1 INTRODUCTION

1.1 What is a Programming Language?
1.2 Abstractions in Programming Languages
1.3 Computational Paradigms
1.4 Language Definition
1.5 Language Translation
1.6 Language Design

How we communicate influences how we think, and vice versa. Similarly, how we program computers influences how we think about them, and vice versa. Over the last several decades a great deal of experience has been accumulated in the design and use of programming languages. Although there are still aspects of the design of programming languages that are not completely understood, the basic principles and concepts now belong to the fundamental body of knowledge of computer science. A study of these principles is as essential to the programmer and computer scientist as the knowledge of a particular programming language such as C or Java. Without this knowledge it is impossible to gain the needed perspective and insight into the effect programming languages and their design have on the way we communicate with computers and the ways we think about computers and computation.

It is the goal of this text to introduce the major principles and concepts underlying all programming languages without concentrating on one particular language. Specific languages are used as examples and illustrations. These languages include C, Java, C++, Ada, ML, LISP, FORTRAN, Pascal and Prolog. It is not necessary for the reader to be familiar with all these languages, or even any of them, to understand the concepts being illustrated. At most the reader is required to be experienced in only one programming language and to have some general knowledge of data structures, algorithms, and computational processes.

In this chapter we will introduce the basic notions of programming languages and outline some of the basic concepts. We will also briefly discuss the role of language translators. However, the techniques used in building language translators will not be discussed in detail in this book.

1.1 *WHAT IS A PROGRAMMING LANGUAGE?*

A definition often advanced for a programming language is "a notation for communicating to a computer what we want it to do."

But this definition is inadequate. Before the 1940s computers were programmed by being "hard-wired": switches were set by the programmer to connect the internal wiring of a computer to perform the requested tasks. This effectively communicated to the computer what computations were desired, yet switch settings can hardly be called a programming language.

A major advance in computer design occurred in the 1940s, when John von Neumann had the idea that a computer should not be "hardwired" to do particular things, but that a series of codes stored as data would determine the actions taken by a central processing unit. Soon programmers realized that it would be a tremendous help to attach symbols to the instruction codes, as well as to memory locations, and **assembly language** was born, with instructions such as

```
LDA #2
STA X
```

But assembly language, because of its machine dependence, low level of abstraction, and difficulty in being written and understood, is also not what we usually think of as a programming language and will not be studied further in this text. (Sometimes, assembly language is referred to as a **low-level language** to distinguish it from the **high-level languages,** which are the subject of this text.) Indeed, programmers soon realized that a higher level of abstraction would improve their ability to write concise, understandable instructions that could be used with little change from

machine to machine. Certain standard constructions, such as assignment, loops, and selections or choices, were constantly being used and had nothing to do with the particular machine; these constructions should be expressible in simple standard phrases that could be translated into machine-usable form, such as the C code for the previous assembly language instructions (indicating assignment of the value 2 to the location with name **X**)

```
X = 2
```

Programs thus became relatively machine independent, but the language still reflected the underlying architecture of the von Neumann model of a machine: an area of memory where both programs and data are stored and a separate central processing unit that sequentially executes instructions fetched from memory. Most modern programming languages still retain the flavor of this processor model of computation. With increasing abstraction, and with the development of new architectures, particularly parallel processors, came the realization that programming languages need not be based on any particular model of computation or machine, but need only describe computation or processing in general.

A parallel evolution has also led away from the use of programming languages to communicate solely from humans to computers to the use of such languages to commmuncate from human to human. Indeed, while it is still an important requirement that a programming language allow humans to easily *write* instructions, in the modern world of very large programming projects, it is even more important that other programmers be able to *read* the instructions as well.

This leads us to state the following definition.

> **Definition:** A **programming language** is a notational system for describing computation in machine-readable and human-readable form.

We will discuss the three key concepts in this definition.

Computation. Computation is usually defined formally using the mathematical concept of a **Turing machine,** which is a kind of computer whose operation is simple enough to be described with great precision. Such a machine needs also to be powerful enough to perform any computation that a computer can, and Turing machines are known to be able to carry out any computation that current computers are capable of (though certainly not as efficiently). In fact, the generally accepted **Church's thesis** states that it is not possible to build a machine that is inherently more powerful than a Turing machine.

Our own view of computation in this text is less formal. We will think of computation as any process that can be carried out by a computer. Note, however, that computation does not mean simply mathematical calculation, such as the computation of the product of two numbers or the logarithm of a number. Computation instead includes *all* kinds of computer operations, including data manipulation, text processing, and information storage and retrieval. In this sense, computation is used as a synonym for processing of any kind on a computer. Sometimes a programming language will be designed with a particular kind of processing in mind, such as report generation, graphics, or database maintenance. Although such **special-purpose languages** may be able to express more general kinds of computations, in this text we will concentrate on the **general-purpose languages** that are designed to be used for general processing and not for particular purposes.

Machine readability. For a language to be machine-readable, it must have a simple enough structure to allow for efficient translation. This is not something that depends on the notion of any particular machine, but is a general requirement that can be stated precisely in terms of definiteness and complexity of translation. First, there must be an **algorithm** to translate a language, that is, a step-by-step process that is unambiguous and finite. Second, the algorithm cannot have too great a complexity: most programming languages can be translated in time that is proportional to the size of the program. Otherwise, a computer might spend more time on the translation process than on the

actual computation being described. Usually, machine readability is ensured by restricting the structure of a programming language to that of the so-called **context-free languages,** which are studied in Chapter 4, and by insisting that all translation be based on this structure.

Human readability. Unlike machine readability, this is a much less precise notion, and it is also less understood. It requires that a programming language provide **abstractions** of the actions of computers that are easy to understand, even by persons not completely familiar with the underlying details of the machine.[1] One consequence of this is that programming languages tend to resemble natural languages (like English or Chinese), at least superficially. This way, a programmer can rely on his or her natural understanding to gain immediate insight into the computation being described. (Of course, this can lead to serious misunderstandings as well.)

Human readability acquires a new dimension as the size of a program increases. (Some programs are now as large as the largest novels.) The readability of large programs requires suitable mechanisms for reducing the amount of detail required to understand the program as a whole. For example, in a large program we would want to localize the effect a small change in one part of the program would have—it should not require major changes to the entire program. This requires the collection of local information in one place and the prevention of this information from being used indiscriminately throughout the program. The development of such abstraction mechanisms has been one of the important advances in programming language design over the past two decades, and we will study such mechanisms in detail in Chapter 9.

Large programs also often require the use of large groups of programmers, who simultaneously write separate parts of the programs. This substantially changes the view that must be taken of a programming language. A programming language is no longer a way of describing computation, but it becomes part of a **software development environment** that promotes and enforces a software design methodology. Software development environments not only contain facilities for writing and translating programs in one or more programming languages, but also have facilities for manipulating program files, keeping records of changes, and performing debugging, testing, and analysis. Programming languages thus become part of the study of **software engineering.** Our view of programming languages, however, will be focused on the languages themselves rather than on their place as part of such a software development environment. The design issues involved in integrating a programming language into a software development environment can be treated more adequately in a software engineering text.

1.2 *ABSTRACTIONS IN PROGRAMMING LANGUAGES*

We have noted the essential role that abstraction plays in providing human readability of programs. In this section we briefly describe common abstractions that programming languages provide to express computation and give an indication of where they are studied in more detail in subsequent chapters. Programming language abstractions fall into two general categories: **data abstraction** and **control abstraction.** Data abstractions abstract properties of the data, such as character strings, numbers, or search trees, which is the subject of computation. Control abstractions abstract properties of the transfer of control, that is, the modification of the execution path of a program based on the situation at hand. Examples of control abstractions are loops, conditional statements, and procedure calls.

