2

Evolution of the Major Programming Languages

This chapter describes the development of a collection of programming languages. It explores the environment in which each was designed and focuses on the contributions of the language and the motivation for its development. Overall language descriptions are not included; rather, we discuss only some of the new features introduced by each language. Of particular interest are the features that most influenced subsequent languages or the field of computer science.

This chapter does not include an in-depth discussion of any language feature or concept; that is left for later chapters. Brief, informal explanations of features will suffice for our trek through the development of these languages.

This chapter discusses a wide variety of languages and language concepts that will not be familiar to many readers. These topics are discussed in detail only in later chapters. Those who find this unsettling may prefer to delay reading this chapter until the rest of the book has been studied.

The choice as to which languages to discuss here was subjective, and some readers will unhappily note the absence of one or more of their favorites. However, to keep this historical coverage to a reasonable size, it was necessary to leave out some languages that some regard highly. The choices were based on our estimate of each language's importance to language development and the computing world as a whole. We also include brief discussions of some other languages that are referenced later in the book.

The organization of this chapter is as follows: The initial versions of languages generally are discussed in chronological order. However, subsequent versions of languages appear with their initial version, rather than in later sections. For example, Fortran 2003 is discussed in the section with Fortran I (1956). Also, in some cases, languages of secondary importance that are related to a language that has its own section appear in that section.

This chapter includes listings of 14 complete example programs, each in a different language. These programs are not described in this chapter; they are meant simply to illustrate the appearance of programs in these languages. Readers familiar with any of the common imperative languages should be able to read and understand most of the code in these programs, except those in LISP, COBOL, and Smalltalk. (A Scheme function similar to the LISP example is discussed in Chapter 15.) The same problem is solved by the Fortran, ALGOL 60, PL/I, BASIC, Pascal, C, Perl, Ada, Java, JavaScript, and C# programs. Note that most of the contemporary languages in this list support dynamic arrays, but because of the simplicity of the example problem, we did not use them in the example programs. Also, in the Fortran 95 program, we avoided using the features that could have avoided the use of loops altogether, in part to keep the program simple and readable and in part just to illustrate the basic loop structure of the language.

Figure 2.1 is a chart of the genealogy of the high-level languages discussed in this chapter.

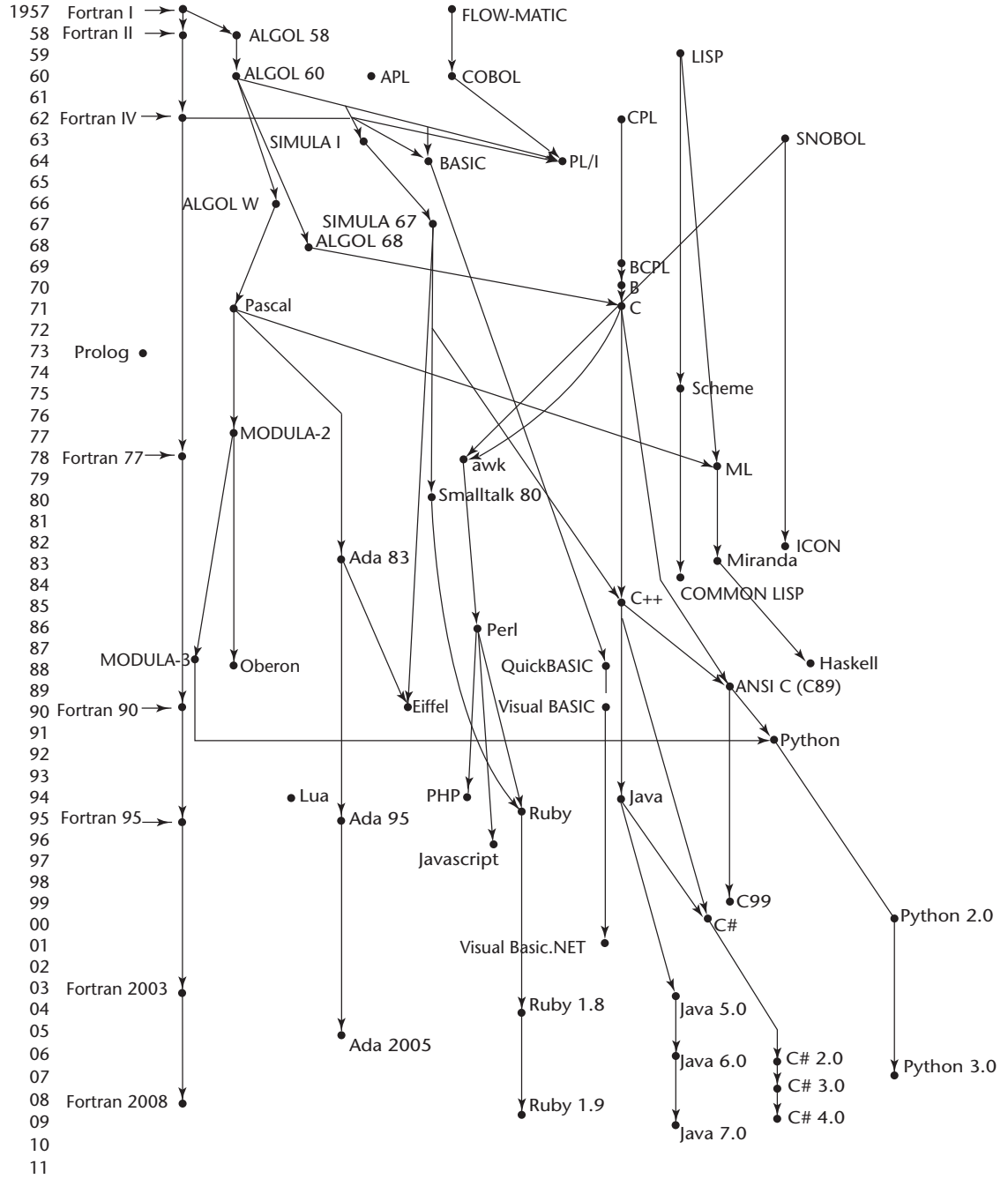


Figure 2.1

Genealogy of common high-level programming languages

2.1 Zuse's Plankalkül

The first programming language discussed in this chapter is highly unusual in several respects. For one thing, it was never implemented. Furthermore, although developed in 1945, its description was not published until 1972. Because so few people were familiar with the language, some of its capabilities did not appear in other languages until 15 years after its development.

2.1.1 Historical Background

Between 1936 and 1945, German scientist Konrad Zuse (pronounced "Tsoo-zuh") built a series of complex and sophisticated computers from electromechanical relays. By early 1945, Allied bombing had destroyed all but one of his latest models, the Z4, so he moved to a remote Bavarian village, Hinterstein, and his research group members went their separate ways.

Working alone, Zuse embarked on an effort to develop a language for expressing computations for the Z4, a project he had begun in 1943 as a proposal for his Ph.D. dissertation. He named this language Plankalkül, which means *program calculus*. In a lengthy manuscript dated 1945 but not published until 1972 (Zuse, 1972), Zuse defined Plankalkül and wrote algorithms in the language to solve a wide variety of problems.

2.1.2 Language Overview

Plankalkül was remarkably complete, with some of its most advanced features in the area of data structures. The simplest data type in Plankalkül was the single bit. Integer and floating-point numeric types were built from the bit type. The floating-point type used twos-complement notation and the "hidden bit" scheme currently used to avoid storing the most significant bit of the normalized fraction part of a floating-point value.

In addition to the usual scalar types, Plankalkül included arrays and records (called *structs* in the C-based languages). The records could include nested records.

Although the language had no explicit goto, it did include an iterative statement similar to the Ada **`for`**. It also had the command **`Fin`** with a superscript that specified an exit out of a given number of iteration loop nestings or to the beginning of a new iteration cycle. Plankalkül included a selection statement, but it did not allow an `else` clause.

One of the most interesting features of Zuse's programs was the inclusion of mathematical expressions showing the current relationships between program variables. These expressions stated what would be true during execution at the points in the code where they appeared. These are very similar to the assertions of Java and in those in axiomatic semantics, which is discussed in Chapter 3.

Zuse's manuscript contained programs of far greater complexity than any written prior to 1945. Included were programs to sort arrays of numbers; test the connectivity of a given graph; carry out integer and floating-point operations, including square root; and perform syntax analysis on logic formulas that had parentheses and operators in six different levels of precedence. Perhaps most remarkable were his 49 pages of algorithms for playing chess, a game in which he was not an expert.

If a computer scientist had found Zuse's description of Plankalkül in the early 1950s, the single aspect of the language that would have hindered its implementation as defined would have been the notation. Each statement consisted of either two or three lines of code. The first line was most like the statements of current languages. The second line, which was optional, contained the subscripts of the array references in the first line. The same method of indicating subscripts was used by Charles Babbage in programs for his Analytical Engine in the middle of the nineteenth century. The last line of each Plankalkül statement contained the type names for the variables mentioned in the first line. This notation is quite intimidating when first seen.

The following example assignment statement, which assigns the value of the expression `A[4] +1` to `A[5]`, illustrates this notation. The row labeled `V` is for subscripts, and the row labeled `S` is for the data types. In this example, `1.n` means an integer of *n* bits:

```
   | A + 1 => A
V  | 4       5
S  | 1.n     1.n
```

We can only speculate on the direction that programming language design might have taken if Zuse's work had been widely known in 1945 or even 1950. It is also interesting to consider how his work might have been different had he done it in a peaceful environment surrounded by other scientists, rather than in Germany in 1945 in virtual isolation.

2.2 Pseudocodes

First, note that the word *pseudocode* is used here in a different sense than its contemporary meaning. We call the languages discussed in this section pseudocodes because that's what they were named at the time they were developed and used (the late 1940s and early 1950s). However, they are clearly not pseudocodes in the contemporary sense.

The computers that became available in the late 1940s and early 1950s were far less usable than those of today. In addition to being slow, unreliable, expensive, and having extremely small memories, the machines of that time were difficult to program because of the lack of supporting software.

There were no high-level programming languages or even assembly languages, so programming was done in machine code, which is both tedious and

error prone. Among its problems is the use of numeric codes for specifying instructions. For example, an ADD instruction might be specified by the code 14 rather than a connotative textual name, even if only a single letter. This makes programs very difficult to read. A more serious problem is absolute addressing, which makes program modification tedious and error prone. For example, suppose we have a machine language program stored in memory. Many of the instructions in such a program refer to other locations within the program, usually to reference data or to indicate the targets of branch instructions. Inserting an instruction at any position in the program other than at the end invalidates the correctness of all instructions that refer to addresses beyond the insertion point, because those addresses must be increased to make room for the new instruction. To make the addition correctly, all instructions that refer to addresses that follow the addition must be found and modified. A similar problem occurs with deletion of an instruction. In this case, however, machine languages often include a "no operation" instruction that can replace deleted instructions, thereby avoiding the problem.

These are standard problems with all machine languages and were the primary motivations for inventing assemblers and assembly languages. In addition, most programming problems of that time were numerical and required floating-point arithmetic operations and indexing of some sort to allow the convenient use of arrays. Neither of these capabilities, however, was included in the architecture of the computers of the late 1940s and early 1950s. These deficiencies naturally led to the development of somewhat higher-level languages.

2.2.1 Short Code

The first of these new languages, named Short Code, was developed by John Mauchly in 1949 for the BINAC computer, which was one of the first successful stored-program electronic computers. Short Code was later transferred to a UNIVAC I computer (the first commercial electronic computer sold in the United States) and, for several years, was one of the primary means of programming those machines. Although little is known of the original Short Code because its complete description was never published, a programming manual for the UNIVAC I version did survive (Remington-Rand, 1952). It is safe to assume that the two versions were very similar.

The words of the UNIVAC I's memory had 72 bits, grouped as 12 six-bit bytes. Short Code consisted of coded versions of mathematical expressions that were to be evaluated. The codes were byte-pair values, and many equations could be coded in a word. The following operation codes were included:

```
01  -     06  abs value    1n  (n+2)nd power
02  )     07  +            2n  (n+2)nd root
03  =     08  pause        4n  if <= n
04  /     09  (            58  print and tab
```

Variables were named with byte-pair codes, as were locations to be used as constants. For example, `X0` and `Y0` could be variables. The statement

```
X0 = SQRT(ABS(Y0))
```

would be coded in a word as `00 X0 03 20 06 Y0`. The initial `00` was used as padding to fill the word. Interestingly, there was no multiplication code; multiplication was indicated by simply placing the two operands next to each other, as in algebra.

Short Code was not translated to machine code; rather, it was implemented with a pure interpreter. At the time, this process was called *automatic programming*. It clearly simplified the programming process, but at the expense of execution time. Short Code interpretation was approximately 50 times slower than machine code.

2.2.2 Speedcoding

In other places, interpretive systems were being developed that extended machine languages to include floating-point operations. The Speedcoding system developed by John Backus for the IBM 701 is an example of such a system (Backus, 1954). The Speedcoding interpreter effectively converted the 701 to a virtual three-address floating-point calculator. The system included pseudoinstructions for the four arithmetic operations on floating-point data, as well as operations such as square root, sine, arc tangent, exponent, and logarithm. Conditional and unconditional branches and input/output conversions were also part of the virtual architecture. To get an idea of the limitations of such systems, consider that the remaining usable memory after loading the interpreter was only 700 words and that the add instruction took 4.2 milliseconds to execute. On the other hand, Speedcoding included the novel facility of automatically incrementing address registers. This facility did not appear in hardware until the UNIVAC 1107 computers of 1962. Because of such features, matrix multiplication could be done in 12 Speedcoding instructions. Backus claimed that problems that could take two weeks to program in machine code could be programmed in a few hours using Speedcoding.

