CHAPTER

6

Programming Languages

In this chapter we study programming languages. Our purpose is not to focus on a particular language, although we will continue to use examples from Python where appropriate. Rather it is to learn *about* programming languages. We want to appreciate the commonality as well as the diversity among programming languages and their associated methodologies.

The development of complex software systems such as operating systems, network software, and the vast array of application software available today would likely be impossible if humans were forced to write programs in machine language. Dealing with the intricate detail associated with such languages while trying to organize complex systems would be a taxing experience, to say the least.

Consequently, many programming languages have been developed that allow algorithms to be expressed in a form that is both palatable to humans and easily convertible into machine language instructions. Our goal in this chapter is to explore the sphere of computer science that deals with the design and implementation of these languages.

6.1 Historical Perspective

We begin our study by tracing the historical development of programming languages.

Early Generations

As we learned in Chapter 2, programs for modern computers consist of sequences of instructions that are encoded as numeric digits. Such an encoding system is known as a machine language. Unfortunately, writing programs in a machine language is a tedious task that often leads to errors that must be located and corrected (a process known as **debugging**) before the job is finished.

In the 1940s, researchers simplified the programming process by developing notational systems by which instructions could be represented in mnemonic rather than numeric form. For example, the instruction

Move the contents of register 5 to register 6

would be expressed as

```
4056
```

using the machine language introduced in Chapter 2, whereas in a mnemonic system it might appear as

```
MOV R5, R6
```

As a more extensive example, the machine language routine

```
156C
166D
5056
306E
C000
```

which adds the contents of memory cells 6C and 6D and stores the result at location 6E (Figure 2.7 of Chapter 2) might be expressed as

```
LD R5,Price
LD R6,ShippingCharge
ADDI R0,R5 R6
ST R0,TotalCost
HLT
```

using mnemonics. (Here we have used `LD`, `ADDI`, `ST`, and `HLT` to represent *load, add, store,* and *halt.* Moreover, we have used the descriptive names `Price`, `ShippingCharge`, and `TotalCost` to refer to the memory cells at locations 6C, 6D, and 6E, respectively. Such descriptive names are often called program variables or **identifiers.**) Note that the mnemonic form, although still lacking, does a better job of representing the meaning of the routine than does the numeric form.

Once such a mnemonic system was established, programs called **assemblers** were developed to convert mnemonic expressions into machine language instructions. Thus, rather than being forced to develop a program directly in machine language, a human could develop a program in mnemonic form and then have it converted into machine language by means of an assembler.

A mnemonic system for representing programs is collectively called an **assembly language.** At the time assembly languages were first developed, they represented a giant step forward in the search for better programming techniques. In fact, assembly languages were so revolutionary that they became known as second-generation languages, the first generation being the machine languages themselves.

Although assembly languages have many advantages over their machine language counterparts, they still fall short of providing the ultimate programming environment. After all, the primitives used in an assembly language are essentially the same as those found in the corresponding machine language. The difference is simply in the syntax used to represent them. Thus a program written in an assembly language is inherently machine dependent—that is, the instructions within the program are expressed in terms of a particular machine's attributes. In turn, a program written in assembly language cannot be easily transported to another computer design because it must be rewritten to conform to the new computer's register configuration and instruction set.

Another disadvantage of an assembly language is that a programmer, although not required to code instructions in numeric form, is still forced to think in terms of the small, incremental steps of the machine's language. The situation is analogous to designing a house in terms of boards, nails, bricks, and so on. It is true that the actual construction of the house ultimately requires a description based on these elementary pieces, but the design process is easier if we think in terms of larger units such as rooms, windows, doors, and so on.

In short, the elementary primitives in which a product must ultimately be constructed are not necessarily the primitives that should be used during the product's design. The design process is better suited to the use of high-level primitives, each representing a concept associated with a major feature of the product. Once the design is complete, these primitives can be translated to lower-level concepts relating to the details of implementation.

Following this philosophy, computer scientists began developing programming languages that were more conducive to software development than were the low-level assembly languages. The result was the emergence of a third generation of programming languages that differed from previous generations in that their primitives were both higher level (in that they expressed instructions in larger increments) and **machine independent** (in that they did not rely on the characteristics of a particular machine). The best-known early examples are FORTRAN (FORmula TRANslator), which was developed for scientific and engineering applications, and COBOL (COmmon Business-Oriented Language), which was developed by the U.S. Navy for business applications.

In general, the approach to third-generation programming languages was to identify a collection of high-level primitives (in essentially the same spirit with which we developed our pseudocode in Chapter 5) in which software could be developed. Each of these primitives was designed so that it could be implemented as a sequence of the low-level primitives available in machine languages. For example, the statement

```
assign TotalCost the value Price + ShippingCharge
```

expresses a high-level activity without reference to how a particular machine should perform the task, yet it can be implemented by the sequence of machine instructions discussed earlier. Thus, our pseudocode structure

```
identifier = expression
```

is a potential high-level primitive.

Once this collection of high-level primitives had been identified, a program, called a **translator,** was written that translated programs expressed in these high-level primitives into machine-language programs. Such a translator was similar to the second-generation assemblers, except that it often had to compile several machine instructions into short sequences to simulate the activity requested by a single high-level primitive. Thus, these translation programs were often called **compilers.**

An alternative to translators, called **interpreters,** emerged as another means of implementing third-generation languages. These programs were similar to translators except that they executed the instructions as they were translated instead of recording the translated version for future use. That is, rather than producing a machine-language copy of a program that would be executed later, an interpreter actually executed a program from its high-level form.

As a side issue, we should note that the task of promoting third-generation programming languages was not as easy as might be imagined. The thought of writing programs in a form similar to a natural language was so revolutionary that many in managerial positions fought the notion at first. Grace Hopper, who is recognized as the developer of the first compiler, often told the story of demonstrating a translator for a third-generation language in which German terms, rather than English, were used. The point was that the programming language was constructed around a small set of primitives that could be expressed in a variety of natural languages with only simple modifications to the translator. But she was surprised to find that many in the audience were shocked that, in the years surrounding World War II, she would be teaching a computer to "understand" German. Today we know that understanding a natural language involves much, much more than responding to a few rigorously defined primitives. Indeed, **natural languages** (such as English, German, and Latin) are distinguished from **formal languages** (such as programming languages) in that the latter are precisely defined by grammars (Section 6.4), whereas the former evolved over time without formal grammatical analysis.

Machine Independence and Beyond

With the development of third-generation languages, the goal of machine independence was largely achieved. Since the statements in a third-generation language did not refer to the attributes of any particular machine, they could be

Cross-Platform Software

A typical application program must rely on the operating system to perform many of its tasks. It may require the services of the window manager to communicate with the computer user, or it may use the file manager to retrieve data from mass storage. Unfortunately, different operating systems dictate that requests for these services be made in different ways. Thus for programs to be transferred and executed across networks and internets involving different machine designs and different operating systems, the programs must be operating-system independent as well as machine independent. The term *cross-platform* is used to reflect this additional level of independence. That is, cross-platform software is software that is independent of an operating system's design as well as the machine's hardware design and is therefore executable throughout a network.

compiled as easily for one machine as for another. A program written in a third-generation language could theoretically be used on any machine simply by applying the appropriate compiler.

Reality, however, has not proven to be this simple. When a compiler is designed, particular characteristics of the underlying machine are sometimes reflected as conditions on the language being translated. For example, the different ways in which machines handle I/O operations have historically caused the "same" language to have different characteristics, or dialects, on different machines. Consequently, it is often necessary to make at least minor modifications to a program to move it from one machine to another.

Compounding this problem of portability is the lack of agreement in some cases as to what constitutes the correct definition of a particular language. To aid in this regard, the American National Standards Institute and the International Organization for Standardization have adopted and published standards for many of the popular languages. In other cases, informal standards have evolved because of the popularity of a certain dialect of a language and the desire of other compiler writers to produce compatible products. However, even in the case of highly standardized languages, compiler designers often provide features, sometimes called language extensions, that are not part of the standard version of the language. If a programmer takes advantage of these features, the program produced will not be compatible with environments using a compiler from a different vendor.

In the overall history of programming languages, the fact that third-generation languages fell short of true machine independence is actually of little significance for two reasons. First, they were close enough to being machine independent that software could be transported from one machine to another with relative ease. Second, the goal of machine independence turned out to be only a seed for more demanding goals. Indeed, the realization that machines could respond to such high-level statements as

```
assign TotalCost the value Price + ShippingCharge
```

led computer scientists to dream of programming environments that would allow humans to communicate with machines in terms of abstract concepts rather than forcing them to translate these concepts into machine-compatible form.

Moreover, computer scientists wanted machines that could perform much of the algorithm discovery process rather than just algorithm execution. The result has been an ever-expanding spectrum of programming languages that challenges a clear-cut classification in terms of generations.

Programming Paradigms

The generation approach to classifying programming languages is based on a linear scale (Figure 6.1) on which a language's position is determined by the degree to which the user of the language is freed from the world of computer gibberish and allowed to think in terms associated with the problem being solved. In reality, the development of programming languages has not progressed in this manner but has developed along different paths as alternative approaches to the programming process (called **programming paradigms**) have surfaced and been pursued. Consequently, the historical development of programming languages is better represented by a multiple-track diagram as shown in Figure 6.2, in which different paths resulting from different paradigms are shown to emerge and progress independently. In particular, the figure presents four paths representing the functional, object-oriented, imperative, and declarative paradigms, with various languages associated with each paradigm positioned in a manner that indicates their births relative to other languages. (It does not imply that one language necessarily evolved from a previous one.)

We should note that although the paradigms identified in Figure 6.2 are called *programming* paradigms, these alternatives have ramifications beyond the programming process. They represent fundamentally different approaches to building solutions to problems and therefore affect the entire software development process. In this sense, the term *programming paradigm* is a misnomer. A more realistic term would be *software development paradigm.*

The **imperative paradigm,** also known as the **procedural paradigm,** represents the traditional approach to the programming process. It is the paradigm on which Python and our pseudocode of Chapter 5 are based as well as the machine language discussed in Chapter 2. As the name suggests, the imperative paradigm defines the programming process to be the development of a sequence of commands that, when followed, manipulate data to produce the desired result. Thus the imperative paradigm tells us to approach the programming process by finding an algorithm to solve the problem at hand and then expressing that algorithm as a sequence of commands.

Figure 6.1 Generations of programming languages

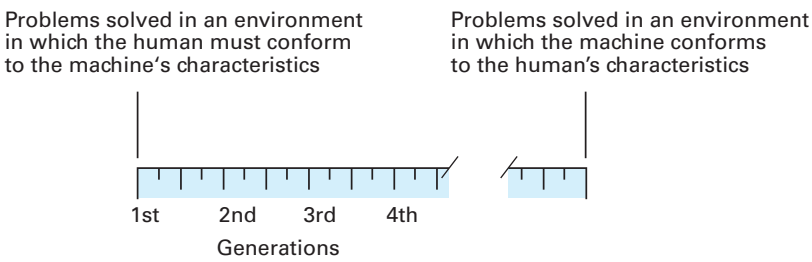
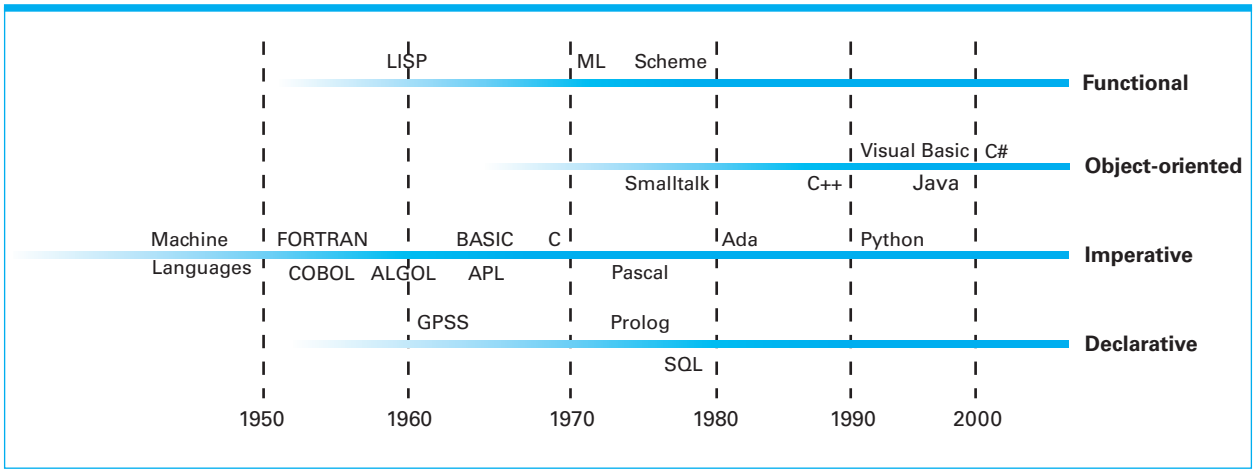


Figure 6.2 The evolution of programming paradigms



In contrast to the imperative paradigm is the **declarative paradigm,** which asks a programmer to describe the problem to be solved rather than an algorithm to be followed. More precisely, a declarative programming system applies a pre-established general-purpose problem-solving algorithm to solve problems presented to it. In such an environment the task of a programmer becomes that of developing a precise statement of the problem rather than of describing an algorithm for solving the problem.

A major obstacle in developing programming systems based on the declarative paradigm is the need for an underlying problem-solving algorithm. For this reason early declarative programming languages tended to be special-purpose in nature, designed for use in particular applications. For example, the declarative approach has been used for many years to simulate a system (political, economic, environmental, and so on) in order to test hypotheses or to obtain predictions. In these settings, the underlying algorithm is essentially the process of simulating the passage of time by repeatedly recomputing values of parameters (gross domestic product, trade deficit, and so on) based on the previously computed values. Thus, implementing a declarative language for such simulations requires that one first implement an algorithm that performs this repetitive function. Then the only task required of a programmer using the system is to describe the situation to be simulated. In this manner, a weather forecaster does not need to develop an algorithm for forecasting the weather but merely describes the current weather status, allowing the underlying simulation algorithm to produce weather predictions for the near future.

A tremendous boost was given to the declarative paradigm with the discovery that the subject of formal logic within mathematics provides a simple problem-solving algorithm suitable for use in a general-purpose declarative programming system. The result has been increased attention to the declarative paradigm and the emergence of **logic programming,** a subject discussed in Section 6.7.

Another programming paradigm is the **functional paradigm.** Under this paradigm a program is viewed as an entity that accepts inputs and produces outputs. Mathematicians refer to such entities as functions, which is the reason this approach is called the functional paradigm. Under this paradigm a program is constructed by connecting smaller predefined program units (predefined

functions) so that each unit's outputs are used as another unit's inputs in such a way that the desired overall input-to-output relationship is obtained. In short, the programming process under the functional paradigm is that of building functions as nested complexes of simpler functions.

As an example, Figure 6.3 shows how a function for balancing your checkbook can be constructed from two simpler functions. One of these, called `Find_sum`, accepts values as its input and produces the sum of those values as its output. The other, called `Find_diff`, accepts two input values and computes their difference. The structure displayed in Figure 6.3 can be represented in the LISP programming language (a prominent functional programming language) by the expression

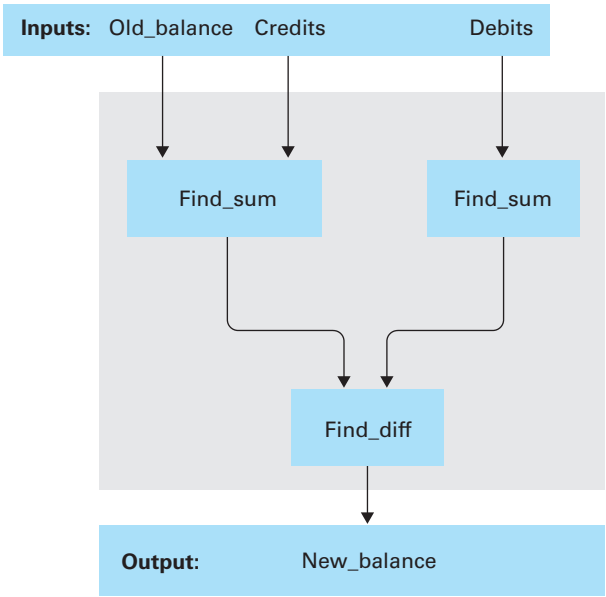
```
(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))
```

The nested structure of this expression (as indicated by parentheses) reflects the fact that the inputs to the function `Find_diff` are produced by two applications of `Find_sum`. The first application of `Find_sum` produces the result of adding all the `Credits` to the `Old_balance`. The second application of `Find_sum` computes the total of all `Debits`. Then, the function `Find_diff` uses these results to obtain the new checkbook balance.

To more fully understand the distinction between the functional and imperative paradigms, let us compare the functional program for balancing a checkbook to the following pseudocode program obtained by following the imperative paradigm:

```
Total_credits = sum of all Credits
Temp_balance = Old_balance + Total_credits
Total_debits = sum of all Debits
Balance = Temp_balance – Total_debits
```

Figure 6.3 A function for checkbook balancing constructed from simpler functions



Note that this imperative program consists of multiple statements, each of which requests that a computation be performed and that the result be stored for later use. In contrast, the functional program consists of a single statement in which the result of each computation is immediately channeled into the next. In a sense, the imperative program is analogous to a collection of factories, each converting its raw materials into products that are stored in warehouses. From these warehouses, the products are later shipped to other factories as they are needed. But the functional program is analogous to a collection of factories that are coordinated so that each produces only those products that are ordered by other factories and then immediately ships those products to their destinations without intermediate storage. This efficiency is one of the benefits proclaimed by proponents of the functional paradigm.