Abstractions also fall into **levels,** which can be viewed as measures of the amount of information contained in the abstraction. **Basic abstractions** collect together the most localized machine information. **Structured abstractions** collect more global information about the structure of the program. **Unit abstractions** collect information about entire pieces of a program.

In the following paragraphs we classify common abstractions according to the levels of abstraction, for both data abstraction and control abstraction.

[1] Human *writability* also requires such abstractions to reduce the effort of expressing a computation. Writability is related to but not the same as readability; see Chapter 3.

1.2.1 Data Abstractions

Basic abstractions. Basic data abstractions in programming languages abstract the internal representation of common data values in a computer. For example, integer data values are often stored in a computer using a two's complement representation, and standard operations such as addition and multiplication are provided. Similarly, a real, or floating-point, data value is usually provided. Locations in computer memory that contain data values are abstracted by giving them names and are called **variables.** The kind of data value is also given a name and is called a **data type.** Data types of basic data values are usually given names that are variations of their corresponding mathematical values, such as **`int`** or **`integer`** and **`real`** or **`float`**. Variables are given names and data types using a **declaration,** such as the Pascal

```
var x : integer;
```

or the equivalent C declaration

```
int x;
```

In this example, **`x`** is established as the name of a variable and is given the data type *integer.* Data types are studied in Chapter 6 and declarations in Chapter 5.

Structured abstractions. The **data structure** is the principal method for abstracting collections of data values that are related. For example, an employee record may consist of a name, address, phone number, and salary, each of which may be a different data type, but together represent the record as a whole. Another example is that of a group of items, all of which have the same data type and which need to be kept together for purposes of sorting or searching. A typical data structure provided by programming languages is the **array,** which collects data into a sequence of individually indexed items. Variables can be given a data structure in a declaration, as in the C

```
int a[10];
```

or the FORTRAN

```
INTEGER a(10)
```

which establish the variable **`a`** as containing an array of ten integer values. Data structures can also be viewed as new data types that are not internal, but are constructed by the programmer as needed. In many languages these types can also be given type names, just as the basic types, and this is done in a **type declaration,** such as the C

```
typedef int Intarray[10];
```

which defines the new name **`Intarray`** for the type *array of integer*, with space for ten values. Such data types are called **structured types.** The different ways of creating and using structured types are studied in Chapter 6.

Unit abstractions. In a large program, it is useful and even necessary to collect related code into specific locations within a program, either as separate files or as separate language structures within a file. Typically such abstractions include access conventions and restrictions, which traditionally have been referred to as **data encapsulation** and **information hiding**. These mechanisms vary widely from language to language, but are often associated with structured types in some way, and thus are related (but not identical) to **abstract data types**. Examples include the **module** of ML and Haskell, and the **package** of Ada and Java. A somewhat

intermediate abstraction more closely related to abstract data types is the **class** mechanism of object-oriented languages, which could be categorized as either a structured abstraction or a unit abstraction (or both), depending on the language. Classes generally offer data encapsulation and possibly also information hiding, and may also have some of the characteristics of modules or packages. In this text we study modules and abstract data types in Chapter 9, while classes (and their relation to abstract data types) are studied in Chapter 10.

An additional property of a unit data abstraction that has become increasingly important is its **reusability** -- the ability to reuse the data abstraction in different programs, thus saving the cost of writing abstractions from scratch for each program. Typically such data abstractions represent **components** (operationally complete pieces of a program or user interface) or **containers** (data structures containing other user-defined data) and are entered into a library of available components and containers. As such, unit data abstractions become the basis for language **library** mechanisms (the library mechanism itself, as well as certain standard libraries, may or may not be part of the language itself), and their ability to be easily combined with each other (their **interoperability**) is enhanced by providing standard conventions for their interfaces. Many interface standards have been developed, either independent of the programming language, or sometimes tied to a specific language. Most of these apply to the class structure of object-oriented languages, since classes have proven to be more flexible for reuse than most other language structures (see the next section and Chapter 10). Two language independent standards for communicating processes (that is, programs that are running separately) are Microsoft's Component Object Model (COM) and the Common Object Request Broker Architecture (CORBA) recognized as an international standard. A somewhat different interface standard is the JavaBeans application programming interface tied to the Java programming language. A study of such interface standards is beyond the scope of this book, however.

1.2.2 Control Abstractions

Basic abstractions. Typical basic control abstractions are those statements in a language that combine a few machine instructions into a more understandable abstract statement. We have already mentioned the as**signment statement** as a typical instruction that abstracts the computation and storage of a value into the location given by a variable, as for example,

```
x = x + 3
```

This assignment statement represents the fetching of the value of the variable **`x`**, adding the integer 3 to it, and storing the result in the location of **`x`**. Assignment is studied in Chapter 5.

Another typical basic control statement is the **goto** statement, which abstracts the jump operation of a computer or the transfer of control to a statement elsewhere in a program, such as the FORTRAN

```
      . . .
      GOTO 10
C     this part skipped
      . . .
C     control goes here
10    CONTINUE
      . . .
```

Goto statements today are considered too close to the actual operation of a computer to be a useful abstraction mechanism (except in special situations), so most modern languages provide only very limited forms of this statement. See Chapter 7 for a discussion.

Structured abstractions. Structured control abstractions divide a program into groups of instructions that are nested within tests that govern their execution. Typical examples are selection statements,

such as the **if-statement** of many languages, the **case-statement** of Pascal, and the **switch-statement** of C. For example, in the following C code,

```
if (x > 0)
{  numSolns = 2;
   r1 = sqrt (x);
   r2 = - r1;
}
else
{  numSolns = 0;
}
```

the three statements within the first set of curly brackets are executed if **x** > 0, and the single statement within the second set of curly brackets otherwise. (C allows the second set of brackets to be deleted, since they only surround a single statement.)

Some languages provide additional support for such structured control by, for example, allowing indentation to substitute for the curly brackets, as in the Haskell

```
case numSolns(x) of
   0     ->  []
   2     ->  [sqrt(x),-sqrt(x)]
```

which is shorthand for

```
case numSolns(x) of
{  0     ->  [] ;
   2     ->  [sqrt(x),-sqrt(x)]
}
```

(This is called the *layout rule* in Haskell.)

Ada goes one step farther in structured control, in that the opening of a group of nested statements is automatic and does not require a "begin":

```
if x > 0.0 then
   numSolns := 2;
   r1 := sqrt(x);
   r2 := -r1;
else
   numSolns := 0;
end if;
```

(Note the required keywords **end if** at the end of the if-statement.)

One advantage of structured control structures is that they can be nested within other control structures, usually to any desired depth, as in the following C code (which is a modification of the foregoing example):

```
if (x > 0)
{  numSolns = 2;
   r1 = sqrt (x);
   r2 = - r1;
}
else
{  if (x == 0)
   {  numSolns = 1;
      r1 = 0.0;
```

```
    }
    else
    {   numSolns = 0;
    }
}
```

which is usually written more compactly as

```
if (x > 0)
{   numSolns = 2;
    r1 = sqrt (x);
    r2 = - r1;
}
else if (x == 0)
{   numSolns = 1;
    r1 = 0.0;
}
else numSolns = 0;
```

or the Ada,

```
if x > 0.0 then
  numSolns:= 2;
  r1 := sqrt (x);
  r2 := r1;
elsif x = 0.0 then
  numSolns := 1;
  r1 := 0.0;
else
  numSolns:= 0;
end if;
```

Structured looping mechanisms come in many forms, including the **while**, **for**, and **do** loops of C and C++, the **repeat** loops of Pascal, and the **loop** statement of Ada. For example, the following program fragments, first in C and then Ada, both compute **x** to be the greatest common divisor of **u** and **v** using Euclid's algorithm (for example, the greatest common divisor of 8 and 20 is 4, and the greatest common divisor of 3 and 11 is 1):

```
/* C example */
x = u;
y = v;
while (y != 0)  /* not equal in C */
{  t = y;
   y = x % y;   /* the integer remainder operation in C */
   x = t;
}

-- Ada example
x := u;
y := v;
loop
  exit when y = 0;
   t := y;
   y := x mod y; -- modulo, similar to % in C
   x := t;
end loop;
```

Structured selection and loop mechanisms are studied in Chapter 7.

