1

Preliminaries

1.1 Reasons for Studying Concepts of Programming Languages
1.2 Programming Domains
1.3 Language Evaluation Criteria
1.4 Influences on Language Design
1.5 Language Categories
1.6 Language Design Trade-Offs
1.7 Implementation Methods
1.8 Programming Environments

Before we begin discussing the concepts of programming languages, we must consider a few preliminaries. First, we explain some reasons why computer science students and professional software developers should study general approaches to language design and evaluation. This discussion is especially valuable for those who believe that a working knowledge of one or two programming languages is sufficient for computer scientists. Then, we briefly describe the major programming domains. Next, because the book evaluates language constructs and features, we present a list of criteria that can serve as a basis for such judgments. Then, we discuss the two major influences on language design: machine architecture and program design methodologies. After that, we introduce the major categories of programming languages. Next, we describe a few of the most important trade-offs that must be considered during language design.

Because this book is also about the implementation of programming languages, this chapter includes an overview of the most common general approaches to implementation. Finally, we briefly describe a few examples of programming environments and discuss their impact on software production.

1.1 Reasons for Studying Concepts of Programming Languages

It is natural for students to wonder how they will benefit from the study of programming language concepts. After all, many other topics in computer science are worthy of serious study. In fact, many now believe that there are more important areas of computing for study than can be covered in a four-year college curriculum. The following is what we believe to be a compelling list of potential benefits of studying concepts of programming languages:

- *Increased capacity to express ideas.* It is widely believed that the depth at which people can think is influenced by the expressive power of the language in which they communicate their thoughts. Those with only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction. In other words, it is difficult for people to conceptualize structures they cannot describe, verbally or in writing.

  Programmers, in the process of developing software, are similarly constrained. The language in which they develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited. Awareness of a wider variety of programming language features can reduce such limitations in software development. Programmers can increase the range of their software development thought processes by learning new language constructs.

  It might be argued that learning the capabilities of other languages does not help a programmer who is forced to use a language that lacks those capabilities. That argument does not hold up, however, because often, language constructs can be simulated in other languages that do not support those constructs directly. For example, a C (Harbison and Steele, 2002) programmer who had learned the structure and uses of associative

arrays in Perl (Christianson et al., 2013) might design structures that simulate associative arrays in that language. In other words, the study of programming language concepts builds an appreciation for valuable language features and constructs and encourages programmers to use them, even when the language they are using does not directly support such features and constructs.

- *Improved background for choosing appropriate languages.* Some professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through in-house training programs. Such training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization. Other programmers received their formal training years ago. The languages they learned then are no longer widely used, and many features now available in programming languages were not commonly known at the time. The result is that many programmers, when given a choice of languages for a new project, use the language with which they are most familiar, even if it is poorly suited for the project at hand. If these programmers were familiar with a wider range of languages and language constructs, they would be better able to choose the language with the features that best address the problem.

  Some of the features of one language often can be simulated in another language. However, it is preferable to use a feature whose design has been integrated into a language than to use a simulation of that feature, which is often less elegant, more cumbersome, and less safe.

- *Increased ability to learn new languages.* Computer programming is still a relatively young discipline, and design methodologies, software development tools, and programming languages are still in a state of continuous evolution. This makes software development an exciting profession, but it also means that continuous learning is essential. The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general. Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned. For example, programmers who understand the concepts of object-oriented programming will have a much easier time learning Ruby (Thomas et al., 2013) than those who have never used those concepts.

  The same phenomenon occurs in natural languages. The better you know the grammar of your native language, the easier it is to learn a second language. Furthermore, learning a second language has the benefit of teaching you more about your first language.

  The TIOBE Programming Community issues an index (`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm`) that is an indicator of the relative popularity of programming languages. For example, according to the index, Java, C, C++ (Lippman et al., 2012), and

C# (Albahari and Abrahari, 2012) were the four most popular languages in use in February 2017.[1] However, dozens of other languages were widely used at the time. The index data also show that the distribution of usage of programming languages is always changing. The number of languages in use and the dynamic nature of the statistics imply that every software developer must be prepared to learn different languages.

Finally, it is essential that practicing programmers know the vocabulary and fundamental concepts of programming languages so they can read and understand programming language descriptions and evaluations, as well as promotional literature for languages and compilers. These are the sources of information needed in order to choose and learn a language.

- *Better understanding of the significance of implementation.* In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts. In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. In turn, this knowledge leads to the ability to use a language more intelligently, as it was designed to be used. We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.

  Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details. Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs. In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program. For example, programmers who know little about the complexity of the implementation of subprogram calls often do not realize that a small subprogram that is frequently called can be a highly inefficient design choice.

  Because this book touches on only a few of the issues of implementation, the previous two paragraphs also serve well as rationale for studying compiler design.
- *Better use of languages that are already known.* Most contemporary programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses. By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.
- *Overall advancement of computing.* Finally, there is a global view of computing that can justify the study of programming language concepts. Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular

1. Note that this index is only one measure of the popularity of programming languages, and its accuracy is not universally accepted.

languages are not always the best available. In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.

For example, many people believe it would have been better if ALGOL 60 (Backus et al., 1963) had displaced Fortran (ISO/IEC 1539-1, 2010) in the early 1960s, because it was more elegant and had much better control statements, among other reasons. That it did not, is due partly to the programmers and software development managers of that time, many of whom did not clearly understand the conceptual design of ALGOL 60. They found its description difficult to read (which it was) and even more difficult to understand. They did not appreciate the benefits of block structure, recursion, and well-structured control statements, so they failed to see the benefits of ALGOL 60 over Fortran.

Of course, many other factors contributed to the lack of acceptance of ALGOL 60, as we will see in Chapter 2. However, the fact that computer users were generally unaware of the benefits of the language played a significant role.

In general, if those who choose languages were well informed, perhaps better languages would eventually squeeze out poorer ones.

1.2 Programming Domains

Computers have been applied to a myriad of different areas, from controlling nuclear power plants to providing video games in mobile phones. Because of this great diversity in computer use, programming languages with very different goals have been developed. In this section, we briefly discuss a few of the most common areas of computer applications and their associated languages.

1.2.1 Scientific Applications

The first digital computers, which appeared in the late 1940s and early 1950s, were invented and used for scientific applications. Typically, the scientific applications of that time used relatively simple data structures, but required large numbers of floating-point arithmetic computations. The most common data structures were arrays and matrices; the most common control structures were counting loops and selections. The early high-level programming languages invented for scientific applications were designed to provide for those needs. Their competition was assembly language, so efficiency was a primary concern. The first language for scientific applications was Fortran. ALGOL 60 and most of its descendants were also intended to be used in this area, although they were designed to be used in related areas as well. For some scientific applications where efficiency is the primary concern, such as those that were common in the 1950s and 1960s, no subsequent language is significantly better than Fortran, which explains why Fortran is still used.