Still another programming paradigm (and the most prominent one in today's software development) is the **object-oriented paradigm,** which is associated with the programming process called **object-oriented programming (OOP).** Following this paradigm, a software system is viewed as a collection of units, called **objects,** each of which is capable of performing the actions that are immediately related to itself as well as requesting actions of other objects. Together, these objects interact to solve the problem at hand.

As an example of the object-oriented approach at work, consider the task of developing a graphical user interface. In an object-oriented environment, the icons that appear on the screen would be implemented as objects. Each of these objects would encompass a collection of functions (called **methods** in the object-oriented vernacular) describing how that object is to respond to the occurrence of various events, such as being selected by a click of the mouse button or being dragged across the screen by the mouse. Thus the entire system would be constructed as a collection of objects, each of which knows how to respond to the events related to it.

To contrast the object-oriented paradigm with the imperative paradigm, consider a program involving a list of names. In the traditional imperative paradigm, this list would be merely a collection of data. Any program unit accessing the list would have to contain the algorithms for performing the required manipulations. In the object-oriented approach, however, the list would be constructed as an object that consisted of the list together with a collection of methods for manipulating the list. (This might include functions for inserting a new entry in the list, deleting an entry from the list, detecting if the list is empty, and sorting the list.) In turn, another program unit that needed to manipulate the list would not contain algorithms for performing the pertinent tasks. Instead, it would make use of the functions provided in the object. In a sense, rather than sorting the list as in the imperative paradigm, the program unit would ask the list to sort itself. Although we will discuss the object-oriented paradigm in more detail in Section 6.5, its significance in today's software development arena dictates that we include the concept of a class in this introduction. To this end, recall that an object can consist of data (such as a list of names) together with a collection of methods for performing activities (such as inserting new names in the list). These features must be described by statements in the written program. This description of the object's properties is called a **class.** Once a class has been constructed, it can be applied anytime an object with those characteristics is needed. Thus, several objects can be based on (that is, built from) the same class. Just like

identical twins, these objects would be distinct entities but would have the same characteristics because they are constructed from the same template (the same class). (An object that is based on a particular class is said to be an **instance** of that class.)

It is because objects are well-defined units whose descriptions are isolated in reusable classes that the object-oriented paradigm has gained popularity. Indeed, proponents of object-oriented programming argue that the object-oriented paradigm provides a natural environment for the "building block" approach to software development. They envision software libraries of predefined classes from which new software systems can be constructed in the same way that many traditional products are constructed from off-the-shelf components. Building and expanding such libraries is an ongoing process, as we will learn in Chapter 7.

In closing, we should note that the methods within an object are essentially small imperative program units. This means that most programming languages based on the object-oriented paradigm contain many of the features found in imperative languages. For instance, the popular object-oriented language C++ was developed by adding object-oriented features to the imperative language known as C. Moreover, since Java and C# are derivatives of C++, they too have inherited this imperative core. In Sections 6.2 and 6.3 we will explore many of these imperative features, and in so doing, we will be discussing concepts that permeate a vast majority of today's object-oriented software. Then, in Section 6.5, we will consider features that are unique to the object-oriented paradigm.

Questions & Exercises

1. In what sense is a program in a third-generation language machine independent? In what sense is it still machine dependent?
2. What is the difference between an assembler and a compiler?
3. We can summarize the imperative programming paradigm by saying that it places emphasis on describing a process that leads to the solution of the problem at hand. Give a similar summary of the declarative, functional, and object-oriented paradigms.
4. In what sense are the third-generation programming languages at a higher level than the earlier generations?

6.2 Traditional Programming Concepts

In this section we consider some of the concepts found in imperative as well as object-oriented programming languages. For this purpose we will draw examples from the languages Ada, C, C++, C#, FORTRAN, and Java. Our goal is not to become entangled in the details of any particular language but merely to demonstrate how common language features appear in a variety of actual languages. Our collection of languages is therefore chosen to be representative of the landscape. C is a third-generation imperative language. C++ is an object-oriented language that

was developed as an extension of the language C. Java and C# are object-oriented languages derived from C++. (Java was developed at Sun Microsystems, which was later purchased by Oracle, whereas C# is a product of Microsoft.) FORTRAN and Ada were originally designed as third-generation imperative languages although their newer versions have expanded to encompass most of the object-oriented paradigm. Appendix D contains a brief background of each of these languages.

Even though we are including object-oriented languages such as C++, Java, and C# among our example languages, we will approach this section as though we were writing a program in the imperative paradigm, because many units within an object-oriented program (such as the functions describing how an object should react to an outside stimulus) are essentially short imperative programs. Later, in Section 6.5, we will focus on features unique to the object-oriented paradigm.

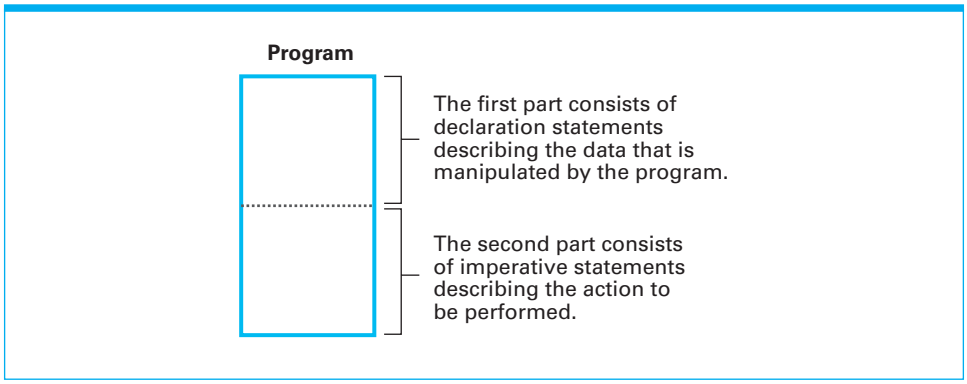
Generally, a program consists of a collection of statements that tend to fall into three categories: declarative statements, imperative statements, and comments. **Declarative statements** define customized terminology that is used later in the program, such as the names used to reference data items; **imperative statements** describe steps in the underlying algorithms; and **comments** enhance the readability of a program by explaining its esoteric features in a more human-compatible form. Normally, an imperative program (or an imperative program unit within an object-oriented program) can be thought of as having the structure depicted in Figure 6.4. It begins with a collection of declarative statements describing the data to be manipulated by the program. This preliminary material is followed by imperative statements that describe the algorithm to be executed. Many languages now allow the declarative and imperative statements to be freely intermingled, but the conceptual distinction remains. Comment statements are dispersed as needed to clarify the program.

Following this lead, we approach our study of programming concepts by considering statement categories in the order in which we might encounter them in a program, beginning with concepts associated with declaration statements.

Variables and Data Types

As we learned in Section 1.8, high-level programming languages allow locations in main memory to be referenced by descriptive names rather than by numeric addresses. Such a name is known as a **variable,** in recognition of the fact that

Figure 6.4 The composition of a typical imperative program or program unit



Scripting Languages

A subset of the imperative programming languages is the collection of languages known as **scripting languages.** These languages are typically used to perform administrative tasks rather than to develop complex programs. The expression of such a task is known as a **script,** which explains the term *scripting language.* For example, the administrator of a computer system might write a script to describe a sequence of record-keeping activities that should be performed every evening, or the user of a PC might write a script to direct the execution of a sequence of programs required to read pictures from a digital camera, index the pictures by date, and store copies of them in an archival storage system. The origin of scripting languages can be traced to the job control languages of the 1960s that were used to direct an operating system in the scheduling of batch processing jobs (see Section 3.1). Even today, many consider scripting languages to be languages for directing the execution of other programs, which is a rather restrictive view of current scripting languages. Examples of scripting languages include Perl and PHP, both of which are popular in controlling server-side Web applications (see Section 4.3), as well as VBScript, which is a dialect of Visual Basic that was developed by Microsoft and is used in Windows-specific situations.

by changing the value stored at the location, the value associated with the name changes as the program executes. Unlike Python, our example languages in this chapter require that variables be identified via a declarative statement prior to being used elsewhere in the program. These declarative statements also require that the programmer describe the type of data that will be stored at the memory location associated with the variable.

Such a type is known as a **data type** and encompasses both the manner in which the data item is encoded and the operations that can be performed on that data. For example, the type **integer** refers to numeric data consisting of whole numbers, probably stored using two's complement notation. Operations that can be performed on integer data include the traditional arithmetic operations and comparisons of relative size, such as determining whether one value is greater than another. The type **float** (sometimes called **real**) refers to numeric data that might contain values other than whole numbers, probably stored in floating-point notation. Operations performed on data of type float are similar to those performed on data of type integer. Recall, however, that the activity required for adding two items of type float differs from that for adding two items of type integer.

Suppose, then, that we wanted to use the variable `WeightLimit` in a program to refer to an area of main memory containing a numeric value encoded in two's complement notation. In the languages C, C++, Java, and C# we would declare our intention by inserting the statement

```
int WeightLimit;
```

toward the beginning of the program. This statement means "The name `WeightLimit` will be used later in the program to refer to a memory area containing a value stored in two's complement notation." Multiple variables of the same

type can normally be declared in the same declaration statement. For example, the statement

```
int Height, Width;
```

would declare both `Height` and `Width` to be variables of type integer. Moreover, most languages allow a variable to be assigned an initial value when it is declared. Thus,

```
int WeightLimit = 100;
```

would not only declare `WeightLimit` to be a variable of type integer but also assign it the starting value 100. In contrast, dynamically typed languages like Python allow variables to be assigned without first declaring their type; such variables are checked for correct type later, when operations are performed upon them.

Other common data types include character and Boolean. The type character refers to data consisting of symbols, probably stored using ASCII or Unicode. Operations performed on such data include comparisons such as determining whether one symbol occurs before another in alphabetical order, testing to see whether one string of symbols appears inside another, and concatenating one string of symbols at the end of another to form one long string. The statement

```
char Letter, Digit;
```

could be used in the languages C, C++, C#, and Java to declare the variables `Letter` and `Digit` to be of type character.

The type **Boolean** refers to data items that can take on only the values true or false. Operations on data of type Boolean include inquiries as to whether the current value is true or false. For example, if the variable `LimitExceeded` was declared to be of type Boolean, then a statement of the form

```
if (LimitExceeded) then (...) else (...)
```

would be reasonable.

The data types that are included as primitives in a programming language, such as `int` for integer and `char` for character, are called **primitive data types.** As we have learned, the types integer, float, character, and Boolean are common primitives. Other data types that have not yet become widespread primitives include images, audio, video, and hypertext. However, types such as GIF, JPEG, and HTML might soon become as common as integer and float. Later (Sections 6.5 and Section 8.4) we will learn how the object-oriented paradigm enables a programmer to extend the repertoire of available data types beyond the primitive types provided in a language. Indeed, this ability is a celebrated trait of the object-oriented paradigm.

In summary, the following program segment, expressed in the language C and its derivatives C++, C#, and Java, declares the variables `Length` and `Width` to be of type `float`, the variables `Price`, `Tax`, and `Total` to be of type integer, and the variable `Symbol` to be of type character.

```
float Length, Width;
int   Price, Tax, Total;
char  Symbol;
```

In Section 6.4 we will see how a translator uses the knowledge that it gathers from such declaration statements to help it translate a program from a high-level language into machine language. For now, we note that such information can be used

to identify errors. For example, if a translator found a statement requesting the addition of two variables that had been declared earlier to be of type Boolean, it should probably consider the statement to be in error and report this finding to the user.

Data Structure

In addition to data type, variables in a program are often associated with **data structure,** which is the conceptual shape or arrangement of data. For example, text is normally viewed as a long string of characters, whereas sales records might be envisioned as a rectangular table of numeric values, where each row represents the sales made by a particular employee and each column represents the sales made on a particular day.

One common data structure is the **array,** which is a block of elements of the same type such as a one-dimensional list, a two-dimensional table with rows and columns, or tables with higher dimensions. To establish such an array in a program, many programming languages require that the declaration statement declaring the name of the array also specify the length of each dimension of the array. For example, Figure 6.5 displays the conceptual structure declared by the statement

```
int Scores[2][9];
```

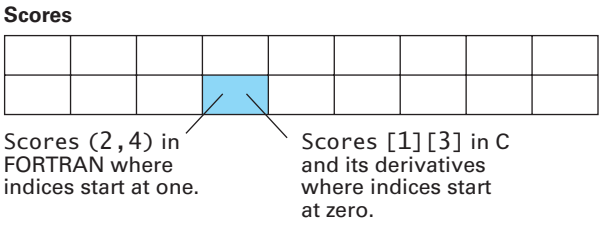
in the language C, which means "The variable `Scores` will be used in the following program unit to refer to a two-dimensional array of integers having two rows and nine columns." The same statement in FORTRAN would be written as

```
INTEGER Scores(2,9)
```

Once an array has been declared, it can be referenced elsewhere in the program by its name, or an individual element can be identified by means of integer values called **indices** that specify the row, column, and so on, desired. However, the range of these indices varies from language to language. For example, in C (and its derivatives C++, Java, and C#) indices start at 0, meaning that the entry in the second row and fourth column of the array called `Scores` (as declared above) would be referenced by `Scores[1][3]`, and the entry in the first row and first column would be `Scores[0][0]`. In contrast, indices start at 1 in a FORTRAN program so the entry in the second row and fourth column would be referenced by `Scores(2,4)` (see again Figure 6.5).

In contrast to an array in which all data items are the same type, an **aggregate type** (also called a **structure,** a **record,** or sometimes a **heterogeneous array**)

Figure 6.5 A two-dimensional array with two rows and nine columns



is a block of data in which different elements can have different types. For instance, a block of data referring to an employee might consist of an entry called `Name` of type character, an entry called `Age` of type integer, and an entry called `SkillRating` of type float. Such an aggregate type would be declared in C by the statement

```
struct { char  Name[25];
         int Age;
         float SkillRating;
       } Employee;
```

which says that the variable `Employee` is to refer to a structure (abbreviated `struct`) consisting of three components called `Name` (a string of 25 characters), `Age`, and `SkillRating` (Figure 6.6). Once such an aggregate has been declared, a programmer can use the structure name (`Employee`) to refer to the entire aggregate or can reference individual **fields** within the aggregate by means of the structure name followed by a period and the field name (such as `Employee.Age`).

In Chapter 8 we will see how conceptual constructs such as arrays are actually implemented inside a computer. In particular, we will learn that the data contained in an array might be scattered over a wide area of main memory or mass storage. This is why we refer to data structure as being the *conceptual* shape or arrangement of data. Indeed, the actual arrangement within the computer's storage system might be quite different from its conceptual arrangement.

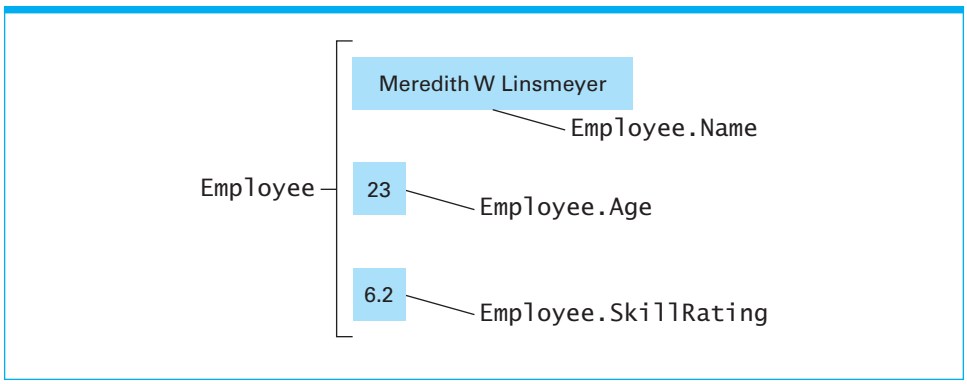
Constants and Literals

Sometimes a fixed, predetermined value is used in a program. For example, a program for controlling air traffic in the vicinity of a particular airport might contain numerous references to that airport's altitude above sea level. When writing such a program, one can include this value, say 645 feet, literally each time it is required. Such an explicit appearance of a value is called a **literal.** The use of literals leads to program statements such as

```
EffectiveAlt = Altimeter + 645
```

where `EffectiveAlt` and `Altimeter` are assumed to be variables and 645 is a literal. Thus, this statement asks that the variable `EffectiveAlt` be assigned the result of adding 645 to the value assigned to the variable `Altimeter`.

Figure 6.6 The conceptual layout of the structure Employee



In most programming languages, literals consisting of text are delineated with single or double quotation marks to distinguish them from other program components. For instance, the statement

```
LastName = 'Smith'
```

might be used to assign the text "Smith" to the variable `LastName`, whereas the statement

```
LastName = Smith
```

would be used to assign the value of the variable `Smith` to the variable `LastName`.

Often, the use of literals is not good programming practice because literals can mask the meaning of the statements in which they appear. How, for instance, can a reader of the statement

```
EffectiveAlt = Altimeter + 645
```

know what the value 645 represents? Moreover, literals can complicate the task of modifying the program should it become necessary. If our air traffic program is moved to another airport, all references to the airport's altitude must be changed. If the literal 645 is used in each reference to that altitude, each such reference throughout the program must be located and changed. The problem is compounded if the literal 645 also occurs in reference to a quantity other than the airport's altitude. How do we know which occurrences of 645 to change and which to leave alone?