A further, powerful mechanism for structuring control is the **procedure**, sometimes also called a **subprogram** or **subroutine**. This allows a programmer to consider a sequence of actions as a single action and to control the interaction of these actions with other parts of the program. Procedure abstraction involves two things. First, a procedure must be defined by giving it a name and associating with it the actions that are to be performed. This is called **procedure declaration,** and it is similar to variable and type declaration, mentioned earlier. Second, the procedure must actually be called at the point where the actions are to be performed. This is sometimes also referred to as procedure invocation or procedure **activation.**

As an example, consider the sample code fragment that computes the greatest common divisor of integers `u` and `v`. We can make this into a procedure in Ada with the procedure declaration as given in Figure 1.1.

Figure 1.1. An Ada `gcd` procedure

```
(1)   procedure gcd (u, v: in integer; x: out integer) is
(2)   y, t, z: integer;
(3)   begin
(4)     z := u;
(5)     y := v;
(6)     loop
(7)       exit when y = 0;
(8)       t := y;
(9)       y := z mod y;
(10)      z := t;
(11)    end loop;
(12)    x := z;
(13)  end gcd;
```

In this declaration, `u, v,` and `x` have become **parameters** to the procedure (line 1), that is, things that can change from call to call. This procedure can now be **called** by simply naming it and supplying appropriate **actual parameters** or **arguments,** as in

```
gcd (8, 18, d);
```

which gives `d` the value 2. (The parameter `x` is given the `out` label in line 1 to indicate that its value is computed by the procedure itself and will change the value of the corresponding actual parameter of the caller.)

In FORTRAN, by contrast, a procedure is declared as a subroutine,

```
SUBROUTINE gcd (u, v, x)
. . .
END
```

and is called using an explicit call-statement:

```
CALL gcd (a, b, d)
```

Procedure call is a more complex mechanism than selection or looping, since it requires the storing of information about the condition of the program at the point of the call and the way the called procedure operates. Such information is stored in a **runtime environment.** Procedure calls, parameters, and runtime environments are all studied in Chapter 8. (The basic kinds of runtime environments are also mentioned in Section 1.5 of this chapter.)

An abstraction mechanism closely related to procedures is the **function**, which can be viewed simply as a procedure that returns a value or result to its caller. For example, the Ada code for the `gcd` procedure in Figure 1.1 can be more appropriately be written as a function as given in Figure 1.2.

Figure 1.2 An Ada `gcd` function

```
(1)   function gcd (u, v: in integer) return integer is
(2)     y, t, z: integer;
(3)    begin
(4)     z := u;
(5)     y := v;
(6)     loop
(7)       exit when y = 0;
(8)       t := y;
(9)       y := z mod y;
(10)      z := t;
(11)    end loop;
(12)    return z;
(13)  end gcd;
```

In some languages, procedures are viewed as special cases of functions that return no value, as in C and C++, where procedures are called **void functions** (since the return value is declared to be void, or non-existent). However, the importance of functions is much greater than this correspondence to procedures implies, since functions can be written in such a way that they correspond more closely to the mathematical abstraction of a function, and thus, unlike procedures, can be understood independently of any concept of a computer or runtime environment. This is the basis for the functional programming paradigm and the functional languages mentioned in the next section, and discussed in detail in Chapter 11.

Unit abstractions. Control can also be abstracted to include a collection of procedures that provide logically related services to other parts of a program and that form a **unit,** or stand-alone, part of the program. For example, a data management program may require the computation of statistical indices for stored data, such as mean, median, and standard deviation. The procedures that provide these operations can be collected into a program unit that can be translated separately and used by other parts of the program through a carefully controlled interface. This allows the program to be understood as a whole without needing to know the details of the services provided by the unit.

Note that what we have just described is essentially the same as a unit-level data abstraction, and is usually implemented using the same kind of module or package language mechanism. The only difference is that here the focus is on the operations rather than the data. But the goals of reusability and library building remain the same.

One kind of control abstraction that is difficult to fit into any one abstraction level is that of parallel programming mechanisms. Many modern computers have several processors or processing elements and are capable of processing different pieces of data simultaneously. A number of programming languages include mechanisms that allow for the parallel execution of parts of programs, as well as providing for synchronization and communication among such program parts. Java has mechanisms for declaring **threads** (separately executed control paths within the Java system) and **processes** (other programs executing outside the Java system). Ada provides the **task** mechanism for parallel execution. Ada's tasks are essentially a unit abstraction, while Java's threads and processes are classes and so structured abstractions, albeit part of the standard **java.lang** package. Other languages provide different levels of parallel abstractions, even down to the statement level. Parallel programming mechanisms are surveyed in Chapter 14.

It is worth noting that almost all abstraction mechanisms are provided for human readability. If a programming language needs to describe only computation, then it needs only enough mechanisms to be able to describe all the computations that a Turing machine can perform, since, as we noted previously, a Turing machine can perform any known computation on a computer. Such a language is called **Turing complete.** As the following property shows, Turing completeness can be achieved with very few language mechanisms:

> A programming language is Turing complete provided it has integer variables and arithmetic and sequentially executes statements, which include assignment, selection (if) and loop (while) statements.

1.3 *COMPUTATIONAL PARADIGMS*

Programming languages began by imitating and abstracting the operations of a computer. It is not surprising that the kind of computer for which they were written had a significant effect on their design. In most cases the computer in question was the von Neumann model mentioned in Section 1.1: a single central processing unit that sequentially executes instructions that operate on values stored in memory. Indeed, the result on Turing completeness of the previous section explicitly referred to sequential execution and the use of variables and assignment. These are typical features of a language based on the von Neumann model: variables represent memory values, and assignment allows the program to operate on these memory values.

A programming language that is characterized by these three properties—the sequential execution of instructions, the use of variables—representing memory locations, and the use of assignment to change the values of variables—is called an **imperative** language, since its primary feature is a sequence of statements that represent commands, or imperatives. Sometimes such languages are also called **procedural** but this has nothing explicitly to do with the concept of procedures discussed earlier.

Most programming languages today are imperative. But it is not necessary for a programming language to describe computation in this way. Indeed, the requirement that computation be described as a sequence of instructions, each operating on a single piece of data, is sometimes referred to as the **von Neumann bottleneck,** since it restricts the ability of a language to indicate parallel computation, that is, computation that can be applied to many different pieces of data simultaneously, and nondeterministic computation, or computation that does not depend on order.[2] Thus it is reasonable to ask if there are ways to describe computation that are less dependent on the von Neumann model of a computer. Indeed there are, and these will be described shortly. Imperative programming languages therefore become only one **paradigm,** or pattern, for programming languages to follow.

Two alternative paradigms for describing computation come from mathematics. The **functional** paradigm comes from mathematics and is based on the abstract notion of a function as studied in the lambda calculus. The **logic** paradigm is based on symbolic logic. Each of these will be the subject of a subsequent chapter, but we will discuss them in a little more detail here. The importance of these paradigms is their correspondence to mathematical foundations, which allows for program behavior to be described abstractly and precisely, thus making it much easier to judge (even without a complete theoretical analysis) that a program will execute correctly, and permitting very concise code to be written even for very complex tasks.