1.2.2 Business Applications

The use of computers for business applications began in the 1950s. Special computers were developed for this purpose, along with special languages. The first successful high-level language for business was COBOL (ISO/IEC, 2002), the initial version of which appeared in 1960. It probably still is the most commonly used language for these applications. Business languages are characterized by facilities for producing elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify decimal arithmetic operations.

There have been few developments in business application languages outside the development and evolution of COBOL. Therefore, this book includes only limited discussions of the structures in COBOL.

1.2.3 Artificial Intelligence

Artificial intelligence (AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations. Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. Also, symbolic computation is more conveniently done with linked lists of data rather than arrays. This kind of programming sometimes requires more flexibility than other programming domains. For example, in some AI applications the ability to create and execute code segments during execution is convenient.

The first widely used programming language developed for AI applications was the functional language Lisp (McCarthy et al., 1965), which appeared in 1959. Most AI applications developed prior to 1990 were written in Lisp or one of its close relatives. During the early 1970s, however, an alternative approach to some of these applications appeared—logic programming using the Prolog (Clocksin and Mellish, 2013) language. More recently, some AI applications have been written in systems languages such as Python (Lutz, 2013). Scheme (Dybvig, 2011), a dialect of Lisp, and Prolog are introduced in Chapters 15 and 16, respectively.

1.2.4 Web Software

The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript (Flanagan, 2011) or PHP (Tatroe et al., 2013). There are also some markup-like languages that have been extended to include constructs that control document processing, which are discussed in Section 1.5 and in Chapter 2.

1.3 Language Evaluation Criteria

As noted previously, the purpose of this book is to examine carefully the underlying concepts of the various constructs and capabilities of programming languages. We will also evaluate these features, focusing on their impact on the software development process, including maintenance. To do this, we need a set of evaluation criteria. Such a list of criteria is necessarily controversial, because it is difficult to get even two computer scientists to agree on the value of some given language characteristic relative to others. In spite of these differences, most would agree that the criteria discussed in the following subsections are important.

Some of the characteristics that influence three of the four most important of these criteria are shown in Table 1.1, and the criteria themselves are discussed in the following sections.[2] Note that only the most important characteristics are included in the table, mirroring the discussion in the following subsections. One could probably make the case that if one considered less important characteristics, virtually all table positions could include "bullets."

Note that some of these characteristics are broad and somewhat vague, such as writability, whereas others are specific language constructs, such as exception handling. Furthermore, although the discussion might seem to imply that the criteria have equal importance, that implication is not intended, and it is clearly not the case.

1.3.1 Readability

One of the most important criteria for judging a programming language is the ease with which programs can be read and understood. Before 1970, software development was largely thought of in terms of writing code. The primary

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

2. The fourth primary criterion is cost, which is not included in the table because it is only slightly related to the other criteria and the characteristics that influence them.

positive characteristic of programming languages was efficiency. Language constructs were designed more from the point of view of the computer than of the computer users. In the 1970s, however, the software life-cycle concept (Booch, 1987) was developed; coding was relegated to a much smaller role, and maintenance was recognized as a major part of the cycle, particularly in terms of cost. Because ease of maintenance is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages. This was an important juncture in the evolution of programming languages. There was a distinct crossover from a focus on machine orientation to a focus on human orientation.

Readability must be considered in the context of the problem domain. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural and convoluted, making it unusually difficult to read.

The following subsections describe characteristics that contribute to the readability of a programming language.

1.3.1.1 Overall Simplicity

The overall simplicity of a programming language strongly affects its readability. A language with a large number of basic constructs is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features. This learning pattern is sometimes used to excuse the large number of language constructs, but that argument is not valid. Readability problems occur whenever the program's author has learned a different subset from that subset with which the reader is familiar.

A second complicating characteristic of a programming language is **feature multiplicity**—that is, having more than one way to accomplish a particular operation. For example, in Java, a user can increment a simple integer variable in four different ways:

```
count = count + 1
count += 1
count++
++count
```

Although the last two statements have slightly different meanings from each other and from the others in some contexts, all of them have the same meaning when used as stand-alone expressions. These variations are discussed in Chapter 7.

A third potential problem is **operator overloading**, in which a single operator symbol has more than one meaning. Although this is often useful, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly. For example, it is clearly acceptable to overload + to use it for both integer and floating-point addition. In fact, this overloading

simplifies a language by reducing the number of operators. However, suppose the programmer defined + used between single-dimensioned array operands to mean the sum of all elements of both arrays. Because the usual meaning of vector addition is quite different from this, this unusual meaning could confuse both the author and the program's readers. An even more extreme example of program confusion would be a user defining + between two vector operands to mean the difference between their respective first elements. Operator overloading is further discussed in Chapter 7.

Simplicity in languages can, of course, be carried too far. For example, the form and meaning of most assembly language statements are models of simplicity, as you can see when you consider the statements that appear in the next section. This very simplicity, however, makes assembly language programs less readable. Because they lack more complex control statements, program structure is less obvious; because the statements are simple, far more of them are required than in equivalent programs in a high-level language. These same arguments apply to the less extreme case of high-level languages with inadequate control and data-structuring constructs.

1.3.1.2 Orthogonality

Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful. For example, consider data types. Suppose a language has four primitive data types (integer, float, double, and character) and two type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined.

The meaning of an orthogonal language feature is independent of the context of its appearance in a program. (The word *orthogonal* comes from the mathematical concept of orthogonal vectors, which are independent of each other.) Orthogonality follows from a symmetry of relationships among primitives. A lack of orthogonality leads to exceptions to the rules of the language. For example, in a programming language that supports pointers, it should be possible to define a pointer to point to any specific type defined in the language. However, if pointers are not allowed to point to arrays, many potentially useful user-defined data structures cannot be defined.

We can illustrate the use of orthogonality as a design concept by comparing one aspect of the assembly languages of the IBM mainframe computers and the VAX series of minicomputers. We consider a single simple situation: adding two 32-bit integer values that reside in either memory or registers and replacing one of the two values with the sum. The IBM mainframes have two instructions for this purpose, which have the forms

```
A Reg1, memory_cell
AR Reg1, Reg2
```

where `Reg1` and `Reg2` represent registers. The semantics of these are

```
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)
```

The VAX addition instruction for 32-bit integer values is

```
ADDL operand_1, operand_2
```

whose semantics is

```
operand_2 ← contents(operand_1) + contents(operand_2)
```

In this case, either operand can be a register or a memory cell.

The VAX instruction design is orthogonal in that a single instruction can use either registers or memory cells as its operands. There are two ways to specify operands, which can be combined in all possible ways. The IBM design is not orthogonal. Only two out of four operand combinations possibilities are legal, and the two require different instructions, `A` and `AR`. The IBM design is more restricted and therefore less writable. For example, you cannot add two values and store the sum in a memory location. Furthermore, the IBM design is more difficult to learn because of the restrictions and the additional instruction.

Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand. Anyone who has learned a significant part of the English language can testify to the difficulty of learning its many rule exceptions (for example, *i* before *e* except after *c*).

As examples of the lack of orthogonality in a high-level language, consider the following rules and exceptions in C. Although C has two kinds of structured data types, arrays and records (**`struct`**s), records can be returned from functions but arrays cannot. A member of a structure can be any data type except **`void`** or a structure of the same type. An array element can be any data type except **`void`** or a function. Parameters are passed by value, unless they are arrays, in which case they are, in effect, passed by reference (because the appearance of an array name without a subscript in a C program is interpreted to be the address of the array's first element).

As an example of context dependence, consider the C expression

```
a + b
```

This expression often means that the values of `a` and `b` are fetched and added together. However, if `a` happens to be a pointer and `b` is an integer, it affects the value of `b`. For example, if `a` points to a float value that occupies four bytes, then the value of `b` must be scaled—in this case multiplied by 4—before it is

added to a. Therefore, the type of a affects the treatment of the value of b. The context of b affects its meaning.

Too much orthogonality can also cause problems. Perhaps the most orthogonal programming language is ALGOL 68 (van Wijngaarden et al., 1969). Every language construct in ALGOL 68 has a type, and there are no restrictions on those types. In addition, most constructs produce values. This combinational freedom allows extremely complex constructs. For example, a conditional can appear as the left side of an assignment, along with declarations and other assorted statements, as long as the result is an address. This extreme form of orthogonality leads to unnecessary complexity. Furthermore, because languages require a large number of primitives, a high degree of orthogonality results in an explosion of combinations. So, even if the combinations are simple, their sheer numbers lead to complexity.

Simplicity in a language, therefore, is at least in part the result of a combination of a relatively small number of primitive constructs and a limited use of the concept of orthogonality.

Some believe that functional languages offer a good combination of simplicity and orthogonality. A functional language, such as Lisp, is one in which computations are made primarily by applying functions to given parameters. In contrast, in imperative languages such as C, C++, and Java, computations are usually specified with variables and assignment statements. Functional languages offer potentially the greatest overall simplicity, because they can accomplish everything with a single construct, the function call, which can be combined simply with other function calls. This simple elegance is the reason why some language researchers are attracted to functional languages as the primary alternative to complex nonfunctional languages such as Java. Other factors, the most important of which is probably efficiency, however, have prevented functional languages from becoming more widely used.

1.3.1.3 Data Types

The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability. For example, suppose a numeric type is used for an indicator flag because there is no Boolean type in the language. In such a language, for example, in the original version of C, we might have an assignment such as the following:

```
timeout = 1
```

The meaning of this statement is unclear, whereas in a language that includes Boolean types, we would have the following:

```
timeout = true
```

The meaning of this statement is perfectly clear.

1.3.1.4 Syntax Design

The syntax, or form, of the elements of a language has a significant effect on the readability of programs. Following are some examples of syntactic design choices that affect readability:

- *Special words.* Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, **`while`**, **`class`**, and **`for`**). Especially important is the method of forming compound statements, or statement groups, primarily in control constructs. Some languages have used matching pairs of special words or symbols to form groups. C and its descendants use braces to specify compound statements. All of these languages have diminished readability because statement groups are always terminated in the same way, which makes it difficult to determine which group is being ended when an **`end`** or a right brace appears. Fortran 95 and Ada (ISO/IEC, 2014) make this clearer by using a distinct closing syntax for each type of statement group. For example, Ada uses **`end if`** to terminate a selection construct and **`end loop`** to terminate a loop construct. This is an example of the conflict between simplicity that results in fewer reserved words, as in Java, and the greater readability that can result from using more reserved words, as in Ada.

  Another important issue is whether the special words of a language can be used as names for program variables. If so, then the resulting programs can be very confusing. For example, in Fortran 95, special words, such as `Do` and `End`, are legal variable names, so the appearance of these words in a program may or may not connote something special.

- *Form and meaning.* Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability. Semantics, or meaning, should follow directly from syntax, or form. In some cases, this principle is violated by two language constructs that are identical or similar in appearance but have different meanings, depending perhaps on context. In C, for example, the meaning of the reserved word **`static`** depends on the context of its appearance. If used on the definition of a variable inside a function, it means the variable is created at compile time. If used on the definition of a variable that is outside all functions, then it means the variable is visible only in the file in which its definition appears; that is, it is not exported from that file.

  One of the primary complaints about the shell commands of UNIX (Robbins, 2005) is that their appearance does no t always suggest their function. For example, the meaning of the UNIX command `grep` can be deciphered only through prior knowledge, or perhaps cleverness and familiarity with the UNIX editor, `ed`. The appearance of `grep` connotes nothing to UNIX beginners. (In `ed`, the command */regular_expression/* searches for a substring that matches the regular expression. Preceding this with `g` makes it a global command, specifying that the scope of the search is the whole file being edited. Following the command with `p` specifies that lines with the matching substring are to be printed. So `g`*/regular_expression/*`p`, which can obviously be abbreviated as `grep`, prints all lines in a file that contain substrings that match its operand, which is a regular expression.)

1.3.2 Writability

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability. This follows directly from the fact that the process of writing a program requires the programmer frequently to reread the part of the program that is already written.

As is the case with readability, writability must be considered in the context of the target problem domain of a language. It simply is not fair to compare the writability of two languages in the realm of a particular application when one was designed for that application and the other was not. For example, the writabilities of Visual BASIC (VB) (Halvorson, 2013) and C are dramatically different for creating a program that has a graphical user interface (GUI), for which VB is ideal. Their writabilities are also quite different for writing systems programs, such as an operating system, for which C was designed.

The following subsections describe the most important characteristics influencing the writability of a language.

1.3.2.1 Simplicity and Orthogonality

If a language has a large number of different constructs, some programmers who use the language might not be familiar with all of them. This situation can lead to a misuse of some features and a disuse of others that may be either more elegant or more efficient, or both, than those that are used. It may even be possible, as noted by Hoare (1973), to use unknown features accidentally, with bizarre results. Therefore, a smaller number of primitive constructs and a consistent set of rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives. A programmer can design a solution to a complex problem after learning only a simple set of primitive constructs.

On the other hand, too much orthogonality can be a detriment to writability. Errors in programs can go undetected when nearly any combination of primitives is legal. This can lead to code absurdities that cannot be discovered by the compiler.

1.3.2.2 Expressivity

Expressivity in a language can refer to several different characteristics. In a language such as APL (Gilman and Rose, 1983), it means that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program. More commonly, it means that a language has relatively convenient, rather than cumbersome, ways of specifying computations. For example, in C, the notation `count++` is more convenient and shorter than `count = count + 1`. Also, the **`and then`** Boolean operator in Ada is a convenient way of specifying short-circuit evaluation of a Boolean expression. The inclusion of the **`for`** statement in Java makes writing counting loops easier than with the use of **`while`**, which is also possible. All of these increase the writability of a language.