2.2.3 The UNIVAC "Compiling" System

Between 1951 and 1953, a team led by Grace Hopper at UNIVAC developed a series of "compiling" systems named A-0, A-1, and A-2 that expanded a pseudocode into machine code subprograms in the same way as macros are expanded into assembly language. The pseudocode source for these "compilers" was still quite primitive, although even this was a great improvement over machine code because it made source programs much shorter. Wilkes (1952) independently suggested a similar process.

2.2.4 Related Work

Other means of easing the task of programming were being developed at about the same time. At Cambridge University, David J. Wheeler (1950) developed a method of using blocks of relocatable addresses to solve, at least partially, the problem of absolute addressing, and later, Maurice V. Wilkes (also at Cambridge) extended the idea to design an assembly program that could combine chosen subroutines and allocate storage (Wilkes et al., 1951, 1957). This was indeed an important and fundamental advance.

We should also mention that assembly languages, which are quite different from the pseudocodes discussed, evolved during the early 1950s. However, they had little impact on the design of high-level languages.

2.3 The IBM 704 and Fortran

Certainly one of the greatest single advances in computing came with the introduction of the IBM 704 in 1954, in large measure because its capabilities prompted the development of Fortran. One could argue that if it had not been IBM with the 704 and Fortran, it would soon thereafter have been some other organization with a similar computer and related high-level language. However, IBM was the first with both the foresight and the resources to undertake these developments.

2.3.1 Historical Background

One of the primary reasons why the slowness of interpretive systems was tolerated from the late 1940s to the mid-1950s was the lack of floating-point hardware in the available computers. All floating-point operations had to be simulated in software, a very time-consuming process. Because so much processor time was spent in software floating-point processing, the overhead of interpretation and the simulation of indexing were relatively insignificant. As long as floating-point had to be done by software, interpretation was an acceptable expense. However, many programmers of that time never used interpretive systems, preferring the efficiency of hand-coded machine (or assembly) language. The announcement of the IBM 704 system, with both indexing and floating-point instructions in hardware, heralded the end of the interpretive era, at least for scientific computation. The inclusion of floating-point hardware removed the hiding place for the cost of interpretation.

Although Fortran is often credited with being the first compiled high-level language, the question of who deserves credit for implementing the first such language is somewhat open. Knuth and Pardo (1977) give the credit to Alick E. Glennie for his Autocode compiler for the Manchester Mark I computer. Glennie developed the compiler at Fort Halstead, Royal Armaments Research Establishment, in England. The compiler was operational by September 1952. However, according to John Backus (Wexelblat, 1981, p. 26),

Glennie's Autocode was so low level and machine oriented that it should not be considered a compiled system. Backus gives the credit to Laning and Zierler at the Massachusetts Institute of Technology.

The Laning and Zierler system (Laning and Zierler, 1954) was the first algebraic translation system to be implemented. By algebraic, we mean that it translated arithmetic expressions, used separately coded subprograms to compute transcendental functions (e.g., sine and logarithm), and included arrays. The system was implemented on the MIT Whirlwind computer, in experimental prototype form, in the summer of 1952 and in a more usable form by May 1953. The translator generated a subroutine call to code each formula, or expression, in the program. The source language was easy to read, and the only actual machine instructions included were for branching. Although this work preceded the work on Fortran, it never escaped MIT.

In spite of these earlier works, the first widely accepted compiled high-level language was Fortran. The following subsections chronicle this important development.

2.3.2 Design Process

Even before the 704 system was announced in May 1954, plans were begun for Fortran. By November 1954, John Backus and his group at IBM had produced the report titled "The IBM Mathematical FORmula TRANslating System: FORTRAN" (IBM, 1954). This document described the first version of Fortran, which we refer to as Fortran 0, prior to its implementation. It also boldly stated that Fortran would provide the efficiency of hand-coded programs and the ease of programming of the interpretive pseudocode systems. In another burst of optimism, the document stated that Fortran would eliminate coding errors and the debugging process. Based on this premise, the first Fortran compiler included little syntax error checking.

The environment in which Fortran was developed was as follows: (1) Computers had small memories and were slow and relatively unreliable; (2) the primary use of computers was for scientific computations; (3) there were no existing efficient and effective ways to program computers; and (4) because of the high cost of computers compared to the cost of programmers, speed of the generated object code was the primary goal of the first Fortran compilers. The characteristics of the early versions of Fortran follow directly from this environment.

2.3.3 Fortran I Overview

Fortran 0 was modified during the implementation period, which began in January 1955 and continued until the release of the compiler in April 1957. The implemented language, which we call Fortran I, is described in the first Fortran *Programmer's Reference Manual*, published in October 1956 (IBM, 1956). Fortran I included input/output formatting, variable names of up to six characters (it had been just two in Fortran 0), user-defined subroutines, although they

could not be separately compiled, the `If` selection statement, and the `Do` loop statement.

All of Fortran I's control statements were based on 704 instructions. It is not clear whether the 704 designers dictated the control statement design of Fortran I or whether the designers of Fortran I suggested these instructions to the 704 designers.

There were no data-typing statements in the Fortran I language. Variables whose names began with `I`, `J`, `K`, `L`, `M`, and `N` were implicitly integer type, and all others were implicitly floating-point. The choice of the letters for this convention was based on the fact that at that time scientists and engineers used letters as variable subscripts, usually *i*, *j*, and *k*. In a gesture of generosity, Fortran's designers threw in the three additional letters.

The most audacious claim made by the Fortran development group during the design of the language was that the machine code produced by the compiler would be about half as efficient as what could be produced by hand.[1] This, more than anything else, made skeptics of potential users and prevented a great deal of interest in Fortran before its actual release. To almost everyone's surprise, however, the Fortran development group nearly achieved its goal in efficiency. The largest part of the 18 worker-years of effort used to construct the first compiler had been spent on optimization, and the results were remarkably effective.

The early success of Fortran is shown by the results of a survey made in April 1958. At that time, roughly half of the code being written for 704s was being written in Fortran, in spite of the skepticism of most of the programming world only a year earlier.

2.3.4 Fortran II

The Fortran II compiler was distributed in the spring of 1958. It fixed many of the bugs in the Fortran I compilation system and added some significant features to the language, the most important being the independent compilation of subroutines. Without independent compilation, any change in a program required that the entire program be recompiled. Fortran I's lack of independent-compilation capability, coupled with the poor reliability of the 704, placed a practical restriction on the length of programs to about 300 to 400 lines (Wexelblat, 1981, p. 68). Longer programs had a poor chance of being compiled completely before a machine failure occurred. The capability of including precompiled machine language versions of subprograms shortened the compilation process considerably and made it practical to develop much larger programs.

1. In fact, the Fortran team believed that the code generated by their compiler could be no less than half as fast as handwritten machine code, or the language would not be adopted by users.

2.3.5 Fortrans IV, 77, 90, 95, 2003, and 2008

A Fortran III was developed, but it was never widely distributed. Fortran IV, however, became one of the most widely used programming languages of its time. It evolved over the period 1960 to 1962 and was standardized as Fortran 66 (ANSI, 1966), although that name was rarely used. Fortran IV was an improvement over Fortran II in many ways. Among its most important additions were explicit type declarations for variables, a logical `If` construct, and the capability of passing subprograms as parameters to other subprograms.

Fortran IV was replaced by Fortran 77, which became the new standard in 1978 (ANSI, 1978a). Fortran 77 retained most of the features of Fortran IV and added character string handling, logical loop control statements, and an `If` with an optional `Else` clause.

Fortran 90 (ANSI, 1992) was dramatically different from Fortran 77. The most significant additions were dynamic arrays, records, pointers, a multiple selection statement, and modules. In addition, Fortran 90 subprograms could be recursively called.

A new concept that was included in the Fortran 90 definition was that of removing some language features from earlier versions. While Fortran 90 included all of the features of Fortran 77, the language definition included a list of constructs that were recommended for removal in the next version of the language.

Fortran 90 included two simple syntactic changes that altered the appearance of both programs and the literature describing the language. First, the required fixed format of code, which required the use of specific character positions for specific parts of statements, was dropped. For example, statement labels could appear only in the first five positions and statements could not begin before the seventh position. This rigid formatting of code was designed around the use of punch cards. The second change was that the official spelling of FORTRAN became Fortran. This change was accompanied by the change in convention of using all uppercase letters for keywords and identifiers in Fortran programs. The new convention was that only the first letter of keywords and identifiers would be uppercase.

Fortran 95 (INCITS/ISO/IEC, 1997) continued the evolution of the language, but only a few changes were made. Among other things, a new iteration construct, `Forall`, was added to ease the task of parallelizing Fortran programs.

Fortran 2003 (Metcalf et al., 2004), added support for object-oriented programming, parameterized derived types, procedure pointers, and interoperability with the C programming language.

The latest version of Fortran, Fortran 2008 (ISO/IEC 1539-1, 2010) added support for blocks to define local scopes, co-arrays, which provide a parallel execution model, and the `DO CONCURRENT` construct, to specify loops without interdependencies.

2.3.6 Evaluation

The original Fortran design team thought of language design only as a necessary prelude to the critical task of designing the translator. Furthermore, it never occurred to them that Fortran would be used on computers not

manufactured by IBM. Indeed, they were forced to consider building Fortran compilers for other IBM machines only because the successor to the 704, the 709, was announced before the 704 Fortran compiler was released. The effect that Fortran has had on the use of computers, along with the fact that all subsequent programming languages owe a debt to Fortran, is indeed impressive in light of the modest goals of its designers.

One of the features of Fortran I, and all of its successors before 90, that allows highly optimizing compilers was that the types and storage for all variables are fixed before run time. No new variables or space could be allocated during execution time. This was a sacrifice of flexibility to simplicity and efficiency. It eliminated the possibility of recursive subprograms and made it difficult to implement data structures that grow or change shape dynamically. Of course, the kinds of programs that were being built at the time of the development of the early versions of Fortran were primarily numerical in nature and were simple in comparison with more recent software projects. Therefore, the sacrifice was not a great one.

The overall success of Fortran is difficult to overstate: It dramatically changed the way computers are used. This is, of course, in large part due to its being the first widely used high-level language. In comparison with concepts and languages developed later, early versions of Fortran suffer in a variety of ways, as should be expected. After all, it would not be fair to compare the performance and comfort of a 1910 Model T Ford with the performance and comfort of a 2013 Ford Mustang. Nevertheless, in spite of the inadequacies of Fortran, the momentum of the huge investment in Fortran software, among other factors, has kept it in use for more than a half century.

Alan Perlis, one of the designers of ALGOL 60, said of Fortran in 1978, "Fortran is the *lingua franca* of the computing world. It is the language of the streets in the best sense of the word, not in the prostitutional sense of the word. And it has survived and will survive because it has turned out to be a remarkably useful part of a very vital commerce" (Wexelblat, 1981, p. 161).

