually, H has an infinite number of fixed points. What we want (and what denotational semantics will give us) is the *least* fixed point: the one that defines a value for as few strings of symbols as possible, while still producing the "correct" value for numbers and other simple strings. Another example of least fixed points appears in Section ◎16.4.2.

```
(define simulate
  (lambda (dfa input)
    (cons (current-state dfa)                ; start state
          (if (null? input)
              (if (infinal? dfa) '(accept) '(reject))
            (simulate (move dfa (car input)) (cdr input))))))

;; access functions for machine description:
(define current-state car)
(define transition-function cadr)
(define final-states caddr)
(define infinal?
  (lambda (dfa)
    (memq (current-state dfa) (final-states dfa))))

(define move
  (lambda (dfa symbol)
    (let ((cs (current-state dfa)) (trans (transition-function dfa)))
      (list
       (if (eq? cs 'error)
           'error
         (let ((pair (assoc (list cs symbol) trans)))
           (if pair (cadr pair) 'error)))   ; new start state
       trans                                ; same transition function
       (final-states dfa)))))               ; same final states
```

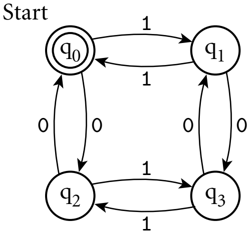
Figure 10.1 Scheme program to simulate the actions of a DFA. Given a machine description and an input symbol *i*, function `move` searches for a transition labeled *i* from the start state to some new state *s*. It then returns a new machine with the same transition function and final states, but with *s* as its "start" state. The main function, `simulate`, tests to see if it is in a final state. If not, it passes the current machine description and the first symbol of input to `move`, and then calls itself recursively on the new machine and the remainder of the input. The functions `cadr` and `caddr` are defined as `(lambda (x) (car (cdr x)))` and `(lambda (x) (car (cdr (cdr x))))`, respectively. Scheme provides a large collection of such abbreviations.

If we change the input string to `010010`, the interpreter will print

```
(q0 q2 q3 q1 q3 q2 q0 accept)
```

☑ **CHECK YOUR UNDERSTANDING**

1. What mathematical formalism underlies functional programming?
2. List several distinguishing characteristics of functional programming languages.
3. Briefly describe the behavior of the Lisp/Scheme *read-eval-print* loop.
4. What is a *first-class* value?
5. Explain the difference between `let`, `let*`, and `letrec` in Scheme.



```
(define zero-one-even-dfa
 '(q0                                                    ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)     ; transition fn
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0)))                                                ; final states
```

Figure 10.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.

6. Explain the difference between `eq?`, `eqv?`, and `equal?`.
7. Describe three ways in which Scheme programs can depart from a purely functional programming model.
8. What is an *association list*?
9. What does it mean for a language to be *homoiconic*?
10. What is an *S-expression*?
11. Outline the behavior of `eval` and `apply`.

10.4 Evaluation Order Revisited

In Section 6.6.2 we observed that the subcomponents of many expressions can be evaluated in more than one order. In particular, one can choose to evaluate function arguments before passing them to a function, or to pass them unevaluated. The former option is called *applicative-order* evaluation; the latter is called *normal-order* evaluation. Like most imperative languages, Scheme uses applicative order in most cases. Normal order, which arises in the macros and call-by-name parameters of imperative languages, is available in special cases.

EXAMPLE 10.22

Applicative and normal-order evaluation

Suppose, for example, that we have defined the following function:

```
(define double (lambda (x) (+ x x)))
```

Evaluating the expression `(double (* 3 4))` in applicative order (as Scheme does), we have

```
    (double (* 3 4))
⟹ (double 12)
⟹ (+ 12 12)
⟹ 24
```

Under normal-order evaluation we would have

```
    (double (* 3 4))
⟹ (+ (* 3 4) (* 3 4))
⟹ (+ 12 (* 3 4))
⟹ (+ 12 12)
⟹ 24
```

Here we end up doing extra work: normal order causes us to evaluate `(* 3 4)` twice. ■

EXAMPLE 10.23

Normal-order avoidance of unnecessary work

In other cases, applicative-order evaluation can end up doing extra work. Suppose we have defined the following:

```
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
        ((= x 0) b)
        ((> x 0) c))))
```

Evaluating the expression `(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))` in applicative order, we have

```
    (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⟹ (switch -1 3 (+ 2 3) (+ 3 4))
⟹ (switch -1 3 5 (+ 3 4))
⟹ (switch -1 3 5 7)
⟹ (cond   ((< -1 0) 3)
          ((= -1 0) 5)
          ((> -1 0) 7))
⟹ (cond   (#t 3)
          ((= -1 0) 5)
          ((> -1 0) 7))
⟹ 3
```

(Here we have assumed that `cond` is built in, and evaluates its arguments lazily, even though `switch` is doing so eagerly.) Under normal-order evaluation we would have

```
    (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⟹ (cond   ((< -1 0) (+ 1 2))
          ((= -1 0) (+ 2 3))
          ((> -1 0) (+ 3 4)))
⟹ (cond   (#t (+ 1 2))
          ((= -1 0) (+ 2 3))
          ((> -1 0) (+ 3 4)))
⟹ (+ 1 2)
⟹ 3
```

Here normal-order evaluation avoids evaluating `(+ 2 3)` or `(+ 3 4)`. (In this case, we have assumed that arithmetic and logical functions such as `+` and `<` are built in, and force the evaluation of their arguments.)

In our overview of Scheme we have differentiated on several occasions between special forms and functions. Arguments to functions are always passed by sharing (Section 8.3.1), and are evaluated before they are passed (i.e., in applicative order). Arguments to special forms are passed unevaluated—in other words, by name. Each special form is free to choose internally when (and if) to evaluate its parameters. `Cond`, for example, takes a sequence of unevaluated pairs as arguments. It evaluates their `car`s internally, one at a time, stopping when it finds one that evaluates to `#t`.