A fourth programming paradigm has also become of enormous importance over the last ten years, and that is the **object-oriented** paradigm. Languages using this paradigm have been very successful in allowing programmers to write reusable, extensible code that operates in a way that mimics the real world, thus allowing programmers to use their natural intuition about the world to understand the behavior of a program and construct appropriate code (as with "natural language" syntax, this can also lead to misunderstanding and confusion, however, if relied on too extensively). In a sense, though, the object-oriented paradigm is an extension of the imperative paradigm, in that it

[2] Parallel and nondeterministic computations are related concepts. See Chapter 14.

relies primarily on the same sequential execution with a changing set of memory locations. The difference is that the resulting programs consist of a large number of very small pieces whose interactions are carefully controlled and yet easily changed. Still, the behavior of object-oriented programs is often even harder to understand fully and especially to describe abstractly, so that, along with practical success comes a certain difficulty in precisely predicting behavior and determining correctness. Still, the object-oriented paradigm has essentially become the new standard, much as the imperative paradigm was in the past, and so will feature prominently throughout this book.

We treat each of these paradigms in a little more detail in the following, in roughly decreasing order of importance in current practice: object-oriented programming first, then functional programming, and finally logic programming. Later, an entire chapter will be devoted to each of these paradigms.

Object-oriented programming. The object-oriented paradigm is based on the notion of an object, which can be loosely described as a collection of memory locations together with all the operations that can change the values of these memory locations. The standard simple example of an object is a variable, with operations to assign it a value and to fetch its value. It represents computation as the interaction among, or communication between, a group of objects, each of which behaves like its own computer, with its own memory and its own operations. In many object-oriented languages, objects are grouped into **classes** that represent all the objects with the same properties. Classes are defined using declarations, much as structured types are declared in a language like C or Pascal. Objects are then created as particular examples, or **instances,** of a class.

Consider the example of the greatest common divisor of two integers from the previous section, which we wrote as either a procedure or a function with two incoming integer values and an outgoing integer result. The object-oriented view of the **gcd** is to consider it to be an operation, or **method**, available for integer-like objects. Indeed, in Java, such a **gcd** method is already available for **BigInteger** objects from the standard library (a **BigInteger** is an integer with an arbitrarily large number of digits). Here, we would like to show an implementation of **gcd** for ordinary integers in Java. Unfortunately, ordinary integers in Java are not "real" objects (an efficiency tradeoff, similar to C++, but unlike Smalltalk, in which all data values are implicitly objects).[3] Thus, we must include the integers themselves in the class that defines integer objects with a **gcd** method, as in the Java code in Figure 1.3.

Figure 1.3 A Java **gcd** class

```
(1)  public class IntWithGcd
(2)  { public IntWithGcd( int val ) { value = val; }
(3)    public int intValue() { return value; }
(4)    public int gcd ( int v )
(5)    { int z = value;
(6)      int y = v;
(7)      while ( y != 0 )
(8)      { int t = y;
(9)        y = z % y;
(10)       z = t;
(11)     }
(12)     return z;
(13)   }
(14)   private int value;
```

[3] For those who know Java, it may appear reasonable to use the standard **Integer** class already available. Unfortunately, this class is defined to be "final", so a **gcd** method cannot be added to it. See, however, the **BigInteger** class in Java, which contains a **gcd** method similar to this one.

```
(15)    }
```

This class defines four things within it. First, a **constructor** is defined on line 2 (by definition it has the same name as the class, but no return type). A constructor allocates memory and provides initial values for the data of an object. In this case, the constructor needs to be provided with an integer, which is the "value" of the object. Second, a method is provided to access (but not change) this value (the `intValue` method on line 3). Third, the `gcd` method is defined in lines 4-11, but with only *one* integer parameter (since the first parameter is implicitly the value of the object on which `gcd` will be called). Finally, the integer `value` is defined on line 14 (this is called a *field* in Java). Note also that the constructor and methods are defined to have `public` access on lines 2, 3, and 4 (so they can be called by users), while the data field on line 14 is defined to be `private` (and thus inaccessible to the "outside world"). This is a feature of Java classes that relate to the encapsulation and information hiding properties of unit abstractions discussed in the previous section.

The `gcd` class can be used by defining a variable name to hold an object of the class as follows:

```
IntWithGcd x;
```

At first there is no actual object contained in `x`[4]; we must create, or **instantiate,** the object with the following statement:

```
x = new IntWithGcd(8);
```

Then we can ask `x` to provide the greatest common divisor of its own value and that of another integer by calling the `gcd` method (using a familiar *dot notation*):

```
int y = x.gcd(18); // y is now 2, the gcd of 8 & 18
```

Internally, the `gcd` method works essentially the same as the Ada procedure or function of Figures 1--1 and 1-2 of the previous section. The main difference is that the data object -- in this case the one contained in `x` -- is emphasized by placing it first in the call: `x.gcd(18)` instead of `gcd(x,18)`, and then giving `gcd` only one parameter instead of two (since the data in the first parameter of the imperative version of `gcd` is now already stored in the object contained in `x`).

We examine object-oriented languages in more detail in Chapter 10, including Java, C++, and Smalltalk.

Functional programming. The functional paradigm bases the description of computation on the evaluation of functions or the application of functions to known values. For this reason, functional languages are sometimes called **applicative** languages. A functional programming language has as its basic mechanism the evaluation of a function, or the **function call.** This involves, besides the actual evaluation of functions, the passing of values as parameters to functions and the obtaining of the resultant values as **returned values** from functions. The functional paradigm involves no notion of variable or assignment to variables. In a sense, functional programming is the opposite of object-oriented programming: it concentrates on values and functions rather than memory locations. Also, repetitive operations are not expressed by loops (which require control variables to terminate) but by recursive functions. Indeed the study of **recursive function theory** in mathematics has established the following property:

> A programming language is Turing complete if it has integer values, arithmetic functions on those values, and if it has a mechanism for defining new functions using existing functions, selection, and recursion.

[4] To be precise, x does not actually contain an object, but a reference to an object. This is studied in detail in Chapters 5 and 10.

It may seem surprising that a programming language can completely do away with variables and loops, but that is exactly what the functional paradigm does, and there are advantages to doing so. We have already stated two: that the language becomes more independent of the machine model, and that, because functional programs resemble mathematics, it is easier to draw precise conclusions about their behavior. Exactly how this is possible is left to later chapters. We content ourselves here with one example of functional programming.

Returning to the Ada procedure to compute the greatest common divisor of two integers that we gave in the last section, a functional version of this procedure is given in Figure 1.4.

Figure 1.4 A functional version of the **gcd** function in Ada.

```
(1)   function gcd (u, v: in integer) return integer is
(2)   begin
(3)     if v = 0 then
(4)       return u;
(5)     else
(6)       return gcd(v, u mod v);
(7)     end if;
(8)   end gcd;
```

Note that this code does not use any local variables or loops, but does use recursion (it calls itself with a different set of parameters on line 6).

In an even more functionally oriented programming language, such as LISP, this function would be written as in Figure 1.5 (here and throughout the book we use the Scheme dialect of LISP).

Figure 1.5 A **gcd** function in the Scheme dialect of LISP.

```
(1) (define (gcd u v)
(2)  (if (= v 0) u
(3)      (gcd v (modulo u v))))
```

A few comments about this Scheme code may be worthwhile.

In LISP, programs are list expressions, that is, sequences of things separated by spaces and surrounded by parentheses, as in `(+ 2 3)`. Programs are run by evaluating them as expressions, and expressions are evaluated by applying the first item in a list, which must be a function, to the rest of the items as arguments. Thus `(gcd 8 18)` applies the **gcd** function to arguments 8 and 18. Similarly 2 + 3 is written `(+ 2 3)`, which applies the **+** function to the values 2 and 3.

In the definition of the **gcd** function we have used the if-then-else function (lines 2 and 3 in Figure 1.5), which is just called "if"—the "then" and "else" are dispensed with. Thus (`if a b c`) means "if *a* then *b* else *c*." Note that the "if" function represents control as well as the computation of a value: first **a** is evaluated and, depending on the result, either **b** or **c** is evaluated, with the resulting value becoming the returned value of the function. (This differs from the "if" statement of C or Ada, which does not have a value.)