The following is an example of a Fortran 95 program:

```
! Fortran 95 Example program
!  Input:  An integer, List_Len, where List_Len is less
!          than 100, followed by List_Len-Integer values
!  Output: The number of input values that are greater
!          than the average of all input values
Implicit none
Integer Dimension(99) :: Int_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Read input data into an array and compute its sum
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
```

```
   End Do
! Compute the average
   Average = Sum / List_Len
! Count the values that are greater than the average
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
      End If
   End Do
! Print the result
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

2.4 Functional Programming: LISP

The first functional programming language was invented to provide language features for list processing, the need for which grew out of the first applications in the area of artificial intelligence (AI).

2.4.1 The Beginnings of Artificial Intelligence and List Processing

Interest in AI appeared in the mid-1950s in a number of places. Some of this interest grew out of linguistics, some from psychology, and some from mathematics. Linguists were concerned with natural language processing. Psychologists were interested in modeling human information storage and retrieval, as well as other fundamental processes of the brain. Mathematicians were interested in mechanizing certain intelligent processes, such as theorem proving. All of these investigations arrived at the same conclusion: Some method must be developed to allow computers to process symbolic data in linked lists. At the time, most computation was on numeric data in arrays.

The concept of list processing was developed by Allen Newell, J. C. Shaw, and Herbert Simon at the RAND Corporation. It was first published in a classic paper that describes one of the first AI programs, the Logic Theorist,[2] and a language in which it could be implemented (Newell and Simon, 1956). The language, named IPL-I (Information Processing Language I), was never implemented. The next version, IPL-II, was implemented on a RAND Johnniac computer. Development of IPL continued until 1960, when the description of IPL-V was published (Newell and Tonge, 1960). The low level of the IPL languages prevented their widespread use. They were actually assembly languages for a hypothetical computer, implemented with an interpreter, in which

2. Logic Theorist discovered proofs for theorems in propositional calculus.

list-processing instructions were included. Another factor that kept the IPL languages from becoming popular was their implementation on the obscure Johnniac machine.

The contributions of the IPL languages were in their list design and their demonstration that list processing was feasible and useful.

IBM became interested in AI in the mid-1950s and chose theorem proving as a demonstration area. At the time, the Fortran project was still underway. The high cost of the Fortran I compiler convinced IBM that their list processing should be attached to Fortran, rather than in the form of a new language. Thus, the Fortran List Processing Language (FLPL) was designed and implemented as an extension to Fortran. FLPL was used to construct a theorem prover for plane geometry, which was then considered the easiest area for mechanical theorem proving.

2.4.2 LISP Design Process

John McCarthy of MIT took a summer position at the IBM Information Research Department in 1958. His goal for the summer was to investigate symbolic computations and to develop a set of requirements for doing such computations. As a pilot example problem area, he chose differentiation of algebraic expressions. From this study came a list of language requirements. Among them were the control flow methods of mathematical functions: recursion and conditional expressions. The only available high-level language of the time, Fortran I, had neither of these.

Another requirement that grew from the symbolic-differentiation investigation was the need for dynamically allocated linked lists and some kind of implicit deallocation of abandoned lists. McCarthy simply would not allow his elegant algorithm for differentiation to be cluttered with explicit deallocation statements.

Because FLPL did not support recursion, conditional expressions, dynamic storage allocation, or implicit deallocation, it was clear to McCarthy that a new language was needed.

When McCarthy returned to MIT in the fall of 1958, he and Marvin Minsky formed the MIT AI Project, with funding from the Research Laboratory for Electronics. The first important effort of the project was to produce a software system for list processing. It was to be used initially to implement a program proposed by McCarthy called the Advice Taker.[3] This application became the impetus for the development of the list-processing language LISP. The first version of LISP is sometimes called "pure LISP" because it is a purely functional language. In the following section, we describe the development of pure LISP.

---

3. Advice Taker represented information with sentences written in a formal language and used a logical inferencing process to decide what to do.

2.4.3 Language Overview

2.4.3.1 Data Structures

Pure LISP has only two kinds of data structures: atoms and lists. Atoms are either symbols, which have the form of identifiers, or numeric literals. The concept of storing symbolic information in linked lists is natural and was used in IPL-II. Such structures allow insertions and deletions at any point, operations that were then thought to be a necessary part of list processing. It was eventually determined, however, that LISP programs rarely require these operations.

Lists are specified by delimiting their elements with parentheses. Simple lists, in which elements are restricted to atoms, have the form

```
(A B C D)
```

Nested list structures are also specified by parentheses. For example, the list

```
(A (B C) D (E (F G)))
```

is composed of four elements. The first is the atom `A`; the second is the sublist `(B C)`; the third is the atom `D`; the fourth is the sublist `(E (F G))`, which has as its second element the sublist `(F G)`.

Internally, lists are stored as single-linked list structures, in which each node has two pointers and represents a list element. A node containing an atom has its first pointer pointing to some representation of the atom, such as its symbol or numeric value, or a pointer to a sublist. A node for a sublist element has its first pointer pointing to the first node of the sublist. In both cases, the second pointer of a node points to the next element of the list. A list is referenced by a pointer to its first element.

The internal representations of the two lists shown earlier are depicted in Figure 2.2. Note that the elements of a list are shown horizontally. The last element of a list has no successor, so its link is `NIL`, which is represented in Figure 2.2 as a diagonal line in the element. Sublists are shown with the same structure.

2.4.3.2 Processes in Functional Programming

LISP was designed as a functional programming language. All computation in a purely functional program is accomplished by applying functions to arguments. Neither the assignment statements nor the variables that abound in imperative language programs are necessary in functional language programs. Furthermore, repetitive processes can be specified with recursive function calls, making iteration (loops) unnecessary. These basic concepts of functional programming make it significantly different from programming in an imperative language.

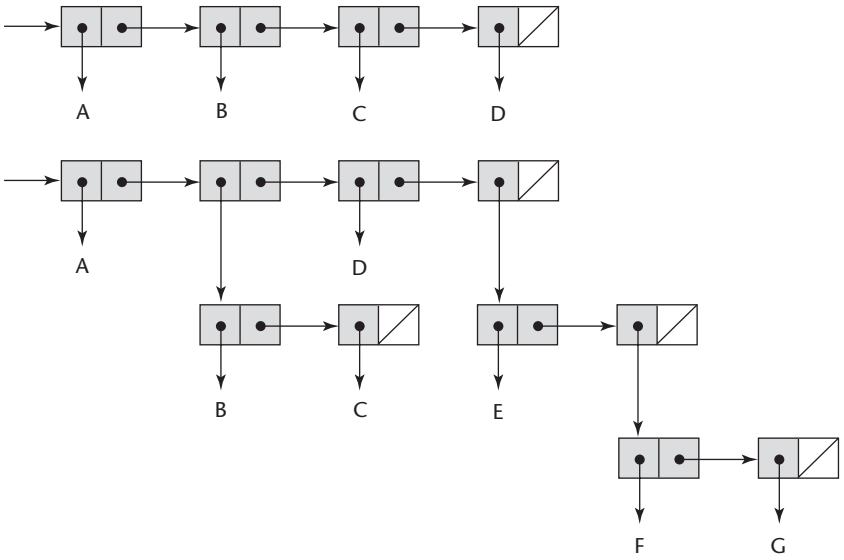


Figure 2.2

Internal representation of two LISP lists

2.4.3.3 The Syntax of LISP

LISP is very different from the imperative languages, both because it is a functional programming language and because the appearance of LISP programs is so different from those in languages like Java or C++. For example, the syntax of Java is a complicated mixture of English and algebra, while LISP's syntax is a model of simplicity. Program code and data have exactly the same form: parenthesized lists. Consider again the list

```
(A B C D)
```

When interpreted as data, it is a list of four elements. When viewed as code, it is the application of the function named `A` to the three parameters `B`, `C`, and `D`.

2.4.4 Evaluation

LISP completely dominated AI applications for a quarter century. Much of the cause of LISP's reputation for being highly inefficient has been eliminated. Many contemporary implementations are compiled, and the resulting code is much faster than running the source code on an interpreter. In addition to its success in AI, LISP pioneered functional programming, which has proven to be a lively area of research in programming languages. As stated in Chapter 1, many programming language researchers believe functional programming is a much better approach to software development than procedural programming using imperative languages.

The following is an example of a LISP program:

```
;  LISP Example function
;  The following code defines a LISP predicate function
;  that  takes two lists as arguments and returns True
;  if the two lists are equal, and NIL (false) otherwise
   (DEFUN equal_lists (lis1 lis2)
    (COND
      ((ATOM lis1) (EQ lis1 lis2))
      ((ATOM lis2) NIL)
      ((equal_lists (CAR lis1) (CAR lis2))
                  (equal_lists (CDR lis1) (CDR lis2)))
      (T NIL)
    )
)
```

2.4.5 Two Descendants of LISP

Two dialects of LISP are now widely used, Scheme and Common LISP. These are briefly discussed in the following subsections.

2.4.5.1 Scheme

The Scheme language emerged from MIT in the mid-1970s (Dybvig, 2003). It is characterized by its small size, its exclusive use of static scoping (discussed in Chapter 5), and its treatment of functions as first-class entities. As first-class entities, Scheme functions can be assigned to variables, passed as parameters, and returned as the values of function applications. They can also be the elements of lists. Early versions of LISP did not provide all of these capabilities, nor did they use static scoping.

As a small language with simple syntax and semantics, Scheme is well suited to educational applications, such as courses in functional programming and general introductions to programming. Scheme is described in some detail in Chapter 15.

2.4.5.2 Common LISP

During the 1970s and early 1980s, a large number of different dialects of LISP were developed and used. This led to the familiar problem of lack of portability among programs written in the various dialects. Common LISP (Graham, 1996) was created in an effort to rectify this situation. Common LISP was designed by combining the features of several dialects of LISP developed in the early 1980s, including Scheme, into a single language. Being such an amalgam, Common LISP is a relatively large and complex language. Its basis, however, is pure LISP, so its syntax, primitive functions, and fundamental nature come from that language.

Recognizing the flexibility provided by dynamic scoping as well as the simplicity of static scoping, Common LISP allows both. The default scoping for variables is static, but by declaring a variable to be `special`, that variable becomes dynamically scoped.

Common LISP has a large number of data types and structures, including records, arrays, complex numbers, and character strings. It also has a form of packages for modularizing collections of functions and data providing access control.

Common LISP is further described in Chapter 15.

2.4.6 Related Languages

ML (*M*eta*L*anguage; Ullman, 1998) was originally designed in the 1980s by Robin Milner at the University of Edinburgh as a metalanguage for a program verification system named Logic for Computable Functions (LCF; Milner et al., 1990). ML is primarily a functional language, but it also supports imperative programming. Unlike LISP and Scheme, the type of every variable and expression in ML can be determined at compile time. Types are associated with objects rather than names. Types of names and expressions are inferred from their context.

Unlike LISP and Scheme, ML does not use the parenthesized functional syntax that originated with lambda expressions. Rather, the syntax of ML resembles that of the imperative languages, such as Java and C++.

Miranda was developed by David Turner (1986) at the University of Kent in Canterbury, England, in the early 1980s. Miranda is based partly on the languages ML, SASL, and KRC. Haskell (Hudak and Fasel, 1992) is based in large part on Miranda. Like Miranda, it is a purely functional language, having no variables and no assignment statement. Another distinguishing characteristic of Haskell is its use of lazy evaluation. This means that no expression is evaluated until its value is required. This leads to some surprising capabilities in the language.

Caml (Cousineau et al., 1998) and its dialect that supports object-oriented programming, OCaml (Smith, 2006), descended from ML and Haskell. Finally, F# is a relatively new typed language based directly on OCaml. F# (Syme et al., 2010) is a .NET language with direct access to the whole .NET library. Being a .NET language also means it can smoothly interoperate with any other .NET language. F# supports both functional programming and procedural programming. It also fully supports object-oriented programming.

ML, Haskell, and F# are further discussed in Chapter 15.

2.5 The First Step Toward Sophistication: ALGOL 60

ALGOL 60 has had much influence on subsequent programming languages and is therefore of central importance in any historical study of languages.

2.5.1 Historical Background

ALGOL 60 was the result of efforts to design a universal programming language for scientific applications. By late 1954, the Laning and Zierler algebraic system had been in operation for over a year, and the first report on Fortran had been published. Fortran became a reality in 1957, and several other high-level languages were being developed. Most notable among them were IT, which was designed by Alan Perlis at Carnegie Tech, and two languages for the UNIVAC computers, MATH-MATIC and UNICODE. The proliferation of languages made program sharing among users difficult. Furthermore, the new languages were all growing up around single architectures, some for UNIVAC computers and some for IBM 700-series machines. In response to this blossoming of machine-dependent languages, several major computer user groups in the United States, including SHARE (the IBM scientific user group) and USE (UNIVAC Scientific Exchange, the large-scale UNIVAC scientific user group), submitted a petition to the Association for Computing Machinery (ACM) on May 10, 1957, to form a committee to study and recommend action to create a machine-independent scientific programming language. Although Fortran might have been a candidate, it could not become a universal language, because at the time it was solely owned by IBM.

Previously, in 1955, GAMM (a German acronym for Society for Applied Mathematics and Mechanics) had formed a committee to design one universal, machine-independent algorithmic language. The desire for this new language was in part due to the Europeans' fear of being dominated by IBM. By late 1957, however, the appearance of several high-level languages in the United States convinced the GAMM subcommittee that their effort had to be widened to include the Americans, and a letter of invitation was sent to ACM. In April 1958, after Fritz Bauer of GAMM presented the formal proposal to ACM, the two groups officially agreed to a joint language design project.

2.5.2 Early Design Process

GAMM and ACM each sent four members to the first design meeting. The meeting, which was held in Zurich from May 27 to June 1, 1958, began with the following goals for the new language:

- The syntax of the language should be as close as possible to standard mathematical notation, and programs written in it should be readable with little further explanation.
- It should be possible to use the language for the description of algorithms in printed publications.
- Programs in the new language must be mechanically translatable into machine language.

The first goal indicated that the new language was to be used for scientific programming, which was the primary computer application area at that time. The second was something entirely new to the computing business. The last goal is an obvious necessity for any programming language.

The Zurich meeting succeeded in producing a language that met the stated goals, but the design process required innumerable compromises, both among individuals and between the two sides of the Atlantic. In some cases, the compromises were not so much over great issues as they were over spheres of influence. The question of whether to use a comma (the European method) or a period (the American method) for a decimal point is one example.

2.5.3 ALGOL 58 Overview

The language designed at the Zurich meeting was named the International Algorithmic Language (IAL). It was suggested during the design that the language be named ALGOL, for ALGOrithmic Language, but the name was rejected because it did not reflect the international scope of the committee. During the following year, however, the name was changed to ALGOL, and the language subsequently became known as ALGOL 58.

In many ways, ALGOL 58 was a descendant of Fortran, which is quite natural. It generalized many of Fortran's features and added several new constructs and concepts. Some of the generalizations had to do with the goal of not tying the language to any particular machine, and others were attempts to make the language more flexible and powerful. A rare combination of simplicity and elegance emerged from the effort.

ALGOL 58 formalized the concept of data type, although only variables that were not floating-point required explicit declaration. It added the idea of compound statements, which most subsequent languages incorporated. Some features of Fortran that were generalized were the following: Identifiers were allowed to have any length, as opposed to Fortran I's restriction to six or fewer characters; any number of array dimensions was allowed, unlike Fortran I's limitation to no more than three; the lower bound of arrays could be specified by the programmer, whereas in Fortran it was implicitly 1; nested selection statements were allowed, which was not the case in Fortran I.

ALGOL 58 acquired the assignment operator in a rather unusual way. Zuse used the form

expression => variable

for the assignment statement in Plankalkül. Although Plankalkül had not yet been published, some of the European members of the ALGOL 58 committee were familiar with the language. The committee dabbled with the Plankalkül assignment form but, because of arguments about character set limitations,[4] the greater-than symbol was changed to a colon. Then, largely at the insistence of the Americans, the whole statement was turned around to the Fortran form

variable := expression

The Europeans preferred the opposite form, but that would be the reverse of Fortran.

4. The card punches of that time did not include the greater-than symbol.

2.5.4 Reception of the ALGOL 58 Report

In December 1958, publication of the ALGOL 58 report (Perlis and Samelson, 1958) was greeted with a good deal of enthusiasm. In the United States, the new language was viewed more as a collection of ideas for programming language design than as a universal standard language. Actually, the ALGOL 58 report was not meant to be a finished product but rather a preliminary document for international discussion. Nevertheless, three major design and implementation efforts used the report as their basis. At the University of Michigan, the MAD language was born (Arden et al., 1961). The U.S. Naval Electronics Group produced the NELIAC language (Huskey et al., 1963). At System Development Corporation, JOVIAL was designed and implemented (Shaw, 1963). JOVIAL, an acronym for Jules' Own Version of the International Algebraic Language, represents the only language based on ALGOL 58 to achieve widespread use (Jules was Jules I. Schwartz, one of JOVIAL's designers). JOVIAL became widely used because it was the official scientific language for the U.S. Air Force for a quarter century.

The rest of the U.S. computing community was not so kind to the new language. At first, both IBM and its major scientific user group, SHARE, seemed to embrace ALGOL 58. IBM began an implementation shortly after the report was published, and SHARE formed a subcommittee, SHARE IAL, to study the language. The subcommittee subsequently recommended that ACM standardize ALGOL 58 and that IBM implement it for all of the 700-series computers. The enthusiasm was short-lived, however. By the spring of 1959, both IBM and SHARE, through their Fortran experience, had had enough of the pain and expense of getting a new language started, both in terms of developing and using the first-generation compilers and in terms of training users in the new language and persuading them to use it. By the middle of 1959, both IBM and SHARE had developed such a vested interest in Fortran that they decided to retain it as *the* scientific language for the IBM 700-series machines, thereby abandoning ALGOL 58.

2.5.5 ALGOL 60 Design Process

During 1959, ALGOL 58 was furiously debated in both Europe and the United States. Large numbers of suggested modifications and additions were published in the European *ALGOL Bulletin* and in *Communications of the ACM.* One of the most important events of 1959 was the presentation of the work of the Zurich committee to the International Conference on Information Processing, for there Backus introduced his new notation for describing the syntax of programming languages, which later became known as BNF (Backus-Naur form). BNF is described in detail in Chapter 3.

In January 1960, the second ALGOL meeting was held, this time in Paris. The purpose of the meeting was to debate the 80 suggestions that had been formally submitted for consideration. Peter Naur of Denmark had become heavily involved in the development of ALGOL, even though he had not been

a member of the Zurich group. It was Naur who created and published the *ALGOL Bulletin*. He spent a good deal of time studying Backus's paper that introduced BNF and decided that BNF should be used to describe formally the results of the 1960 meeting. After making a few relatively minor changes to BNF, he wrote a description of the new proposed language in BNF and handed it out to the members of the 1960 group at the beginning of the meeting.

2.5.6 ALGOL 60 Overview

Although the 1960 meeting lasted only six days, the modifications made to ALGOL 58 were dramatic. Among the most important new developments were the following:

- The concept of block structure was introduced. This allowed the programmer to localize parts of programs by introducing new data environments, or scopes.
- Two different means of passing parameters to subprograms were allowed: pass by value and pass by name.
- Procedures were allowed to be recursive. The ALGOL 58 description was unclear on this issue. Note that although this recursion was new for the imperative languages, LISP had already provided recursive functions in 1959.
- Stack-dynamic arrays were allowed. A stack-dynamic array is one for which the subscript range or ranges are specified by variables, so that the size of the array is set at the time storage is allocated to the array, which happens when the declaration is reached during execution. Stack-dynamic arrays are described in detail in Chapter 6.

Several features that might have had a dramatic impact on the success or failure of the language were proposed and rejected. Most important among these were input and output statements with formatting, which were omitted because they were thought to be machine-dependent.

The ALGOL 60 report was published in May 1960 (Naur, 1960). A number of ambiguities still remained in the language description, and a third meeting was scheduled for April 1962 in Rome to address the problems. At this meeting the group dealt only with problems; no additions to the language were allowed. The results of this meeting were published under the title "Revised Report on the Algorithmic Language ALGOL 60" (Backus et al., 1963).

2.5.7 Evaluation

In some ways, ALGOL 60 was a great success; in other ways, it was a dismal failure. It succeeded in becoming, almost immediately, the only acceptable formal means of communicating algorithms in computing literature, and it remained that for more than 20 years. Every imperative programming language designed since 1960 owes something to ALGOL 60. In fact, most are direct

or indirect descendants; examples include PL/I, SIMULA 67, ALGOL 68, C, Pascal, Ada, C++, Java, and C#.

The ALGOL 58/ALGOL 60 design effort included a long list of firsts. It was the first time that an international group attempted to design a programming language. It was the first language that was designed to be machine independent. It was also the first language whose syntax was formally described. This successful use of the BNF formalism initiated several important fields of computer science: formal languages, parsing theory, and BNF-based compiler design. Finally, the structure of ALGOL 60 affected machine architecture. In the most striking example of this, an extension of the language was used as the systems language of a series of large-scale computers, the Burroughs B5000, B6000, and B7000 machines, which were designed with a hardware stack to implement efficiently the block structure and recursive subprograms of the language.

On the other side of the coin, ALGOL 60 never achieved widespread use in the United States. Even in Europe, where it was more popular than in the United States, it never became the dominant language. There are a number of reasons for its lack of acceptance. For one thing, some of the features of ALGOL 60 turned out to be too flexible; they made understanding difficult and implementation inefficient. The best example of this is the pass-by-name method of passing parameters to subprograms, which is explained in Chapter 9. The difficulties of implementing ALGOL 60 are evidenced by Rutishauser's statement in 1967 that few, if any, implementations included the full ALGOL 60 language (Rutishauser, 1967, p. 8).

The lack of input and output statements in the language was another major reason for its lack of acceptance. Implementation-dependent input/output made programs difficult to port to other computers.

Ironically, one of the most important contributions to computer science associated with ALGOL 60, BNF, was also a factor in its lack of acceptance. Although BNF is now considered a simple and elegant means of syntax description, in 1960 it seemed strange and complicated.

Finally, although there were many other problems, the entrenchment of Fortran among users and the lack of support by IBM were probably the most important factors in ALGOL 60's failure to gain widespread use.

The ALGOL 60 effort was never really complete, in the sense that ambiguities and obscurities were always a part of the language description (Knuth, 1967).

The following is an example of an ALGOL 60 program:

```
comment ALGOL 60 Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all the input values  ;
begin
  integer array intlist [1:99];
```

```
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ∧ (listlen < 100) then
    begin
comment Read input into an array and compute the average;
    for counter := 1 step 1 until listlen do
      begin
      readint (intlist[counter]);
      sum := sum + intlist[counter]
      end;
comment Compute the average;
    average := sum / listlen;
comment Count the input values that are > average;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
        then result := result + 1;
comment Print result;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error—input list length is not legal";
end
```

2.6 Computerizing Business Records: COBOL

The story of COBOL is, in a sense, the opposite of that of ALGOL 60. Although it has been used more than any other programming language, COBOL has had little effect on the design of subsequent languages, except for PL/I. It may still be the most widely used language,[5] although it is difficult to be sure one way or the other. Perhaps the most important reason why COBOL has had little influence is that few have attempted to design a new language for business applications since it appeared. That is due in part to how well COBOL's capabilities meet the needs of its application area. Another reason is that a great deal of growth in business computing over the past 30 years has occurred in small businesses. In these businesses, very little software development has taken place. Instead, most of the software used is purchased as off-the-shelf packages for various general business applications.

5. In the late 1990s, in a study associated with the Y2K problem, it was estimated that there were approximately 800 million lines of COBOL in use in the 22 square miles of Manhattan.

2.6.1 Historical Background

The beginning of COBOL is somewhat similar to that of ALGOL 60, in the sense that the language was designed by a committee of people meeting for relatively short periods of time. At the time, in 1959, the state of business computing was similar to the state of scientific computing several years earlier, when Fortran was being designed. One compiled language for business applications, FLOW-MATIC, had been implemented in 1957, but it belonged to one manufacturer, UNIVAC, and was designed for that company's computers. Another language, AIMACO, was being used by the U.S. Air Force, but it was only a minor variation of FLOW-MATIC. IBM had designed a programming language for business applications, COMTRAN (COMmercial TRANslator), but it had not yet been implemented. Several other language design projects were being planned.

2.6.2 FLOW-MATIC

The origins of FLOW-MATIC are worth at least a brief discussion, because it was the primary progenitor of COBOL. In December 1953, Grace Hopper at Remington-Rand UNIVAC wrote a proposal that was indeed prophetic. It suggested that "mathematical programs should be written in mathematical notation, data processing programs should be written in English statements" (Wexelblat, 1981, p. 16). Unfortunately, in 1953, it was impossible to convince nonprogrammers that a computer could be made to understand English words. It was not until 1955 that a similar proposal had some hope of being funded by UNIVAC management, and even then it took a prototype system to do the final convincing. Part of this selling process involved compiling and running a small program, first using English keywords, then using French keywords, and then using German keywords. This demonstration was considered remarkable by UNIVAC management and was instrumental in their acceptance of Hopper's proposal.

2.6.3 COBOL Design Process

The first formal meeting on the subject of a common language for business applications, which was sponsored by the Department of Defense, was held at the Pentagon on May 28 and 29, 1959 (exactly one year after the Zurich ALGOL meeting). The consensus of the group was that the language, then named CBL (Common Business Language), should have the following general characteristics: Most agreed that it should use English as much as possible, although a few argued for a more mathematical notation. The language must be easy to use, even at the expense of being less powerful, in order to broaden the base of those who could program computers. In addition to making the language easy to use, it was believed that the use of English would allow managers to read programs. Finally, the design should not be overly restricted by the problems of its implementation.

One of the overriding concerns at the meeting was that steps to create this universal language should be taken quickly, as a lot of work was already being done to create other business languages. In addition to the existing languages, RCA and Sylvania were working on their own business applications languages. It was clear that the longer it took to produce a universal language, the more difficult it would be for the language to become widely used. On this basis, it was decided that there should be a quick study of existing languages. For this task, the Short Range Committee was formed.

There were early decisions to separate the statements of the language into two categories—data description and executable operations—and to have statements in these two categories be in different parts of programs. One of the debates of the Short Range Committee was over the inclusion of subscripts. Many committee members argued that subscripts were too complex for the people in data processing, who were thought to be uncomfortable with mathematical notation. Similar arguments revolved around whether arithmetic expressions should be included. The final report of the Short Range Committee, which was completed in December 1959, described the language that was later named COBOL 60.

The language specifications for COBOL 60, published by the Government Printing Office in April 1960 (Department of Defense, 1960), were described as "initial." Revised versions were published in 1961 and 1962 (Department of Defense, 1961, 1962). The language was standardized by the American National Standards Institute (ANSI) group in 1968. The next three revisions were standardized by ANSI in 1974, 1985, and 2002. The language continues to evolve today.

2.6.4 Evaluation

The COBOL language originated a number of novel concepts, some of which eventually appeared in other languages. For example, the `DEFINE` verb of COBOL 60 was the first high-level language construct for macros. More important, hierarchical data structures (records), which first appeared in Plankalkül, were first implemented in COBOL. They have been included in most of the imperative languages designed since then. COBOL was also the first language that allowed names to be truly connotative, because it allowed both long names (up to 30 characters) and word-connector characters (hyphens).

Overall, the data division is the strong part of COBOL's design, whereas the procedure division is relatively weak. Every variable is defined in detail in the data division, including the number of decimal digits and the location of the implied decimal point. File records are also described with this level of detail, as are lines to be output to a printer, which makes COBOL ideal for printing accounting reports. Perhaps the most important weakness of the original procedure division was in its lack of functions. Versions of COBOL prior to the 1974 standard also did not allow subprograms with parameters.

Our final comment on COBOL: It was the first programming language whose use was mandated by the Department of Defense (DoD). This mandate came after its initial development, because COBOL was not designed specifically for the DoD. In spite of its merits, COBOL probably would not have

survived without that mandate. The poor performance of the early compilers simply made the language too expensive to use. Eventually, of course, compilers became more efficient and computers became much faster and cheaper and had much larger memories. Together, these factors allowed COBOL to succeed, inside and outside DoD. Its appearance led to the electronic mechanization of accounting, an important revolution by any measure.

The following is an example of a COBOL program. This program reads a file named BAL-FWD-FILE that contains inventory information about a certain collection of items. Among other things, each item record includes the number currently on hand (BAL-ON-HAND) and the item's reorder point (BAL-REORDER-POINT). The reorder point is the threshold number of items on hand at which more must be ordered. The program produces a list of items that must be reordered as a file named REORDER-LISTING.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE   ASSIGN TO READER.
    SELECT REORDER-LISTING  ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO        PICTURE IS 9(5).
    02 BAL-ITEM-DESC      PICTURE IS X(20).
    02 FILLER             PICTURE IS X(5).
    02 BAL-UNIT-PRICE     PICTURE IS 999V99.
    02 BAL-REORDER-POINT  PICTURE IS 9(5).
    02 BAL-ON-HAND        PICTURE IS 9(5).
    02 BAL-ON-ORDER       PICTURE IS 9(5).
    02 FILLER             PICTURE IS X(30).
FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
```

```
    02 RL-ITEM-NO         PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(5).
    02 RL-ITEM-DESC       PICTURE IS X(20).
    02 FILLER             PICTURE IS X(5).
    02 RL-UNIT-PRICE      PICTURE IS ZZZ.99.
    02 FILLER             PICTURE IS X(5).
    02 RL-AVAILABLE-STOCK PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(5).
    02 RL-REORDER-POINT   PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02 CARD-EOF-SWITCH    PICTURE IS X.
01  WORK-FIELDS.
    02 AVAILABLE-STOCK    PICTURE IS 9(5).

PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
    PERFORM 100-PRODUCE-REORDER-LINE
        UNTIL CARD-EOF-SWITCH IS EQUAL TO "Y".
    CLOSE BAL-FWD-FILE.
    CLOSE REORDER-LISTING.
    STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"]
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        AT END
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
ADD BAL-ON-HAND BAL-ON-ORDER
    GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE             TO REORDER-LINE.
```

```
MOVE BAL-ITEM-NO       TO RL-ITEM-NO.
MOVE BAL-ITEM-DESC     TO RL-ITEM-DESC.
MOVE BAL-UNIT-PRICE    TO RL-UNIT-PRICE.
MOVE AVAILABLE-STOCK   TO RL-AVAILABLE-STOCK.
MOVE BAL-REORDER-POINT TO RL-REORDER-POINT.
WRITE REORDER-LINE.
```