Together, special forms and functions are known as *expression types* in Scheme. Some expression types are *primitive*, in the sense that they must be built into the language implementation. Others are *derived*; they can be defined in terms of primitive expression types. In an `eval`/`apply`–based interpreter, primitive special forms are built into `eval`; primitive functions are recognized by `apply`. We have seen how the special form `lambda` can be used to create derived functions, which can be bound to names with `let`. Scheme provides an analogous special form, `syntax-rules`, that can be used to create derived special forms. These can then be bound to names with `define-syntax` and `let-syntax`. Derived special forms are known as *macro*s in Scheme, but unlike most other macros, they are *hygienic*—lexically scoped, integrated into the language's semantics, and immune from the problems of mistaken grouping and variable capture described in Section 3.7. Like C++ templates (Section ◎8.4.4), Scheme macros are Turing complete. They behave like functions whose arguments are passed by name (Section ◎8.3.2) instead of by sharing. They are implemented, however, via logical expansion in the interpreter's parser and semantic analyzer, rather than by delayed evaluation with thunks.

10.4.1 Strictness and Lazy Evaluation

Evaluation order can have an effect not only on execution speed, but on program correctness as well. A program that encounters a dynamic semantic error or an infinite regression in an "unneeded" subexpression under applicative-order evaluation may terminate successfully under normal-order evaluation. A (side-effect-free) function is said to be *strict* if it is undefined (fails to terminate, or encounters an error) when any of its arguments is undefined. Such a function can safely evaluate all its arguments, so its result will not depend on evaluation order. A function is said to be *nonstrict* if it does not impose this requirement—that is, if it is sometimes defined even when one of its arguments is not. A *language* is said to be strict if it is defined in such a way that functions are always strict. A language is said to be nonstrict if it permits the definition of nonstrict functions. If a language always evaluates expressions in applicative order, then every function is guaranteed to be strict, because whenever an argument is undefined, its evaluation will fail and so will the function to which it is being passed. Contrapositively, a nonstrict language cannot use applicative order; it must use normal order to avoid evaluating unneeded arguments. ML and (with the exception of macros) Scheme are strict. Miranda and Haskell are nonstrict.

Lazy evaluation (as described here—see the footnote on page 276) gives us the advantage of normal-order evaluation (not evaluating unneeded subexpressions) while running within a constant factor of the speed of applicative-order evaluation for expressions in which everything is needed. The trick is to tag every argument internally with a "memo" that indicates its value, if known. Any attempt to evaluate the argument sets the value in the memo as a side effect, or returns the value (without recalculating it) if it is already set.

EXAMPLE 10.24

Avoiding work with lazy evaluation

Returning to the expression of Example 10.22, `(double (* 3 4))` will be compiled as `(double (f))`, where `f` is a hidden closure with an internal side effect:

```
(define f
  (lambda ()
    (let ((done #f)                       ; memo initially unset
          (memo '())
          (code (lambda () (* 3 4))))
      (if done memo                       ; if memo is set, return it
        (begin
          (set! memo (code))              ; remember value
          memo)))))                       ; and return it
...
    (double (f))
⟹  (+ (f) (f))
⟹  (+ 12 (f))         ; first call computes value
⟹  (+ 12 12)          ; second call returns remembered value
⟹  24
```

Here `(* 3 4)` will be evaluated only once. While the cost of manipulating memos will clearly be higher than that of the extra multiplication in this case, if we were to replace `(* 3 4)` with a very expensive operation, the savings could be substantial.

DESIGN & IMPLEMENTATION

Lazy evaluation

One of the beauties of a purely functional language is that it makes lazy evaluation a completely transparent performance optimization: the programmer can think in terms of nonstrict functions and normal-order evaluation, counting on the implementation to avoid the cost of repeated evaluation. For languages with imperative features, however, this characterization does not hold: lazy evaluation is *not* transparent in the presence of side effects.

Lazy evaluation is particularly useful for "infinite" data structures, as described in Section 6.6.2. It can also be useful in programs that need to examine only a prefix of a potentially long list (see Exercise 10.10). Lazy evaluation is used for all arguments in Miranda and Haskell. It is available in Scheme through explicit use of `delay` and `force`. (Recall that the first of these is a special form that creates a [memo, closure] pair; the second is a function that returns the value in the memo, using the closure to calculate it first if necessary.) Where normal-order evaluation can be thought of as function evaluation using call-by-name parameters, lazy evaluation is sometimes said to employ "call-by-need." In addition to Miranda and Haskell, call-by-need can be found in the R scripting language, widely used by statisticians.

The principal problem with lazy evaluation is its behavior in the presence of side effects. If an argument contains a reference to a variable that may be modified by an assignment, then the value of the argument will depend on whether it is evaluated before or after the assignment. Likewise, if the argument contains an assignment, values elsewhere in the program may depend on when evaluation occurs. These problems do not arise in Miranda or Haskell because they are purely functional: there are no side effects. Scheme leaves the problem up to the programmer, but requires that every use of a `delay`-ed expression be enclosed in `force`, making it relatively easy to identify the places where side effects are an issue. ML provides no built-in mechanism for lazy evaluation. The same effect can be achieved with assignment and explicit functions (Exercise 10.11), but the code is rather awkward.

10.4.2 I/O: Streams and Monads

A major source of side effects can be found in traditional I/O, including the built-in functions `read` and `display` of Scheme: `read` will generally return a different value every time it is called, and multiple calls to `display`, though they never return a value, must occur in the proper order if the program is to be considered correct.

One way to avoid these side effects is to model input and output as *streams*—unbounded-length lists whose elements are generated lazily. We saw an example of a stream in Section 6.6.2, where we used Scheme's `delay` and `force` to implement a "list" of the natural numbers. Similar code in ML appears in Exercise 10.11.[7]

EXAMPLE 10.25
Stream-based program execution

If we model input and output as streams, then a program takes the form

```
(define output (my_prog input))
```

When it needs an input value, function `my_prog` forces evaluation of the `car` of `input`, and passes the `cdr` on to the rest of the program. To drive execution,

[7] Note that `delay` and `force` automatically *memoize* their stream, so that values are never computed more than once. Exercise 10.11 asks the reader to write a memoizing version of a nonmemoizing stream.

the language implementation repeatedly forces evaluation of the car of output, prints it, and repeats:

```
(define driver (lambda (s)
  (if (null? s) '()      ; nothing left
    (display (car s))
    (driver (cdr s)))))
(driver output)
```

EXAMPLE 10.26
Interactive I/O with streams

To make things concrete, suppose we want to write a purely functional program that prompts the user for a sequence of numbers (one at a time!) and prints their squares. If Scheme employed lazy evaluation of input and output streams (it doesn't), then we could write:

```
(define squares (lambda (s)
  (cons "please enter a number\n"
        (let ((n (car s)))
          (if (eof-object? n) '()
            (cons (* n n) (cons #\newline (squares (cdr s)))))))))
(define output (squares input)))
```

Prompts, inputs, and outputs (i.e., squares) would be interleaved naturally in time. In effect, lazy evaluation would *force* things to happen in the proper order: The car of output is the first prompt. The cadr of output is the first square, a value that requires evaluation of the car of input. The caddr of output is the second prompt. The cadddr of output is the second square, a value that requires evaluation of the cadr of input.