1.3.3 Reliability

A program is said to be reliable if it performs to its specifications under all conditions. The following subsections describe several language features that have a significant effect on the reliability of programs in a given language.

1.3.3.1 Type Checking

Type checking is simply testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability. Because run-time type checking is expensive, compile-time type checking is more desirable. Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs. The design of Java requires checks of the types of nearly all variables and expressions at compile time. This virtually eliminates type errors at run time in Java programs. Types and type checking are discussed in depth in Chapter 6.

One example of how failure to type check, at either compile time or run time, has led to countless program errors is the use of subprogram parameters in the original C language (Kernighan and Ritchie, 1978). In this language, the type of an actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function. An `int` type variable could be used as an actual parameter in a call to a function that expected a `float` type as its formal parameter, and neither the compiler nor the run-time system would detect the inconsistency. For example, because the bit string that represents the integer 23 is essentially unrelated to the bit string that represents a floating-point 23, if an integer 23 is sent to a function that expects a floating-point parameter, any uses of the parameter in the function will produce nonsense. Furthermore, such problems are often difficult to diagnose.[3] The current version of C has eliminated this problem by requiring all parameters to be type checked. Subprograms and parameter-passing techniques are discussed in Chapter 9.

1.3.3.2 Exception Handling

The ability of a program to intercept run-time errors (as well as other unusual conditions detectable by the program), take corrective measures, and then continue is an obvious aid to reliability. This language facility is called **exception handling**. Ada, C++, Java, and C# include extensive capabilities for exception handling, but such facilities are practically nonexistent in some widely used languages, for example, C. Exception handling is discussed in Chapter 14.

1.3.3.3 Aliasing

Loosely defined, **aliasing** is having two or more distinct names in a program that can be used to access the same memory cell. It is now generally accepted that aliasing is a dangerous feature in a programming language. Most

3. In response to this and other similar problems, UNIX systems include a utility program named `lint` that checks C programs for such problems.

programming languages allow some kind of aliasing—for example, two pointers (or references) set to point to the same variable, which is possible in most languages. In such a program, the programmer must always remember that changing the value pointed to by one of the two changes the value referenced by the other. Some kinds of aliasing, as described in Chapters 5 and 9, can be prohibited by the design of a language.

In some languages, aliasing is used to overcome deficiencies in the language's data abstraction facilities. Other languages greatly restrict aliasing to increase their reliability.

1.3.3.4 Readability and Writability

Both readability and writability influence reliability. A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural approaches. Unnatural approaches are less likely to be correct for all possible situations. The easier a program is to write, the more likely it is to be correct.

Readability affects reliability in both the writing and maintenance phases of the life cycle. Programs that are difficult to read are difficult both to write and to modify.

1.3.4 Cost

The total cost of a programming language is a function of many of its characteristics.

First, there is the cost of training programmers to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers. Although more powerful languages are not necessarily more difficult to learn, they often are.

Second, there is the cost of writing programs in the language. This is a function of the writability of the language, which depends in part on its closeness in purpose to the particular application. The original efforts to design and implement high-level languages were driven by the desire to lower the costs of creating software.

Both the cost of training programmers and the cost of writing programs in a language can be significantly reduced in a good programming environment. Programming environments are discussed in Section 1.8.

Third, the cost of executing programs written in a language is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler. Although execution efficiency was the foremost concern in the design of early languages, it is now considered to be less important.

A simple trade-off can be made between compilation cost and execution speed of the compiled code. **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce. If little or no optimization is done, compilation can be done much faster than if a significant effort is made to

produce optimized code. The choice between the two alternatives is influenced by the environment in which the compiler will be used. In a laboratory for beginning programming students, who often compile their programs many times during development but use little execution time (their programs are small and they must execute correctly only once), little or no optimization should be done. In a production environment, where compiled programs are executed many times after development, it is better to pay the extra cost to optimize the code.

Fourth, there is the cost of poor reliability. If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high. The failures of noncritical systems can also be very expensive in terms of lost future business or lawsuits over defective software systems.

The final consideration is the cost of maintaining programs, which includes both corrections and modifications to add new functionality. The cost of software maintenance depends on a number of language characteristics, primarily readability. Because maintenance is often done by individuals other than the original author of the software, poor readability can make the task extremely challenging.

The importance of software maintainability cannot be overstated. It has been estimated that for large software systems with relatively long lifetimes, maintenance costs can be as high as two to four times as much as development costs (Sommerville, 2010).

Of all the contributors to language costs, three are most important: program development, maintenance, and reliability. Because these are functions of writability and readability, these two evaluation criteria are, in turn, the most important.

Of course, a number of other criteria could be used for evaluating programming languages. One example is **portability**, or the ease with which programs can be moved from one implementation to another. Portability is most strongly influenced by the degree of standardization of the language. Some languages are not standardized at all, making programs in these languages very difficult to move from one implementation to another. This problem is alleviated in some cases by the fact that implementations for some languages now have single sources. Standardization is a time-consuming and difficult process. A committee began work on producing a standard version of C++ in 1989. It was approved in 1998.

Generality (the applicability to a wide range of applications) and **well-definedness** (the completeness and precision of the language's official defining document) are two other criteria.

Most criteria, particularly readability, writability, and reliability, are neither precisely defined nor accurately measurable. Nevertheless, they are useful concepts and they provide valuable insight into the design and evaluation of programming languages.

A final note on evaluation criteria: language design criteria are weighed differently from different perspectives. Language implementors are concerned

primarily with the difficulty of implementing the constructs and features of the language. Language users are worried about writability first and readability later. Language designers are likely to emphasize elegance and the ability to attract widespread use. These characteristics often conflict with one another.

1.4 Influences on Language Design

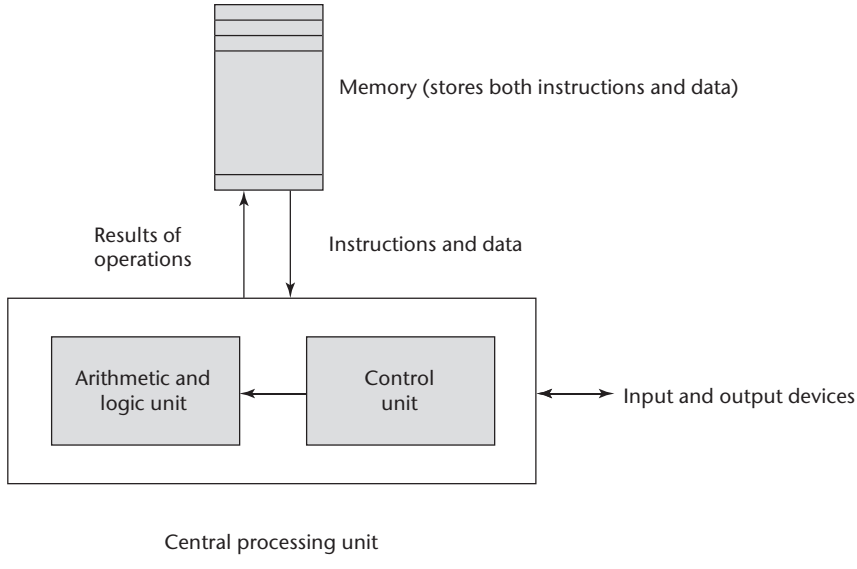
In addition to those factors described in Section 1.3, several other factors influence the basic design of programming languages. The most important of these are computer architecture and programming design methodologies.