2.7 The Beginnings of Timesharing: BASIC

BASIC (Mather and Waite, 1971) is another programming language that has enjoyed widespread use but has gotten little respect. Like COBOL, it has largely been ignored by computer scientists. Also, like COBOL, in its earliest versions it was inelegant and included only a meager set of control statements.

BASIC was very popular on microcomputers in the late 1970s and early 1980s. This followed directly from two of the main characteristics of early versions of BASIC. It was easy for beginners to learn, especially those who were not science oriented, and its smaller dialects can be implemented on computers with very small memories.[6] When the capabilities of microcomputers grew and other languages were implemented, the use of BASIC waned. A strong resurgence in the use of BASIC began with the appearance of Visual Basic (Microsoft, 1991) in the early 1990s.

2.7.1 Design Process

BASIC (Beginner's All-purpose Symbolic Instruction Code) was originally designed at Dartmouth College (now Dartmouth University) in New Hampshire by two mathematicians, John Kemeny and Thomas Kurtz, who, in the early 1960s, developed compilers for a variety of dialects of Fortran and ALGOL 60. Their science students generally had little trouble learning or using those languages in their studies. However, Dartmouth was primarily a liberal arts institution, where science and engineering students made up only about 25 percent of the student body. It was decided in the spring of 1963 to design a new language especially for liberal arts students. This new language would use terminals as the method of computer access. The goals of the system were as follows:

1. It must be easy for nonscience students to learn and use.
2. It must be "pleasant and friendly."
3. It must provide fast turnaround for homework.

6. Some early microcomputers included BASIC interpreters that resided in 4096 bytes of ROM.

4. It must allow free and private access.
5. It must consider user time more important than computer time.

The last goal was indeed a revolutionary concept. It was based at least partly on the belief that computers would become significantly cheaper as time went on, which of course they did.

The combination of the second, third, and fourth goals led to the time-shared aspect of BASIC. Only with individual access through terminals by numerous simultaneous users could these goals be met in the early 1960s.

In the summer of 1963, Kemeny began work on the compiler for the first version of BASIC, using remote access to a GE 225 computer. Design and coding of the operating system for BASIC began in the fall of 1963. At 4:00 A.M. on May 1, 1964, the first program using the timeshared BASIC was typed in and run. In June, the number of terminals on the system grew to 11, and by the fall it had ballooned to 20.

2.7.2 Language Overview

The original version of BASIC was very small and, oddly, was not interactive: There was no way for an executing program to get input data from the user. Programs were typed in, compiled, and run, in a sort of batch-oriented way. The original BASIC had only 14 different statement types and a single data type—floating-point. Because it was believed that few of the targeted users would appreciate the difference between integer and floating-point types, the type was referred to as "numbers." Overall, it was a very limited language, though quite easy to learn.

2.7.3 Evaluation

The most important aspect of the original BASIC was that it was the first widely used language that was used through terminals connected to a remote computer.[7] Terminals had just begun to be available at that time. Before then, most programs were entered into computers through either punched cards or paper tape.

Much of the design of BASIC came from Fortran, with some minor influence from the syntax of ALGOL 60. Later, it grew in a variety of ways, with little or no effort made to standardize it. The American National Standards Institute issued a Minimal BASIC standard (ANSI, 1978b), but this represented only the bare minimum of language features. In fact, the original BASIC was very similar to Minimal BASIC.

Although it may seem surprising, Digital Equipment Corporation used a rather elaborate version of BASIC named BASIC-PLUS to write significant

7. LISP initially was used through terminals, but it was not widely used in the early 1960s.

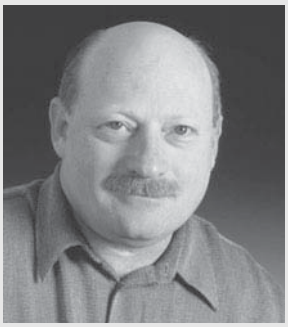
portions of their largest operating system for the PDP-11 minicomputers, RSTS, in the 1970s.

BASIC has been criticized for the poor structure of programs written in it, among other things. By the evaluation criteria discussed in Chapter 1, specifically readability and reliability, the language does indeed fare very poorly. Clearly, the early versions of the language were not meant for and should not have been used for serious programs of any significant size. Later versions are much better suited to such tasks.

The resurgence of BASIC in the 1990s was driven by the appearance of Visual BASIC (VB). VB became widely used in large part because it provided a simple way of building graphical user interfaces (GUIs), hence the name Visual BASIC. Visual Basic .NET, or just VB.NET, is one of Microsoft's .NET languages. Although it is a significant departure from VB, it quickly displaced the older language. Perhaps the most important difference between VB and VB.NET is that VB.NET fully supports object-oriented programming.

The following is an example of a BASIC program:

```
REM  BASIC Example Program
REM  Input:  An integer, listlen, where listlen is less
REM          than 100, followed by listlen-integer values
REM  Output: The number of input values that are greater
REM          than the average of all input values
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF listlen > 0 AND listlen < 100 THEN
REM  Read input into an array and compute the sum
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Compute the average
    average = sum / listlen
REM  Count the number of input values that are > average
    FOR counter = 1 TO listlen
      IF intlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Print the result
    PRINT "The number of values that are > average is:";
          result
  ELSE
    PRINT "Error—input list length is not legal"
  END IF
END
```



User Design and Language Design

ALAN COOPER

Best-selling author of *About Face: The Essentials of User Interface Design,* Alan Cooper also had a large hand in designing what can be touted as the language with the most concern for user interface design, Visual Basic. For him, it all comes down to a vision for humanizing technology.

SOME INFORMATION ON THE BASICS

How did you get started in all of this? I'm a high school dropout with an associate degree in programming from a California community college. My first job was as a programmer for American President Lines (one of the United States' oldest ocean transportation companies) in San Francisco. Except for a few months here and there, I've remained self-employed.

What is your current job? Founder and chairman of Cooper, the company that humanizes technology (www.cooper.com).

What is or was your favorite job? Interaction design consultant.

You are very well known in the fields of language design and user interface design. Any thoughts on designing languages versus designing software, versus designing anything else? It's pretty much the same in the world of software: Know your user.

ABOUT THAT EARLY WINDOWS RELEASE

In the 1980s, you started using Windows and have talked about being lured by its plusses: the graphical user interface support and the dynamically linked library that let you create tools that configured themselves. What about the parts of Windows that you eventually helped shape? I was very impressed by Microsoft's inclusion of support for practical multitasking in Windows. This included dynamic relocation and interprocess communications. MSDOS.exe was the shell program for the first few releases of Windows. It was a terrible program, and I believed that it could be improved dramatically, and I was the guy to do it. In my spare time, I immediately began to write a better shell program than the one Windows came with. I called it Tripod. Microsoft's original shell, MSDOS.exe, was one of the main stumbling blocks to the initial success of Windows. Tripod attempted to solve the problem by being easier to use and to configure.

When was that "Aha!" moment? It wasn't until late in 1987, when I was interviewing a corporate client, that the key design strategy for Tripod popped into my head. As the IS manager explained to me his need to create and publish a wide range of shell solutions to his disparate user base, I realized the conundrum that there is no such thing as an ideal shell. Every user would need their own personal shell, configured to their own needs and skill levels. In an instant, I perceived the solution to the shell design problem: It would be a shell construction set; a tool where each user would be able to construct exactly the shell that he or she needed for a unique mix of applications and training.

What is so compelling about the idea of a shell that can be individualized? Instead of me telling the users what the ideal shell was, they could design their own, personalized ideal shell. With a customizable shell, a programmer would create a shell that was powerful and wide ranging but also somewhat dangerous, whereas an IT manager would create a shell that could be given to a desk clerk that exposed only those few application-specific tools that the clerk used.

> *"MSDOS.exe was the shell program for the first few releases of Windows. It was a terrible program, and I believed that it could be improved dramatically, and I was the guy to do it. In my spare time, I immediately began to write a better shell program than the one Windows came with."*

How did you get from writing a shell program to collaborating with Microsoft? Tripod and Ruby are the same thing. After I signed a deal with Bill Gates, I changed the name of the prototype from Tripod to Ruby. I then used the Ruby prototype as prototypes should be used: as a disposable model for constructing release-quality code. Which is what I did. MS took the release version of Ruby and added QuickBASIC to it, creating VB. All of those original innovations were in Tripod/Ruby.

RUBY AS THE INCUBATOR FOR VISUAL BASIC

Let's revisit your interest in early Windows and that DLL feature. The DLL wasn't a thing, it was a facility in the OS. It allowed a programmer to build code objects that could be linked to at run time as opposed to only at compile time. This is what allowed me to invent the dynamically extensible parts of VB, where controls can be added by third-party vendors.

The Ruby product embodied many significant advances in software design, but two of them stand out as exceptionally successful. As I mentioned, the dynamic linking capability of Windows had always intrigued me, but having the tools and knowing what to do with them were two different things. With Ruby, I finally found two practical uses for dynamic linking, and the original program contained both. First, the language was both installable and could be extended dynamically. Second, the palette of gizmos could be added to dynamically.

Was your language in Ruby the first to have a dynamic linked library and to be linked to a visual front end? As far as I know, yes.

Using a simple example, what would this enable a programmer to do with his or her program? Purchase a control, such as a grid control, from a third-party vendor, install it on his or her computer, and have the grid control appear as an integral part of the language, including the visual programming front end.

Why do they call you "the father of Visual Basic"? Ruby came with a small language, one suited only for executing the dozen or so simple commands that a shell program needs. However, this language was implemented as a chain of DLLs, any number of which could be installed at run time. The internal parser would identify a verb and then pass it along the chain of DLLs until one of them acknowledged that it knew how to process the verb. If all of the DLLs passed, there was a syntax error. From our earliest discussions, both Microsoft and I had entertained the idea of growing the language, possibly even replacing it altogether with a "real" language. C was the candidate most frequently mentioned, but eventually, Microsoft took advantage of this dynamic interface to unplug our little shell language and replace it entirely with QuickBASIC. This new marriage of language to visual front end was static and permanent, and although the original dynamic interface made the coupling possible, it was lost in the process.

SOME FINAL COMMENTS ON NEW IDEAS

In the world of programming and programming tools, including languages and environments, what projects most interest you? I'm interested in creating programming tools that are designed to help users instead of programmers.

What's the most critical rule, famous quote, or design idea to keep in mind? Bridges are not built by engineers. They are built by ironworkers.

Similarly, software programs are not built by engineers. They are built by programmers.

2.8 Everything for Everybody: PL/I

PL/I represents the first large-scale attempt to design a language that could be used for a broad spectrum of application areas. All previous and most subsequent languages have focused on one particular application area, such as science, artificial intelligence, or business.

2.8.1 Historical Background

Like Fortran, PL/I was developed as an IBM product. By the early 1960s, the users of computers in industry had settled into two separate and quite different camps: scientific and business. From the IBM point of view, scientific programmers could use either the large-scale 7090 or the small-scale 1620 IBM computers. This group used floating-point data and arrays extensively. Fortran was the primary language, although some assembly language was also used. They had their own user group, SHARE, and had little contact with anyone who worked on business applications.