Streams formed the basis of the I/O system in early versions of Haskell. Unfortunately, while they successfully encapsulate the imperative nature of interaction at a terminal, streams don't work very well for graphics or random access to files. They also make it difficult to accommodate I/O of different kinds (since all elements of a list in Haskell must be of a single type). More recent versions of Haskell employ a more general concept known as *monads*. Monads are drawn from a branch of mathematics known as *category theory*, but one doesn't need to understand the theory to appreciate their usefulness in practice. In Haskell, monads are essentially a clever use of higher-order functions, coupled with a bit of syntactic sugar, that allow the programmer to chain together a sequence of *actions* (function calls) that have to happen in order. The power of the idea comes from the ability to carry a hidden, structured value of arbitrary complexity from one action to the next. In many applications of monads, this extra hidden value plays the role of mutable state: differences between the values carried to successive actions act as side effects.

EXAMPLE 10.27
Pseudorandom numbers in Haskell

As a motivating example somewhat simpler than I/O, consider the possibility of creating a pseudorandom number generator (RNG) along the lines of Example 6.42 (page 247). In that example we assumed that rand() would modify hidden state as a side effect, allowing it to return a different value every time it is called. This idiom isn't possible in a pure functional language, but we can obtain a similar effect by passing the state to the function and having it return new state along with the random number. This is exactly how the built-in function `random` works in Haskell. The following code calls `random` twice to illustrate its interface.

```
twoRandomInts :: StdGen -> ([Integer], StdGen)
    -- type signature: twoRandomInts is a function that takes an
    -- StdGen (the state of the RNG) and returns a tuple containing
    -- a list of Integers and a new StdGen.
twoRandomInts gen = let
        (rand1, gen2) = random gen
        (rand2, gen3) = random gen2
    in ([rand1, rand2], gen3)

main = let
        gen = mkStdGen 123              -- new RNG, seeded with 123
        ints = fst (twoRandomInts gen)  -- extract first element
    in print ints                           -- of returned tuple
```

Note that `gen2`, one of the return values from the first call to `random`, has been passed as an argument to the second call. Then `gen3`, one of the return values from the second call, is returned to `main`, where it could, if we wished, be passed to another function. This mechanism works, but it's far from pretty: copies of the RNG state must be "threaded through" every function that needs a random number. This is particularly complicated for deeply nested functions. It is easy to make a mistake, and difficult to verify that one has not.

Monads provide a more general solution to the problem of threading mutable state through a functional program. Here is our example rewritten to use Haskell's standard `IO` monad, which includes a random number generator:

```
twoMoreRandomInts :: IO [Integer]
    -- twoMoreRandomInts returns a list of Integers.  It also
    -- implicitly accepts, and returns, all the state of the IO monad.
twoMoreRandomInts = do
    rand1 <- randomIO
    rand2 <- randomIO
    return [rand1, rand2]

main = do
    moreInts <- twoMoreRandomInts
    print moreInts
```

There are several differences here. First, the type of the `twoMoreRandomInts` function has become `IO [Integer]`. This identifies it as an *IO action*—a function that (in addition to returning an explicit list of integers) invisibly accepts and returns the state of the `IO` monad (including the standard RNG). Similarly, the type of `randomIO` is `IO Integer`. To thread the `IO` state from one action to the next, the bodies of `twoMoreRandomInts` and `main` use `do` notation rather than let. A do block packages a sequence of actions together into a single, compound action. At each step along the way, it passes the (potentially modified) state of the monad from one action to the next. It also supports the "assignment" operator, <-, which separates the explicit return value from the hidden state and opens a nested scope for its left-hand side, so all values "assigned" earlier in the sequence are visible to actions later in the sequence.

The return operator in twoMoreRandomInts packages an explicit return value (in our case, a two-element list) together with the hidden state, to be returned to the caller. A similar use of return presumably appears inside randomIO. Everything we have done is purely functional—do and <- are simply syntactic sugar—but the bookkeeping required to pass the state of the RNG from one invocation of random to the next has been hidden in a way that makes our code look imperative. ■

EXAMPLE 10.28

The state of the IO monad

So what does this have to do with I/O? Consider the getChar function, which reads a character from standard input. Like rand, we expect it to return a different value every time we call it. Haskell therefore arranges for getChar to be of type IO Char: it returns a character, but also accepts, and passes on, the hidden state of the monad.

In most Haskell monads, hidden state can be explicitly extracted and examined. The IO monad, however, is *abstract*: only part of its state is defined in library header files; the rest is implemented by the language run-time system. This is unavoidable, because, in effect, *the hidden state of the* IO *monad encompasses the real world.* If this state were visible, a program could capture and reuse it, with the nonsensical expectation that we could "go back in time" and see what the user would have done in response to a different prompt last Tuesday. Unfortunately, IO state hiding means that a value of type IO T is permanently tainted: it can never be extracted from the monad to produce a "pure T." ■

EXAMPLE 10.29

Functional composition of actions

Because IO actions are just ordinary values, we can manipulate them in the same way as values of other data types. The most basic output action is putChar, of type Char -> IO () (monadic function with an explicit character argument and no explicit return). Given putChar, we can define putStr:

```
putStr :: String -> IO ()
putStr s = sequence_ (map putChar s)
```

Strings in Haskell are simply lists of characters. The map function takes a function *f* and a list *l* as argument, and returns a list that contains the results of applying *f* to the elements of *l*:

```
map :: (a->b) -> [a] -> [b]
map f [] = []                      -- base case
map f (h:t) = f h : map f t        -- tail recursive case
                                       -- ':' is like cons in Scheme
```