1.4.1 Computer Architecture

The basic architecture of computers has had a profound effect on language design. Most of the popular languages of the past 60 years have been designed around the prevalent computer architecture, called the **von Neumann architecture**, after one of its originators, John von Neumann (pronounced "von Noyman"). These languages are called **imperative** languages. In a von Neumann computer, both data and programs are stored in the same memory. The central processing unit (CPU), which executes instructions, is separate from the memory. Therefore, instructions and data must be transmitted, or piped, from memory to the CPU. Results of operations in the CPU must be moved back to memory. Nearly all digital computers built since the 1940s have been based on the von Neumann architecture. The overall structure of a von Neumann computer is shown in Figure 1.1.

Figure 1.1

The von Neumann computer architecture



Because of the von Neumann architecture, the central features of imperative languages are variables, which model the memory cells; assignment statements, which are based on the piping operation; and the iterative form of repetition, which is the most efficient way to implement repetition on this architecture. Operands in expressions are piped from memory to the CPU, and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment. Iteration is fast on von Neumann computers because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction. This efficiency discourages the use of recursion for repetition, although recursion is sometimes more natural.

The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**. As stated earlier, programs reside in memory but are executed in the CPU. Each instruction to be executed must be moved from memory to the processor. The address of the next instruction to be executed is maintained in a register called the **program counter**. The fetch-execute cycle can be simply described by the following algorithm:

```
initialize the program counter
repeat forever
    fetch the instruction pointed to by the program counter
    increment the program counter to point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

The "decode the instruction" step in the algorithm means the instruction is examined to determine what action it specifies. Program execution terminates when a stop instruction is encountered, although on an actual computer a stop instruction is rarely executed. Rather, control transfers from the operating system to a user program for its execution and then back to the operating system when the user program execution is complete. In a computer system in which more than one user program may be in memory at a given time, this process is far more complex.

As stated earlier, a functional, or applicative, language is one in which the primary means of computation is applying functions to given parameters. Programming can be done in a functional language without the kind of variables that are used in imperative languages, without assignment statements, and without iteration. Although many computer scientists have expounded on the myriad benefits of functional languages, such as Scheme, it is unlikely that they will displace the imperative languages until a non–von Neumann computer is designed that allows efficient execution of programs in functional languages. Among those who have bemoaned this fact, perhaps the most eloquent was John Backus (1978), the principal designer of the original version of Fortran.

In spite of the fact that the structure of imperative programming languages is modeled on a machine architecture, rather than on the abilities and inclinations of the users of programming languages, some believe that using imperative languages is somehow more natural than using a functional language. So, these people believe that even if functional programs were as efficient as imperative programs, the use of imperative programming languages would still dominate.

1.4.2 Programming Design Methodologies

The late 1960s and early 1970s brought an intense analysis, begun in large part by the structured-programming movement, of both the software development process and programming language design.

An important reason for this research was the shift in the major cost of computing from hardware to software, as hardware costs decreased and programmer costs increased. Increases in programmer productivity were relatively small. In addition, progressively larger and more complex problems were being solved by computers. Rather than simply solving sets of equations to simulate satellite tracks, as in the early 1960s, programs were being written for large and complex tasks, such as controlling large petroleum-refining facilities and providing worldwide airline reservation systems.

The new software development methodologies that emerged as a result of the research of the 1970s were called top-down design and stepwise refinement. The primary programming language deficiencies that were discovered were incompleteness of type checking and inadequacy of control statements (requiring the extensive use of gotos).

In the late 1970s, a shift from procedure-oriented to data-oriented program design methodologies began. Simply put, data-oriented methods emphasize data design, focusing on the use of abstract data types to solve problems.

For data abstraction to be used effectively in software system design, it must be supported by the languages used for implementation. The first language to provide even limited support for data abstraction was SIMULA 67 (Birtwistle et al., 1973), although that language certainly was not propelled to popularity because of it. The benefits of data abstraction were not widely recognized until the early 1970s. However, most languages designed since the late 1970s support data abstraction, which is discussed in detail in Chapter 11.

The latest step in the evolution of data-oriented software development, which began in the early 1980s, is object-oriented design. Object-oriented methodology begins with data abstraction, which encapsulates processing with data objects and controls access to data, and adds inheritance and dynamic method binding. Inheritance is a powerful concept that greatly enhances the potential reuse of existing software, thereby providing the possibility of significant increases in software development productivity. This is an important factor in the increase in popularity of object-oriented languages. Dynamic (run-time) method binding allows more flexible use of inheritance.

Object-oriented programming developed along with a language that supported its concepts: Smalltalk (Goldberg and Robson, 1989). Although Smalltalk never became as widely used as many other languages, support for object-oriented programming is now part of most popular imperative languages, including Java, C++, and C#. Object-oriented concepts have also found their way into functional programming in CLOS (Bobrow et al., 1988) and F# (Syme et al., 2010), as well as logic programming in Prolog++ (Moss, 1994). Language support for object-oriented programming is discussed in detail in Chapter 12.

Procedure-oriented programming is, in a sense, the opposite of data-oriented programming. Although data-oriented methods now dominate software development, procedure-oriented methods have not been abandoned. On the contrary, in recent years, a good deal of research has occurred in procedure-oriented programming, especially in the area of concurrency. These research efforts brought with them the need for language facilities for creating and controlling concurrent program units. Java and C# include such capabilities. Concurrency is discussed in detail in Chapter 13.

All of these evolutionary steps in software development methodologies led to new language constructs to support them.

1.5 Language Categories

Programming languages are often categorized into four bins: imperative, functional, logic, and object oriented. However, we do not consider languages that support object-oriented programming to form a separate category of languages. We have described how the most popular languages that support object-oriented programming grew out of imperative languages. Although the object-oriented software development paradigm differs significantly from the procedure-oriented paradigm usually used with imperative languages, the extensions to an imperative language required to support object-oriented programming are not intensive. For example, the expressions, assignment statements, and control statements of C and Java are nearly identical. (On the other hand, the arrays, subprograms, and semantics of Java are very different from those of C.) Similar statements can be made for functional languages that support object-oriented programming.

