CHAPTER 3

Chapter 1 mentioned several different styles of programming, such as functional programming, logic programming, and object-oriented programming. Different languages have evolved to support each style of programming. Each type of language in turn rests on a distinct model of computation, which is different from the underlying von Neumann model reflected by most programming languages and most hardware platforms. In the next three chapters, we explore each of these alternative styles of programming, the types of languages that support them, and their underlying models of computation. In so doing, we will come to a better understanding of power and limits of languages that reflect the von Neumann model as well.

We begin with functional programming. Unlike other styles of programming, functional programming provides a uniform view of programs as functions, the treatment of functions as data, and the prevention of side effects. Generally speaking, a functional programming language has simpler semantics and a simpler model of computation than an imperative language. These advantages make functional languages popular for rapid prototyping, artificial intelligence, mathematical proof systems, and logic applications.

Until recently, the major drawback to functional languages was inefficient execution. Because of their semantics, such languages at first were interpreted rather than compiled, with a resulting substantial loss in execution speed. In the last 20 years, however, advances in compilation techniques for functional languages, plus advances in interpreter technology in cases where compilation is unsuitable or unavailable, have made functional languages very attractive for general programming. Because functional programs lend themselves very well to parallel execution, functional languages have ironically acquired an efficiency advantage over imperative languages in the present era of multicore hardware architectures (see Chapter 13). Additionally, modern functional languages such as those discussed in this chapter include mature application libraries, such as windowing systems, graphics packages, and networking capabilities. If speed is a special concern for part of a program, you can link a functional program to compiled code from other languages such as C. Thus, there is no reason today not to choose a functional language for implementing complex systems—even Web applications. A functional language is a particularly good choice in situations where development time is short and you need to use clear, concise code with predictable behavior.

Still, although functional languages have been around since the 1950s, they have never become mainstream languages. In fact, with the rise of object orientation, programmers have been even less inclined to use functional languages. Why is that? One possible reason is that programmers typically learn to program using an imperative or object-oriented language, so that functional programming seems inherently foreign and intimidating. Another is that object-oriented languages provide a strong organizing principle for structuring code and controlling complexity in ways that mirror the everyday experience of

real objects. Functional programming, too, has strong mechanisms for controlling complexity and structuring code, but they are more abstract and mathematical and, thus, not as easily mastered by the majority of programmers.

Even if you never expect to write "real" applications in a functional language, you should take the time to become familiar with functional programming. Functional methods, such as recursion, functional abstraction, and higher-order functions, have influenced and become part of many programming languages. They should be part of your arsenal of programming techniques.

In this chapter, we review the concept of a function, focusing on how functions can be used to structure programs. We give a brief introduction to the basic principles of functional programming. We then survey three modern functional languages—Scheme, ML, and Haskell—and discuss some of their properties. We conclude with a short introduction to lambda calculus, the underlying mathematical model for functional languages.

3.1 Programs as Functions

A program is a description of a specific computation. If we ignore the details of the computation—the "how" of the computation—and focus on the result being computed—the "what" of the computation—then a program becomes a virtual black box that transforms input into output. From this point of view, a program is essentially equivalent to a mathematical function.

DEFINITION: A ***function*** is a rule that associates to each x from some set X of values a unique y from a set Y of values. In mathematical terminology, if f is the name of the function, we write:

$$y = f(x)$$

or

$$f: X \rightarrow Y$$

The set X is called the **domain** of f, while the set Y is called the **range** of f. The x in $f(x)$, which represents any value from X, is called the **independent variable**, while the y from the set Y, defined by the equation $y = f(x)$, is called the **dependent** variable. Sometimes f is not defined for all x in X, in which case it is called a **partial function** (and a function that is defined for all x in X is called **total**).

Programs, procedures, and functions in a programming language can all be represented by the mathematical concept of a function. In the case of a program, x represents the input and y represents the output. In the case of a procedure or function, x represents the parameters and y represents the returned values. In either case, we can refer to x as "input" and y as "output." Thus, the functional view of programming makes no distinction between a program, a procedure, and a function. It always makes a distinction, however, between input and output values.

In programming languages, we must also distinguish between **function definition** and **function application**. A function definition describes how a value is to be computed using formal

parameters. A function application is a call to a defined function using actual parameters, or the values that the formal parameters assume for a particular computation. Note that, in mathematics, we don't always clearly distinguish between a parameter and a variable. Instead, the term **independent variable** is often used for parameters. For example, in mathematics, we write:

$$\text{square}(x) = x * x$$

for the definition of the squaring function. Then we frequently apply the function to a variable x representing an actual value:

$$\text{Let } x \text{ be such that square}(x) + 2 \ldots$$

In fact, one major difference between imperative programming and functional programming is in how each style of programming treats the concept of a variable. In mathematics, variables always stand for actual values, while in imperative programming languages, variables refer to memory locations that store values. Assignment allows these memory locations to be reset with new values. In mathematics, there are no concepts of memory location and assignment, so that a statement such as

$$x = x + 1$$

makes no sense. Functional programming takes a mathematical approach to the concept of a variable. In functional programming, variables are bound to values, not to memory locations. Once a variable is bound to a value, that variable's value cannot change. This also eliminates assignment, which can reset the value of a variable, as an available operation.

Despite the fact that most functional programming languages retain some notion of assignment, it is possible to create a **pure functional program**—that is, one that takes a strictly mathematical approach to variables.