The result of map putChar s is a list of actions, each of which prints a character: it has type [IO ()]. The built-in function sequence_ converts this to a single action that prints a list. It could be defined as follows.

```
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()                     -- base case
sequence_ (a:more) = do a; sequence_ more   -- tail recursive case
```

As before, do provides a convenient way to chain actions together. For brevity, we have written the actions on a single line, separated by a semicolon. ■

The entry point of a Haskell program is always the function main. It has type IO (). Because Haskell is lazy (nonstrict), the action sequence returned by main remains hypothetical until the run-time system forces its evaluation. In practice, Haskell programs tend to have a small top-level structure of IO monad code that sequences I/O operations. The bulk of the program—both the computation of values *and the determination of the order in which I/O actions should occur*—is then purely functional. For a program whose I/O can be expressed in terms of streams, the top-level structure may consist of a single line:

EXAMPLE 10.30
Streams and the I/O monad

```
main = interact my_program
```

The library function interact is of type (String -> String) -> IO (). It takes as argument a function from strings to strings (in this case my_program). It calls this function, passing the contents of standard input as argument, and writes the result to standard output. Internally, interact uses the function getContents, which returns the program's input as a lazily evaluated string: a stream. In a more sophisticated program, main may orchestrate much more complex I/O actions, including graphics and random access to files. ■

DESIGN & IMPLEMENTATION

Monads

Monads are very heavily used in Haskell. The IO monad serves as the central repository for imperative language features—not only I/O and random numbers, but also mutable global variables and shared-memory synchronization. Additional monads (with accessible hidden state) support partial functions and various container classes (lists and sets). When coupled with lazy evaluation, monadic containers in turn provide a natural foundation for backtracking search, nondeterminism, and the functional equivalent of iterators. (In the list monad, for example, hidden state can carry the continuation needed to generate the tail of an infinite list.)

The inability to extract values from the IO monad reflects the fact that the physical world is imperative, and that a language that needs to interact with the physical world in nontrivial ways must include imperative features. Put another way, the IO monad (unlike monads in general) is more than syntactic sugar: by hiding the state of the physical world it makes it possible to express things that could not otherwise be expressed in a functional way, provided that we are willing to enforce a sequential evaluation order. The beauty of monads is that they confine sequentiality to a relatively small fraction of the typical program, so that side effects cannot interfere with the bulk of the computation.

10.5 Higher-Order Functions

A function is said to be a *higher-order function* (also called a *functional form*) if it takes a function as an argument, or returns a function as a result. We have seen several examples already of higher-order functions: `call/cc` (Section 6.2.2), `for-each` (Example 10.18), `compose` (Example 10.19), and `apply` (page 518). We also saw a Haskell version of the higher-order function `map` in Section 10.4.2.

EXAMPLE 10.31
Map function in Scheme

The Scheme version of `map` is slightly more general. Like `for-each`, it takes as argument a function and a *sequence* of lists. There must be as many lists as the function takes arguments, and the lists must all be of the same length. `Map` calls its function argument on corresponding sets of elements from the lists:

```
(map * '(2 4 6) '(3 5 7))  ⟹ (6 20 42)
```

Where `for-each` is executed for its side effects, and has an implementation-dependent return value, `map` is purely functional: it returns a list composed of the values returned by its function argument. ■

EXAMPLE 10.32
Folding (reduction) in Scheme

Programmers in Scheme (or in ML, Haskell, or other functional languages) can easily define other higher-order functions. Suppose, for example, that we want to be able to "fold" the elements of a list together, using an associative binary operator:

```
(define fold (lambda (f i l)
  (if (null? l) i     ; i is commonly the identity element for f
    (f (car l) (fold f i (cdr l))))))
```

Now `(fold + 0 '(1 2 3 4 5))` gives us the sum of the first five natural numbers, and `(fold * 1 '(1 2 3 4 5))` gives us their product. ■

EXAMPLE 10.33
Combining higher-order functions

One of the most common uses of higher-order functions is to build new functions from existing ones:

```
(define total (lambda (l) (fold + 0 l)))
(total '(1 2 3 4 5))                        ⟹ 15

(define total-all (lambda (l)
  (map total l)))
(total-all '((1 2 3 4 5)
             (2 4 6 8 10)
             (3 6 9 12 15)))                ⟹ (15 30 45)

(define make-double (lambda (f) (lambda (x) (f x x))))
(define twice (make-double +))
(define square (make-double *))
```
■

Currying

EXAMPLE 10.34

Partial application with currying

A common operation, named for logician Haskell Curry, is to replace a multiargument function with a function that takes a single argument and returns a function that expects the remaining arguments:

```
(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4)                        ⟹ 7
(define plus-3 (curried-plus 3))
(plus-3 4)                                  ⟹ 7
```

Among other things, currying gives us the ability to pass a "partially applied" function to a higher-order function:

```
(map (curried-plus 3) '(1 2 3))             ⟹ (4 5 6)
```

■

EXAMPLE 10.35

General-purpose curry function

It turns out that we can write a general-purpose function that "curries" its (binary) function argument:

```
(define curry (lambda (f) (lambda (a) (lambda (b) (f a b)))))
(((curry +) 3) 4)                           ⟹ 7
(define curried-plus (curry +))
```

■

EXAMPLE 10.36

Tuples as ML function arguments

ML and its descendants (Miranda, Haskell, Caml, F#) make it especially easy to define curried functions. Consider the following function in ML:

```
fun plus (a, b) : int = a + b;
==> val plus = fn : int * int -> int
```

DESIGN & IMPLEMENTATION

Higher-order functions

If higher-order functions are so powerful and useful, why aren't they more common in imperative programming languages? There would appear to be at least two important answers. First, much of the power of first-class functions depends on the ability to create new functions on the fly, and for that we need a function *constructor*—something like Scheme's `lambda` or ML's `fn`. Though they appear in certain recent languages, notably Python and C#, function constructors are a significant departure from the syntax and semantics of traditional imperative languages. Second, the ability to specify functions as return values, or to store them in variables (if the language has side effects) requires either that we eliminate function nesting (something that would again erode the ability of programs to create functions with desired behaviors on the fly), or that we give local variables unlimited extent, thereby increasing the cost of storage management.