To solve these problems, programming languages allow descriptive names to be assigned to specific, nonchangeable values. Such a name is called a **constant.** As an example, in C++ and C#, the declarative statement

```
const int AirportAlt = 645;
```

associates the identifier `AirportAlt` with the fixed value 645 (which is considered to be of type integer). The similar concept in Java is expressed by

```
final int AirportAlt = 645;
```

Following such declarations, the descriptive name `AirportAlt` can be used in lieu of the literal 645. Using such a constant in our pseudocode, the statement

```
EffectiveAlt = Altimeter + 645
```

could be rewritten as

```
EffectiveAlt = Altimeter + AirportAlt
```

which better represents the meaning of the statement. Moreover, if such constants are used in place of literals and the program is moved to another airport whose altitude is 267 feet, then changing the single declarative statement in which the constant is defined is all that is needed to convert all references to the airport's altitude to the new value.

Assignment Statements

Once the special terminology to be used in a program (such as the variables and constants) has been declared, a programmer can begin to describe the algorithms involved. This is done by means of imperative statements. The most basic imperative statement is the **assignment statement,** which requests that a value be assigned to a variable (or more precisely, stored in the memory area identified

by the variable). Such a statement normally takes the syntactic form of a variable, followed by a symbol representing the assignment operation, and then by an expression indicating the value to be assigned. The semantics of such a statement is that the expression is to be evaluated and the result stored as the value of the variable. For example, the statement

```
Z = X + Y;
```

in C, C++, C#, and Java requests that the sum of X and Y be assigned to the variable Z. The semicolon at the end of the line, which is used to separate statements in many imperative languages, is the only syntactic difference from an equivalent Python assignment statement. In some other languages (such as Ada) the equivalent statement would appear as

```
Z := X + Y;
```

Note that these statements differ only in the syntax of the assignment operator, which in C, C++, C#, and Java is merely an equal sign but in Ada is a colon followed by an equal sign. Perhaps a better notation for the assignment operator is found in APL, a language that was designed by Kenneth E. Iverson in 1962. (APL stands for A Programming Language.) It uses an arrow to represent assignment. Thus, the preceding assignment would be expressed as

```
Z ← X + Y
```

in APL. Unfortunately, the "←" symbol has historically not been available on most keyboards.

Much of the power of assignment statements comes from the scope of expressions that can appear on the right side of the statement. In general, any algebraic expression can be used, with the arithmetic operations of addition, subtraction, multiplication, and division typically represented by the symbols +, −, *, and /, respectively. In some languages (including Ada and Python, but not including C or Java), the combination ** is used to represent exponentiation, so the expression

```
x ** 2
```

represents x^2. Languages differ, however, in the manner in which algebraic expressions are interpreted. For example, the expression 2 * 4 + 6/2 could produce the value 14 if it is evaluated from right to left, or 7 if evaluated from left to right. These ambiguities are normally resolved by rules of **operator precedence,** meaning that certain operations are given precedence over others. The traditional rules of algebra dictate that multiplication and division have precedence over addition and subtraction. That is, multiplications and divisions are performed before additions and subtractions. Following this convention, the preceding expression would produce the value 11. In most languages, parentheses can be used to override the language's operator precedence. Thus 2 * (4 + 6)/2 would produce the value 10.

Many programming languages allow the use of one symbol to represent more than one operation. In these cases the meaning of the symbol is determined by the data type of the operands. For example, the symbol + traditionally indicates addition when its operands are numeric, but in some languages, such as Java and Python, the symbol indicates concatenation when its operands are character strings. That is, the result of the expression

```
'abra' + 'cadabra'
```

is *abracadabra.* Such multiple use of an operation symbol is called **overloading.** While many languages provide built-in overloading of a few common operators,

others such as Ada, C++, and C# may allow programmers to define additional overloaded meanings or even add additional operators.

Control Statements

A **control statement** alters the execution sequence of the program. Of all the programming constructs, those from this group have probably received the most attention and generated the most controversy. The major villain is the simplest control statement of all, the `goto` statement. It provides a means of directing the execution sequence to another location that has been labeled for this purpose by a name or number. It is therefore nothing more than a direct application of the machine-level JUMP instruction. The problem with such a feature in a high-level programming language is that it allows programmers to write a rat's nest like

```
    goto 40
20  Evade()
    goto 70
40  if (KryptoniteLevel < LethalDose) then goto 60
    goto 20
60  RescueDamsel()
70  ...
```

when a single statement such as

```
if (KryptoniteLevel < LethalDose):
  RescueDamsel()
else:
  Evade()
```

does the job.

To avoid such complexities, modern languages are designed with control statements that allow an entire branching pattern to be expressed within a single lexical structure. The choice of which control statements to incorporate into a language is a design decision. The goal is to provide a language that not only allows algorithms to be expressed in a readable form but also assists the programmer in obtaining such readability. This is done by restricting the use of those features that have historically led to sloppy programming while encouraging the use of better-designed features. The result is the practice known as **structured programming,** which encompasses an organized design methodology combined with the appropriate use of the language's control statements. The idea is to produce a program that can be readily comprehended and shown to meet its specifications.

We have already met two popular branching structures in Chapter 5, represented by the `if-else` and `while` statements. These are present in almost all imperative, functional, or object-oriented languages. More precisely, the Python statements

```
if (condition):
  statementA
else:
  statementB
```

and

```
while (condition):
  body
```

Programming Language Cultures

As with natural languages, users of different programming languages tend to develop cultural differences and often debate the merits of their perspectives. Sometimes these differences are significant as, for instance, when different programming paradigms are involved. In other cases, the distinctions are subtle. For example, whereas Pascal distinguishes between procedures and functions (Section 6.3), C and Python programmers refer to both as functions. This is because a procedure in a Python program is declared in the same way as a function, but without defining a returned value. A similar example is that C++ programmers refer to a procedure within an object as a member function, whereas the generic term for this is method. This discrepancy can be traced to the fact that C++ was developed as an extension of C. Another cultural difference is that programs in Ada are normally typeset with reserved words in either uppercase or bold—a tradition that is not widely practiced by users of C, C++, C#, FORTRAN, or Java.

Although this book uses Python as an exemplar in most chapters, each specific example is presented in a form that is compatible with the style of the language involved. As you encounter these examples, you should keep in mind that they are presented as examples of how generic ideas appear in actual languages—not as a means of teaching the details of a particular language. Try to look at the forest rather than the trees.

would be written as

```
if (condition) statementA; else statementB;
```

and

```
while (condition) { body }
```

in C, C++, C#, and Java. Note that the fact that these statements are identical in all four languages is a consequence of the fact that C++, C#, and Java are object-oriented extensions of the imperative language C. In contrast, the corresponding statements would be written as

```
IF condition THEN
  statementA;
ELSE
  statementB;
END IF;
```

and

```
WHILE condition LOOP
  body
END LOOP;
```

in the language Ada.

Another common branching structure is often represented by a `switch` or `case` statement. It provides a means of selecting one statement sequence among several options, depending on the value assigned to a designated variable. For example, the statement

```
switch (variable) {
  case 'A': statementA; break;
  case 'B': statementB; break;
  case 'C': statementC; break;
  default: statementD;}
```

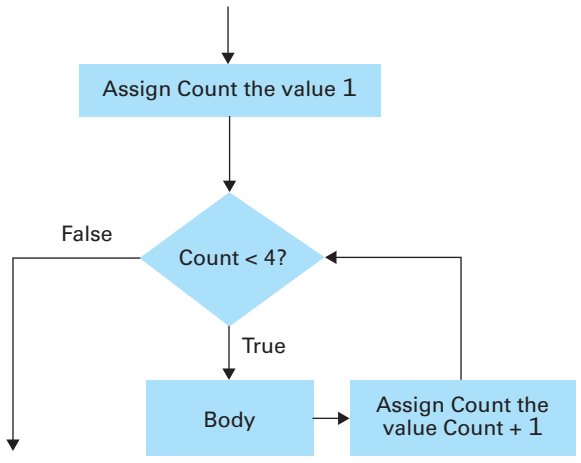
in C, C++, C#, and Java requests the execution of `statementA`, `statementB`, or `statementC` depending on whether the current value of `variable` is A, B, or C, respectively or the execution of `statementD` if the value of `variable` is something else. The same structure would be expressed as

```
CASE variable IS
  WHEN 'A'=> statementA;
  WHEN 'B'=> statementB;
  WHEN 'C'=> statementC;
  WHEN OTHERS=> statementD;
END CASE;
```

in Ada.

Still another common control structure, often called the `for` loop, is shown in Figure 6.7 along with its representation in C++, C#, and Java. This structure differs from the Python `for` structure introduced in Chapter 5. Instead of implicitly setting up the initialization, modification, and termination components for a loop that iterates through a list of data, the C family `for` structure explicitly incorporates initialization, modification, and termination of the loop in a single statement. Such a statement is convenient when the body of the loop is to be performed once for each value within a specific range. In particular, the statements in Figure 6.7 direct that the loop body be performed repeatedly—first with the value of `Count` being 1, then with the value of `Count` being 2, and again with the value of `Count`

Figure 6.7 The for loop structure and its representation in C++, C#, and Java




```
for (int Count = 1; Count < 4; Count++)
     body ;
```

being 3. Programming languages that incorporate both kinds of `for` structure may differentiate between them with syntax, such as `foreach` or `for...in`.

The point to be made from the examples we have cited is that common branching structures appear, with slight variations, throughout the gamut of imperative and object-oriented programming languages. A somewhat surprising result from theoretical computer science is that only a few of these structures are needed to ensure that a programming language provides a means of expressing a solution to any problem that has an algorithmic solution. We will investigate this claim in Chapter 12. For now, we merely point out that learning a programming language is not an endless task of learning different control statements. Most of the control structures found in today's programming languages are essentially variations of those we have identified here.

Comments

No matter how well a programming language is designed and how well the language's features are applied in a program, additional information is usually helpful or mandatory when a human tries to read and understand the program. For this reason, programming languages provide ways of inserting explanatory statements, called **comments,** within a program. These statements are ignored by a translator, and therefore their presence or absence does not affect the program from a machine's point of view. The machine-language version of the program produced by a translator will be the same with or without comments, but the information provided by these statements constitutes an important part of the program from a human's perspective. Without such documentation, large, complex programs can easily thwart the comprehension of a human programmer.

There are two common ways of inserting comments within a program. One is to surround the entire comment by special markers, one at the beginning of the comment and one at the end. The other is to mark only the beginning of the comment and allow the comment to occupy the remainder of the line to the right of the marker. We find examples of both these techniques in C++, C#, and Java. They allow comments to be bracketed by `/*` and `*/`, but they also allow a comment to begin with `//` and extend through the remainder of the line.
Thus both

```
/* This is a comment. */
```

and

```
// This is a comment.
```

are valid comment statements.

A few words are in order about what constitutes a meaningful comment. Beginning programmers, when told to use comments for internal documentation, tend to follow a program statement such as

```
ApproachAngle = SlipAngle + HyperSpaceIncline;
```

with a comment such as "Calculate `ApproachAngle` by adding `HyperSpaceIncline` and `SlipAngle`." Such redundancy adds length rather than clarity to a program. The purpose of a comment is to explain the program, not to repeat it. A more appropriate comment in this case might be to explain why `ApproachAngle` is being calculated (if that is not obvious). For example, the comment, "`ApproachAngle`

is used later to compute `ForceFieldJettisonVelocity` and is not needed after that," is more helpful than the previous one.

Additionally, comments that are scattered among a program's statements can sometimes hamper a human's ability to follow the program's flow and thus make it harder to comprehend the program than if no comments had been included. A good approach is to collect comments that relate to a single program unit in one place, perhaps at the beginning of the unit. This provides a central place where the reader of the program unit can look for explanations. It also provides a location in which the purpose and general characteristics of the program unit can be described. If this format is adopted for all program units, the entire program is given a degree of uniformity in which each unit consists of a block of explanatory statements followed by the formal presentation of the program unit. Such uniformity in a program enhances its readability.

Questions & Exercises

1. Why is the use of a constant considered better programming style than the use of a literal?
2. What is the difference between a declarative statement and an imperative statement?
3. List some common data types.
4. Identify some common control structures found in imperative and object-oriented programming languages.
5. What is the difference between an array and an aggregate type?

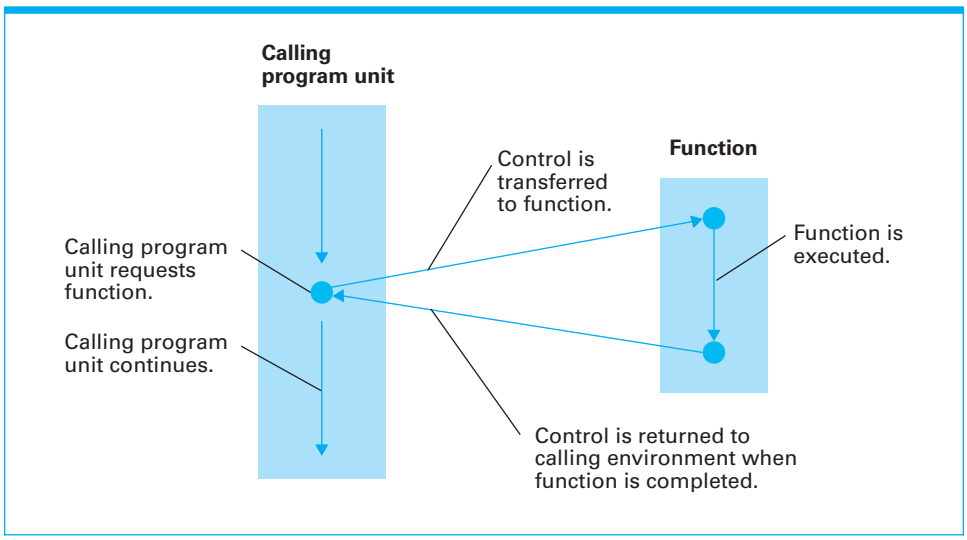
6.3 Procedural Units

In previous chapters we have seen advantages to dividing large programs into manageable units. In this section we focus on the concept of a function, which is the major technique for obtaining a modular representation of a program in an imperative language. As mentioned in Chapter 5, programming languages through the ages have used many terms for this essential concept: subprogram, subroutine, procedure, method, function, sometimes with subtle shades of different meaning. Among the strictly imperative languages, the term *function* has come to predominate, whereas in object-oriented languages, the term *method* is often preferred when programmers specify how objects should respond to various stimuli.

Functions

A **function,** in its generic sense, is a set of instructions for performing a task that can be used as an abstract tool by other program units. Control is transferred to the function at the time its services are required and then returned to the original

Figure 6.8 The flow of control involving a function



program unit after the function has finished (Figure 6.8). The process of transferring control to a function is often referred to as *calling* or *invoking* the function. We will refer to a program unit that requests the execution of a function as the *calling* unit.

As in our Python examples of Chapter 5, functions are usually written as individual program units. The unit begins with a statement, known as the function's **header,** that identifies, among other things, the name of the function. Following this header are the statements that define the function's details. These statements tend to be arranged in the same manner as those in a traditional imperative program, beginning with declaration statements that describe the variables used in the function followed by imperative statements that describe the steps to be performed when the function is executed.

As a general rule, a variable declared within a function is a **local variable,** meaning that it can be referenced only within that function. This eliminates any confusion that might occur if two functions, written independently, happen to use variables of the same name. (The portion of a program in which a variable can be referenced is called the **scope** of the variable. Thus, the scope of a local variable is the function in which it is declared. Variables whose scopes are not restricted to a particular part of a program are called **global variables.** Most programming languages provide a means of specifying whether a variable is to be local or global.)

Most modern programming languages allow functions to be called by merely stating the function's name. For example, if `GetNames`, `SortNames`, and `WriteNames` were the names of functions for acquiring, sorting, and printing a list of names, then a program to get, sort, and print the list could be written as

```
GetNames()
SortNames()
WriteNames()
```

Note that by assigning each function a name that indicates the action performed by the function, this condensed form appears as a sequence of commands that reflect the meaning of the program.

Parameters

Functions are often written using generic terms that are made specific when the function is applied. For example, Figure 5.11 of the preceding chapter is expressed in terms of a generic list rather than a specific list. In our pseudocode, we agreed to identify such generic terms within parentheses in the function's header. Thus the function in Figure 5.11 begins with the header

```
def Sort(List):
```

and then proceeds to describe the sorting process using the term `List` to refer to the list being sorted. If we want to apply the function to sort a wedding guest list, we need merely follow the directions in the function, assuming that the generic term `List` refers to the wedding guest list. If, however, we want to sort a membership list, we need merely interpret the generic term `List` as referring to the membership list.

Such generic terms within functions are called **parameters.** More precisely, the terms used within the function are called **formal parameters** and the precise meanings assigned to these formal parameters when the function is applied are called **actual parameters.** In a sense, the formal parameters represent slots in the function into which actual parameters are plugged when the function is requested.

As in Python, many programming languages require that, when defining a function, the formal parameters be listed in parentheses in the function's header. As another example, Figure 6.9 presents the definition of a function named `ProjectPopulation` as it might be written in the programming language C. The function expects to be given a specific yearly growth rate when it is called. Based on this rate, the function computes the projected population of a species, assuming an initial population of 100, for the next 10 years, and stores these values in a global array called `Population`.