Some authors refer to scripting languages as a separate category of programming languages. However, languages in this category are bound together more by their implementation method, partial or full interpretation, than by a common language design. The languages that are typically called scripting languages, among them Perl, JavaScript, and Ruby (Flanagan and Matsumoto, 2008), are imperative languages in every sense.

A logic programming language is an example of a rule-based language. In an imperative language, an algorithm is specified in great detail, and the specific order of execution of the instructions or statements must be included. In a rule-based language, however, rules are specified in no particular order, and the language implementation system must choose an order in which the

rules are used to produce the desired result. This approach to software development is radically different from those used with the other two categories of languages and clearly requires a completely different kind of language. Prolog, the most commonly used logic programming language, and logic programming are discussed in Chapter 16.

In recent years, a new category of languages has emerged, the markup/programming hybrid languages. Markup languages are not programming languages. For instance, HTML, the most widely used markup language, is used to specify the layout of information in Web documents. However, some programming capability has crept into some extensions to HTML and XML. Among these are the Java Server Pages Standard Tag Library (JSTL) and eXtensible Stylesheet Language Transformations (XSLT). Both of these are briefly introduced in Chapter 2. Those languages cannot be compared to any of the complete programming languages and therefore will not be discussed after Chapter 2.

1.6 Language Design Trade-Offs

The programming language evaluation criteria described in Section 1.3 provide a framework for language design. Unfortunately, that framework is self-contradictory. In his insightful paper on language design, Hoare (1973) stated that "there are so many important but conflicting criteria, that their reconciliation and satisfaction is a major engineering task."

Two criteria that conflict are reliability and cost of execution. For example, the Java language definition demands that all references to array elements be checked to ensure that the index or indices are in their legal ranges. This step adds a great deal to the cost of execution of Java programs that contain large numbers of references to array elements. C does not require index range checking, so C programs execute faster than semantically equivalent Java programs, although Java programs are more reliable. The designers of Java traded execution efficiency for reliability.

As another example of conflicting criteria that leads directly to design trade-offs, consider the case of APL. APL includes a powerful set of operators for array operands. Because of the large number of operators, a significant number of new symbols had to be included in APL to represent the operators. Also, many APL operators can be used in a single, long, complex expression. One result of this high degree of expressivity is that, for applications involving many array operations, APL is very writable. Indeed, a huge amount of computation can be specified in a very small program. Another result is that APL programs have very poor readability. A compact and concise expression has a certain mathematical beauty but it is difficult for anyone other than the programmer to understand. Well-known author Daniel McCracken (1970) once noted that it took him four hours to read and understand a four-line APL program. The designer of APL traded readability for writability.

The conflict between writability and reliability is a common one in language design. The pointers of C++ can be manipulated in a variety of ways, which supports highly flexible addressing of data. Because of the potential reliability problems with pointers, they are not included in Java.

Examples of conflicts among language design (and evaluation) criteria abound; some are subtle, others are obvious. It is therefore clear that the task of choosing constructs and features when designing a programming language requires many compromises and trade-offs.

1.7 Implementation Methods

As described in Section 1.4.1, two of the primary components of a computer are its internal memory and its processor. The internal memory is used to store programs and data. The processor is a collection of circuits that provides a realization of a set of primitive operations, or machine instructions, such as those for arithmetic and logic operations. In most computers, some of these instructions, which are sometimes called macroinstructions, are actually implemented with a set of instructions called microinstructions, which are defined at an even lower level. Because microinstructions are never seen by software, they will not be discussed further here.

The machine language of the computer is its set of instructions. In the absence of other supporting software, its own machine language is the only language that most hardware computers "understand." Theoretically, a computer could be designed and built with a particular high-level language as its machine language, but it would be very complex and expensive. Furthermore, it would be highly inflexible, because it would be difficult (but not impossible) to use it with other high-level languages. The more practical machine design choice implements in hardware a very low-level language that provides the most commonly needed primitive operations and requires system software to create an interface to programs in higher-level languages.

A language implementation system cannot be the only software on a computer. Also required is a large collection of programs, called the operating system, which supplies higher-level primitives than those of the machine language. These primitives provide system resource management, input and output operations, a file management system, text and/or program editors, and a variety of other commonly needed functions. Because language implementation systems need many of the operating system facilities, they interface with the operating system rather than directly with the processor (in machine language).

The operating system and language implementations are layered over the machine language interface of a computer. These layers can be thought of as virtual computers, providing interfaces to the user at higher levels. For example, an operating system and a C compiler provide a virtual C computer. With other compilers, a machine can become other kinds of virtual computers. Most computer systems provide several different virtual computers. User programs form another layer over the top of the layer of virtual computers. The layered view of a computer is shown in Figure 1.2.

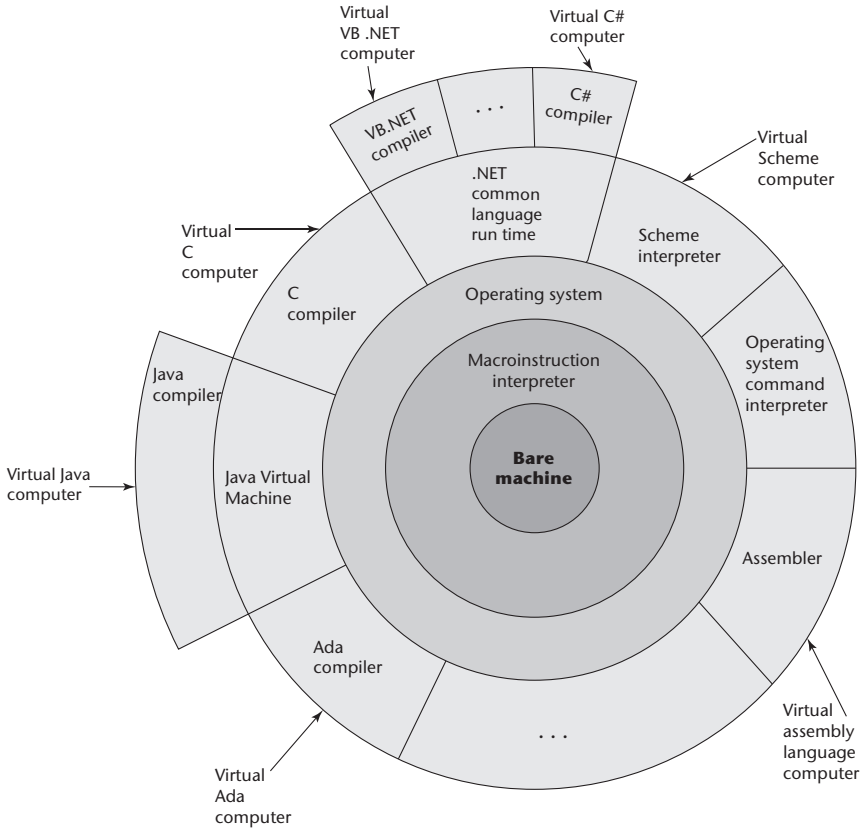


Figure 1.2

Layered interface of virtual computers, provided by a typical computer system

The implementation systems of the first high-level programming languages, constructed in the late 1950s, were among the most complex software systems of that time. In the 1960s, intensive research efforts were made to understand and formalize the process of constructing these high-level language implementations. The greatest success of those efforts was in the area of syntax analysis, primarily because that part of the implementation process is an application of parts of automata theory and formal language theory that were then well understood.

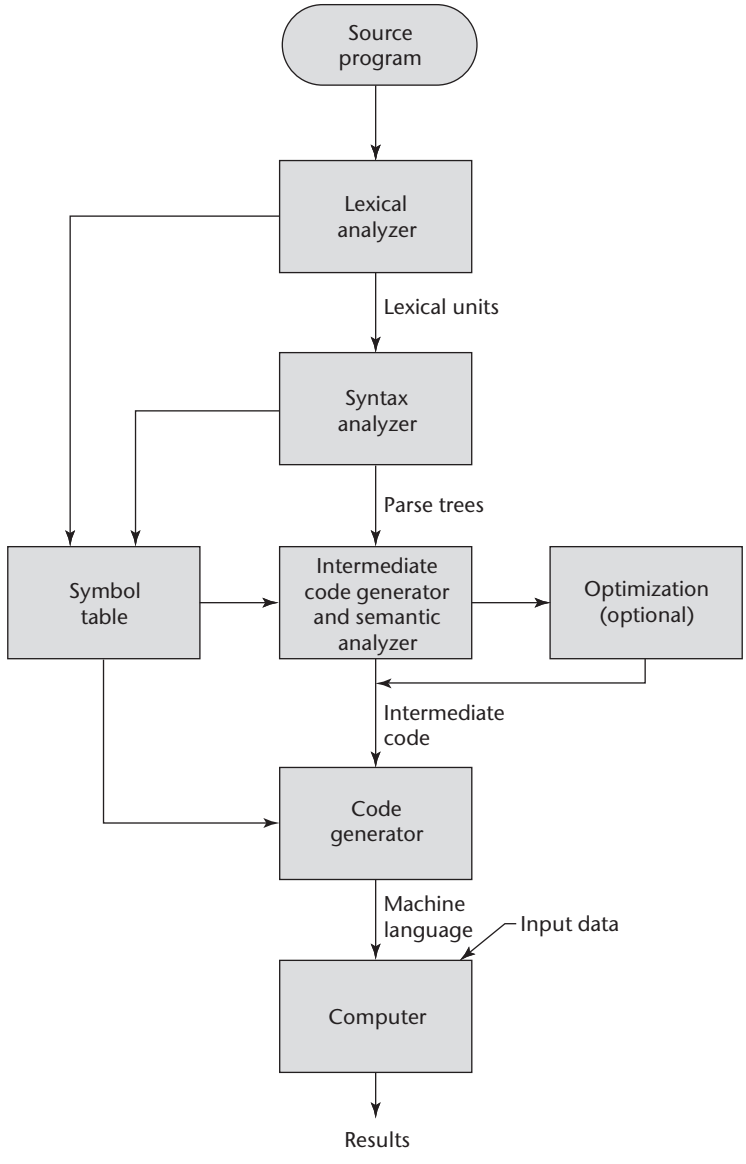
1.7.1 Compilation

Programming languages can be implemented by any of three general methods. At one extreme, programs can be translated into machine language, which can be executed directly on the computer. This method is called a **compiler implementation** and has the advantage of very fast program execution, once the translation process is complete. Most production implementations of languages, such as C, COBOL, and C++, are by compilers.