The last line is printed by the ML interpreter, and indicates the inferred type of `plus`. The type declaration is required to disambiguate the overloaded + operator. Though one may think of `plus` as a function of two arguments, the ML definition says that all functions take a *single* argument. What we have declared is a function that takes a two-element *tuple* as argument. To call `plus`, we juxtapose its name and the tuple that is its argument:

```
plus (3, 4);
==> val it = 7 : int
```

The parentheses here are not part of the function call syntax; they delimit the tuple `(3, 4)`. ■

EXAMPLE 10.37

Optional parentheses on singleton arguments

We can declare a single-argument function without parenthesizing its formal argument:

```
fun twice n : int = n + n;
==> val twice = fn : int -> int
twice 2;
==> val it = 4 : int
```

We can add parentheses in either the declaration or the call if we want, but because there is no comma inside, no tuple is implied:

```
fun double (n) : int = n + n;
twice (2);
==> val it = 4 : int
twice 2;
==> val it = 4 : int
double (2);
==> val it = 4 : int
double 2;
==> val it = 4 : int
```

Ordinary parentheses can be placed around any expression in ML. ■

EXAMPLE 10.38

Simple curried function in ML

Now consider the definition of a curried function:

```
fun curried_plus a = fn b : int => a + b;
==> val curried_plus = fn : int -> int -> int
```

Note the type of `curried_plus`: `int -> int -> int` groups implicitly as `int -> (int -> int)`. Where `plus` is a function mapping a pair (tuple) of integers to an integer, `curried_plus` is a function mapping an integer to a function that maps an integer to an integer:

```
curried_plus 3;
==> val it = fn : int -> int
```

```
plus 3;
==> Error: operator domain (int * int) and operand (int) don’t agree
```

EXAMPLE 10.39
Shorthand notation for currying

To make it easier to declare functions like curried_plus, ML allows a sequence of operands in the formal parameter position of a function declaration:

```
fun curried_plus a b : int = a + b;
==> val curried_plus = fn : int -> int -> int
```

This form is simply shorthand for the declaration in the previous example; it does not declare a function of two arguments. Curried_plus has a single formal parameter, a. Its return value is a function with formal parameter b that in turn returns a + b.

EXAMPLE 10.40
Folding (reduction) in ML

Using tuple notation, our fold function might be declared as follows in ML:

```
fun fold (f, i, l) =
    case l of
        nil => i
      | h :: t => f (h, fold (f, i, t));
==> val fold = fn : (’a * ’b -> ’b) * ’b * ’a list -> ’b
```

EXAMPLE 10.41
Curried fold in ML

The curried version would be declared as follows:

```
fun curried_fold f i l =
    case l of
        nil => i
      | h :: t => f (h, curried_fold f i t);
==> val fold = fn : (’a * ’b -> ’b) -> ’b -> ’a list -> ’b

curried_fold plus;
==> val it = fn : int -> int list -> int
curried_fold plus 0;
==> val it = fn : int list -> int
curried_fold plus 0 [1, 2, 3, 4, 5];
==> val it = 15 : int
```

Note again the difference in the inferred types of the functions.

EXAMPLE 10.42
Currying in ML vs Scheme

It is of course possible to define curried_fold by nesting occurrences of the explicit fn notation within the function’s body. The shorthand notation, however, is substantially more intuitive and convenient. Note also that ML’s syntax for function calls—juxtaposition of function and argument—makes the use of a curried function more intuitive and convenient than it is in Scheme:

```
curried_fold plus 0 [1, 2, 3, 4, 5];    (* ML *)
(((curried_fold +) 0) ’(1 2 3 4 5))     ; Scheme
```

10.6 Theoretical Foundations

EXAMPLE 10.43

Declarative (nonconstructive) function definition

Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*). In conventional notation, we indicate the domain and range of, say, the square root function by writing

$$\mathsf{sqrt} : \mathcal{R} \longrightarrow \mathcal{R}$$

We can also define functions using conventional set notation:

$$\mathsf{sqrt} \equiv \left\{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y > 0 \wedge x = y^2\right\}$$

Unfortunately, this notation is *nonconstructive*: it doesn't tell us how to *compute* square roots. Church designed the lambda calculus to address this limitation. ■

IN MORE DEPTH

Lambda calculus is a *constructive* notation for function definitions. We consider it in more detail on the PLP CD. Any computable function can be written as a lambda expression. Computation amounts to macro substitution of arguments into the function definition, followed by reduction to simplest form via simple and mechanical rewrite rules. The order in which these rules are applied captures the distinction between applicative and normal-order evaluation, as described in Section 6.6.2. Conventions on the use of certain simple functions (e.g., the identity function) allow selection, structures, and even arithmetic to be captured as lambda expressions. Recursion is captured through the notion of *fixed points*.

10.7 Functional Programming in Perspective

Side-effect–free programming is a very appealing idea. As discussed in Sections 6.1.2 and 6.3, side effects can make programs both hard to read and hard to compile. By contrast, the lack of side effects makes expressions *referentially transparent*—independent of evaluation order. Programmers and compilers of a purely functional language can employ *equational reasoning*, in which the equivalence of two expressions at any point in time implies their equivalence at all times. Equational reasoning in turn is highly appealing for parallel execution: In a purely functional language, the arguments to a function can safely be evaluated in parallel with each other. In a lazy functional language, they can be evaluated in parallel with (the beginning of) the function to which they are passed. We will consider these possibilities further in Section 12.4.5.

Unfortunately, there are common programming idioms in which the canonical side effect—assignment—plays a central role. Critics of functional programming often point to these idioms as evidence of the need for imperative language features. I/O is one example. We have seen (in Section 10.4) that sequential access to files can be modeled in a functional manner using streams. For graphics and random file access we have also seen that the monads of Haskell can cleanly isolate the invocation of actions from the bulk of the language, and allow the full power of equational reasoning to be applied to both the computation of values and the determination of the order in which I/O actions should occur.

Other commonly cited examples of "naturally imperative" idioms include:

Initialization of complex structures: The heavy reliance on lists in Lisp, ML, and Haskell reflects the ease with which functions can build new lists out of the components of old lists. Other data structures—multidimensional arrays in particular—are much less easy to put together incrementally, particularly if the natural order in which to initialize the elements is not strictly row-major or column-major.