Many programming languages also use parenthetical notation to identify the actual parameters when a function is called. That is, the statement requesting the execution of a function consists of the function name followed by a list of the actual parameters enclosed in parentheses. Thus, rather than a statement such as

```
Call ProjectPopulation() using a growth rate of 0.03
```

that we might use in pseudocode, the statement

```
ProjectPopulation(0.03);
```

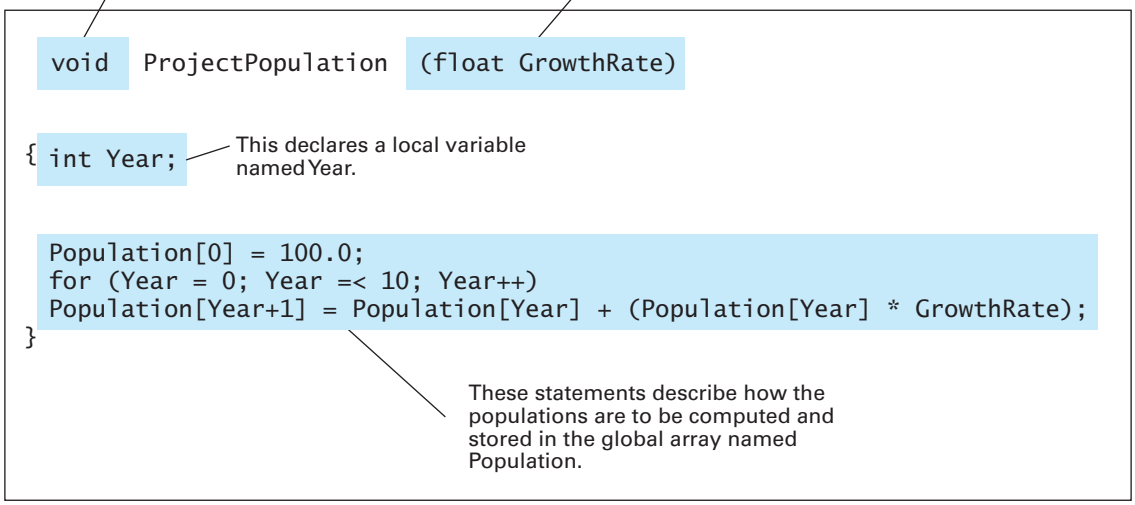
would be used in a C program to call the function `ProjectPopulation` of Figure 6.9 using a growth rate of 0.03. The syntax is the same in Python, but without the trailing semicolon.

When more than one parameter is involved, the actual parameters are associated, entry by entry, with the formal parameters listed in the function's header—the first actual parameter is associated with the first formal parameter, and so on. Then, the values of the actual parameters are effectively transferred to their corresponding formal parameters, and the function is executed.

To emphasize this point, suppose the function `PrintCheck` was defined with a header such as

```
def PrintCheck(Payee, Amount):
```

Figure 6.9 The function ProjectPopulation written in the programming language C



where `Payee` and `Amount` are formal parameters used within the function to refer to the person to whom the check is to be payable and the amount of the check, respectively. Then, calling the function with the statement

```
PrintCheck('John Doe', 150)
```

would cause the function to be executed with the formal parameter `Payee` being associated with the actual parameter John Doe and the formal parameter `Amount` being associated with the value 150. However, calling the function with the statement

```
PrintCheck(150, 'John Doe')
```

would cause the value 150 to be assigned to the formal parameter `Payee` and the name John Doe to be assigned to the formal parameter `Amount`, which would lead to erroneous results.

The task of transferring data between actual and formal parameters is handled in a variety of ways by different programming languages. In some languages a duplicate of the data represented by the actual parameters is produced and given to the function. Using this approach, any alterations to the data made by the function are reflected only in the duplicate—the data in the calling program unit are never changed. We often say that such parameters are **passed by value.** Note that passing parameters by value protects the data in the calling unit from being mistakenly altered by a poorly designed function. For example, if the calling unit passed an employee's name to a function, it might not want the function to change that name.

Unfortunately, passing parameters by value is inefficient when the parameters represent large blocks of data. A more efficient technique is to give the

function direct access to the actual parameters by telling it the addresses of the actual parameters in the calling program unit. In this case we say that the parameters are **passed by reference.** Note that passing parameters by reference allows the function to modify the data residing in the calling environment. Such an approach would be desirable in the case of a function for sorting a list since the point of calling such a function would be to cause changes in the list.

As an example, let us suppose that the function Demo was defined as

```
def Demo (Formal):
  Formal = Formal + 1
```

Moreover, suppose that the variable `Actual` was assigned the value 5 and we called Demo with the statement

```
Demo(Actual)
```

Then, if parameters were passed by value, the change to `Formal` in the function would not be reflected in the variable Actual (Figure 6.10). But, if parameters were passed by reference, the value of Actual would be incremented by one (Figure 6.11).

Figure 6.10 Executing the function Demo and passing parameters by value

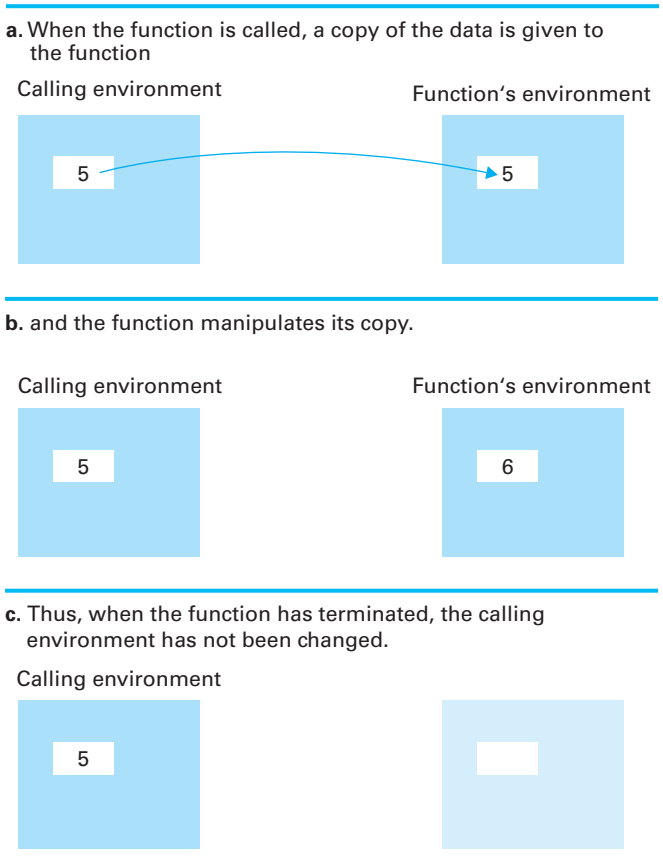
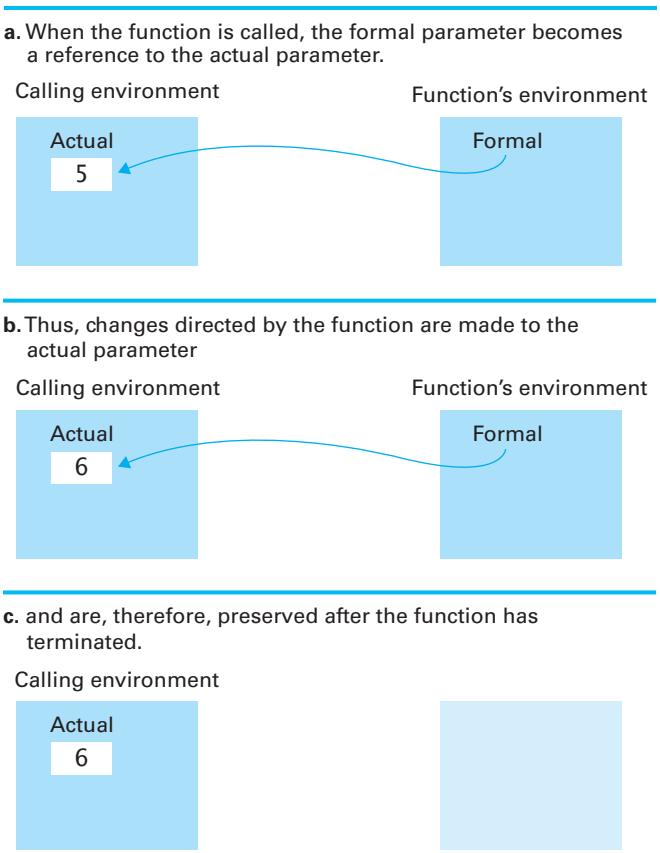


Figure 6.11 Executing the function Demo and passing parameters by reference



Visual Basic

Visual Basic is an object-oriented programming language that was developed by Microsoft as a tool by which users of Microsoft's Windows operating system could develop their own GUI applications. Actually, Visual Basic is more than a language—it is an entire software development package that allows a programmer to construct applications from predefined components (such as buttons, check boxes, text boxes, scroll bars, etc.) and to customize these components by describing how they should react to various events. In the case of a button, for example, the programmer would describe what should happen when that button is clicked. In Chapter 7 we will learn that this strategy of constructing software from predefined components represents the current trend in software development techniques.

The growing popularity of the Windows operating system combined with the convenience of the Visual Basic development package made the original Visual Basic a widely used programming language in the 1990s. Visual Basic successors, such as VB.NET, remain popular choices of language for rapid prototyping of software with graphical user interfaces.

Different programming languages provide different parameter-passing techniques, but in all cases the use of parameters allows a function to be written in a generic sense and applied to specific data at the appropriate time.

Fruitful Functions

Let us pause to consider a slight variation of the function concept that is found in many programming languages. At times the purpose of a function is to produce a value rather than perform an action. (Consider the subtle distinction between a function whose purpose is to estimate the number of widgets that will be sold as opposed to a function for playing a simple game—the emphasis in the former is to produce a value, the emphasis in the latter is to perform an action.) Indeed, the term *function* in computer science is derived from the mathematical concept of *function,* which by default is a relationship between a set of inputs and outputs. In this sense, all of the Python and pseudocode examples of functions we have defined thus far have been a special case of function, in which there is no output or returned value to be concerned about. (Technically, all functions in Python do return a value, and thus are suited to the use of the mathematical term *function.* Functions that do not include a `return` statement return the default value `None`.) In the family tree of programming languages, many terms have been used to differentiate these two different types of program unit. In the influential language Pascal, the term **procedure** was specifically used to refer to a subprogram that returned no value. C and Java use the keyword **void** to label functions or methods that return no value. Among Python programmers, the term **fruitful function** is sometimes used to differentiate functions that do specify a return a value. Regardless of the language-specific terms, most languages have names for the same idea— a program unit that transfers a value back to the calling program unit as "the value of the function." That is, as a consequence of executing the function, a value will be computed and sent back to the calling program unit. This value can then be stored in a variable for later reference or used immediately in a computation. For example, a C, C++, Java, or C# programmer might write

```
ProjectedJanSales = EstimatedSales(January);
```

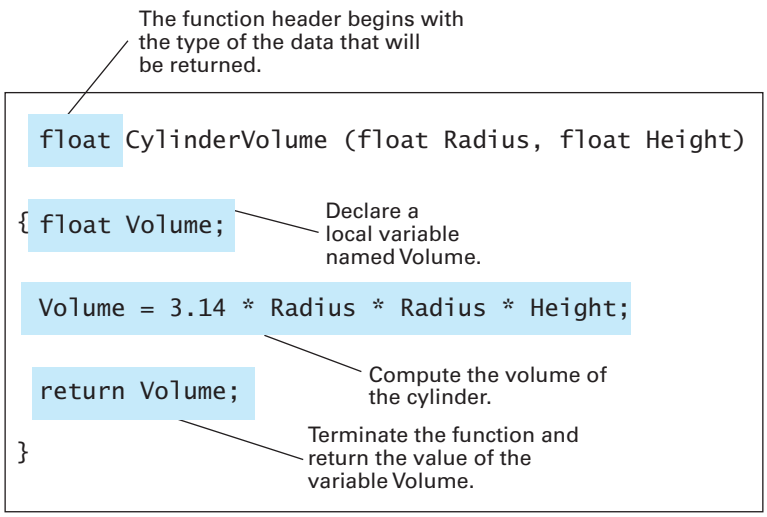
to request that the variable `ProjectedJanSales` be assigned the result of applying the function `EstimatedSales` to determine how many widgets are expected to be sold in January. Or, the programmer might write

```
if (LastJanSales < EstimatedSales(January)) ...
else ...
```

to cause different actions to be performed depending on whether this January's sales are expected to be better than those of last January. Note that in the second case, the value computed by the function is used to determine which branch should be taken, but it is never stored.

Functions that return a value are defined within a program in much the same way as void functions. The difference is that a function header usually begins by specifying the data type of the value that is to be returned, and the function definition usually ends with a `return` statement in which the value to be returned is specified. Figure 6.12 presents a definition of a function named `CylinderVolume` as it might be written in the language C. (Actually, a C programmer would use a more succinct form, but we will use this somewhat verbose version for pedagogical reasons.) When called, the function receives specific values for the formal

Figure 6.12 The fruitful function CylinderVolume written in the programming language C



parameters `Radius` and `Height` and returns the result of computing the volume of a cylinder with those dimensions. Thus the function could be used elsewhere in the program in a statement such as

```
Cost = CostPerVolUnit * CylinderVolume(3.45, 12.7);
```

to determine the cost of the contents of a cylinder with radius 3.45 and height 12.7.

We have used several examples of built-in Python functions that return values in previous chapters, including the `input()` and `math.sqrt()` functions, but we have not defined one of our own. The Python version of the `CylinderVolume` function from Figure 6.12 would be

```
def CylinderVolume(Radius, Height):
  Volume = math.pi * Radius * Radius * Height
  return Volume
```

Event-Driven Software Systems

In the text, we have considered cases in which functions are activated as the result of statements elsewhere in the program that explicitly call the function. There are cases, however, in which functions are activated implicitly by the occurrence of an event. Examples are found in GUIs where the function that describes what should happen when a button is clicked is not activated by a call from another program unit, but instead is activated as the result of the button being clicked. Software systems in which functions are activated by events rather than explicit requests are called **event-driven** systems. In short, an event-driven software system consists of functions that describe what should happen as the result of various events. When the system is executed, these functions lie dormant until their respective event occurs—then they become active, perform their task, and return to dormancy.

Questions & Exercises

1. What is meant by the "scope" of a variable?
2. What is the difference between a function and a fruitful function?
3. Why do many programming languages implement I/O operations as if they were calls to functions?
4. What is the difference between a formal parameter and an actual parameter?
5. What is the difference between a function that passes parameters using call by reference and a function that uses call by value?

6.4 Language Implementation

In this section we investigate the process of converting a program written in a high-level language into a machine-executable form.

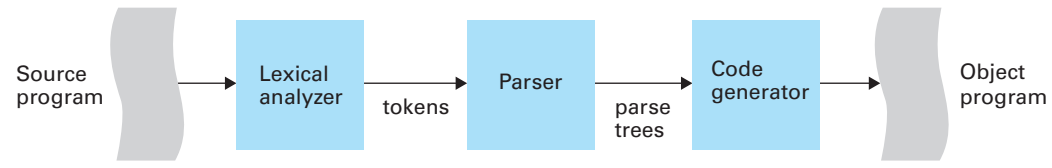
The Translation Process

The process of converting a program from one language to another is called **translation.** The program in its original form is the **source program**; the translated version is the **object program.** The translation process consists of three activities—lexical analysis, parsing, and code generation—that are performed by units in the translator known as the **lexical analyzer, parser,** and **code generator** (Figure 6.13).

Lexical analysis is the process of recognizing which strings of symbols from the source program represent a single entity, or **token.** For example, the three symbols 153 should not be interpreted as a 1, a 5, and a 3 but should be recognized as representing a single numeric value. Likewise, a word appearing in the program, although composed of individual symbols, should be interpreted as a single unit. Most humans perform lexical analysis with little conscious effort. When asked to read aloud, we pronounce words rather than individual characters.

Thus the lexical analyzer reads the source program symbol by symbol, identifying which groups of symbols represent tokens, and classifying those tokens according to whether they are numeric values, words, arithmetic operators, and so on. The lexical analyzer encodes each token with its classification and hands them to the parser. During this process, the lexical analyzer skips over all comment statements.

Figure 6.13 The translation process



Thus the parser views the program in terms of lexical units (tokens) rather than individual symbols. It is the parser's job to group these units into statements. Indeed, parsing is the process of identifying the grammatical structure of the program and recognizing the role of each component. It is the technicalities of parsing that cause one to hesitate when reading the sentence

> The man the horse that won the race threw was not hurt.

(Try this one: "That that is is. That that is not is not. That that is not is not that that is."!)

To simplify the parsing process, early programming languages insisted that each program statement be positioned in a particular manner on the printed page. Such languages were known as **fixed-format** languages. Today, many programming languages are **free-format** languages, meaning that the positioning of statements is not critical. The advantage of free-format languages lies in a programmer's ability to organize the written program in a way that enhances readability from a human's point of view. In these cases it is common to use indentation to help a reader grasp the structure of a statement. Rather than writing

```
if Cost < CashOnHand then pay with cash else use credit card
```

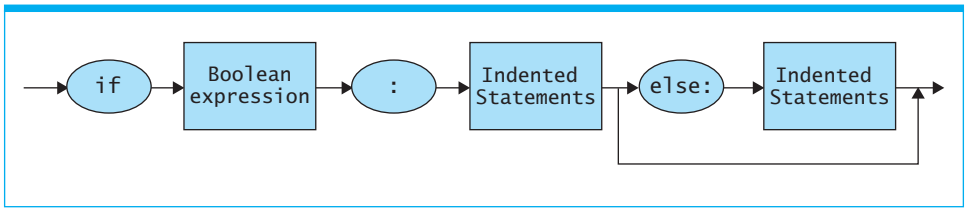
a programmer might write

```
if Cost < CashOnHand
  then pay with cash
  else use credit card
```

For a machine to parse a program written in a free-format language, the syntax of the language must be designed so that the structure of a program can be identified regardless of the spacing used in the source program. To this end, most free-format languages use punctuation marks such as semicolons to mark the ends of statements, as well as **key words** such as `if`, `then`, and `else` to mark the beginning of individual phrases. These key words are often **reserved words,** meaning that they cannot be used by the programmer for other purposes within the program. Python is unusual in this respect, in that it has aspects of free-format languages, but strictly requires indentation to mark structure, rather than punctuation marks like semicolons and curly braces.