Finally, LISP does not require a return-statement to indicate the value returned by a function. Simply stating the value itself implies that it is returned by the function.

An even more succinct version of this function can be written in the functional language Haskell:

```
gcd u v = if v == 0 then u else gcd v (u `mod` v)
```

Note the minimal use of parentheses (no parentheses are needed for parameters) and the use of backquotes (`mod`) to allow a function name to be used as an infix operator (written between its arguments).

Chapter 11 examines functional programming in detail, including Scheme and Haskell.

Logic programming. This language paradigm is based on symbolic logic. In a logic programming language, a program consists of a set of statements that describe what is true about a desired result, as opposed to giving a particular sequence of statements that must be executed in a fixed order to produce the result. A pure logic programming language has no need for control abstractions such as loops or selection. Control is supplied by the underlying system. All that is needed in a logic program is the statement of the properties of the computation. For this reason, logic programming is sometimes called **declarative programming**[5], since properties are declared, but no execution sequence is specified. (Since there is such a removal from the details of machine execution, these languages are sometimes also referred to as **very-high-level languages**.)

In our running example of the greatest common divisor, we can state the properties of gcd in a form similar to that of a logic program as follows:

The gcd of *u* and *v* is *u* if *v* = 0.
The gcd of *u* and *v* is the same as the gcd of *v* and *u* mod *v* if *v* is not = 0.

A number of logic programming languages have been developed, but only one has become widely used: Prolog. The gcd statements given translate into Prolog as in Figure 1.6[6].

Figure 1.6 Prolog clauses defining a **`gcd`** function.

```
(1)   gcd(U, V, U) :- V = 0.
(2)   gcd(U, V, X) :- not (V = 0),
(3)                   Y is U mod V,
(4)                   gcd (V, Y, X).
```

In Prolog, the form of a program is a sequence of statements, called **clauses,** which are of the form

```
a :- b, c, d
```

Such a clause roughly corresponds to the assertion that *a* is true if *b* and *c* and *d* are true. Unlike functional programming (and more like imperative programming), Prolog requires values to be represented by variables. However, variables do not represent memory locations as they do in imperative programming, but behave more as names for the results of partial computations, as they do in mathematics.

In the Prolog program, **`gcd`** has three parameters instead of two: the third represents the computed value (much as if **`gcd`** were a procedure), since **`gcd`** itself can only be true or false (that is, it can only succeed or fail). Note also the use of uppercase names for variables. This is a requirement in Prolog, where variables must be syntactically distinct from other language elements.

The first of the two clauses for **`gcd`** in (Figure 1.6, line 1) states that the **`gcd`** of **`U`** and **`V`** is **`U`**, provided **`V`** is equal to 0. The second clause (lines 2, 3, and 4) states that the **`gcd`** of **`U`** and **`V`** is **`X`**, provided **`V`** is not equal to 0 (line 2), and that **`X`** is the result of the **`gcd`** of **`V`** and **`Y`** (line 4), where **`Y`** is equal to **`U mod V`** (line 3). (The "is" clause for **`Y`** in line 3 is somewhat like assignment in an ordinary programming language and gives **`Y`** the value of **`U mod V`**.)

Details of Prolog and logic programming are treated in Chapter 12.

It needs to be stressed that, even though a programming language may exhibit most or all of the properties of one of the four paradigms just discussed, few languages adhere purely to one paradigm,

[5] Functional programming is sometimes also referred to as "declarative," for similar reasons.
[6] A Prolog programmer would actually write this program in a somewhat more efficient form; see Chapter 12.

but usually contain features of several paradigms. Indeed, as we saw, we were able to write a functional version of the `gcd` function in Ada, a language that is considered to be more of an imperative language. Nevertheless, it and most other modern imperative languages permit the definition of recursive functions, a mechanism that is generally considered to be functional. Similarly, the Scheme dialect of LISP, which is considered to be a functional language, does permit variables to be declared and assigned to, which is definitely an imperative feature. Scheme programs can also be written in an object-oriented style that closely approximates the object-oriented paradigm. Thus we can refer to a programming **style** as following one (or more) of the paradigms. In a language that permits the expression of several different paradigms, which one is used depends on the kind of computation desired and the requirements of the development environment.

1.4 *LANGUAGE DEFINITION*

A programming language needs a complete, precise description. As obvious as that sounds, in the past many programming languages began with only informal English descriptions. Even today most languages are defined by a **reference manual** in English, although the language in such manuals has become increasingly formalized. Such manuals will always be needed, but there has been increasing acceptance of the need for programming languages to have definitions that are formally precise. Such definitions have, in a few cases, been completely given, but currently it is customary to give a formal definition only of parts of a programming language.

The importance of a precise definition for a programming language should be clear from its use to describe computation. Without a clear notion of the effect of language constructs, we have no clear idea of what computation is actually being performed. Moreover, it should be possible to reason mathematically about programs, and to do this requires formal verification or proof of the behavior of a program. Without a formal definition this is impossible.

But there are other compelling reasons for the need for a formal definition. We have already mentioned the need for machine or implementation independence. The best way to achieve this is through standardization, which requires an independent and precise language definition that is universally accepted. Standards organizations such as ANSI (American National Standards Institute) and ISO (International Organization for Standardization) have published definitions for many languages, including Pascal, FORTRAN, C, C++, Ada, and Prolog.

A further reason for a formal definition is that, inevitably in the programming process, difficult questions arise about program behavior and interaction. Programmers need an adequate reference to answer such questions besides the often-used trial-and-error process: it can happen that such questions need to be answered already at the design stage and may result in major design changes.

Finally, the requirements of a formal definition provide discipline during the design of a language. Often a language designer will not realize the consequences of design decisions until he or she is required to produce a clear definition.

Language definition can be loosely divided into two parts: **syntax,** or structure, and **semantics,** or meaning. We discuss each of these categories in turn.

Language syntax. The syntax of a programming language is in many ways like the grammar of a natural language. It is the description of the ways different parts of the language may be combined to form other parts. As an example, the syntax of the if-statement in C may be described in words as follows:

> An if-statement consists of the word "if" followed by an expression inside parentheses, followed by a statement, followed by an optional else part consisting of the word "else" and another statement.

The description of language syntax is one of the areas where formal definitions have gained acceptance, and the syntax of almost all languages is now given using **context-free grammars.** For example, a context-free grammar rule for the C if-statement can be written as follows:

<if-statement> ::= if (<expression>) <statement>
[else <statement>]

or (using special characters and formatting):

if-statement → **`if (`** *expression* **`)`** *statement*
[**`else`** *statement*]

An issue closely related to the syntax of a programming language is its **lexical structure.** This is similar to spelling in a natural language. The lexical structure of a programming language is the structure of the words of the language, which are usually called **tokens.** In the example of a C if-statement, the words "if" and "else" are tokens. Other tokens in programming languages include identifiers (or names), symbols for operations, such as " + " and "<=," and special punctuation symbols such as the semicolon (";") and the period (".").

In this book we shall consider syntax and lexical structure together, and a more detailed study is found in Chapter 4.

Language semantics. Syntax represents only the surface structure of a language and thus is only a small part of a language definition. The semantics, or meaning, of a language is much more complex and difficult to describe precisely. The first difficulty is that "meaning" can be defined in many different ways; typically describing the meaning of a piece of code involves some kind of description of the effects of executing it, and there is no standard way to do this. Moreover, the meaning of a particular mechanism may involve interactions with other mechanisms in the language, so that a comprehensive description of its meaning in all contexts may become extremely complex.