For business applications, people used the large 7080 or the small 1401 IBM computers. They needed the decimal and character string data types, as well as elaborate and efficient input and output facilities. They used COBOL, although in early 1963 when the PL/I story begins, the conversion from assembly language to COBOL was far from complete. This category of users also had its own user group, GUIDE, and seldom had contact with scientific users.

In early 1963, IBM planners perceived the beginnings of a change in this situation. The two widely separated computer user groups were moving toward each other in ways that were thought certain to create problems. Scientists began to gather large files of data to be processed. This data required more sophisticated and more efficient input and output facilities. Business applications people began to use regression analysis to build management information systems, which required floating-point data and arrays. It began to appear that computing installations would soon require two separate computers and technical staffs, supporting two very different programming languages.[8]

These perceptions naturally led to the concept of designing a single universal computer that would be capable of doing both floating-point and decimal arithmetic, and therefore both scientific and business applications. Thus was born the concept of the IBM System/360 line of computers. Along with this came the idea of a programming language that could be used for both business and scientific applications. For good measure, features to support systems programming and list processing were thrown in. Therefore, the new language was to replace Fortran, COBOL, LISP, and the systems applications of assembly language.

8. At the time, large computer installations required both full-time hardware and full-time system software maintenance staff.

2.8.2 Design Process

The design effort began when IBM and SHARE formed the Advanced Language Development Committee of the SHARE Fortran Project in October 1963. This new committee quickly met and formed a subcommittee called the 3 × 3 Committee, so named because it had three members from IBM and three from SHARE. The 3 × 3 Committee met for three or four days every other week to design the language.

As with the Short Range Committee for COBOL, the initial design was scheduled for completion in a remarkably short time. Apparently, regardless of the scope of a language design effort, in the early 1960s the prevailing belief was that it could be done in three months. The first version of PL/I, which was then named Fortran VI, was supposed to be completed by December, less than three months after the committee was formed. The committee pleaded successfully on two different occasions for extensions, moving the due date back to January and then to late February 1964.

The initial design concept was that the new language would be an extension of Fortran IV, maintaining compatibility, but that goal was dropped quickly along with the name Fortran VI. Until 1965, the language was known as NPL (New Programming Language). The first published report on NPL was given at the SHARE meeting in March 1964. A more complete description followed in April, and the version that would actually be implemented was published in December 1964 (IBM, 1964) by the compiler group at the IBM Hursley Laboratory in England, which was chosen to do the implementation. In 1965, the name was changed to PL/I to avoid the confusion of the name NPL with the National Physical Laboratory in England. If the compiler had been developed outside the United Kingdom, the name might have remained NPL.

2.8.3 Language Overview

Perhaps the best single-sentence description of PL/I is that it included what were then considered the best parts of ALGOL 60 (recursion and block structure), Fortran IV (separate compilation with communication through global data), and COBOL 60 (data structures, input/output, and report-generating facilities), along with an extensive collection of new constructs, all somehow cobbled together. Because PL/I is no longer a popular language, we will not attempt, even briefly, to discuss all the features of the language, or even its most controversial constructs. Instead, we will mention some of the language's contributions to the pool of knowledge of programming languages.

PL/I was the first programming language to have the following facilities:

- Programs were allowed to create concurrently executing subprograms. Although this was a good idea, it was poorly developed in PL/I.
- It was possible to detect and handle 23 different types of exceptions, or run-time errors.

- Subprograms were allowed to be used recursively, but the capability could be disabled, allowing more efficient linkage for nonrecursive subprograms.
- Pointers were included as a data type.
- Cross-sections of arrays could be referenced. For example, the third row of a matrix could be referenced as if it were a single-dimensioned array.

2.8.4 Evaluation

Any evaluation of PL/I must begin by recognizing the ambitiousness of the design effort. In retrospect, it appears naive to think that so many constructs could have been combined successfully. However, that judgment must be tempered by acknowledging that there was little language design experience at the time. Overall, the design of PL/I was based on the premise that any construct that was useful and could be implemented should be included, with insufficient concern about how a programmer could understand and make effective use of such a collection of constructs and features. Edsger Dijkstra, in his Turing Award Lecture (Dijkstra, 1972), made one of the strongest criticisms of the complexity of PL/I: "I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control."

In addition to the problem with the complexity due to its large size, PL/I suffered from a number of what are now considered to be poorly designed constructs. Among these were pointers, exception handling, and concurrency, although we must point out that in all cases, these constructs had not appeared in any previous language.

In terms of usage, PL/I must be considered at least a partial success. In the 1970s, it enjoyed significant use in both business and scientific applications. It was also widely used during that time as an instructional vehicle in colleges, primarily in several subset forms, such as PL/C (Cornell, 1977) and PL/CS (Conway and Constable, 1976).

The following is an example of a PL/I program:

```
/* PL/I PROGRAM EXAMPLE
 INPUT:  AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
         100, FOLLOWED BY LISTLEN-INTEGER VALUES
 OUTPUT: THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
         THE AVERAGE OF ALL INPUT VALUES   */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
```

```
    DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
    DO COUNTER = 1 TO LISTLEN;
      GET LIST (INTLIST (COUNTER));
      SUM = SUM + INTLIST (COUNTER);
    END;
/* COMPUTE THE AVERAGE */
    AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
    DO COUNTER = 1 TO LISTLEN;
      IF INTLIST (COUNTER) > AVERAGE THEN
        RESULT = RESULT + 1;
    END;
/* PRINT RESULT */
    PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
    PUT LIST (RESULT);
    END;
  ELSE
    PUT SKIP LIST ('ERROR—INPUT LIST LENGTH IS ILLEGAL');
  END PLIEX;
```

2.9 Two Early Dynamic Languages: APL and SNOBOL

The structure of this section is different from that of the other sections because the languages discussed here are very different. Neither APL nor SNOBOL had much influence on later mainstream languages.[9] Some of the interesting features of APL are discussed later in the book.

In appearance and in purpose, APL and SNOBOL are quite different. They share two fundamental characteristics, however: dynamic typing and dynamic storage allocation. Variables in both languages are essentially untyped. A variable acquires a type when it is assigned a value, at which time it assumes the type of the value assigned. Storage is allocated to a variable only when it is assigned a value, because before that there is no way to know the amount of storage that will be needed.

2.9.1 Origins and Characteristics of APL

APL (Brown et al., 1988) was designed around 1960 by Kenneth E. Iverson at IBM. It was not originally designed to be an implemented programming language but rather was intended to be a vehicle for describing computer architecture.

9. However, they have some influence on some nonmainstream languages (J is based on APL, ICON is based on SNOBOL, and AWK is partially based on SNOBOL).

APL was first described in the book from which it gets its name, *A Programming Language* (Iverson, 1962). In the mid-1960s, the first implementation of APL was developed at IBM.

APL has a large number of powerful operators that are specified with a large number of symbols, which created a problem for implementors. Initially, APL was used through IBM printing terminals. These terminals had special print balls that provided the odd character set required by the language. One reason APL has so many operators is that it provides a large number of unit operations on arrays. For example, the transpose of any matrix is done with a single operator. The large collection of operators provides very high expressivity but also makes APL programs difficult to read. Therefore, people think of APL as a language that is best used for "throw-away" programming. Although programs can be written quickly, they should be discarded after use because they are difficult to maintain.

APL has been around for nearly 50 years and is still used today, although not widely. Furthermore, it has not changed a great deal over its lifetime.

2.9.2 Origins and Characteristics of SNOBOL

SNOBOL (pronounced "snowball"; Griswold et al., 1971) was designed in the early 1960s by three people at Bell Laboratories: D. J. Farber, R. E. Griswold, and I. P. Polonsky (Farber et al., 1964). It was designed specifically for text processing. The heart of SNOBOL is a collection of powerful operations for string pattern matching. One of the early applications of SNOBOL was for writing text editors. Because the dynamic nature of SNOBOL makes it slower than alternative languages, it is no longer used for such programs. However, SNOBOL is still a live and supported language that is used for a variety of text-processing tasks in several different application areas.

2.10 The Beginnings of Data Abstraction: SIMULA 67

Although SIMULA 67 never achieved widespread use and had little impact on the programmers and computing of its time, some of the constructs it introduced make it historically important.

2.10.1 Design Process

Two Norwegians, Kristen Nygaard and Ole-Johan Dahl, developed the language SIMULA I between 1962 and 1964 at the Norwegian Computing Center (NCC) in Oslo. They were primarily interested in using computers for simulation but also worked in operations research. SIMULA I was designed exclusively for system simulation and was first implemented in late 1964 on a UNIVAC 1107 computer.

As soon as the SIMULA I implementation was completed, Nygaard and Dahl began efforts to extend the language by adding new features and modifying some existing constructs in order to make the language useful for general-purpose applications. The result of this work was SIMULA 67, whose design was first presented publicly in March 1967 (Dahl and Nygaard, 1967). We will discuss only SIMULA 67, although some of the features of interest in SIMULA 67 are also in SIMULA I.

2.10.2 Language Overview

SIMULA 67 is an extension of ALGOL 60, taking both block structure and the control statements from that language. The primary deficiency of ALGOL 60 (and other languages at that time) for simulation applications was the design of its subprograms. Simulation requires subprograms that are allowed to restart at the position where they previously stopped. Subprograms with this kind of control are known as **coroutines** because the caller and called subprograms have a somewhat equal relationship with each other, rather than the rigid master/slave relationship they have in most imperative languages.

To provide support for coroutines in SIMULA 67, the class construct was developed. This was an important development because the concept of data abstraction began with it. Furthermore, data abstraction provides the foundation for object-oriented programming.

It is interesting to note that the important concept of data abstraction was not developed and attributed to the class construct until 1972, when Hoare (1972) recognized the connection.

2.11 Orthogonal Design: ALGOL 68

ALGOL 68 was the source of several new ideas in language design, some of which were subsequently adopted by other languages. We include it here for that reason, even though it never achieved widespread use in either Europe or the United States.

2.11.1 Design Process

The development of the ALGOL family did not end when the revised report on ALGOL 60 appeared in 1962, although it was six years until the next design iteration was published. The resulting language, ALGOL 68 (van Wijngaarden et al., 1969), was dramatically different from its predecessor.

One of the most interesting innovations of ALGOL 68 was one of its primary design criteria: orthogonality. Recall our discussion of orthogonality in Chapter 1. The use of orthogonality resulted in several innovative features of ALGOL 68, one of which is described in the following section.

2.11.2 Language Overview

One important result of orthogonality in ALGOL 68 was its inclusion of user-defined data types. Earlier languages, such as Fortran, included only a few basic data structures. PL/I included a larger number of data structures, which made it harder to learn and difficult to implement, but it obviously could not provide an appropriate data structure for every need.

The approach of ALGOL 68 to data structures was to provide a few primitive types and structures and allow the user to combine those primitives into a large number of different structures. This provision for user-defined data types was carried over to some extent into all of the major imperative languages designed since then. User-defined data types are valuable because they allow the user to design data abstractions that fit particular problems very closely. All aspects of data types are discussed in Chapter 6.

As another first in the area of data types, ALGOL 68 introduced the kind of dynamic arrays that will be termed *implicit heap-dynamic* in Chapter 5. A dynamic array is one in which the declaration does not specify subscript bounds. Assignments to a dynamic array cause allocation of required storage. In ALGOL 68, dynamic arrays are called **`flex`** arrays.

2.11.3 Evaluation

ALGOL 68 includes a significant number of features that had not been previously used. Its use of orthogonality, which some may argue was overdone, was nevertheless revolutionary.

ALGOL 68 repeated one of the sins of ALGOL 60, however, and it was an important factor in its limited popularity. The language was described using an elegant and concise but also unknown metalanguage. Before one could read the language-describing document (van Wijngaarden et al., 1969), he or she had to learn the new metalanguage, called van Wijngaarden grammars, which were far more complex than BNF. To make matters worse, the designers invented a collection of words to explain the grammar and the language. For example, keywords were called *indicants*, substring extraction was called *trimming*, and the process of a subprogram execution was called a *coercion of deproceduring*, which might be *meek*, *firm*, or something else.

It is natural to contrast the design of PL/I with that of ALGOL 68, because they appeared only a few years apart. ALGOL 68 achieved writability by the principle of orthogonality: a few primitive concepts and the unrestricted use of a few combining mechanisms. PL/I achieved writability by including a large number of fixed constructs. ALGOL 68 extended the elegant simplicity of ALGOL 60, whereas PL/I simply threw together the features of several languages to attain its goals. Of course, it must be remembered that the goal of PL/I was to provide a unified tool for a broad class of problems, whereas ALGOL 68 was targeted to a single class: scientific applications.

PL/I achieved far greater acceptance than ALGOL 68, due largely to IBM's promotional efforts and the problems of understanding and implementing

ALGOL 68. Implementation was a difficult problem for both, but PL/I had the resources of IBM to apply to building a compiler. ALGOL 68 enjoyed no such benefactor.

2.12 Some Early Descendants of the ALGOLs

All imperative languages owe some of their design to ALGOL 60 and/or ALGOL 68. This section discusses some of the early descendants of these languages.