The parsing process is based on a set of rules that define the syntax of the programming language. Collectively, these rules are called a **grammar.** One way of expressing these rules is by means of **syntax diagrams,** which are pictorial representations of a language's grammatical structure. Figure 6.14 shows a syntax diagram of the if-else statement from Python in Chapter 5. This diagram indicates that an if-else structure begins with the word `if`, followed by a Boolean expression, followed by a colon, followed by Indented Statements. This combination might

Figure 6.14 A syntax diagram of Python's if-else statement



or might not be followed by the word `else`, a colon, and Indented Statements. Notice that terms that actually appear in an if-else statement are enclosed in ovals, whereas terms that require further description, such as *Boolean expression* and *Indented Statements,* are enclosed in rectangles. Terms that require further description (those in rectangles) are called **nonterminals;** terms that appear in ovals are called **terminals.** In a complete description of a language's syntax the nonterminals are described by additional diagrams.

As a more complete example, Figure 6.15 presents a set of syntax diagrams that describes the syntax of a structure called *Expression,* which is intended to be the structure of simple arithmetic expressions. The first diagram describes an *Expression* as consisting of a *Term* that might or might not be followed by either a + or − symbol followed by another *Expression.* The second diagram describes a *Term* as consisting of either a single *Factor* or a *Factor* followed by a × or ÷ symbol, followed by another *Term.* Finally, the last diagram describes a *Factor* as one of the symbols x, y, or z.

The manner in which a particular string conforms to a set of syntax diagrams can be represented in a pictorial form by a **parse tree,** as demonstrated in Figure 6.16, which presents a parse tree for the string

x+y×z

based on the set of diagrams in Figure 6.15. Note that the tree starts at the top with the nonterminal *Expression* and at each level shows how the nonterminals at that level are decomposed until the symbols in the string itself are obtained. In particular, the figure shows that (according to the first diagram in Figure 6.15) an

Figure 6.15 Syntax diagrams describing the structure of a simple algebraic expression

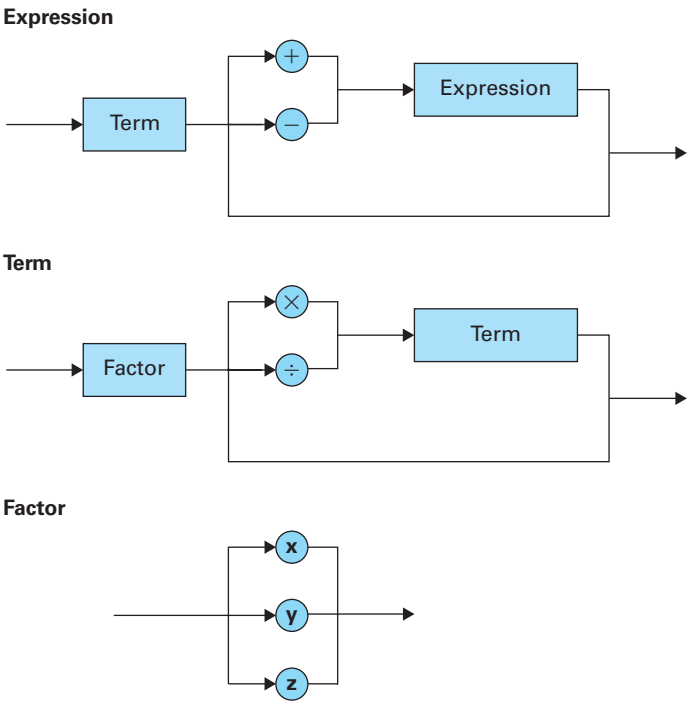
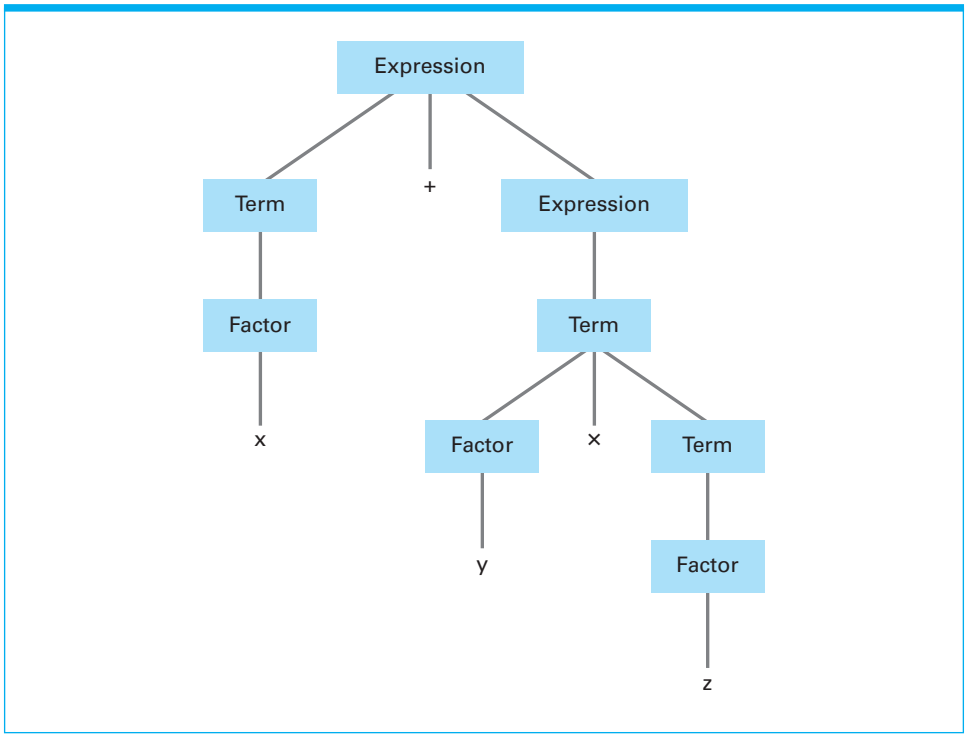


Figure 6.16 The parse tree for the string x + y × z based on the syntax diagrams in Figure 6.15



Expression can be decomposed as a *Term,* followed by the + symbol, followed by an *Expression.* In turn, the *Term* can be decomposed (using the second diagram in Figure 6.15) as a *Factor* (which turns out to be the symbol x), and the final *Expression* can be decomposed (using the third diagram in Figure 6.15) as a *Term* (which turns out to be y × z).

Implementation of Java and C#

In some cases, such as in the control of an animated Web page, software must be transferred over the Internet and executed on distant machines. If this software is supplied in source program form, additional delays will result at the destination because the software will have to be translated into the proper machine language before it is executed. However, supplying the software in machine-language form would mean that a different version of the software would have to be provided depending on the machine language used by the distant computer.

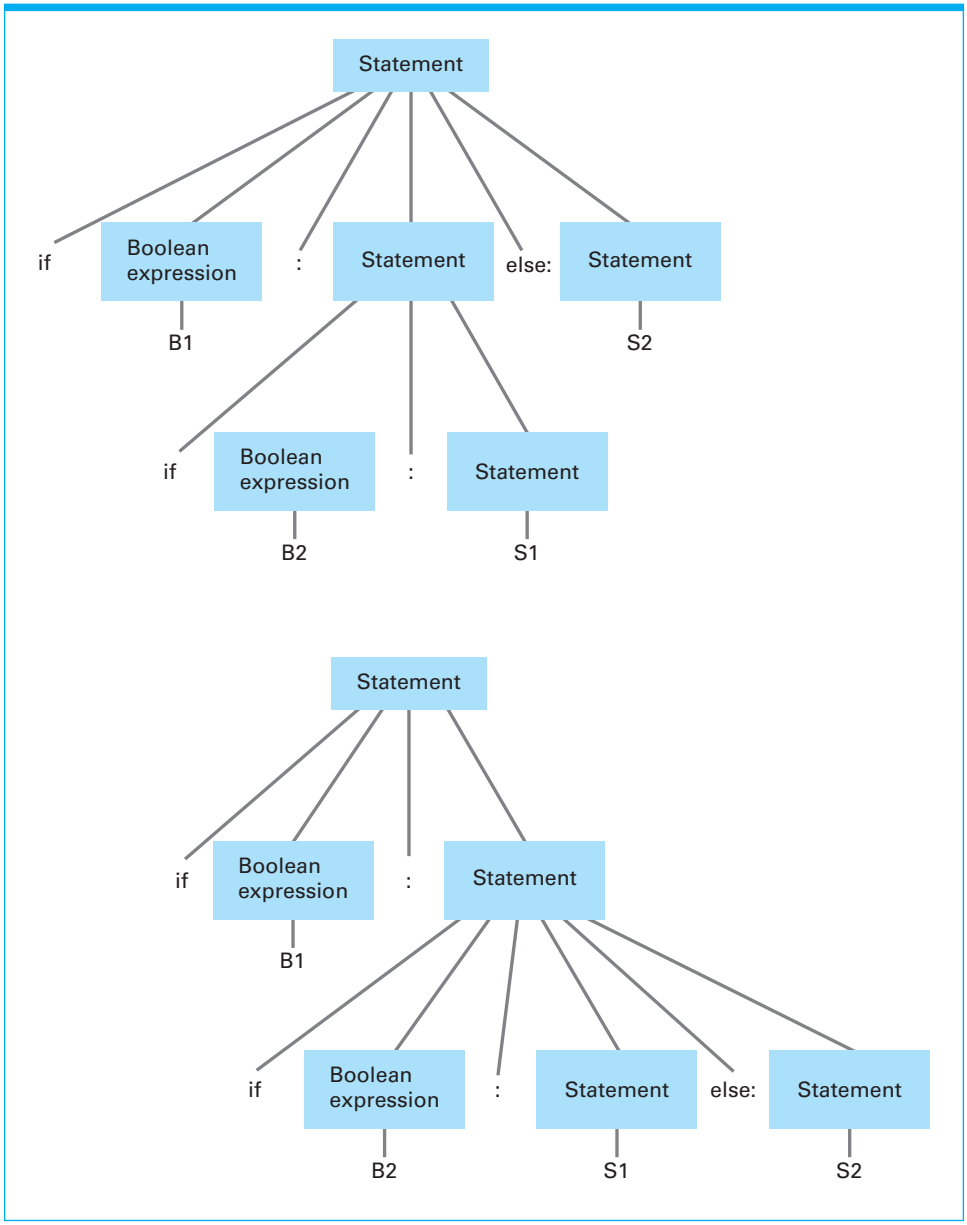
Sun Microsystems and Microsoft have resolved this problem by designing "universal machine languages" (called bytecode in the case of Java and .NET Common Intermediate Language in the case of C#) into which source programs can be translated. Although these languages are not really machine languages, they are designed to be quickly translatable. Thus if software written in Java or C# is translated into the appropriate "universal machine language," then it can be transferred to other machines in the Internet where it can be executed efficiently. In some cases this execution is performed by an interpreter. In other cases, it is quickly translated prior to execution, a process known as **just-in-time compilation.**

The process of parsing a program is essentially that of constructing a parse tree for the source program. Indeed, a parse tree represents the parser's interpretation of the program's grammatical composition. For this reason the syntax rules describing a program's grammatical structure must not allow two distinct parse trees for one string, since this would lead to ambiguities within the parser. A grammar that does allow two distinct parse trees for one string is said to be an **ambiguous grammar.**

Ambiguities in grammars can be quite subtle. In fact, the rule in Figure 6.14 contains such a flaw. It allows both the parse trees in Figure 6.17 for the single statement

```
if B1: if B2: S1 else: S2
```

Figure 6.17 Two distinct parse trees for the statement if B1: if B2: S1 else: S2



Note that these interpretations are significantly different. The first implies that statement *S2* is to execute if *B1* is false; the second implies that *S2* is to execute only if *B1* is true and *B2* is false.

The syntax definitions of formal programming languages are designed to avoid such ambiguities. In Python we avoid such problems by using indentation. In particular, we might write

```
if B1:
  if B2:
    S1
  else:
    S2
```

and

```
if B1:
  if B2:
    S1
else:
  S2
```

to distinguish between the two possible interpretations.

As a parser analyzes the grammatical structure of a program, it is able to identify individual statements and to distinguish between the declarative statements and imperative statements. As it recognizes the declarative statements, it records the information being declared in a table called the **symbol table.** Thus the symbol table contains such information as the names of the variables appearing in the program as well as what data types and data structures are associated with those variables. The parser then relies on this information when analyzing imperative statements such as

```
z = x + y
```

In particular, to determine the meaning of the symbol +, the parser must know the data type associated with `x` and `y`. If `x` is of type float and `y` is of type character, then adding `x` and `y` makes little sense and should be reported as an error. If `x` and `y` are both of type integer, then the parser will request that the code generator build a machine-language instruction using the machine's integer addition op-code; if both are of type float, the parser will request that floating-point addition op-code be used; or if both are of type character, the parser might request that the code generator build the sequence of machine- language instructions needed to perform the concatenation operation.

A somewhat special case arises if `x` is of type integer and `y` is of type float. Then the concept of addition is applicable but the values are not encoded in compatible forms. In this case the parser might choose to have the code generator build the instructions to convert one value to the other type and then perform the addition. Such implicit conversion between types is called **coercion.**

Coercion is frowned upon by many language designers, because implicit type conversion can alter the value of a data item, resulting in subtle program bugs. They argue that the need for coercion usually indicates a flaw in the program's design and therefore should not be accommodated by the parser. The result is that most modern languages are **strongly typed,** which means that all activities requested by a program must involve data of agreeable types. Some languages, such as Java, will allow coercion as long as it is a **type promotion,** meaning that it involves

converting a low precision value to a higher precision value. Implicit coercions that might alter a value are reported as errors. In most cases a programmer can still request these type conversions by making an explicit **type cast,** which notifies the compiler that the programmer is aware that a type conversion will be applied.

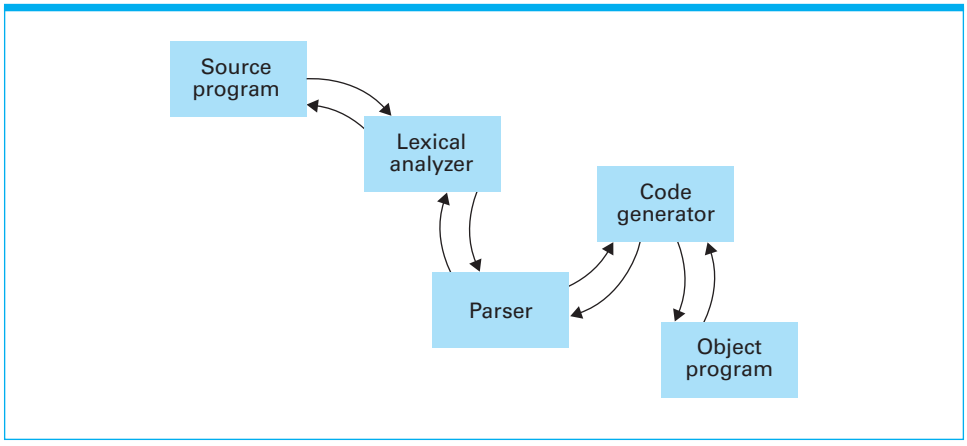
The final activity in the translation process is **code generation,** which is the process of constructing the machine-language instructions to implement the statements recognized by the parser. This process involves numerous issues, one being that of producing efficient machine-language versions of programs. For example, consider the task of translating the two-statement sequence

```
x = y + z
w = x + z
```

If these statements are translated as individual statements, each would require that data be transferred from main memory into the CPU before the indicated addition takes place. However, efficiency can be gained by recognizing that once the first statement has been executed, the values of `x` and `z` will already be in the CPU's general-purpose registers and therefore need not be loaded from memory before performing the second addition. Implementing insights such as this is called **code optimization** and is an important task of the code generator.

Finally, we should note that the steps of lexical analysis, parsing, and code generation are not carried out in a strict sequential order. Instead, these activities are intertwined. The lexical analyzer begins by reading characters from the source program and identifying the first token. It hands this token to the parser. Each time the parser receives a token from the lexical analyzer, it analyzes the grammatical structure being read. At this point it might request another token from the lexical analyzer or, if the parser recognizes that a complete phrase or statement has been read, it calls on the code generator to produce the proper machine instructions. Each such request causes the code generator to build machine instructions that are added to the object program. In turn, the task of translating a program from one language to another conforms naturally to the object-oriented paradigm. The source program, lexical analyzer, parser, code generator, and object program are objects that interact by sending messages back and forth as each object goes about performing its task (Figure 6.18).

Figure 6.18 An object-oriented approach to the translation process



Software Development Packages

The software tools, such as editors and translators, used in the software development process are often grouped into a package that functions as one integrated software development system. Such a system would be classified as application software in the classification scheme of Section 3.2. By using this application package, a programmer gains ready access to an editor for writing programs, a translator for converting the programs into machine language, and a variety of debugging tools that allow the programmer to trace the execution of a malfunctioning program to discover where it goes astray.