To continue with our example of the C if-statement, its semantics may be described in words as follows (adapted from Kernighan and Richie [1988]):

> An if-statement is executed by first evaluating its expression, which must have arithmetic or pointer type, including all side effects, and if it compares unequal to 0, the statement following the expression is executed. If there is an else part, and the expression is 0, the statement following the "else" is executed.

This description in itself points out some of the difficulty in specifying semantics, even for a simple mechanism such as the if-statement. The description makes no mention of what happens if the condition evaluates to 0, but there is no else part (presumably nothing happens; that is, the program continues at the point after the if-statement). Another important question is whether the if-statement is "safe" in the sense that there are no other language mechanisms that may permit the statements inside an if-statement to be executed without the corresponding evaluation of the if expression. If so, then the if-statement provides adequate protection from errors during execution, such as division by zero:

```
if (x != 0) y = 1 / x;
```

Otherwise, additional protection mechanisms may be necessary (or at least the programmer must be aware of the possibility of circumventing the if expression).

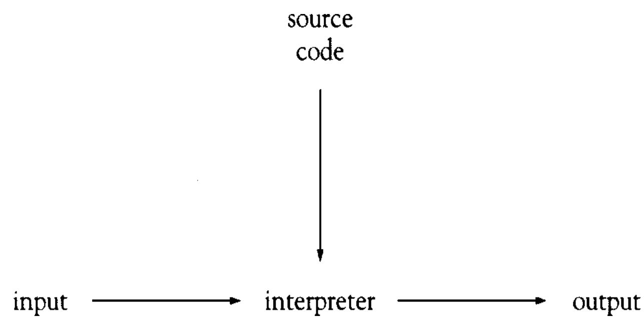
The alternative to this informal description of semantics is to use a formal method. However, no generally accepted method, analogous to the use of context-free grammars for syntax, exists here either. Indeed, it is still not customary for a formal definition of the semantics of a programming language to be given at all. Nevertheless, several notational systems for formal definitions have been developed and are increasingly in use. These include **operational semantics**, **denotational semantics**, and **axiomatic semantics.**

Language semantics are implicit in many of the chapters of this book, but semantic issues are more specifically addressed in Chapters 5 and 9. Chapter 13 discusses formal methods of semantic definition, including operational, denotational, and axiomatic semantics.

1.5 *LANGUAGE TRANSLATION*

For a programming language to be useful, it must have a **translator,** that is, a program that accepts other programs written in the language in question and that either executes them directly or transforms them into a form suitable for execution. A translator that executes a program directly is called an **interpreter,** while a translator that produces an equivalent program in a form suitable for execution is called a **compiler.**

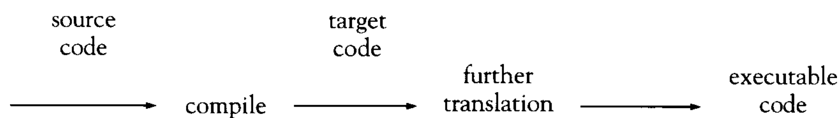
Interpretation is a one-step process, in which both the program and the input are provided to the interpreter, and the output is obtained:



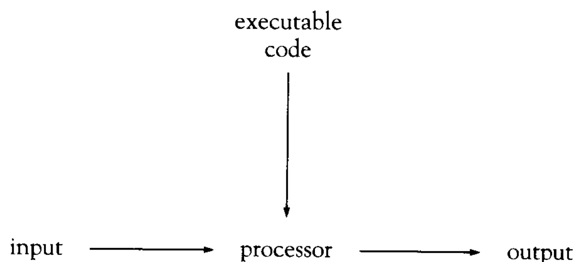
An interpreter can be viewed as a simulator for a machine whose "machine language" is the language being translated.

Compilation, on the other hand, is at least a two-step process: the original program (or **source program)** is input to the compiler, and a new program (or **target program)** is output from the compiler. This target program may then be executed, if it is in a form suitable for direct execution (i.e., in machine language). More commonly, the target language is assembly language, and the target program must be translated by an **assembler** into an object program, and then **linked** with other object programs, and **loaded** into appropriate memory locations before it can be executed. Sometimes the target language is even another programming language, in which case a compiler for that language must be used to obtain an executable object program.

The compilation process can be visualized as follows:



and



It is also possible to have translators that are intermediate between interpreters and compilers: a translator may translate a source program into an intermediate language and then interpret the intermediate language. Such translators could be called **pseudointerpreters,** since they execute the program without producing a target program, but they process the entire source program before execution begins.

It is important to keep in mind that a language is different from a particular translator for that language. It is possible for a language to be defined by the behavior of a particular interpreter or compiler (a so-called **definitional** translator), but this is not common (and even problematic, in view of the need for a formal definition discussed in the last section). More often, a language definition exists independently, and a translator may or may not adhere closely to the language definition (one hopes the former). When writing programs one must always be aware of those features and properties that depend on a specific translator and are not part of the language definition. There are significant advantages to be gained from avoiding nonstandard features as much as possible.

Both compilers and interpreters must perform similar operations when translating a source program. First, a **lexical analyzer,** or **scanner,** must convert the textual representation of the program as a sequence of characters into a form that is easier to process, typically by grouping characters into **tokens** representing basic language entities, such as keywords, identifiers, and constants. Then a **syntax analyzer** or **parser** must determine the structure of the sequence of tokens provided to it by the scanner. Finally, a **semantic analyzer** must determine enough of the meaning of a program to allow execution or generation of a target program to take place. Typically, these phases of translation do not occur separately but are combined in various ways. A language translator must also maintain a **runtime environment,** in which suitable memory space for program data is allocated, and that records the progress of the execution of the program. An interpreter usually maintains the runtime environment internally as a part of its management of the execution of the program, while a compiler must maintain the runtime environment indirectly by adding suitable operations to the target code. Finally, a language may also require a **preprocessor,** which is run prior to translation to transform a program into a form suitable for translation.

The properties of a programming language that can be determined prior to execution are called **static** properties, while properties that can be determined only during execution are called **dynamic** properties. This distinction is not very useful for interpreters, but it is for compilers: a compiler can only make use of a language's static properties. Typical static properties of a language are its lexical and syntactic structure. In some languages, such as C and Ada, important semantic properties are also static: data types of variables are a significant example. (See Chapter 6.)

A programming language can be designed to be more suitable for interpretation or compilation. For instance, a language that is more dynamic, that is, has fewer static properties, is more suitable for interpretation and is more likely to be interpreted. On the other hand, a language with a strong static structure is more likely to be compiled. Historically, imperative languages have had more static properties and have been compiled, while functional and logic programming languages have been more dynamic and have been interpreted. Of course, a compiler or interpreter can exist for any language, regardless of its dynamic or static properties.

The static and dynamic properties of a language can also affect the nature of the runtime environment. In a language with **static allocation** only—all variables are assumed to occupy a fixed position in memory for the duration of the program's execution—a **fully static** environment may be used. For more dynamically oriented languages, a more complex **fully dynamic** environment must be used. Midway between these is the typical **stack-based** environment of languages like C and Ada, which has both static and dynamic aspects. (Chapter 8 describes these in more detail.)

Efficiency may also be an issue in determining whether a language is more likely to be interpreted or compiled. Interpreters are inherently less efficient than compilers, since they must simulate the actions of the source program on the underlying machine. Compilers can also boost the efficiency of the target code by performing code improvements, or **optimizations,** often by making several **passes** over the source program to analyze its behavior in detail. Thus, a programming language that needs efficient execution is likely to be compiled rather than interpreted.