The language that a compiler translates is called the **source language**. The process of compilation and program execution takes place in several phases, the most important of which are shown in Figure 1.3.

Figure 1.3

The compilation process



The lexical analyzer gathers the characters of the source program into lexical units. The lexical units of a program are identifiers, special words, operators, and punctuation symbols. The lexical analyzer ignores comments in the source program because the compiler has no use for them.

The syntax analyzer takes the lexical units from the lexical analyzer and uses them to construct hierarchical structures called **parse trees**. These parse trees represent the syntactic structure of the program. In many cases, no actual parse tree structure is constructed; rather, the information that would be required to

build a tree is generated and used directly. Both lexical units and parse trees are discussed further in Chapter 3. Lexical analysis and syntax analysis, or parsing, are discussed in Chapter 4.

The intermediate code generator produces a program in a different language, at an intermediate level between the source program and the final output of the compiler: the machine language program.[4] Intermediate languages sometimes look very much like assembly languages, and in fact, sometimes are actual assembly languages. In other cases, the intermediate code is at a level somewhat higher than an assembly language. The semantic analyzer is an integral part of the intermediate code generator. The semantic analyzer checks for errors, such as type errors, that are difficult, if not impossible, to detect during syntax analysis.

Optimization, which improves programs (usually in their intermediate code version) by making them smaller or faster or both. Because many kinds of optimization are difficult to do on machine language, most optimization is done on the intermediate code.

The code generator translates the optimized intermediate code version of the program into an equivalent machine language program.

The symbol table serves as a database for the compilation process. The primary contents of the symbol table are the type and attribute information of each user-defined name in the program. This information is placed in the symbol table by the lexical and syntax analyzers and is used by the semantic analyzer and the code generator.

As stated previously, although the machine language generated by a compiler can be executed directly on the hardware, it must nearly always be run along with some other code. Most user programs also require programs from the operating system. Among the most common of these are programs for input and output. The compiler builds calls to required system programs when they are needed by the user program. Before the machine language programs produced by a compiler can be executed, the required programs from the operating system must be found and linked to the user program. The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program. The user and system code together are sometimes called a **load module**, or **executable image**. The process of collecting system programs and linking them to user programs is called **linking and loading**, or sometimes just **linking**. It is accomplished by a systems program called a **linker**.

In addition to systems programs, user programs must often be linked to previously compiled programs that reside in libraries. So the linker not only links a given program to system programs, but also it may link it to other user or system-supplied programs.

The speed of the connection between a computer's memory and its processor often determines the speed of the computer, because instructions often can be executed faster than they can be moved to the processor for execution.

---

4. Note that the words *program* and *code* are often used interchangeably.

This connection is called the **von Neumann bottleneck**; it is the primary limiting factor in the speed of von Neumann architecture computers. The von Neumann bottleneck has been one of the primary motivations for the research and development of parallel computers.

1.7.2 Pure Interpretation

Pure interpretation lies at the opposite end (from compilation) among implementation methods. With this approach, programs are interpreted by another program called an interpreter, with no translation whatever. The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high-level language program statements rather than machine instructions. This software simulation obviously provides a virtual machine for the language.

Pure interpretation has the advantage of allowing easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units. For example, if an array index is found to be out of range, the error message can easily indicate the source line of the error and the name of the array. On the other hand, this method has the serious disadvantage that execution is 10 to 100 times slower than in compiled systems. The primary source of this slowness is the decoding of the high-level language statements, which are far more complex than machine language instructions (although there may be fewer statements than instructions in equivalent machine code). Furthermore, regardless of how many times a statement is executed, it must be decoded every time. Therefore, statement decoding, rather than the connection between the processor and memory, is the bottleneck of a pure interpreter.

Another disadvantage of pure interpretation is that it often requires more space. In addition to the source program, the symbol table must be present during interpretation. Furthermore, the source program may be stored in a form designed for easy access and modification rather than one that provides for minimal size.

Although some simple early languages of the 1960s (APL, SNOBOL (Griswold et al., 1971), and Lisp) were purely interpreted, by the 1980s, the approach was rarely used on high-level languages. However, in recent years, pure interpretation has made a significant comeback with some Web scripting languages, such as JavaScript and PHP, which are now widely used. The process of pure interpretation is shown in Figure 1.4.

1.7.3 Hybrid Implementation Systems

Some language implementation systems are a compromise between compilers and pure interpreters; they translate high-level language programs to an intermediate language designed to allow easy interpretation. This method is faster than pure interpretation because the source language statements are decoded only once. Such implementations are called **hybrid implementation systems**.

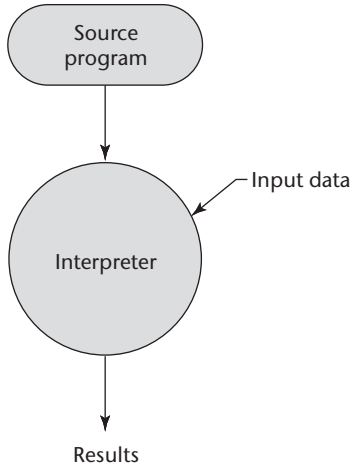


Figure 1.4
Pure interpretation

The process used in a hybrid implementation system is shown in Figure 1.5. Instead of translating intermediate language code to machine code, it simply interprets the intermediate code.

Perl is implemented with a hybrid system. Perl programs are partially compiled to detect errors before interpretation and to simplify the interpreter.

Initial implementations of Java were all hybrid. Its intermediate form, called **byte code**, provides portability to any machine that has a byte code interpreter and an associated run-time system. Together, these are called the Java Virtual Machine. There are now systems that translate Java byte code into machine code for faster execution.

A Just-in-Time (JIT) implementation system initially translates programs to an intermediate language. Then, during execution, it compiles intermediate language methods into machine code when they are called. The machine code version is kept for subsequent calls. JIT systems now are widely used for Java programs. Also, the .NET languages are all implemented with a JIT system.

Sometimes an implementor may provide both compiled and interpreted implementations for a language. In these cases, the interpreter is used to develop and debug programs. Then, after a (relatively) bug-free state is reached, the programs are compiled to increase their execution speed.

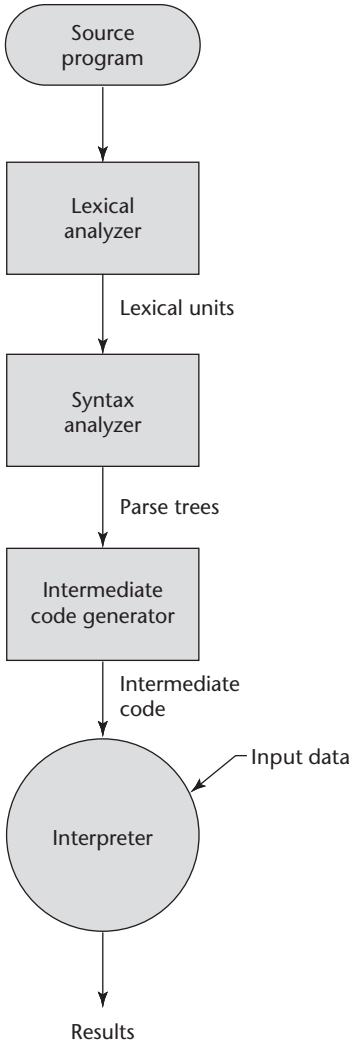
1.7.4 Preprocessors

A **preprocessor** is a program that processes a program just before the program is compiled. Preprocessor instructions are embedded in programs. The preprocessor is essentially a macro expander. Preprocessor instructions are commonly used to specify that the code from another file is to be included. For example, the C preprocessor instruction