The advantages of using such an integrated system are numerous. Perhaps the most obvious is that a programmer can move back and forth between the editor and debugging tools with ease as changes to the program are made and tested. Moreover, many software development packages allow related program units that are under development to be linked in such a way that access to related units is simplified. Some packages maintain records regarding which program units within a group of related units have been altered since the last benchmark was made. Such capabilities are quite advantageous in the development of large software systems in which many interrelated units are developed by different programmers.

On a smaller scale, the editors in software development packages are often customized to the programming language being used. Such an editor will usually provide automatic line indentation that is the de facto standard for the target language and in some cases might recognize and automatically complete key words after the programmer has typed only the first few characters. Moreover, the editor might highlight keywords within source programs (perhaps with color) so that they stand out, making the programs easier to read.

In the next chapter we will learn that software developers are increasingly searching for ways by which new software systems can be constructed from prefabricated blocks called components—leading to a new software development model called component architecture. Software development packages based on the component architecture model often use graphical interfaces in which components can be represented as icons on the display. In this setting a programmer (or component assembler) selects desired components with a mouse. A selected component can then be customized by means of the package's editor and then attached to other components by pointing and clicking with the mouse. Such packages represent a major step forward in the search for better software development tools.

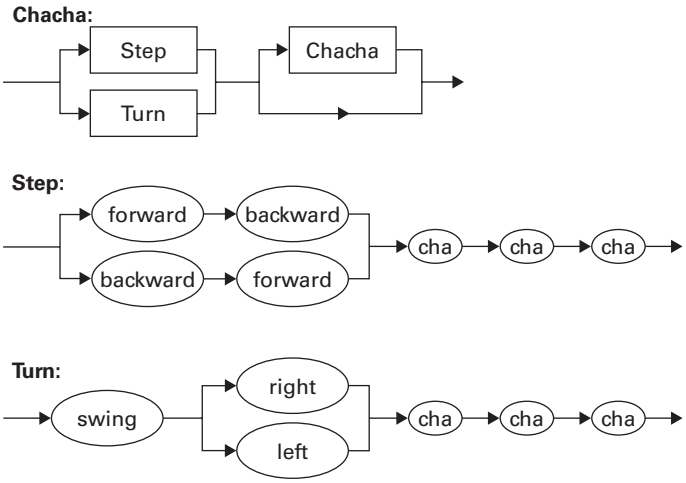
Questions & Exercises

1. Describe the three major steps in the translation process.
2. What is a symbol table?
3. What is the difference between a terminal and a nonterminal?
4. Draw the parse tree for the expression

   x × y + x + z

   based on the syntax diagrams in Figure 6.15.

5. Describe the strings that conform to the structure Chacha according to the following syntax diagrams.



6. Like modern programming language editors, this text uses syntax coloring in an attempt to make code examples easier to read. How many colors have you seen? What does each color seem to mean?

6.5 Object-Oriented Programming

In Section 6.1 we learned that the object-oriented paradigm entails the development of active program units called **objects,** each of which contains functions describing how that object should respond to various stimuli. The object-oriented approach to a problem is to identify the objects involved and describe them as self-contained units. In turn, object-oriented programming languages provide statements for describing objects and their behavior. In this section we will introduce some of these statements as they appear in the languages C++, Java, and C#, which are three of the more prominent object-oriented languages used today.

Classes and Objects

Consider the task of developing a simple computer game in which the player must protect the Earth from falling meteors by shooting them with high-power lasers. Each laser contains a finite internal power source that is partially consumed each time the laser is fired. Once this source is depleted, the laser becomes useless. Each laser should be able to respond to the commands to aim farther to the right, aim farther to the left, and to fire its laser beam.

In the object-oriented paradigm, each laser in the computer game would be implemented as an object that contains a record of its remaining power as well as functions for modifying its aim and firing its laser beam. Since all the laser objects have the same properties, they can be described by means of a common template. In the object-oriented paradigm a template for a collection of objects is called a **class.**

In Chapter 8, we will explore the similarities between classes and data types. For now we simply note that a class describes the common characteristics of a collection of objects in much the same way as the concept of the primitive data type integer encompasses the common characteristics of such numbers as 1, 5, and 82. Once a programmer has included the description of a class in a program, that template can be used to construct and to manipulate objects of that "type" in much the same way that the primitive type integer allows the manipulation of "objects" of type integer.

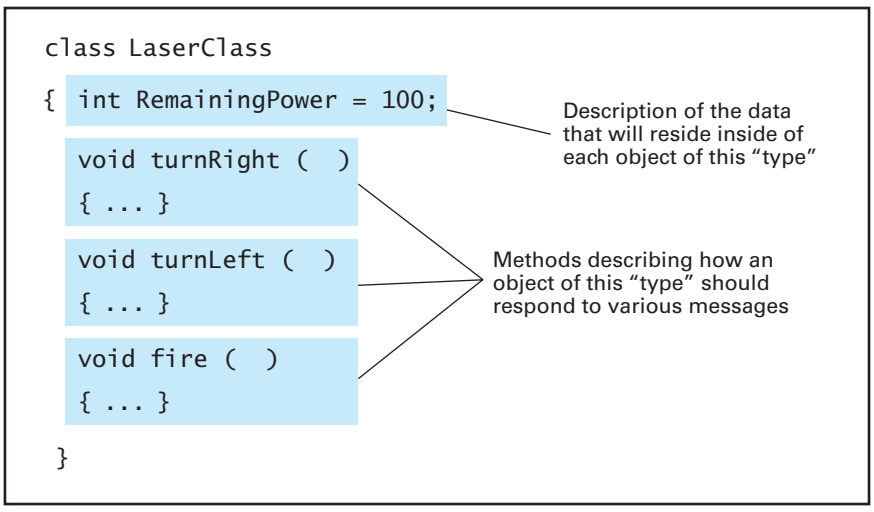
In the languages C++, Java, and C# a class is described by a statement of the form

```
class Name
{
    .
    .
    .
}
```

where `Name` is the name by which the class can be referenced elsewhere in the program. It is within the braces that the properties of the class are described. In particular, a class named `LaserClass` describing the structure of a laser in our computer game is outlined in Figure 6.19. The class consists of the declaration of a variable named `RemainingPower` (of type integer) and three functions named `turnRight`, `turnLeft`, and `fire`. These functions describe the routines to be performed to accomplish the corresponding action. Thus any object that is constructed from this template will have these features: a variable called `RemainingPower` and three functions named `turnRight`, `turnLeft`, and `fire`.

A variable that resides within an object, such as `RemainingPower`, is called an **instance variable** and the functions within an object are called **methods** (or member functions in the C++ vernacular). Note that in Figure 6.19 the instance

Figure 6.19 The structure of a class describing a laser weapon in a computer game



variable RemainingPower is described using a declaration statement similar to those discussed in Section 6.2 and the methods are described in a form reminiscent of functions as discussed in Section 6.3. After all, declarations of instance variables and descriptions of methods are basically imperative programming concepts.

Once we have described the class LaserClass in our game program, we can declare three variables Laser1, Laser2, and Laser3 to be of "type" LaserClass by a statement of the form

```
LaserClass Laser1, Laser2, Laser3;
```

Note that this is the same format as the statement

```
int x, y, z;
```

that would be used to declare three variables named x, y, and z of type integer, as we learned early in Section 6.2. Both consist of the name of a "type" followed by a list of the variables being declared. The difference is that the latter statement says that the variables x, y, and z will be used in the program to refer to items of type integer (which is a primitive type), whereas the former statement says the variables Laser1, Laser2, and Laser3 will be used in the program to refer to items of "type" LaserClass (which is a "type" defined within the program).

Once we have declared the variables Laser1, Laser2, and Laser3 to be of "type" LaserClass, we can assign them values. In this case the values must be objects that conform to the "type" LaserClass. These assignments can be made by assignment statements, but it is often convenient to assign starting values to the variables within the same declaration statements used to declare the variables. Such initial assignments are made automatically in the case of declarations in the language C++. That is, the statement

```
LaserClass Laser1, Laser2, Laser3;
```

not only establishes the variables Laser1, Laser2, and Laser3, but also creates three objects of "type" LaserClass, one as the value of each variable. In the languages Java and C#, such initial assignments are instigated in much the same way that initial assignments are made to variables of primitive types. In particular, whereas the statement

```
int x = 3;
```

not only declares x to be a variable of type integer but also assigns the variable the value 3, the statement

```
LaserClass Laser1 = new LaserClass();
```

declares the variable Laser1 to be of "type" LaserClass and also creates a new object using the LaserClass template and assigns that object as the starting value of Laser1.

At this point we should pause to emphasize the distinction between a class and an object. A class is a template from which objects are constructed. One class can be used to create numerous objects. We often refer to an object as an **instance** of the class from which it was constructed. Thus, in our computer game Laser1, Laser2, and Laser3 are variables whose values are instances of the class LaserClass.

After using declarative statements to create the variables Laser1, Laser2, and Laser3 and assign objects to them, we can continue our game program

by writing imperative statements that activate the appropriate methods within these objects (in object-oriented vernacular, this is called sending messages to the objects). In particular, we could cause the object assigned to the variable `Laser1` to execute its `fire` method using the statement

```
Laser1.fire();
```

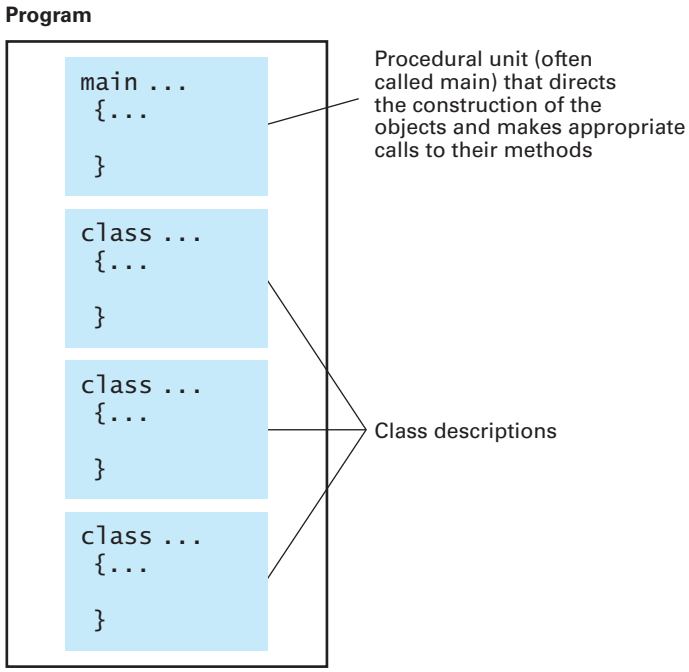
Or we could cause the object assigned to `Laser2` to execute its `turnLeft` method via the statement

```
Laser2.turnLeft();
```

These are actually little more than function calls. Indeed, the former statement is a call to the function (the method) `fire` inside the object assigned to the variable `Laser1`, and the latter statement is a call to the function `turnLeft` inside the object assigned to the variable `Laser2`.

At this stage our meteor game example has given us the background to grasp the overall structure of a typical object-oriented program (Figure 6.20). It will contain a variety of class descriptions similar to Figure 6.19, each describing the structure of one or more objects used in the program. In addition, the program will contain an imperative program segment (usually associated with the name "main") containing the sequence of steps to be performed initially when the program is executed. This segment will contain declaration statements similar to our laser declarations to establish the objects used in the program as well as imperative statements that call for the execution of methods within those objects.

Figure 6.20 The structure of a typical object-oriented program



Constructors

When an object is constructed, often some customizing activities need to be performed. For example, in our meteor computer game we might want the different lasers to have different initial power settings, which would mean that the instance variables named `RemainingPower` within the various objects should be given different starting values. Such initialization needs are handled by defining special methods, called **constructors,** within the appropriate class. Constructors are executed automatically when an object is constructed from the class. A constructor is identified within a class definition by the fact that it is a method with the same name as the class.

Figure 6.21 presents an extension of the `LaserClass` definition originally shown in Figure 6.19. Note that it contains a constructor in the form of a method named `LaserClass`. This method assigns the instance variable `RemainingPower` the value it receives as its parameter. Thus, when an object is constructed from this class, this method will be executed, causing `RemainingPower` to be initialized at the appropriate setting.

The actual parameters to be used by a constructor are identified in a parameter list in the statement causing the creation of the object. Thus, based on the class definition in Figure 6.21, a C++, programmer would write

```
LaserClass Laser1(50), Laser2(100);
```

to create two objects of type `LaserClass`—one known as `Laser1` with an initial power reserve of 50, and the other known as `Laser2` with an initial power

Figure 6.21 A class with a constructor

```
class  LaserClass
{ int RemainingPower;

  LaserClass (InitialPower)
  { RemainingPower = InitialPower;
  }

  void turnRight (  )
  { ... }

  void turnLeft (  )
  { ... }

  void fire (  )
  { ... }

}
```

Constructor assigns a value to RemainingPower when an object is created.

reserve of 100. Java and C# programmers would accomplish the same task with the statements

```
LaserClass Laser1 = new LaserClass(50);
LaserClass Laser2 = new LaserClass(100);
```

Additional Features

Let us now suppose we want to enhance our meteor computer game so that a player who reaches a certain score will be rewarded by recharging some of the lasers to their original power setting. These lasers will have the same properties as the other lasers except that they will be rechargeable.

To simplify the description of objects with similar yet different characteristics, object-oriented languages allow one class to encompass the properties of another through a technique known as **inheritance.** As an example, suppose we were using Java to develop our game program. We could first use the class statement described previously to define a class called `LaserClass` that described those properties that are common to all lasers in the program. Then we could use the statement

```
class RechargeableLaser extends LaserClass
{
    .
    .
    .
}
```

to describe another class called `RechargeableLaser`. (C++ and C# programmers would merely replace the word `extends` with a colon.) Here the extends clause indicates that this class is to inherit the features of the class `LaserClass` as well as contain those features appearing within the braces. In our case, these braces would contain a new method (perhaps named `recharge`) that would describe the steps required to reset the instance variable `RemainingPower` to its original value. Once these classes were defined, we could use the statement

```
LaserClass Laser1, Laser2;
```

to declare `Laser1` and `Laser2` to be variables referring to traditional lasers, and use the statement

```
RechargeableLaser Laser3, Laser4;
```

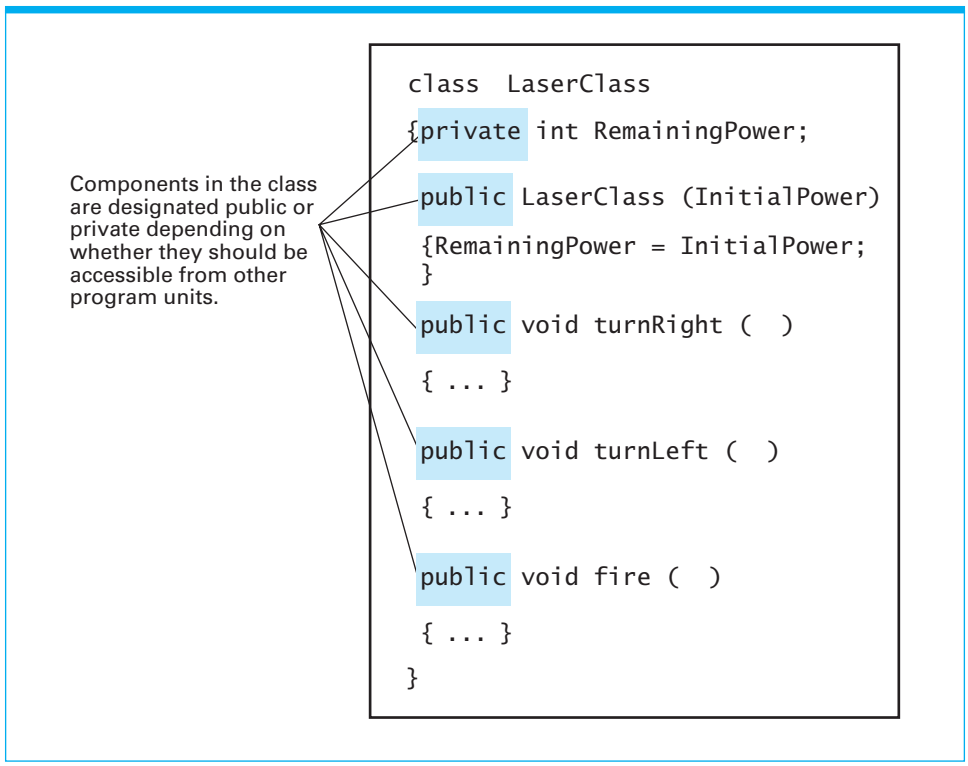
to declare `Laser3` and `Laser4` to be variables referring to lasers having the additional properties described in the `RechargeableLaser` class.

The use of inheritance leads to the existence of a variety of objects with similar yet different characteristics, which in turn leads to a phenomenon reminiscent of overloading, which we met in Section 6.2. (Recall that overloading refers to the use of a single symbol, such as +, for representing different operations depending on the type of its operands.) Suppose that an object-oriented graphics package consists of a variety of objects, each representing a shape (circle, rectangle, triangle, and so on). A particular image might consist of a collection of these objects. Each object "knows" its size, location, and color as well as how

to respond to messages telling it, for example, to move to a new location or to draw itself on the display. To draw an image, we merely send a "draw yourself" message to each object in the image. However, the routine used to draw an object varies according to the shape of the object—drawing a square is not the same process as drawing a circle. This customized interpretation of a message is known as **polymorphism;** the message is said to be polymorphic. Another characteristic associated with object-oriented programming is **encapsulation,** which refers to restricting access to an object's internal properties. To say that certain features of an object are *encapsulated* means that only the object itself is able to access them. Features that are encapsulated are said to be private. Features that are accessible from outside the object are said to be public.