Situations may also exist when an interpreter may be preferred over a compiler. Interpreters usually have an interactive mode, so that the user can enter programs directly from a terminal and also supply input to the program and receive output using the interpreter alone. For example, a Scheme interpreter can be used to provide immediate input to a procedure as follows:

```
> (gcd 8 18) ;; calls gcd with the values 8 and 18
```

```
2           ;; the interpreter prints the returned value
```

By contrast, in a compiled language, such as C, Ada, or Java[7], the programmer must write out by hand the interactive input and output. For example, we provide complete runnable code for the gcd example in these three languages in Figures 1-7, 1-8, and 1-9, respectively[8]. Note in particular the code overhead to get line-oriented input in Java (window-oriented input is actually easier, but not covered here). This overhead, plus the lack of the compilation step, makes an interpreter more suitable than a compiler in some cases for instruction and for program development. By contrast, a language can also be designed to allow **one-pass** compilation, so that the compilation step is efficient enough for instructional use (C has this property).

An important property of a language translator is its response to errors in a source program. Ideally, a translator should attempt to correct errors, but this can be extremely difficult. Failing that, a translator should issue appropriate error messages. It is generally not enough to issue only one error message when the first error is encountered, though some translators do this for efficiency and simplicity. More appropriate is **error recovery,** which allows the translator to proceed with the translation, so that further errors may be discovered.

Figure 1.7 A Complete C Program with a `gcd` function

```
#include <stdio.h>

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

main()
{  int x, y;
   printf("Input two integers:\n");
   scanf("%d%d",&x,&y);
   printf("The gcd of %d and %d is %d\n",x,y,gcd(x,y));
   return 0;
}
```

Figure 1.8 A Complete Ada Program with a `gcd` function

```
with Text_IO; use Text_IO;
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;

procedure gcd_prog is

  function gcd (u, v: in integer) return integer is
  begin
    if v = 0 then
      return u;
    else
      return gcd(v, u mod v);
    end if;
  end gcd;
```

[7] Java code is compiled to a machine-independent set of instruction codes called **bytecode** (because each instruction code occupies a single byte). The bytecode is then interpreted by the **Java Virtual Machine** (JVM). This allows compiled Java code to be run on any machine that has an available JVM.
[8] We do not explain this code here, but it can be used with very little modification to run most of the examples from this book in these languages.

```
    x: Integer;
    y: Integer;

  begin
    put_line("Input two integers:");
    get(x);
    get(y);
    put("The gcd of ");
    put(x);
    put(" and ");
    put(y);
    put(" is ");
    put(gcd(x,y));
    new_line;
  end gcd_prog;
```

Figure 1.9 A Complete Java Program with an **`IntWithGcd`** class

```
import java.io.*;

class IntWithGcd
{ public IntWithGcd( int val ) { value = val; }
  public int getValue() { return value; }
  public int gcd ( int v )
  { int z = value;
    int y = v;
    while ( y != 0 )
    { int t = y;
      y = z % y;
      z = t;
    }
    return z;
  }
  private int value;
}

class GcdProg
{ public static void main (String args[])
  { System.out.println("Input two integers:");
    BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));
    try
    { IntWithGcd x = new IntWithGcd(Integer.parseInt(in.readLine()));
      int y = Integer.parseInt(in.readLine());
      System.out.print("The gcd of " + x.getValue()
                        + " and " + y + " is ");
      System.out.println(x.gcd(y));
    }
    catch ( Exception e)
    { System.out.println(e);
      System.exit(1);
    }
  }
}
```

Errors may be classified according to the stage in translation at which they occur. Lexical errors occur during lexical analysis; these are generally limited to the use of illegal characters. An example in C is

```
x@ = 2
```

The character `@` is not a legal character in the language.

Misspellings such as "whille" for "while" are often not caught by a lexical analyzer, since it will assume that an unknown character string is an identifier, such as the name of a variable. Such an error will be caught by the parser, however. Syntax errors include missing tokens and malformed expressions, such as

```
if (scaleFactor != 0   /* missing right parenthesis */
```

or

```
adjustment = base + * scaleFactor
```

```
/* missing operand between + and * */
```

Semantic errors can be either static (i.e., found prior to execution), such as incompatible types or undeclared variables, or dynamic (found during execution), such as an out-of-range subscript or division by zero.

A further class of errors that may occur in a program are **logic** errors. These are errors that the programmer makes that cause the program to behave in an erroneous or undesirable way. For example, the following C fragment

```
x = u;
y = v;
while (y != 0)
{ t = y;
  y = x * y;
  x = t;
}
```

will cause an infinite loop during execution if `u` and `v` are 1. However, the fragment breaks no rules of the language and must be considered semantically correct from the language viewpoint, even though it does not do what was intended. Thus, logic errors are not errors at all from the point of view of language translation[9].

A language definition will often include a specification of what errors must be caught prior to execution (for compiled languages), what errors must generate a runtime error, and what errors may go undetected. The precise behavior of a translator in the presence of errors is usually unspecified, however.

Finally, a translator needs to provide user options for debugging, for interfacing with the operating system, and perhaps with a software development environment. These options, such as specifying files for inclusion, disabling optimizations, or turning on tracing or debugging information, are the **pragmatics** of a programming language translator. Occasionally, facilities for pragmatic directives, or **pragmas,** are part of the language definition. For example, in Ada the declaration

```
pragma LIST(ON);
```

turns on the generation of a listing by a compiler at the point it is encountered, and

```
pragma LIST(OFF);
```

turns listing generation off again.

1.6 *LANGUAGE DESIGN*

We have spoken of a programming language as a tool for describing computation; we have indicated that differing views of computation can result in widely differing languages but that machine and human **readability** are overriding requirements. It is the challenge of programming language design to achieve the power, expressiveness, and comprehensibility that human readability requires while at the same time retaining the precision and simplicity that is needed for machine translation.

Human readability is a complex and subtle requirement. It depends to a large extent on the facilities a programming language has for abstraction. A. N. Whitehead emphasized the power of abstract notation in 1911: "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems. . . . Civilization advances by extending the number of important operations which we can perform without thinking about them."

---

[9] Syntax and semantic errors could be described as inconsistencies between a program and the specification of the language, while logic errors are inconsistencies between a program and its own specification.

A successful programming language has facilities for the natural expression of the structure of data **(data abstraction)** and for the structure of the computational process for the solution of a problem **(control abstraction).** A good example of the effect of abstraction is the introduction of recursion into the programming language Algol60. C. A. R. Hoare, in his 1980 Turing Award Lecture, describes the effect his attendance at an Algol60 course had on him: "It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded."

The overriding goal of abstraction in programming language design is **complexity control.** A human being can retain only a certain amount of detail at once. To understand and construct complex systems, humans must control how much detail needs to be understood at any one time.

Abelson and Sussman, in their book *Structure and Interpretation of Computer Programs* [1985], have emphasized the importance of complexity control as follows: "We control complexity by building abstractions that hide details when appropriate. We control complexity by establishing conventional interfaces that enable us to construct systems by combining standard, well-understood pieces in a 'mix and match' way. We control complexity by establishing new languages for describing a design, each of which emphasizes particular aspects of the design and deemphasizes others."

In Chapter 3 we study additional language design issues that help to promote readability and complexity control.

Exercises

1.1. The following is a C function that computes the number of (decimal) digits in an integer:

```
int numdigits(int x)
{ int t = x, n = 1;
  while (t >= 10)
  { n++;
    t = t / 10;
  }
  return n;
}
```

Rewrite this function in functional style.

1.2. Write a **numdigits** function (as in the previous exercise) in any of the following languages (or in any language for which you have a translator): (a) Pascal, (b) Scheme, (c) Haskell, (d) Prolog, (e) Ada, (f) FORTRAN, (g) Java, and (h) BASIC.

1.3. Rewrite the **numdigits** function of Exercise 1.1 so that it will compute the number of digits to any base (such as base 2, base 16, base 8). You may do this exercise for any of the following languages or any other language for which you have a translator: (a) C, (b) Pascal, (c) Ada, (d) Java, (e) FORTRAN, (f) Scheme, (g) Haskell, (h) Prolog, and (i) BASIC.