```
#include "myLib.h"
```

Figure 1.5

Hybrid implementation system



causes the preprocessor to copy the contents of `myLib.h` into the program at the position of the `#include`.

Other preprocessor instructions are used to define symbols to represent expressions. For example, one could use

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

to determine the largest of two given expressions. For example, the expression

```
x = max(2 * y, z / 1.73);
```

would be expanded by the preprocessor to

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73);
```

Notice that this is one of those cases where expression side effects can cause trouble. For example, if either of the expressions given to the `max` macro have side effects—such as `z++`—it could cause a problem. Because one of the two expression parameters is evaluated twice, this could result in `z` being incremented twice by the code produced by the macro expansion.

1.8 Programming Environments

A programming environment is the collection of tools used in the development of software. This collection may consist of only a file system, a text editor, a linker, and a compiler. Or it may include a large collection of integrated tools, each accessed through a uniform user interface. In the latter case, the process of the development and maintenance of software is greatly enhanced. Therefore, the characteristics of a programming language are not the only measure of the software development capability of a system. We now briefly describe several programming environments.

UNIX is an older programming environment, first distributed in the middle 1970s, built around a portable multiprogramming operating system. It provides a wide array of powerful support tools for software production and maintenance in a variety of languages. In the past, the most important feature absent from UNIX was a uniform interface among its tools. This made it more difficult to learn and to use. However, UNIX is now often used through a GUI that runs on top of UNIX. Examples of UNIX GUIs are the Solaris Common Desktop Environment (CDE), GNOME, and KDE. These GUIs make the interface to UNIX appear similar to that of Windows and Macintosh systems.

Borland JBuilder is a programming environment that provides an integrated compiler, editor, debugger, and file system for Java development, where all four are accessed through a graphical interface. JBuilder is a complex and powerful system for creating Java software.

Microsoft Visual Studio .NET is a relatively recent step in the evolution of software development environments. It is a large and elaborate collection of software development tools, all used through a windowed interface. This system can be used to develop software in any one of the five .NET languages: C#, Visual Basic.NET, JScript (Microsoft's version of JavaScript), F# (a functional language), and C++/CLI.

NetBeans is a development environment that is primarily used for Java application development but also supports JavaScript, Ruby, and PHP. Both Visual Studio and NetBeans are more than development environments—they are also frameworks, which means they actually provide common parts of the code of the application.