As an example, let us return to our `LaserClass` originally outlined in Figure 6.19. Recall that it described an instance variable `RemainingPower` and three methods `turnRight`, `turnLeft`, and `fire`. These methods are to be accessed by other program units to cause an instance of `LaserClass` to perform the appropriate action. But the value of `RemainingPower` should only be altered by the instance's internal methods. No other program unit should be able to access this value directly. To enforce these rules we need merely designate `RemainingPower` as `private` and `turnRight`, `turnLeft`, and `fire` as `public` as shown in Figure 6.22. With these designations inserted, any attempt to access the value of `RemainingPower` from outside the object in which it resides will be identified as an error when the program is translated—forcing the programmer to correct the problem before proceeding.

Figure 6.22 Our LaserClass definition using encapsulation as it would appear in a Java or C# program



Questions & Exercises

1. What is the difference between an object and a class?
2. What classes of objects other than `LaserClass` might be found in the computer game example used in this section? What instance variables in addition to `RemainingPower` might be found in the class `LaserClass`?
3. Suppose the classes `PartTimeEmployee` and `FullTimeEmployee` inherited the properties of the class `Employee`. What are some features that you might expect to find in each class?
4. What is a constructor?
5. Why are some items within a class designated as `private`?

6.6 Programming Concurrent Activities

Suppose we were asked to design a program to produce animation for an action computer game involving multiple attacking enemy spaceships. One approach would be to design a single program that would control the entire animation screen. Such a program would be charged with drawing each of the spaceships, which (if the animation is to appear realistic) would mean that the program would have to keep up with the individual characteristics of numerous spacecraft. An alternate approach would be to design a program to control the animation of a single spaceship whose characteristics are determined by parameters assigned at the beginning of the program's execution. Then the animation could be constructed by creating multiple activations of this program, each with its own set of parameters. By executing these activations simultaneously, we could obtain the illusion of many individual spaceships streaking across the screen at the same time.

Such simultaneous execution of multiple activations is called **parallel processing** or **concurrent processing.** True parallel processing requires multiple CPU cores, one to execute each activation. When only one CPU is available, the illusion of parallel processing is obtained by allowing the activations to share the time of the single processor in a manner similar to that implemented by multiprogramming systems (Chapter 3).

Many modern computer applications are more easily solved in the context of parallel processing than in the more traditional context involving a single sequence of instructions. In turn, newer programming languages provide syntax for expressing the semantic structures involved in parallel computations. The design of such a language requires the identification of these semantic structures and the development of a syntax for representing them.

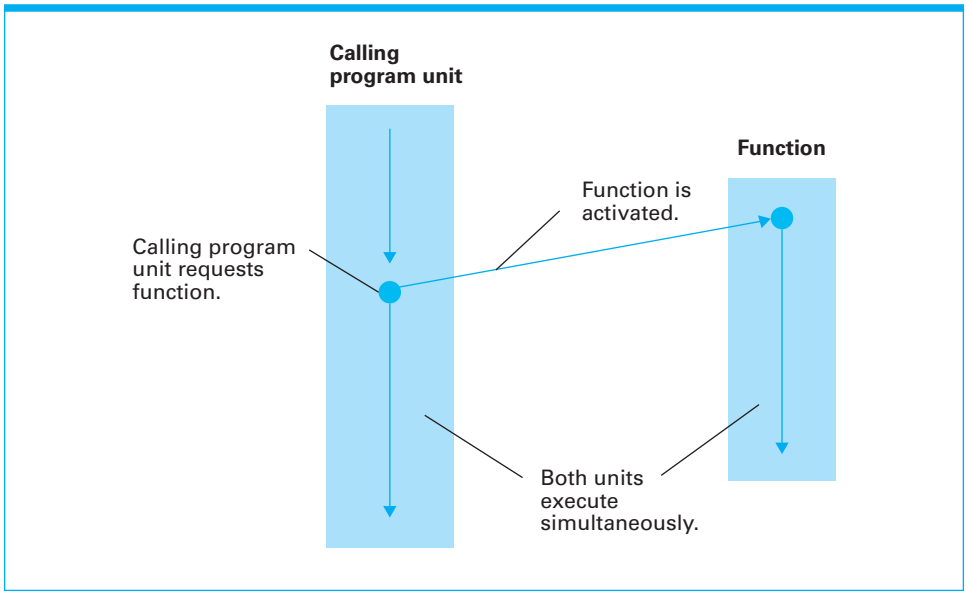
Each programming language tends to approach the parallel processing paradigm from its own point of view, resulting in different terminology. For example, what we have informally referred to as an *activation* is called a *task* in the Ada vernacular and a *thread* in Java. That is, in an Ada program, simultaneous actions are performed by creating multiple *tasks,* whereas in Java one creates multiple *threads.* In either case, the result is that multiple activities are generated and executed in much the same way as processes under the control of a multitasking operating system. We will adopt the Java terminology and refer to such "processes" as threads.

Perhaps the most basic action that must be expressed in a program involving parallel processing is that of creating new threads. If we want multiple activations of the spaceship program to be executed at the same time, we need a syntax for saying so. Such spawning of new threads is similar to that of requesting the execution of a traditional function. The difference is that, in the traditional setting, the program unit that requests the activation of a function does not progress any further until the requested function terminates (recall Figure 6.8), whereas in the parallel context the requesting program unit continues execution while the requested function performs its task (Figure 6.23). Thus to create multiple spaceships streaking across the screen, we would write a main program that simply generates multiple activations of the spaceship program, each provided with the parameters describing the distinguishing characteristics of that spaceship.

A more complex issue associated with parallel processing involves handling communication between threads. For instance, in our spaceship example, the threads representing the different spaceships might need to communicate their locations among themselves in order to coordinate their activities. In other cases one thread might need to wait until another reaches a certain point in its computation, or one thread might need to stop another one until the first has accomplished a particular task.

Such communication needs have long been a topic of study among computer scientists, and many newer programming languages reflect various approaches to thread interaction problems. As an example, let us consider the communication problems encountered when two threads manipulate the same data. (This example is presented in more detail in the optional Section 3.4.) If each of two threads that are executing concurrently need to add the value three to a common item of data, a method is needed to ensure that one thread is allowed to complete its transaction before the other is allowed to perform its task. Otherwise they could both start their individual computations with the same initial value, which would mean that the final result would be incremented by only three rather

Figure 6.23 Spawning threads



Programming Smartphones

Software for hand-held, mobile, and embedded devices is often developed using the same general-purpose programming languages that are used in other contexts. With a larger keyboard and extra patience, some smartphone applications can be written using the smartphone itself. However, in most cases smartphone software is developed on desktop computers using special software systems that provide tools for editing, translating, and testing smartphone software. Simple apps are often written in Java, C++, and C#. However, writing more complex apps or core system software may require additional support for concurrent and event-driven programming.

than six. Data that can be accessed by only one thread at a time is said to have mutually exclusive access.

One way to implement mutually exclusive access is to write the program units that describe the threads involved so that when a thread is using shared data, it blocks other threads from accessing that data until such access is safe. (This is the approach described in the optional Section 3.4, where we identified the portion of a process that accesses shared data as a critical region.) Experience has shown that this approach has the drawback of distributing the task of ensuring mutual exclusion throughout various parts of the program—each program unit accessing the data must be properly designed to enforce mutual exclusion, and thus a mistake in a single segment can corrupt the entire system. For this reason many argue that a better solution is to embody the data item with the ability to control access to itself. In short, instead of relying on the threads that access the data to guard against multiple access, the data item itself is assigned this responsibility. The result is that control of access is concentrated at a single point in the program rather than dispersed among many program units. A data item augmented with the ability to control access to itself is often called a **monitor.**

We conclude that the design of programming languages for parallel processing involves developing ways to express such things as the creation of threads, the pausing and restarting of threads, the identification of critical regions, and the composition of monitors.

In closing, we should note that although animation provides an interesting setting in which to explore the issues of parallel computing, it is only one of many fields that benefit from parallel processing techniques. Other areas include weather forecasting, air traffic control, simulation of complex systems (from nuclear reactions to pedestrian traffic), computer networking, and database maintenance.

Questions & Exercises

1. What are some properties that would be found in a programming language for concurrent processing that would not be found in a more traditional language?
2. Describe two methods for ensuring mutually exclusive access to data.
3. Identify some settings other than animation in which parallel computing is beneficial.

6.7 Declarative Programming

In Section 6.1 we claimed that formal logic provides a general problem-solving algorithm around which a declarative programming system can be constructed. In this section we investigate this claim by first introducing the rudiments of the algorithm and then taking a brief look at a declarative programming language based on it.

Logical Deduction

Suppose we know that either Kermit is on stage or Kermit is sick, and we are told that Kermit is not on stage. We could then conclude that Kermit must be sick. This is an example of a deductive-reasoning principle called **resolution.** Resolution is one of many techniques, called **inference rules,** for deriving a consequence from a collection of statements.

To better understand resolution, let us first agree to represent simple statements by single letters and to indicate the negation of a statement by the symbol ¬. For instance, we might represent the statement "Kermit is a prince" by A and "Miss Piggy is an actress" by B. Then, the expression

A OR B

would mean "Kermit is a prince or Miss Piggy is an actress" and

B AND ¬A

would mean "Miss Piggy is an actress and Kermit is not a prince." We will use an arrow to indicate "implies." For example, the expression

A→B

means "Kermit is a prince implies that Miss Piggy is an actress."

In its general form, the resolution principle states that from two statements of the form

P OR Q

and

R OR ¬Q

we can conclude the statement

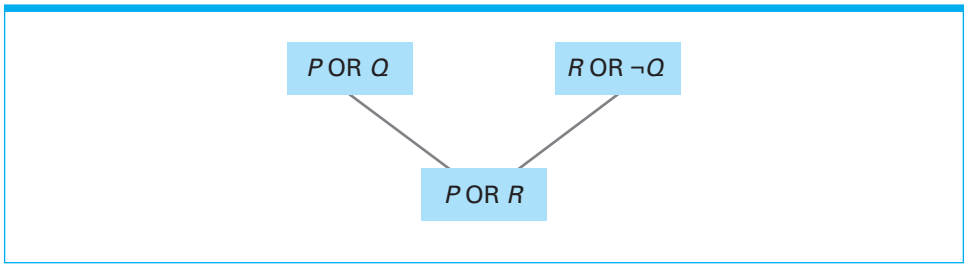
P OR R

In this case we say that the two original statements resolve to form the third statement, which we call the **resolvent.** It is important to observe that the resolvent is a logical consequence of the original statements. That is, if the original statements are true, the resolvent must also be true. (If Q is true, then R must be true; but if Q is false, then P must be true. Thus regardless of the truth or falseness of Q, either P or R must be true.)

We will represent the resolution of two statements pictorially as shown in Figure 6.24, where we write the original statements with lines projecting down to their resolvent. Note that resolution can be applied only to pairs of statements that appear in **clause form**—that is, statements whose elementary components are connected by the Boolean operation OR. Thus

P OR Q

Figure 6.24 Resolving the statements (*P* OR ¬*Q*) and (*R* OR *Q*) to produce (*P* OR *R*)



is in clause form, whereas

P→Q

is not. The fact that this potential problem poses no serious concern is a consequence of a theorem in mathematical logic that states that any statement expressed in the first-order predicate logic (a system for representing statements with extensive expressive powers) can be expressed in clause form. We will not pursue this important theorem here, but for future reference we observe that the statement

P→Q

is equivalent to the clause form statement

Q OR ¬P

A collection of statements is said to be **inconsistent** if it is impossible for all the statements to be true at the same time. In other words, an inconsistent collection of statements is a collection of statements that are self-contradictory. A simple example would be a collection containing the statement P as well as the statement ¬P. Logicians have shown that repeated resolution provides a systematic method of confirming the inconsistency of a set of inconsistent clauses. The rule is that if repeated application of resolution produces the empty clause (the result of resolving a clause of the form P with a clause of the form ¬P), then the original collection of statements must be inconsistent. As an example, Figure 6.25 demonstrates that the collection of statements

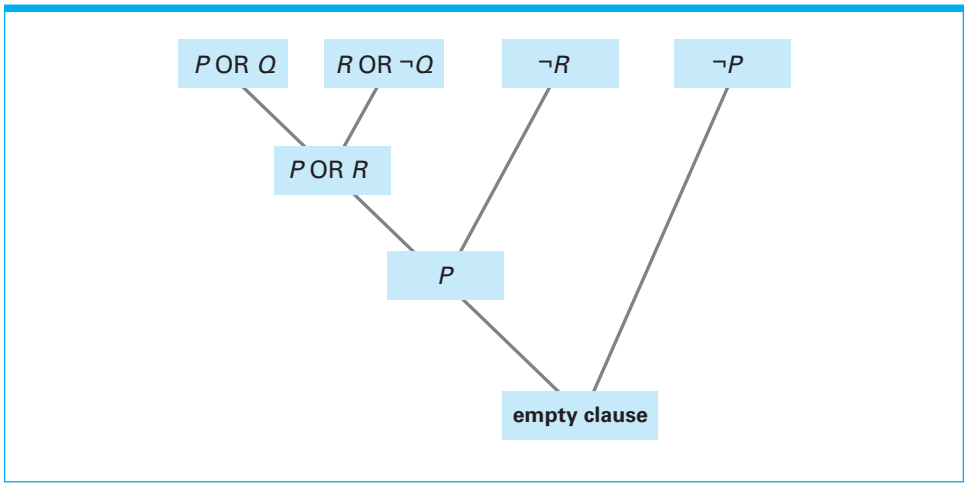
P OR Q
R OR ¬Q
¬R

¬P

is inconsistent.

Suppose now that we want to confirm that a collection of statements implies the statement P. To imply the statement P is the same as contradicting the statement ¬P. Thus, to demonstrate that the original collection of statements implies P, all we need to do is apply resolution to the original statements together with the statement ¬P until an empty clause occurs. Upon obtaining an empty clause, we can conclude that statement ¬P is inconsistent with the original statements, and thus the original statements must imply P.

Figure 6.25 Resolving the statements (*P* OR *Q*), (*R* OR ¬*Q*), ¬*R*, and ¬*P*



One final point remains before we are ready to apply resolution in an actual programming environment. Suppose we have the two statements

`(Mary is at X)` → `(Mary's lamb is at X)`

(where `X` represents any location) and

`Mary is at home`

In clause form the two statements become

`(Mary's lamb is at X) OR ¬(Mary is at X)`

and

`(Mary is at home)`

which at first glance do not have components that can be resolved. On the other hand, the components `(Mary is at home)` and `¬(Mary is at X)` are quite close to being opposites of each other. The problem is to recognize that `Mary is at X`, being a statement about locations in general, is a statement about *home* in particular. Thus a special case of the first statement from above is

`(Mary's lamb is at home) OR ¬(Mary is at home)`

which can be resolved with the statement

`(Mary is at home)`

to produce the statement

`(Mary's lamb is at home)`

The process of assigning values to variables (such as assigning the value *home* to `X`) so that resolution can be performed is called **unification.** It is this process that allows general statements to be applied to specific applications in a deduction system.

Prolog

The programming language Prolog (short for PROgramming in LOGic) is a declarative programming language whose underlying problem-solving algorithm is based on repeated resolution. Such languages are called **logic programming** languages. A program in Prolog consists of a collection of initial statements to which the underlying algorithm applies its deductive reasoning. The components from which these statements are constructed are called **predicates.** A predicate consists of a predicate identifier followed by a parenthetical statement listing the predicate's arguments. A single predicate represents a fact about its arguments, and its identifier is usually chosen to reflect this underlying semantics. Thus if we want to express the fact that Bill is Mary's parent, we can use the predicate form

```
parent(bill, mary)
```

Note that the arguments in this predicate start with lowercase letters, even though they represent proper nouns. This is because Prolog distinguishes arguments that are constants from arguments that are variables by insisting that constants begin with lowercase letters and variables begin with uppercase letters. (Here we have used the terminology of the Prolog culture where the term *constant* is used in place of the more generic term *literal.* More precisely, the term `bill` [note the lowercase] is used in Prolog to represent the literal that might be represented as `"Bill"` in a more generic notation. The term `Bill` [note the uppercase] is used in Prolog to refer to a variable.)

Statements in a Prolog program are either facts or rules, each of which is terminated by a period. A fact consists of a single predicate. For example, the fact that a turtle is faster than a snail could be represented by the Prolog statement

```
faster(turtle, snail).
```

and the fact that a rabbit is faster than a turtle could be represented by

```
faster(rabbit, turtle).
```

A Prolog rule is an "implies" statement. However, instead of writing such a statement in the form `X` $\rightarrow$ `Y`, a Prolog programmer writes "Y if X," except that the symbol `:-` (a colon followed by a dash) is used in place of the word *if.* Thus the rule "X is old implies X is wise" might be expressed by a logician as

```
old(X) → wise(X)
```

but would be expressed in Prolog as

```
wise(X) :- old(X).
```

As another example, the rule

```
(faster(X, Y) AND faster(Y, Z)) → faster(X, Z)
```

would be expressed in Prolog as

```
faster(X, Z) :- faster(X, Y), faster(Y, Z).
```

(The comma separating `faster(X, Y)` and `faster(Y, Z)` represents the conjunction AND.) Although rules such as these are not in clause form, they are allowed in Prolog because they can be easily converted into clause form.