The lack of assignment in functional programming makes loops impossible, because a loop requires a control variable whose value is reset as the loop executes. So, how do we write repeated operations in functional form? Recursion. For example, we can define a greatest common divisor calculation in C using both imperative and functional form, as shown in Figure 3.1.

```
void gcd( int u, int v, int* x)
{ int y, t, z;
  z = u ; y = v;
  while (y != 0)
  { t = y;
    y = z % y;
    z = t;
  }
  *x = z;
}
```

(a) Imperative version using a loop

```
int gcd( int u,  int v)
{ if (v == 0) return u;
  else return gcd(v, u % v);
}
```

(b) Functional version with recursion

Figure 3.1 C code for a greatest common divisor calculation

The second form (Figure 3.1(b)) is close to the mathematical (that is, recursive) definition of the function as

$$\gcd(u, v) = \begin{cases} u & \text{if } v = 0 \\ \gcd(v, u \bmod v) & \text{otherwise} \end{cases}$$

Another consequence of the lack of assignment is that we can have no notion of the internal state of a function. The value of any function depends only on the values of its arguments (and possibly nonlocal variables, which we examine later). For example, the square root of a given number never changes, because the value of the square root function depends only on the value of its argument. The balance of a bank account changes whenever a deposit is made, however, because this value depends not only on the amount deposited but also on the current balance (the internal state of the account).

The value of any function also cannot depend on the order of evaluation of its arguments. This is one reason for using functional programming for concurrent applications. The property whereby a function's value depends only on the values of its arguments (and nonlocal variables) is called **referential transparency**.[1] For example, the `gcd` function is referentially transparent because its value depends only on the value of its arguments. On the other hand, a function `rand`, which returns a pseudo random value, cannot be referentially transparent, because it depends on the state of the machine (and previous calls to itself). Indeed, a referentially transparent function with no parameters must always return the same value and, thus, is no different from a constant. In fact, a functional language is free to consider a function of no parameters to not be a function at all.[2] Referential transparency and the lack of assignment make the semantics of functional programs particularly straightforward. There is no notion of state, since there is no concept of memory locations with changing values. The runtime environment associates names to values only (not memory locations), and once a name enters the environment, its value can never change. These functional programming semantics are sometimes referred to as **value semantics**, to distinguish them from the more usual storage semantics or pointer semantics. Indeed, the lack of local state in functional programming makes it in a sense the opposite of object-oriented programming, where computation proceeds by changing the local state of objects.

Finally, in functional programming we must be able to manipulate functions in arbitrary ways, without arbitrary restrictions—in other words, functions must be general language objects. In particular, functions must be viewed as values themselves, which can be computed by other functions and which can also be parameters to functions. In other words, in functional programming, functions are **first-class data values**.

As an example, one of the essential operations on functions is **composition**. Composition is itself a function that takes two functions as parameters and produces another function as its returned value. Functions like this—that have parameters that are themselves functions or that produce a result that is a function, or both—are called **higher-order functions**.

Mathematically, the composition operator "o" is defined as follows: If f: $X \to Y$ and g: $Y \to Z$, then $g \; o \; f$: $X \to Z$ is given by $(g \; o \; f)(x) = g(f(x))$.

[1] A slightly different but equivalent definition, called the substitution rule, is given in Chapter 10.

[2] Both ML and Haskell (languages discussed later in this chapter) take this approach; Scheme does not: In Scheme, a parameterless function is different from a constant.

We can summarize the qualities of functional programming languages and functional programs as follows:

1. All procedures are functions and clearly distinguish incoming values (parameters) from outgoing values (results).
2. In pure functional programming, there are no assignments. Once a variable is bound to a value, it behaves like a constant.
3. In pure functional programming, there are no loops. Instead, loops are replaced by recursive calls.
4. The value of a function depends only on the value of its parameters and not on the order of evaluation or the execution path that led to the call.
5. Functions are first-class data values.

3.2 Scheme: A Dialect of Lisp

In the late 1950s and early 1960s, a team at MIT led by John McCarthy developed the first language that contained many of the features of modern functional languages. Based on ideas from mathematics, in particular the lambda calculus of Alonzo Church, it was called Lisp (for LISt Processing) because its basic data structure is a list. Lisp, which first existed as an interpreter on an IBM 704, incorporated a number of features that, strictly speaking, are not aspects of functional programming per se, but that were closely associated with functional languages because of the enormous influence of Lisp. These include:

1. The uniform representation of programs and data using a single general data structure—the list.
2. The definition of the language using an interpreter written in the language itself—called a **metacircular interpreter**.
3. The automatic management of all memory by the runtime system.

Unfortunately, no single standard evolved for the Lisp language, and many different Lisp systems have been created over the years. The original version of Lisp and many of its successors lacked a uniform treatment of functions as first-class data values. Also, they used dynamic scoping for nonlocal references. However, two dialects of Lisp that use static scoping and give a more uniform treatment of functions have become standard. The first, Common Lisp, was developed by a committee in the early 1980s. The second, Scheme, was developed by a group at MIT in the mid-1970s. In the following discussion, we use the definition of the Scheme dialect of Lisp given in Abelson et al. [1998], which is the current standard at the time of this writing.

3.2.1 The Elements of Scheme

All programs and data in Scheme are considered expressions. There are two types of expressions: atoms and parenthesized sequences of expressions. Atoms are like the literal constants and identifiers of an imperative language; they include numbers, characters, strings, names, functions, and a few other constructs we will not mention here. A parenthesized expression is simply a sequence of zero or