Summarization: Many programs include code that scans a large data structure or a large amount of input data, counting the occurrences of various items or patterns. The natural way to keep track of the counts is with a dictionary data structure in which one repeatedly updates the count associated with the most recently noticed key.

In-place mutation: In programs with very large data sets, one must economize as much as possible on memory usage, to maximize the amount of data that will fit in memory or the cache. Sorting programs, for example, need to sort in place, rather than copying elements to a new array or list. Matrix-based scientific programs, likewise, need to update values in place.

These last three idioms are examples of what has been called the *trivial update problem*. If the use of a functional language forces the underlying implementation to create a new copy of the entire data structure every time one of its elements must change, then the result will be very inefficient. In imperative programs, the problem is avoided by allowing an existing structure to be modified in place.

One can argue that while the trivial update problem causes trouble in Lisp and its relatives, it does not reflect an inherent weakness of functional programming per se. What is required for a solution is a combination of convenient notation—to access arbitrary elements of a complex structure—and an implementation that is able to determine when the old version of the structure will never be used again, so it can be updated in place instead of being copied.

Sisal, pH, and Single Assignment C (SAC) combine array types and iterative syntax with purely functional semantics. The iterative constructs are defined as syntactic sugar for tail-recursive functions. When nested, these constructs can easily be used to initialize a multidimensional array. The semantics of the language say that each iteration of the loop returns a new copy of the entire array. The compiler can easily verify, however, that the old copy is never used after the return, and can therefore arrange to perform all updates in place. Similar optimizations could be performed in the absence of the imperative syntax, but require somewhat more complex analysis. Cann reports that the Livermore Sisal compiler was able to eliminate 99 to 100% of all copy operations in standard numeric benchmarks [Can92]. Scholz reports performance for SAC competitive with that of carefully optimized modern Fortran programs [Sch03].

Significant strides in both the theory and practice of functional programming have been made in recent years. Wadler [Wad98b] argued in the late 1990s that the principal remaining obstacles to the widespread adoption of functional languages were social and commercial, not technical: most programmers have been trained in an imperative style; software libraries and development environments for functional programming are not yet as mature as those of their imperative cousins. Experience over the past decade appears to have borne out this characterization: with the development of better tools and a growing body of practical experience, functional languages have begun to see much wider use. Functional features have also begun to appear in such mainstream imperative languages as C#, Python, and Ruby.

✓ CHECK YOUR UNDERSTANDING

12. What is the difference between *normal-order* and *applicative-order* evaluation? What is *lazy* evaluation?
13. What is the difference between a function and a *special form* in Scheme?
14. What does it mean for a function to be *strict*?
15. What is *memoization*?

DESIGN & IMPLEMENTATION

Side effects and compilation

As noted in Section 10.2, side-effect freedom has a strong conceptual appeal: it frees the programmer from concern over undocumented access to nonlocal variables, misordered updates, aliases, and dangling pointers. Side-effect freedom also has the potential, at least in theory, to allow the compiler to generate faster code: like aliases, side effects often preclude the caching of values in registers (Section 3.5.1) or the use of constant and copy propagation (Sections ◎16.3 and ◎16.4).

So what are the technical obstacles to generating fast code for functional programs? The trivial update problem is certainly a challenge, as is the cost of heap management for values with unlimited extent. Type checking imposes significant run-time costs in languages descended from Lisp, but not in those descended from ML. Memoization is expensive in Miranda and Haskell, though so-called *strictness analysis* may allow the compiler to eliminate it in cases where applicative order evaluation is provably equivalent. These challenges are all the subject of continuing research.

16. How can one accommodate I/O in a purely functional programming model?
17. What is a *higher-order* function (also known as a *functional form*)? Give three examples.
18. What is *currying*? What purpose does it serve in practical programs?
19. What is the *trivial update problem* in functional programming?
20. Summarize the arguments for and against side-effect–free programming.
21. Why do functional languages make such heavy use of lists?

10.8 Summary and Concluding Remarks

In this chapter we have focused on the functional model of computing. Where an imperative program computes principally through iteration and side effects (i.e., the modification of variables), a functional program computes principally through substitution of parameters into functions. We began by enumerating a list of key issues in functional programming, including first-class and higher-order functions, polymorphism, control flow and evaluation order, and support for list-based data. We then turned to a concrete example—the Scheme dialect of Lisp—to see how these issues may be addressed in a programming language. We also considered, more briefly, ML and its descendants: Miranda, Haskell, Caml, and F#.

For imperative programming languages, the underlying formal model is often taken to be a Turing machine. For functional languages, the model is the lambda calculus. Both models evolved in the mathematical community as a means of formalizing the notion of an effective procedure, as used in constructive proofs. Aside from hardware-imposed limits on arithmetic precision, disk and memory space, and so on, the full power of lambda calculus is available in functional languages. While a full treatment of the lambda calculus could easily consume another book, we provided an overview on the PLP CD. We considered rewrite rules, evaluation order, and the Church-Rosser theorem. We noted that conventions on the use of very simple notation provide the computational power of integer arithmetic, selection, recursion, and structured data types.

For practical reasons, many functional languages extend the lambda calculus with additional features, including assignment, I/O, and iteration. Lisp dialects, moreover, are *homoiconic*: programs look like ordinary data structures, and can be created, modified, and executed on the fly.

Lists feature prominently in most functional programs, largely because they can easily be built incrementally, without the need to allocate and then modify state as separate operations. Many functional languages provide other structured data types as well. In Sisal and Single Assignment C, an emphasis on iterative syntax, tail-recursive semantics, and high-performance compilers allows multidimensional array-based functional programs to achieve performance comparable to that of imperative programs.

10.9 Exercises

10.1 Is the `define` primitive of Scheme an imperative language feature? Why or why not?

10.2 It is possible to write programs in a purely functional subset of an imperative language such as C, but certain limitations of the language quickly become apparent. What features would need to be added to your favorite imperative language to make it genuinely useful as a functional language? (Hint: what does Scheme have that C lacks?)

10.3 Explain the connection between short-circuit Boolean expressions and normal-order evaluation. Why is `cond` a special form in Scheme, rather than a function?