Keep in mind that the Prolog system does not know the meaning of the predicates in a program; it simply manipulates the statements in a totally

symbolic manner according to the resolution inference rule. Thus it is up to the programmer to describe all the pertinent features of a predicate in terms of facts and rules. In this light, Prolog facts tend to be used to identify specific instances of a predicate, whereas rules are used to describe general principles. This is the approach followed by the preceding statements regarding the predicate `faster`. The two facts describe particular instances of "fasterness" while the rule describes a general property. Note that the fact that a rabbit is faster than a snail, although not explicitly stated, is a consequence of the two facts combined with the rule.

When developing software using Prolog, the task of a programmer is to develop the collection of facts and rules that describe the information that is known. These facts and rules constitute the set of initial statements to be used in the deductive system. Once this collection of statements is established, conjectures (called goals in Prolog terminology) can be proposed to the system—usually by typing them at a computer's keyboard. When such a goal is presented to a Prolog system, the system applies resolution to try to confirm that the goal is a consequence of the initial statements. Based on our collection of statements describing the relationship faster, each of the goals

```
faster(turtle, snail).
faster(rabbit, turtle).
faster(rabbit, snail).
```

could be so confirmed because each is a logical consequence of the initial statements. The first two are identical to facts appearing in the initial statements, whereas the third requires a certain degree of deduction by the system.

More interesting examples are obtained if we provide goals whose arguments are variables rather than constants. In these cases Prolog tries to derive the goal from the initial statements while keeping track of the unifications required to do so. Then, if the goal is obtained, Prolog reports these unifications. For example, consider the goal

```
faster(W, snail).
```

In response to this, Prolog reports

```
faster(turtle, snail).
```

Indeed, this is a consequence of the initial statements and agrees with the goal via unification. Furthermore, if we asked Prolog to tell us more, it finds and reports the consequence

```
faster(rabbit, snail).
```

In contrast, we can ask Prolog to find instances of animals that are slower than a rabbit by proposing the goal

```
faster(rabbit, W).
```

In fact, if we started with the goal

```
faster(V, W).
```

Prolog would ultimately seek all the faster relationships that can be derived from the initial statements. This implies that a single Prolog program could be used to confirm that a particular animal is faster than another, to find those animals that

are faster than a given animal, to find those animals that are slower than a given animal, or to find all faster relationships.

This potential versatility is one of the features that has captured the imagination of computer scientists. Unfortunately, when implemented in a Prolog system, the resolution procedure inherits limitations that are not present in its theoretical form, and thus Prolog programs can fail to live up to their anticipated flexibility. To understand what we mean, first note that the diagram in Figure 6.25 displays only those resolutions that are pertinent to the task at hand. There are other directions that the resolution process could pursue. For example, the leftmost and rightmost clauses could be resolved to produce the resolvent Q. Thus, in addition to the statements describing the basic facts and rules involved in an application, a Prolog program often must contain additional statements whose purpose is to guide the resolution process in the correct direction. For this reason actual Prolog programs may not capture the multiplicity of purpose suggested by our previous example.

Questions & Exercises

1. Which of the statements R, S, T, U, and V are logical consequences of the collection of statements (¬R OR T OR S), (¬S OR V), (¬V OR R), (U OR ¬S), (T OR U), and (S OR V)?
2. Is the following collection of statements consistent? Explain your answer.

   P OR Q OR R ¬R OR Q R OR ¬P ¬Q
3. Complete the two rules at the end of the Prolog program below so that the predicate `mother(X, Y)` means "X is the mother of Y" and the predicate `father(X, Y)` means "X is the father of Y."

```
female(carol).
female(sue).
male(bill).
male(john).
parent(john, carol).
parent(sue, carol).
mother(X,Y) :-
father(X,Y) :-
```

4. In the context of the Prolog program in question 3, the following rule is intended to mean that X is Y's sibling if X and Y have a common parent.

```
sibling(X, Y) :- parent(Z, X), parent(Z, Y).
```

   What unexpected conclusion would this definition of the sibling relationship allow Prolog to make?

Chapter Review Problems

Asterisked problems are associated with optional sections.

1. What does it mean to say that a programming language is machine independent?
2. Translate the following Python program into the machine language described in Appendix C.

```
x = 0
while (x < 3):
  x = x + 1
```

3. Translate the statement

```
Halfway = Length + Width
```

into the machine language of Appendix C, assuming that `Length`, `Width`, and `Halfway` are all represented in floating-point notation.

4. Translate the high-level statement

```
if (X == 0):
  Z = Y + W
else:
  Z = Y + X
```

into the machine language of Appendix C, assuming that `W`, `X`, `Y`, and `Z` are all values represented in two's complement notation, each using one byte of memory.

5. Why was it necessary to identify the type of data associated with the variables in question 4 in order to translate the statements? Why do many high-level programming languages require the programmer to identify the type of each variable at the beginning of a program?
6. What are the different levels of programming languages?
7. Suppose the function *f* expects a string as its input and returns the same string as its output by removing the odd-positioned letters from the string. What is the result returned by function *f(f("programming"))*?
8. Suppose the function *f* expects two numeric values as its inputs and returns their addition as its output value, and *g* is a function that returns the subtraction of the two values given as its input. If *a* and *b* represent numeric values, what is the result returned by *f(f(a,b), g(a,b))*?
9. Suppose you are going to write an object-oriented program for calculating grades of students. What data should be stored inside the object representing a student's grades? To what messages should that object be able to respond? What are the different objects that might be used in the program?
10. Summarize the distinction between a machine language and an assembly language.
11. Design an assembly language for the machine described in Appendix C.
12. John Programmer argues that the ability to declare constants within a program is not necessary because variables can be used instead. For example, our example of `AirportAlt` in Section 6.2 could be handled by declaring `AirportAlt` to be a variable and then assigning it the required value at the beginning of the program. Why is this not as good as using a constant?
13. Summarize the distinction between the declaration and the definition of a variable.
14. Explain the differences between a local variable and a global variable.
15. a. What is operator precedence?
    b. Depending on operator precedence, what values could be associated with the expression $6 + 2 \times 3$?
16. Explain the advantages of code reuse.
17. What will be the output of the following C code? Explain your answer.

```
#define sqr(X)  X * X
main() {
int k = sqr (10 + 20);
  printf("%d", k);
}
```

18. Draw a flowchart representing the structure expressed by the following for statement.

```
for (int x = 2; x < 8; ++x)
{ . . . }
```

19. Translate the following `while` statement into an equivalent program segment using the Python `for` statement. Initialization: `x = 1`

```
while ( x! = 100)
{ x = x + 1 }
```

20. If you are familiar with written music, analyze musical notation as a programming language. What are the control structures? What is the syntax for inserting program comments? What music notation has semantics similar to the `for` statement in Figure 6.7?

21. Rewrite the following program segment using a `while` loop instead of a `for` loop.

```
for (i = 0; i<100; i++) {
if (i % 2 == 0)
{
   print(i)
}
}
```

22. Draw a flowchart representing the structure expressed by the following `if` and `else` conditional statements.

```
if (a > b) {
if (c > a) {
if (d > c) { print(d) }
}
} else if (b > a) {
if (c > b) {
if (d > c) { print(d) }
} }
```

23. Summarize the following rat's-nest routine with a single if-else statement:

```
    if X > 5 then goto 80
    X = X + 1
    goto 90
80  X = X + 2
90  stop
```

24. Summarize the basic control structures found in imperative and object-oriented programming languages for performing each of the following activities:

a. Determining which command should be executed next
b. Repeating a collection of commands
c. Changing a variable's value

25. Summarize the distinction between code generation and code optimization.

26. Suppose the variable `P` in a program is of type integer. What will be the output of the following two program statements?

```
P = 2.5 + 3.5
  print(P)
```

27. What does it mean to say that a programming language is strongly typed?

28. Can we return more than one value from a function? Explain your answer.

29. Suppose the Python function `Modify` is defined by

```
def Modify (Y):
  Y = 7
  print(Y)
```

If parameters are passed by value, what will be printed when the following program segment is executed? What if parameters are passed by reference?

```
X = 5
Modify(X)
print(X)
```

30. Suppose the Python function `Modify` is defined by

```
def Modify (Y):
  Y = 9
  print(X)
  print(Y)
```

Also suppose that `X` is a global variable. If parameters are passed by value, what will be printed when the following program segment is executed? What if parameters are passed by reference?

```
X = 5
Modify(X)
print(X)
```

31. Sometimes an actual parameter is passed to a function by producing a duplicate to be used by the function (as when the parameter is passed by value), but when the function is completed the value in the function's copy is transferred to the actual parameter before the calling function continues. In such cases

the parameter is said to be passed by value-result. What would be printed by the program segment in question 30 if parameters were passed by value-result?

32. What is the main difference between passing parameters to a function by value and passing parameters to a function by address? Explain your answer with a proper example for each.

33. What ambiguity exists in the statement

```
P = 9/4 - 1
```

34. Suppose a small company has five employees and is planning to increase the number to six. Moreover, suppose one of the company's programs contained the following assignment statements.

```
DailySalary = TotalSal/5;
AvgSalary   = TotalSal/5;
DailySales  = TotalSales/5;
AvgSales    = TotalSales/5;
```

How would the task of updating the program be simplified if the program had originally been written using constants named `NumberOfEmp` and `WorkWeek` (both set to the value 5) so that the assignment statements could be expressed as

```
DailySalary = TotalSal/DaysWk;
AvgSalary   = TotalSal/NumEmpl;
DailySales  = TotalSales/DaysWk;
AvgSales    = TotalSales/NumEmpl;
```

35. a. What is the distinction between an assembly language and a third-generation programming language?
b. Give an example of each.

36. Draw a syntax diagram representing the structure of the Python nested `if-else` statement.

37. Design a set of syntax diagrams to describe the syntax of telephone numbers in your locality. For instance, in the United States telephone numbers consist of an area code, followed by a regional code, followed by a four-digit number such as (444) 555–1234.

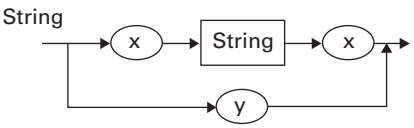
38. Design a set of syntax diagrams to describe simple sentences in your native language.

39. Design a set of syntax diagrams to describe different ways of representing full names such as *firstname - middlename - lastname* or the reverse of it.

40. Design a set of syntax diagrams that describes the grammatical structure of "sentences" that consist of occurrences of the word *yes* followed by the same number of the word *no*. For example, "*yes yes no no*" would be such a sentence, whereas "*no yes*," "*yes no no*," and "*yes no yes*" would not.

41. Give an argument to the effect that a set of syntax diagrams cannot be designed that describes the grammatical structure of "sentences" that consist of occurrences of the word *yes*, followed by the same number of occurrences of the word *no*, followed by the same number of occurrences of the word *maybe*. For example, "*yes no maybe*" and "*yes yes no no maybe maybe*" would be such sentences, whereas "*yes maybe*," "*yes no no maybe maybe*," and "*maybe no*" would not.

42. Write a sentence describing the structure of a string as defined by the following syntax diagram. Then, draw the parse tree for the string *xxyxx*.



43. Add syntax diagrams to those in question 5 of Section 6.4 to obtain a set of diagrams that defines the structure Dance to be either a Chacha or a Waltz, where a Waltz consists of one or more copies of the pattern

forward diagonal close

or

backward diagonal close

44. Draw the parse tree for the expression a ÷ d × b ÷ c + a ÷ d × b ÷ c based on the syntax diagrams in Figure 6.15.

45. What code optimization could be performed by a code generator when building the machine code representing the statement

```
if (X == 5):
  Z = X + 2
```

```
else:
  Z = X + 4
```

46. What will be the output of following code?

```
if (printf ("Hello")) {
  printf("World");
} else {
  printf("Hello");
}
```

47. Simplify the following program segment

```
while (Z < 10):
Z++;
```

48. In an object-oriented programming environment, how are types and classes similar? How are they different?

49. Describe how inheritance might be used to develop classes describing various types of sports.

50. What is the difference between the public and private parts of a class?

51. a. Give an example of a situation in which an instance variable should be private.
b. Give an example of a situation in which an instance variable should be public.
c. Give an example of a situation in which a method should be private.
d. Give an example of a situation in which a method should be public.

52. Describe some objects that might be found in a program for simulating the pedestrian traffic in a hotel lobby. Include explanations of the actions some of the objects should be able to perform.

***53.** What does the term *type cast* mean in the context of a programming language?

***54.** What is the difference between `while` and `do-while` loops in the context of a programming language?

***55.** Draw a diagram (similar to Figure 6.25) representing the resolutions needed to show that the collection of statements `(Q OR ¬R)`, `(T OR R)`, `¬P`, `(P OR ¬T)`, and `(P OR ¬Q)` are inconsistent.

***56.** Is the collection of statements `¬R`, `(T OR R)`, `(P OR ¬Q)`, `(Q OR ¬T)`, and `(R OR ¬P)` consistent? Explain your answer.

***57.** Extend the Prolog program outlined in questions 3 and 4 of Section 6.7 to include the additional family relationships of uncle, aunt, grandparent, and cousin. Also add a rule that defines `parents (X, Y, Z)` to mean that `X` and `Y` are `Z`'s parents.

***58.** Assuming that the first statement in the following Prolog program is intended to mean "Alice likes sports," translate the last two statements of the program. Then, list all the things that, based on this program, Prolog would be able conclude that Alice likes. Explain your list.

```
likes(alice, sports).
likes(alice, music).
likes(carol, music).
likes(david, X) :- likes(X, sports).
likes(alice, X) :- likes(david, X).
```

***59.** What problem would be encountered if the following program segment was executed on a computer in which values are represented in the eight-bit floating-point format described in Section 1.7?

```
X = 0.01
while (X != 1.00):
  print(X)
  X = X + 0.01
```

Social Issues

The following questions are intended as a guide to the ethical/social/legal issues associated with the field of computing. The goal is not merely to answer these questions. You should also consider why you answered as you did and whether your justifications are consistent from one question to the next.

1. In general, copyright laws support ownership rights associated with the expression of an idea but not for the idea itself. As a result, a paragraph in a book is copyrightable but the ideas expressed in the paragraph are not. How should this right extend to source programs and the algorithms they express? To what extent should a person who knows the algorithms used in a commercial software package be allowed to write his or her own program expressing those same algorithms and market this version of the software?
2. By using a high-level programming language a programmer is able to express algorithms using words such as *if, else,* and *while.* To what extent does the computer understand the meaning of those words? Does the ability to respond correctly to the use of words imply an understanding of the words? How do you know when another person has understood what you said?
3. Should a person who develops a new and useful programming language have a right to profit from the use of that language? If so, how can that right be protected? To what extent can a language be owned? To what extent should a company have the right to own the creative, intellectual accomplishments of its employees?
4. With a deadline approaching, is it acceptable for a programmer to forgo documentation via comment statements to get a program running on time? (Beginning students are often surprised to learn how important documentation is considered among professional software developers.)
5. Much of the research in programming languages has been to develop languages that allow programmers to write programs that can be easily read and understood by humans. To what extent should a programmer be required to use such capabilities? That is, to what extent is it good enough for the program to perform correctly even though it is not well written from a human perspective?
6. Suppose an amateur programmer writes a program for his or her own use and in doing so is sloppy in the program's construction. The program does not use the programming language features that would make it more readable, it is not efficient, and it contains shortcuts that take advantage of the particular situation in which the programmer intends to use the program. Over time the programmer gives copies of the program to friends who want to use it themselves, and these friends give it to their friends. To what extent is the programmer liable for problems that might occur?
7. To what extent should a computer professional be knowledgeable in the various programming paradigms? Some companies insist that all software developed in that company be written in the same, predetermined programming language. Does your answer to the original question change if the professional works for such a company?

Additional Reading

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools,* 2nd ed. Boston, MA: Addison-Wesley, 2007.

Barnes, J. *Programming in Ada 2005.* Boston, MA: Addison-Wesley, 2006.

Clocksin, W. F., and C. S. Mellish. *Programming in Prolog,* 5th ed. New York: Springer-Verlag, 2013.

Friedman, D. P., and M. Felleisen. *The Little Schemer,* 4th ed. Cambridge, MA: MIT Press, 1995.

Hamburger, H., and D. Richards. *Logic and Language Models for Computer Science.* Upper Saddle River, NJ: Prentice-Hall, 2002.

Kernighan, B.W., and D.M. Ritchie. *The C Programming Language,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

Metcalf, M., and J. Reid. *Fortran 90/95 Explained,* 2nd ed. Oxford, England: Oxford University Press, 1999.

Pratt, T. W., and M. V. Zelkowitz. *Programming Languages, Design and Implementation,* 4th ed. Upper Saddle River, NJ: Prentice-Hall, 2001.

Savitch, W., and K. Mock. *Absolute C++,* 5th ed. Boston, MA: Addison-Wesley, 2012.

Savitch, W., and K. Mock. *Absolute Java,* 5th ed. Boston, MA: Addison-Wesley, 2012.

Savitch, W. Problem Solving with C++, 8th ed. Boston, MA: Addison-Wesley, 2011.

Scott, M. L. *Programming Language Pragmatics,* 3rd ed. New York: Morgan Kaufmann, 2009.

Sebesta, R. W. *Concepts of Programming Languages,* 10th ed. Boston, MA: Addison Wesley, 2012.

Wu, C. T. *An Introduction to Object-Oriented Programming with Java,* 5th ed. Burr Ridge, IL: McGraw-Hill, 2009.