1.4 Write a program similar to those in Figures 1.7, 1.8, or 1.9 that tests your **numdigits** function from Exercises 1.1, 1.2, or 1.3 in C, Ada, Java, or C++.

1.5. The C and Java versions of the greatest common divisor calculation in this chapter have used the remainder operation **%**, while the Ada, Scheme, Haskell, and Prolog versions have used the modulo operation.

(a) What is the difference between remainder and modulo?
(b) Can this difference lead to any differences in the result of a gcd computation?

1.6. The **numdigits** function of Exercise 1.1 will not work for negative integers. Rewrite it so that it will.

1.7. The following C function computes the factorial of an integer:

```
int fact (int n)
{ if (n <= 1) return 1;
  else return n * fact (n - 1);
}
```

Rewrite this function into imperative style (i.e., using variables and eliminating recursion).

1.8. Write a factorial function in any of the following languages (or in any language for which you have a translator): (a) Pascal, (b) Scheme, (c) Prolog, (d) Java, (e) Ada, (f) FORTRAN, and (g) BASIC.

1.9 Write a program similar to those in Figures 1.7, 1.8, or 1.9 that tests your factorial function from Exercises 1.8 or 1.9 in C, Ada, Java, or C++.

1.10. Factorials grow extremely rapidly, and overflow is soon reached in the factorial function of Exercise 1.7. What happens during execution when overflow occurs? How easy is it to correct this problem in C? In any of the languages you have used in Exercise 1.8?

1.11. For any of the languages of Exercise 1.8, find where (if anywhere) it is specified in your translator manual what happens on integer overflow. Compare this, if possible, to the requirements of the language standard.

1.12. Rewrite the **IntWithGcd** Java class of Section 1.3 to use recursion in the computation of the greatest common divisor.

1.13 The **IntWithGcd** Java class of Section 1.3 has a possible design flaw in that the **gcd** method takes an integer parameter rather than another object of class **IntWithGcd**. Rewrite the **IntWithGcd** class to correct this. Discuss whether you think this new design is better or worse than the one given in the text.

1.14. For any of the following languages, determine if strings are part of the language definition and whether your translator offers string facilities that are not part of the language definition: (a) C, (b) Pascal, (c) Ada, (d) C++, (e) Java, (f) Scheme, (g) Haskell, (h) FORTRAN, and (i) BASIC.

1.15. Add explicit interactive input and output to the Scheme **gcd** function of Section 1.3 (that is, make it into a "compiler-ready" program in a similar manner to the programs in Figures 1.7, 1.8, and 1.9).

1.16. Add explicit interactive input and output to the Prolog program for **gcd** in Section 1.3.

1.17. The following C program differs from Figure 1.7 in that it contains a number of errors. Classify each error as to whether it is lexical, syntactic, static semantic, dynamic semantic, or logical:

```
(1)   #include <stdio.h>

(2)   int gcd(int u#, double v);
(3)   { if (v = 0) return 0;
(4)     else return gcd (v, u# % v);
```

```
(5)   }

(6)   main()
(7)   { int x, y;
(8)     printf("Input two integers: ");
(9)     scanf("%d%d",&x,&y);
(10)    printf("The gcd of %d and %d is %d\n",x,y,Gcd(x,y));
(11)    return;
(12)  }
```

1.18. Describe the syntax of the do-while statement in C.

1.19. Describe the semantics of the do-while statement in C.

1.20. Is it possible in any of the following languages to execute the statements inside an if-statement without evaluating the expression of the if: (a) C, (b) Pascal, (c) FORTRAN, (d) Ada, and (e) Java? Why or why not?

1.21. What are the reasons for the inclusion of many different kinds of loop statements in a programming language? (Address your answer in particular to the need for the while-, do-while-, and for-statements in C and Java, or the while-, loop-, and for-statements in Ada.)

1.22. Given the following properties of a variable in C (or Java or Ada), state which are static and which are dynamic, and why: (a) its value, (b) its data type, and (c) its name.

1.23. Given the following properties of a variable in Scheme, state which are static and which are dynamic, and why: (a) its value, (b) its data type, and (c) its name.

1.24. Prove that a language is Turing complete if it contains integer variables, integer arithmetic, assignment, and while-statements. (Hint: Use the characterization of Turing completeness stated at the end of Section 1.2, and eliminate the need for if-statements.)

1.25. Pick one of the following statements and argue both for and against it:
(a) A programming language is solely a mathematical notation for describing computation.
(b) A programming language is solely a tool for getting computers to perform complex tasks.
(c) A programming language should make it easy for programmers to write code quickly and easily.
(d) A programming language should make it easy for programmers to read and understand code with a minimum of comments.

1.26. Since most languages can be used to express any algorithm, why should it matter which programming language we use to solve a programming problem? (Try arguing both that it should and that it shouldn't matter.)

1.27. Java and C++ are both considered object-oriented languages. To what extent is it possible to write imperative-style code in either of these languages? Functional-style code?

1.28. Stroustrup [1997] contains the following statement (page 9): "A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done." Compare and contrast this statement to the definition of a programming language given in this chapter.

Notes and References

An early description of the von Neumann architecture and the use of a program stored as data to control the execution of a computer is in Burks, Goldstine, and von Neumann [1947]. A similar view of the definition of a programming language we have used is given in Horowitz [1984]. Human readability is discussed in Ghezzi and Jazayeri [1997], but with more emphasis on software engineering. Attempts have been made to study readability from a psychological, or cognitive, perspective, where it is called **comprehension**. See Rosson [1997] for a survey of research in this direction.

References for the major programming languages used as examples in this text are as follows. A reference for the C programming language is Kernighan and Ritchie [1988]. C++ is described in Stroustrup [1994] [1997], and Ellis and Stroustrup [1990]; an introductory text is Lippman and Lajoie [1998]; the international standard for C++ is ISO 14882-1 [1998]. Java is described in many books, including Horstmann and Cornell [1999], Arnold, Gosling, and Holmes [2000], and Flanagan [1999]; the Java language specification is given in Gosling, Joy, Steele and Bracha [2000]. Ada exists in two versions. The original is sometimes called Ada83, and is described by its reference manual (ANSI-1815A [1983]); a newer version is Ada95[10], and is described by its international standard (ISO 8652 [1995]). Standard texts for Ada include Cohen [1996], Barnes [1998] and Feldman and Koffman [1999]. Pascal is described in Cooper [1983]. FORTRAN also exists in several versions: Fortran77, Fortran 90 and Fortran 95. An introductory text that covers all three is Chapman [1997]; a more advanced reference is Metcalf and Reid [1999]. Scheme is described in Dybvig [1996] and Abelson and Sussman [1996]; a language definition can be found in Abelson et al. [1998]. Haskell is covered in Hudak [2000] and Thompson [1999]. The ML functional language (related to Haskell) is covered in Paulson [1996] and Ullman [1997]. The standard reference for Prolog is Clocksin and Mellish [1994]. The logic paradigm is discussed in Kowalski [1979], and the functional paradigm in Backus [1978] and Hudak [1989]. Smalltalk is presented in Lewis [1995] and Sharp [1997].

The Turing completeness property for imperative languages stated on page ?? is proved in Böhm and Jacopini [1966]. The Turing completeness result for functional languages on page ?? can be extracted from results on recursive function theory in such texts as Sipser [1997] and Hopcroft and Ullman [1979]. Language translation techniques are described in Louden [1997] and Aho, Sethi, and Ullman [1986]. The quote from A. N. Whitehead in Section 1.6 is in Whitehead [1911], and Hoare's Turing Award Lecture quote is in Hoare [1981].

[10] Since Ada95 is an extension of Ada83, we will indicate only those features that are specifically Ada95 when they are not part of Ada83.