10.4 Write a program in your favorite imperative language that has the same input and output as the Scheme program of Figure 10.1. Can you make any general observations about the usefulness of Scheme for symbolic computation, based on your experience?

10.5 Suppose we wish to remove adjacent duplicate elements from a list (e.g., after sorting). The following Scheme function accomplishes this goal:

```
(define unique
  (lambda (L)
    (cond
      ((null? L) L)
      ((null? (cdr L)) L)
      ((eqv? (car L) (car (cdr L))) (unique (cdr L)))
      (else (cons (car L) (unique (cdr L)))))))
```

Write a similar function that uses the imperative features of Scheme to modify `L` "in place," rather than building a new list. Compare your function to the code above in terms of brevity, conceptual clarity, and speed.

10.6 Write tail-recursive versions of the following:

(a)
```
;; compute integer log, base 2
;; (number of bits in binary representation)
;; works only for positive integers
(define log2
  (lambda (n)
    (if (= n 1) 0 (+ 1 (log2 (quotient (+ n 1) 2))))))
```

(b)
```
;; find minimum element in a list
(define min
  (lambda (l)
    (cond
      ((null? l) '())
      ((null? (cdr l)) (car l))
      (#t (let ((a (car l))
                (b (min (cdr l))))
            (if (< b a) b a))))))
```

10.7 Write purely functional Scheme functions to

(a) return all *rotations* of a given list. For example, `(rotate '(a b c d e))` should return `((a b c d e) (b c d e a) (c d e a b) (d e a b c) (e a b c d))` (in some order).

(b) return a list containing all elements of a given list that satisfy a given predicate. For example, `(filter (lambda (x) (< x 5)) '(3 9 5 8 2 4 7))` should return `(3 2 4)`.

10.8 Write a purely functional Scheme function that returns a list of all permutations of a given list. For example, given `(a b c)` it should return `((a b c) (b a c) (b c a) (a c b) (c a b) (c b a))` (in some order).

10.9 Modify the Scheme program of Figure 10.1 to simulate an NFA (nondeterministic finite automaton), rather than a DFA. (The distinction between these automata is described in Section 2.2.1.) Since you cannot "guess" correctly in the face of a multivalued transition function, you will need either to use explicitly coded backtracking to search for an accepting series of moves (if there is one), or keep track of *all* possible states that the machine could be in at a given point in time.

10.10 Consider the problem of determining whether two trees have the same *fringe*: the same set of leaves in the same order, regardless of internal structure. An obvious way to solve this problem is to write a function `flatten` that takes a tree as argument and returns an ordered list of its leaves. Then we can say

```
(define same-fringe
  (lambda (T1 T2)
    (equal (flatten T1) (flatten T2))))
```

Write a straightforward version of `flatten` in Scheme. How efficient is `same-fringe` when the trees differ in their first few leaves? How would your answer differ in a language like Haskell, which uses lazy evaluation for all arguments? How hard is it to get Haskell's behavior in Scheme, using `delay` and `force`?

10.11 We can use encapsulation within functions to delay evaluation in ML:

```
datatype 'a delayed_list =
    pair of 'a * 'a delayed_list
    | promise of unit -> 'a * 'a delayed_list;
fun head (pair (h, r)) = h
    | head (promise (f)) = let val (a, b) = f () in a end;
fun rest (pair (h, r)) = r
    | rest (promise (f)) = let val (a, b) = f () in b end;
```

Now given

```
fun next_int (n) = (n, promise (fn () => next_int (n + 1)));
val naturals = promise (fn () => next_int (1));
```

we have

```
head (naturals)                 ⟹ 1
head (rest (naturals))          ⟹ 2
head (rest (rest (naturals)))   ⟹ 3
...
```

The delayed list `naturals` is effectively of unlimited length. It will be computed out only as far as actually needed. If a value is needed more than once, however, it will be recomputed every time. Show how to use pointers and assignment (Example 7.78, page 351) to memoize the values of a `delayed_list`, so that elements are computed only once.

10.12 In Example 10.26 we showed how to implement interactive I/O in terms of the lazy evaluation of streams. Unfortunately, our code would not work as written, because Scheme uses applicative-order evaluation. We can make it work, however, with calls to `delay` and `force`.

Suppose we define `input` to be a function that returns an "istream"—a promise that when forced will yield a pair, the `cdr` of which is an istream:

```
(define input (lambda () (delay (cons (read) (input)))))
```

Now we can define the driver to expect an "ostream"—an empty list or a pair, the `cdr` of which is an ostream:

```
(define driver
  (lambda (s)
    (if (null? s) '()
      (display (car s))
      (driver (force (cdr s))))))
```

Note the use of `force`.

Show how to write the function `squares` so that it takes an istream as argument and returns an ostream. You should then be able to type `(driver (squares (input)))` and see appropriate behavior.

10.13 Write new versions of `cons`, `car`, and `cdr` that operate on streams. Using them, rewrite the code of the previous exercise to eliminate the calls to `delay` and `force`. Note that the stream version of `cons` will need to avoid evaluating its second argument; you will need to learn how to define macros (derived special forms) in Scheme.

10.14 Write the standard quicksort algorithm in Scheme, without using any imperative language features. Be careful to avoid the trivial update problem; your code should run in expected time $n \log n$.

Rewrite your code using arrays (you will probably need to consult a Scheme manual for further information). Compare the running time and space requirements of your two sorts.

10.15 Write `insert` and `find` routines that manipulate binary search trees in Scheme (consult an algorithms text if you need more information). Explain why the trivial update problem does *not* impact the asymptotic performance of `insert`.