2.12.1 Simplicity by Design: Pascal

2.12.1.1 Historical Background

Niklaus Wirth (Wirth is pronounced "Virt") was a member of the International Federation of Information Processing (IFIP) Working Group 2.1, which was created to continue the development of ALGOL in the mid-1960s. In August 1965, Wirth and C. A. R. ("Tony") Hoare contributed to that effort by presenting to the group a somewhat modest proposal for additions and modifications to ALGOL 60 (Wirth and Hoare, 1966). The majority of the group rejected the proposal as being too small an improvement over ALGOL 60. Instead, a much more complex revision was developed, which eventually became ALGOL 68. Wirth, along with a few other group members, did not believe that the ALGOL 68 report should have been released, based on the complexity of both the language and the metalanguage used to describe it. This position later proved to have some validity because the ALGOL 68 documents, and therefore the language, were indeed found to be challenging by the computing community.

The Wirth and Hoare version of ALGOL 60 was named ALGOL-W. It was implemented at Stanford University and was used primarily as an instructional vehicle, but only at a few universities. The primary contributions of ALGOL-W were the value-result method of passing parameters and the `case` statement for multiple selection. The value-result method is an alternative to ALGOL 60's pass-by-name method. Both are discussed in Chapter 9. The `case` statement is discussed in Chapter 8.

Wirth's next major design effort, again based on ALGOL 60, was his most successful: Pascal.[10] The original published definition of Pascal appeared in 1971 (Wirth, 1971). This version was modified somewhat in the implementation process and is described in Wirth (1973). The features that are often ascribed to Pascal in fact came from earlier languages. For example, user-defined data types were introduced in ALGOL 68, the `case` statement in ALGOL-W, and Pascal's records are similar to those of COBOL and PL/I.

10. Pascal is named after Blaise Pascal, a seventeenth-century French philosopher and mathematician who invented the first mechanical adding machine in 1642 (among other things).

2.12.1.2 Evaluation

The largest impact of Pascal was on the teaching of programming. In 1970, most students of computer science, engineering, and science were introduced to programming with Fortran, although some universities used PL/I, languages based on PL/I, and ALGOL-W. By the mid-1970s, Pascal had become the most widely used language for this purpose. This was quite natural, because Pascal was designed specifically for teaching programming. It was not until the late 1990s that Pascal was no longer the most commonly used language for teaching programming in colleges and universities.

Because Pascal was designed as a teaching language, it lacks several features that are essential for many kinds of applications. The best example of this is the impossibility of writing a subprogram that takes as a parameter an array of variable length. Another example is the lack of any separate compilation capability. These deficiencies naturally led to many nonstandard dialects, such as Turbo Pascal.

Pascal's popularity, for both teaching programming and other applications, was based primarily on its remarkable combination of simplicity and expressivity. Although there are some insecurities in Pascal, it is still a relatively safe language, particularly when compared with Fortran or C. By the mid-1990s, the popularity of Pascal was on the decline, both in industry and in universities, primarily due to the rise of Modula-2, Ada, and C++, all of which included features not available in Pascal.

The following is an example of a Pascal program:

```
{Pascal Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Read input into an array and compute the sum }
    for counter := 1 to listlen do
      begin
      readln (intlist[counter]);
      sum := sum + intlist[counter]
      end;
```

```
{ Compute the average }
    average := sum / listlen;
{ Count the number of input values that are > average }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Print the result }
    writeln ('The number of values > average is:',
             result)
    end { of the then clause of if (( listlen > 0 ... }
  else
    writeln ('Error—input list length is not legal')
end.
```

2.12.2 A Portable Systems Language: C

Like Pascal, C contributed little to the previously known collection of language features, but it has been widely used over a long period of time. Although originally designed for systems programming, C is well suited for a wide variety of applications.

2.12.2.1 Historical Background

C's ancestors include CPL, BCPL, B, and ALGOL 68. CPL was developed at Cambridge University in the early 1960s. BCPL is a simple systems language, also developed at Cambridge, this time by Martin Richards in 1967 (Richards, 1969).

The first work on the UNIX operating system was done in the late 1960s by Ken Thompson at Bell Laboratories. The first version was written in assembly language. The first high-level language implemented under UNIX was B, which was based on BCPL. B was designed and implemented by Thompson in 1970.

Neither BCPL nor B is a typed language, which is an oddity among high-level languages, although both are much lower-level than a language such as Java. Being untyped means that all data are considered machine words, which, although simple, leads to many complications and insecurities. For example, there is the problem of specifying floating-point rather than integer arithmetic in an expression. In one implementation of BCPL, the variable operands of a floating-point operation were preceded by periods. Variable operands not preceded by periods were considered to be integers. An alternative to this would have been to use different symbols for the floating-point operators.

This problem, along with several others, led to the development of a new typed language based on B. Originally called NB but later named C, it was designed and implemented by Dennis Ritchie at Bell Laboratories in 1972 (Kernighan and Ritchie, 1978). In some cases through BCPL, and in other cases directly, C was influenced by ALGOL 68. This is seen in its **for**

and **switch** statements, in its assigning operators, and in its treatment of pointers.

The only "standard" for C in its first decade and a half was the book by Kernighan and Ritchie (1978).[11] Over that time span, the language slowly evolved, with different implementors adding different features. In 1989, ANSI produced an official description of C (ANSI, 1989), which included many of the features that implementors had already incorporated into the language. This standard was updated in 1999 (ISO, 1999). This later version includes a few significant changes to the language. Among these are a complex data type, a Boolean data type, and C++-style comments (//). We will refer to the 1989 version, which has long been called ANSI C, as C89; we will refer to the 1999 version as C99.

2.12.2.2 Evaluation

C has adequate control statements and data-structuring facilities to allow its use in many application areas. It also has a rich set of operators that provide a high degree of expressiveness.

One of the most important reasons why C is both liked and disliked is its lack of complete type checking. For example, in versions before C99, functions could be written for which parameters were not type checked. Those who like C appreciate the flexibility; those who do not like it find it too insecure. A major reason for its great increase in popularity in the 1980s was that a compiler for it was part of the widely used UNIX operating system. This inclusion in UNIX provided an essentially free and quite good compiler that was available to programmers on many different kinds of computers.

The following is an example of a C program:

```
/* C Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values */
int main (){
  int intlist[99], listlen, counter, sum, average, result;
  sum = 0;
  result = 0;
  scanf("%d", &listlen);
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
    for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum += intlist[counter];
     }
```

11. This language is often referred to as "K & R C."

```
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
    printf("Number of values > average is:%d\n", result);
   }
  else
    printf("Error—input list length is not legal\n");
 }
```

2.13 Programming Based on Logic: Prolog

Simply put, logic programming is the use of a formal logic notation to communicate computational processes to a computer. Predicate calculus is the notation used in current logic programming languages.

Programming in logic programming languages is nonprocedural. Programs in such languages do not state exactly *how* a result is to be computed but rather describe the necessary form and/or characteristics of the result. What is needed to provide this capability in logic programming languages is a concise means of supplying the computer with both the relevant information and an inferencing process for computing desired results. Predicate calculus supplies the basic form of communication to the computer, and the proof method, named resolution, developed first by Robinson (1965), supplies the inferencing technique.

2.13.1 Design Process

During the early 1970s, Alain Colmerauer and Phillippe Roussel in the Artificial Intelligence Group at the University of Aix-Marseille, together with Robert Kowalski of the Department of Artificial Intelligence at the University of Edinburgh, developed the fundamental design of Prolog. The primary components of Prolog are a method for specifying predicate calculus propositions and an implementation of a restricted form of resolution. Both predicate calculus and resolution are described in Chapter 16. The first Prolog interpreter was developed at Marseille in 1972. The version of the language that was implemented is described in Roussel (1975). The name Prolog is from *pro*gramming *log*ic.

2.13.2 Language Overview

Prolog programs consist of collections of statements. Prolog has only a few kinds of statements, but they can be complex.

One common use of Prolog is as a kind of intelligent database. This application provides a simple framework for discussing the Prolog language.

The database of a Prolog program consists of two kinds of statements: facts and rules. The following are examples of fact statements:

```
mother(joanne, jake).
father(vern, joanne).
```

These state that `joanne` is the `mother` of `jake`, and `vern` is the `father` of `joanne`.

An example of a rule statement is

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

This states that it can be deduced that `X` is the `grandparent` of `Z` if it is true that `X` is the `parent` of `Y` and `Y` is the parent of `Z`, for some specific values for the variables `X`, `Y`, and `Z`.

The Prolog database can be interactively queried with goal statements, an example of which is

```
father(bob, darcie).
```

This asks if `bob` is the `father` of `darcie`. When such a query, or goal, is presented to the Prolog system, it uses its resolution process to attempt to determine the truth of the statement. If it can conclude that the goal is true, it displays "true." If it cannot prove it, it displays "false."

2.13.3 Evaluation

In the 1980s, there was a relatively small group of computer scientists who believed that logic programming provided the best hope for escaping from the complexity of imperative languages, and also from the enormous problem of producing the large amount of reliable software that was needed. So far, however, there are two major reasons why logic programming has not become more widely used. First, as with some other nonimperative approaches, programs written in logic languages thus far have proven to be highly inefficient relative to equivalent imperative programs. Second, it has been determined that it is an effective approach for only a few relatively small areas of application: certain kinds of database management systems and some areas of AI.

There is a dialect of Prolog that supports object-oriented programming—Prolog++ (Moss, 1994). Logic programming and Prolog are described in greater detail in Chapter 16.

2.14 History's Largest Design Effort: Ada

The Ada language is the result of the most extensive and expensive language design effort ever undertaken. The following paragraphs briefly describe the evolution of Ada.

2.14.1 Historical Background

The Ada language was developed for the Department of Defense (DoD), so the state of their computing environment was instrumental in determining its form. By 1974, over half of the applications of computers in DoD were embedded systems. An embedded system is one in which the computer hardware is embedded in the device it controls or for which it provides services. Software costs were rising rapidly, primarily because of the increasing complexity of systems. More than 450 different programming languages were in use for DoD projects, and none of them was standardized by DoD. Every defense contractor could define a new and different language for every contract.[12] Because of this language proliferation, application software was rarely reused. Furthermore, no software development tools were created (because they are usually language dependent). A great many languages were in use, but none was actually suitable for embedded systems applications. For these reasons, in 1974, the Army, Navy, and Air Force each independently proposed the development of a single high-level language for embedded systems.

2.14.2 Design Process

Noting this widespread interest, in January 1975, Malcolm Currie, director of Defense Research and Engineering, formed the High-Order Language Working Group (HOLWG), initially headed by Lt. Col. William Whitaker of the Air Force. The HOLWG had representatives from all of the military services and liaisons with Great Britain, France, and what was then West Germany. Its initial charter was to do the following:

- Identify the requirements for a new DoD high-level language.
- Evaluate existing languages to determine whether there was a viable candidate.
- Recommend adoption or implementation of a minimal set of programming languages.

In April 1975, the HOLWG produced the Strawman requirements document for the new language (Department of Defense, 1975a). This was distributed to military branches, federal agencies, selected industrial and university representatives, and interested parties in Europe.

---

12. This result was largely due to the widespread use of assembly language for embedded systems, along with the fact that most embedded systems used specialized processors.

The Strawman document was followed by Woodenman (Department of Defense, 1975b) in August 1975, Tinman (Department of Defense, 1976) in January 1976, Ironman (Department of Defense, 1977) in January 1977, and finally Steelman (Department of Defense, 1978) in June 1978.

After a tedious process, the many submitted proposals for the language were narrowed down to four finalists, all of which were based on Pascal. In May 1979, the Cii Honeywell/Bull language design proposal was chosen from the four finalists as the design that would be used. The Cii Honeywell/Bull design team in France, the only foreign competitor among the final four, was led by Jean Ichbiah.

In the spring of 1979, Jack Cooper of the Navy Materiel Command recommended the name for the new language, Ada, which was then adopted. The name commemorates Augusta Ada Byron (1815–1851), countess of Lovelace, mathematician, and daughter of poet Lord Byron. She is generally recognized as being the world's first programmer. She worked with Charles Babbage on his mechanical computers, the Difference and Analytical Engines, writing programs for several numerical processes.

The design and the rationale for Ada were published by ACM in its *SIGPLAN Notices* (ACM, 1979) and distributed to a readership of more than 10,000 people. A public test and evaluation conference was held in October 1979 in Boston, with representatives from over 100 organizations from the United States and Europe. By November, more than 500 language reports had been received from 15 different countries. Most of the reports suggested small modifications rather than drastic changes and outright rejections. Based on the language reports, the next version of the requirements specification, the Stoneman document (Department of Defense, 1980a), was released in February 1980.

A revised version of the language design was completed in July 1980 and was accepted as MIL-STD 1815, the standard *Ada Language Reference Manual.* The number 1815 was chosen because it was the year of the birth of Augusta Ada Byron. Another revised version of the *Ada Language Reference Manual* was released in July 1982. In 1983, the American National Standards Institute standardized Ada. This "final" official version is described in Goos and Hartmanis (1983). The Ada language design was then frozen for a minimum of five years.

2.14.3 Language Overview

This subsection briefly describes four of the major contributions of the Ada language.

Packages in the Ada language provide the means for encapsulating data objects, specifications for data types, and procedures. This, in turn, provides the support for the use of data abstraction in program design, as described in Chapter 11.

The Ada language includes extensive facilities for exception handling, which allow the programmer to gain control after any one of a wide variety

of exceptions, or run-time errors, has been detected. Exception handling is discussed in Chapter 14.