10.16 Write an LL(1) parser generator in purely functional Scheme. If you consult Figure 2.23, remember that you will need to use tail recursion in place of iteration. Assume that the input CFG consists of a list of lists, one per nonterminal in the grammar. The first element of each sublist should be the nonterminal; the remaining elements should be the right-hand sides of the productions for which that nonterminal is the left-hand side. You may assume that the sublist for the start symbol will be the first one in the list. If we use quoted strings to represent grammar symbols, the calculator grammar of Figure 2.15 would look like this:

```
'(("program"  ("stmt_list" "$$"))
  ("stmt_list" ("stmt" "stmt_list") ())
  ("stmt"  ("id" ":=" "expr") ("read" "id") ("write" "expr"))
  ("expr"  ("term" "term_tail"))
  ("term"  ("factor" "factor_tail"))
  ("term_tail" ("add_op" "term" "term_tail") ())
  ("factor_tail" ("mult_op" "factor" "FT") ())
  ("add_op" ("+") ("-"))
  ("mult_op" ("*") ("/"))
  ("factor"  ("id") ("number") ("(" "expr" ")")))
```

Your output should be a parse table that has this same format, except that every right-hand side is replaced by a *pair* (a two-element list) whose first element is the predict set for the corresponding production, and whose second element is the right-hand side. For the calculator grammar, the table looks like this:

```
(("program" (("$$" "id" "read" "write") ("stmt_list" "$$")))
 ("stmt_list"
  (("id" "read" "write") ("stmt" "stmt_list"))
  (("$$") ()))
```

```
("stmt"
 (("id") ("id" ":=" "expr"))
 (("read") ("read" "id"))
 (("write") ("write" "expr")))
("expr" (("(" "id" "number") ("term" "term_tail")))
("term" (("(" "id" "number") ("factor" "factor_tail")))
("term_tail"
 (("+" "-") ("add_op" "term" "term_tail"))
 (("$$" ")" "id" "read" "write") ()))
("factor_tail"
 (("*" "/") ("mult_op" "factor" "factor_tail"))
 (("$$" ")" "+" "-" "id" "read" "write") ()))
("add_op" (("+") ("+")) (("-") ("-")))
("mult_op" (("*") ("*")) (("/") ("/")))
("factor"
 (("id") ("id"))
 (("number") ("number"))
 (("(") ("(" "expr" ")"))))
```

(Hint: you may want to define a `right_context` function that takes a nonterminal B as argument and returns a list of all pairs (A, β), where A is a nonterminal and β is a list of symbols, such that for some potentially different list of symbols α, $A \longrightarrow \alpha\ B\ \beta$. This function is useful for computing FOLLOW sets. You may also want to build a tail-recursive function that recomputes FIRST and FOLLOW sets until they converge. You will find it easier if you do not include ϵ in either set, but rather keep a separate estimate, for each nonterminal, of whether it may generate ϵ.)

10.17 Write an ML version of the code in Figure 10.1. Alternatively (or in addition), solve Exercises 10.9, 10.10, 10.14, 10.15, or 10.16 in ML.

10.18–10.21 In More Depth.

10.10 Explorations

10.22 Read the original self-definition of Lisp [MAE^{+}65]. Compare it to a similar definition of Scheme [AS96, Chap. 4]. What is different? What has stayed the same? What is built into `apply` and `eval` in each definition? What do you think of the whole idea? Does a metacircular interpreter really define anything, or is it "circular reasoning"?

10.23 Read the Turing Award lecture of John Backus [Bac78], in which he argues for functional programming. How does his FP notation compare to the Lisp and ML language families?

10.24 Learn more about monads in Haskell. Pay particular attention to the definition of lists. Explain the relationship of the list monad to list comprehensions (Example 7.94), iterators, continuations (Section 6.2.2), and backtracking search.

10.25 Read ahead and learn about *transactional memory* (Section 12.4.4). Then read up on STM Haskell [HMPH05]. Explain how monads facilitate the serialization of updates to locations shared between threads.

10.26 We have seen that Lisp and ML include such imperative features as assignment and iteration. How important are these? What do languages like Haskell give up (conversely, what do they gain) by insisting on a purely functional programming style? In a similar vein, what do you think of attempts in several recent imperative languages (notably Python and C#—see the sidebar on page 531) to facilitate functional programming with function constructors and unlimited extent?

10.27 Investigate the compilation of functional programs. What special issues arise? What techniques are used to address them? Starting places for your search might include the compiler texts of Appel [App97], Wilhelm and Maurer [WM95], and Grune et al. [GBJL01].

◎ **10.28–10.30** In More Depth.

10.11 Bibliographic Notes

Lisp, the original functional programming language, dates from the work of McCarthy and his associates in the late 1950s. Bibliographic references for Caml, Erlang, Haskell, Lisp, Miranda, ML, Scheme, Single Assignment C, and Sisal can be found in Appendix A. Historically important dialects of Lisp include Lisp 1.5 [MAE$^+$65], MacLisp [Moo78] (no relation to the Apple Macintosh), and Interlisp [TM81].

The book by Abelson and Sussman [AS96], long used for introductory programming classes at MIT and elsewhere, is a classic guide to fundamental programming concepts, and to functional programming in particular. Additional historical references can be found in the paper by Hudak [Hud89], which surveys the field from the point of view of Haskell.

The lambda calculus was introduced by Church in 1941 [Chu41]. A classic reference is the text of Curry and Feys [CF58]. Barendregt's book [Bar84] is a standard modern reference. Michaelson [Mic89] provides an accessible introduction to the formalism, together with a clear explanation of its relationship to Lisp and ML. Stansifer [Sta95, Sec. 7.6] provides a good informal discussion and correctness proof for the fixed-point combinator **Y** (see Exercise ◎10.9).

John Backus, one of the original developers of Fortran, argued forcefully for a move to functional programming in his 1977 Turing Award lecture [Bac78]. His functional programming notation is known as FP. Peyton Jones [Pey87, Pey92], Wilhelm and Maurer [WM95, Chap. 3], Appel [App97, Chap. 15], and Grune et al. [GBJL01, Chap. 7] discuss the implementation of functional languages. Peyton Jones's paper on the "awkward squad" [Pey01] is widely considered the definitive introduction to monads in Haskell.

While Lisp dates from the early 1960s, it is only in recent years that functional languages have seen widespread use in large commercial systems. Wadler [Wad98a, Wad98b] describes the situation as of the late 1990s, when the tide began to turn. Descriptions of many subsequent projects can be found in the proceedings of the Commercial Users of Functional Programming workshop (*cufp.galois.com*), held annually since 2004. The *Journal of Functional Programming* also publishes a special category of articles on commercial use. Armstrong reports [Arm07] that the Ericsson AXD301, a telephone switching system comprising more than two million lines of Erlang code, has achieved an astonishing "nine nines" level of reliability—the equivalent of less than 32 ms of downtime per year.