Program units can be generic in Ada. For example, it is possible to write a sort procedure that uses an unspecified type for the data to be sorted. Such a generic procedure must be instantiated for a specified type before it can be used, which is done with a statement that causes the compiler to generate a version of the procedure with the given type. The availability of such generic units increases the range of program units that might be reused, rather than duplicated, by programmers. Generics are discussed in Chapters 9 and 11.

The Ada language also provides for concurrent execution of special program units, named tasks, using the rendezvous mechanism. Rendezvous is the name of a method of intertask communication and synchronization. Concurrency is discussed in Chapter 13.

2.14.4 Evaluation

Perhaps the most important aspects of the design of the Ada language to consider are the following:

- Because the design was competitive, there were no limits on participation.
- The Ada language embodies most of the concepts of software engineering and language design of the late 1970s. Although one can question the actual approaches used to incorporate these features, as well as the wisdom of including such a large number of features in a language, most agree that the features are valuable.
- Although most people did not anticipate it, the development of a compiler for the Ada language was a difficult task. Only in 1985, almost four years after the language design was completed, did truly usable Ada compilers begin to appear.

The most serious criticism of Ada in its first few years was that it was too large and too complex. In particular, Hoare (1981) stated that it should not be used for any application where reliability is critical, which is precisely the type of application for which it was designed. On the other hand, others have praised it as the epitome of language design for its time. In fact, even Hoare eventually softened his view of the language.

The following is an example of an Ada program:

```
-- Ada Example Program
-- Input:  An integer, List_Len, where List_Len is less
--         than 100, followed by List_Len-integer values
-- Output: The number of input values that are greater
--         than the average of all input values
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
```

```
procedure Ada_Ex is
  type Int_List_Type is array (1..99) of Integer;
  Int_List : Int_List_Type;
  List_Len, Sum, Average, Result : Integer;
begin
  Result:= 0;
  Sum := 0;
  Get (List_Len);
  if (List_Len > 0) and (List_Len < 100) then
-- Read input data into an array and compute the sum
    for Counter := 1 .. List_Len loop
      Get (Int_List(Counter));
      Sum := Sum + Int_List(Counter);
    end loop;
-- Compute the average
    Average := Sum / List_Len;
-- Count the number of values that are > average
    for Counter := 1 .. List_Len loop
      if Int_List(Counter) > Average then
        Result:= Result+ 1;
      end if;
    end loop;
-- Print result
    Put ("The number of values > average is:");
    Put (Result);
    New_Line;
  else
    Put_Line ("Error—input list length is not legal");
  end if;
end Ada_Ex;
```

2.14.5 Ada 95 and Ada 2005

Two of the most important new features of Ada 95 are described briefly in the following paragraphs. In the remainder of the book, we will use the name Ada 83 for the original version and Ada 95 (its actual name) for the later version when it is important to distinguish between the two versions. In discussions of language features common to both versions, we will use the name Ada. The Ada 95 standard language is defined in ARM (1995).

The type derivation mechanism of Ada 83 is extended in Ada 95 to allow adding new components to those inherited from a base class. This provides for inheritance, a key ingredient in object-oriented programming languages. Dynamic binding of subprogram calls to subprogram definitions is accomplished through subprogram dispatching, which is based on the tag value of derived types through classwide types. This feature provides for polymorphism,

another principal feature of object-oriented programming. These features of Ada 95 are discussed in Chapter 12.

The rendezvous mechanism of Ada 83 provided only a cumbersome and inefficient means of sharing data among concurrent processes. It was necessary to introduce a new task to control access to the shared data. The protected objects of Ada 95 offer an attractive alternative to this. The shared data is encapsulated in a syntactic structure that controls all access to the data, either by rendezvous or by subprogram call. The new features of Ada 95 for concurrency and shared data are discussed in Chapter 13.

It is widely believed that the popularity of Ada 95 suffered because the Department of Defense stopped requiring its use in military software systems. There were, of course, other factors that hindered its growth in popularity. Most important among these was the widespread acceptance of C++ for object-oriented programming, which occurred before Ada 95 was released.

There were several additions to Ada 95 to get Ada 2005. Among these were interfaces, similar to those of Java, more control of scheduling algorithms, and synchronized interfaces.

Ada is widely used in both commercial and defense avionics, air traffic control, and rail transportation, as well as in other areas.

2.15 Object-Oriented Programming: Smalltalk

Smalltalk was the first programming language that fully supported object-oriented programming. It is therefore an important part of any discussion of the evolution of programming languages.

2.15.1 Design Process

The concepts that led to the development of Smalltalk originated in the Ph.D. dissertation work of Alan Kay in the late 1960s at the University of Utah (Kay, 1969). Kay had the remarkable foresight to predict the future availability of powerful desktop computers. Recall that the first microcomputer systems were not marketed until the mid-1970s, and they were only remotely related to the machines envisioned by Kay, which were seen to execute a million or more instructions per second and contain several megabytes of memory. Such machines, in the form of workstations, became widely available only in the early 1980s.

Kay believed that desktop computers would be used by nonprogrammers and thus would need very powerful human-interfacing capabilities. The computers of the late 1960s were largely batch oriented and were used exclusively by professional programmers and scientists. For use by nonprogrammers, Kay determined, a computer would have to be highly interactive and use sophisticated graphics in its user interface. Some of the graphics concepts came from

the LOGO experience of Seymour Papert, in which graphics were used to aid children in the use of computers (Papert, 1980).

Kay originally envisioned a system he called Dynabook, which was meant to be a general information processor. It was based in part on the Flex language, which he had helped design. Flex was based primarily on SIMULA 67. Dynabook used the paradigm of the typical desk, on which there are a number of papers, some partially covered. The top sheet is often the focus of attention, with the others temporarily out of focus. The display of Dynabook would model this scene, using screen windows to represent various sheets of paper on the desktop. The user would interact with such a display both through keystrokes and by touching the screen with his or her fingers. After the preliminary design of Dynabook earned him a Ph.D., Kay's goal became to see such a machine constructed.

Kay found his way to the Xerox Palo Alto Research Center (Xerox PARC) and presented his ideas on Dynabook. This led to his employment there and the subsequent birth of the Learning Research Group at Xerox. The first charge of the group was to design a language to support Kay's programming paradigm and implement it on the best personal computer then available. These efforts resulted in an "Interim" Dynabook, consisting of a Xerox Alto workstation and Smalltalk-72 software. Together, they formed a research tool for further development. A number of research projects were conducted with this system, including several experiments to teach programming to children. Along with the experiments came further developments, leading to a sequence of languages that ended with Smalltalk-80. As the language grew, so did the power of the hardware on which it resided. By 1980, both the language and the Xerox hardware nearly matched the early vision of Alan Kay.

2.15.2 Language Overview

The Smalltalk world is populated by nothing but objects, from integer constants to large complex software systems. All computing in Smalltalk is done by the same uniform technique: sending a message to an object to invoke one of its methods. A reply to a message is an object, which either returns the requested information or simply notifies the sender that the requested processing has been completed. The fundamental difference between a message and a subprogram call is this: A message is sent to a data object, specifically to one of the methods defined for the object. The called method is then executed, often modifying the data of the object to which the message was sent; a subprogram call is a message to the code of a subprogram. Usually the data to be processed by the subprogram is sent to it as a parameter.[13]

In Smalltalk, object abstractions are classes, which are very similar to the classes of SIMULA 67. Instances of the class can be created and are then the objects of the program.

The syntax of Smalltalk is unlike that of most other programming language, in large part because of the use of messages, rather than arithmetic and

13. Of course, a method call can also pass data to be processed by the called method.

logic expressions and conventional control statements. One of the Smalltalk control constructs is illustrated in the example in the next subsection.

2.15.3 Evaluation

Smalltalk has done a great deal to promote two separate aspects of computing: graphical user interfaces and object-oriented programming. The windowing systems that are now the dominant method of user interfaces to software systems grew out of Smalltalk. Today, the most significant software design methodologies and programming languages are object oriented. Although the origin of some of the ideas of object-oriented languages came from SIMULA 67, they reached maturation in Smalltalk. It is clear that Smalltalk's impact on the computing world is extensive and will be long-lived.

The following is an example of a Smalltalk class definition:

```
"Smalltalk Example Program"
"The following is a class definition, instantiations
of which can draw equilateral polygons of any number of
sides"
class name                    Polygon
superclass                    Object
instance variable names       ourPen
numSides
sideLength
"Class methods"
  "Create an instance"
  new
     ^ super new getPen

  "Get a pen for drawing polygons"
  getPen
     ourPen <- Pen new defaultNib: 2

  "Instance methods"
  "Draw a polygon"
  draw
     numSides timesRepeat: [ourPen go: sideLength;
                            turn: 360 // numSides]

  "Set length of sides"
  length: len
     sideLength <- len

  "Set number of sides"
  sides: num
     numSides <- num
```

2.16 Combining Imperative and Object-Oriented Features: C++

The origins of C were discussed in Section 2.12; the origins of Simula 67 were discussed in Section 2.10; the origins of Smalltalk were discussed in Section 2.15. C++ builds language facilities, borrowed from Simula 67, on top of C to support much of what Smalltalk pioneered. C++ has evolved from C through a sequence of modifications to improve its imperative features and to add constructs to support object-oriented programming.

2.16.1 Design Process

The first step from C toward C++ was made by Bjarne Stroustrup at Bell Laboratories in 1980. The initial modifications to C included the addition of function parameter type checking and conversion and, more significantly, classes, which are related to those of SIMULA 67 and Smalltalk. Also included were derived classes, public/private access control of inherited components, constructor and destructor methods, and friend classes. During 1981, inline functions, default parameters, and overloading of the assignment operator were added. The resulting language was called C with Classes and is described in Stroustrup (1983).

It is useful to consider some goals of C with Classes. The primary goal was to provide a language in which programs could be organized as they could be organized in SIMULA 67—that is, with classes and inheritance. A second important goal was that there should be little or no performance penalty relative to C. For example, array index range checking was not even considered because a significant performance disadvantage, relative to C, would result. A third goal of C with Classes was that it could be used for every application for which C could be used, so virtually none of the features of C would be removed, not even those considered to be unsafe.

By 1984, this language was extended by the inclusion of virtual methods, which provide dynamic binding of method calls to specific method definitions, method name and operator overloading, and reference types. This version of the language was called C++. It is described in Stroustrup (1984).

In 1985, the first available implementation appeared: a system named Cfront, which translated C++ programs into C programs. This version of Cfront and the version of C++ it implemented were named Release 1.0. It is described in Stroustrup (1986).

Between 1985 and 1989, C++ continued to evolve, based largely on user reactions to the first distributed implementation. This next version was named Release 2.0. Its Cfront implementation was released in June 1989. The most important features added to C++ Release 2.0 were support for multiple inheritance (classes with more than one parent class) and abstract classes, along with some other enhancements. Abstract classes are described in Chapter 12.

Release 3.0 of C++ evolved between 1989 and 1990. It added templates, which provide parameterized types, and exception handling. The current version of C++, which was standardized in 1998, is described in ISO (1998).

In 2002, Microsoft released its .NET computing platform, which included a new version of C++, named Managed C++, or MC++. MC++ extends C++ to provide access to the functionality of the .NET Framework. The additions include properties, delegates, interfaces, and a reference type for garbage-collected objects. Properties are discussed in Chapter 11. Delegates are briefly discussed in the introduction to C# in Section 2.19. Because .NET does not support multiple inheritance, neither does MC++.

2.16.2 Language Overview

Because C++ has both functions and methods, it supports both procedural and object-oriented programming.

Operators in C++ can be overloaded, meaning the user can create operators for existing operators on user-defined types. C++ methods can also be overloaded, meaning the user can define more than one method with the same name, provided either the numbers or types of their parameters are different.

Dynamic binding in C++ is provided by virtual methods. These methods define type-dependent operations, using overloaded methods, within a collection of classes that are related through inheritance. A pointer to an object of class A can also point to objects of classes that have class A as an ancestor. When this pointer points to an overloaded virtual method, the method of the current type is chosen dynamically.

Both methods and classes can be templated, which means that they can be parameterized. For example, a method can be written as a templated method to allow it to have versions for a variety of parameter types. Classes enjoy the same flexibility.

C++ supports multiple inheritance. It also includes exception handling that is significantly different from that of Ada. One difference is that hardware-detectable exceptions cannot be handled. The exception-handling constructs of Ada and C++ are discussed in Chapter 14.

2.16.3 Evaluation

C++ rapidly became and remains a widely used language. One factor in its popularity is the availability of good and inexpensive compilers. Another factor is that it is almost completely backward compatible with C (meaning that C programs can be, with few changes, compiled as C++ programs), and in most implementations it is possible to link C++ code with C code—and thus relatively easy for the many C programmers to learn C++. Finally, at the time C++ first appeared, when object-oriented programming began to receive widespread interest, C++ was the only available language that was suitable for large commercial software projects.

On the negative side, because C++ is a very large and complex language, it clearly suffers drawbacks similar to those of PL/I. It inherited most of the insecurities of C, which make it less safe than languages such as Ada and Java.

2.16.4 A Related Language: Objective-C

Objective-C (Kochan, 2009) is another hybrid language with both imperative and object-oriented features. Objective-C was designed by Brad Cox and Tom Love in the early 1980s. Initially, it consisted of C plus the classes and message passing of Smalltalk. Among the programming languages that were created by adding support for object-oriented programming to an imperative language, Objective-C is the only one to use the Smalltalk syntax for that support.

After Steve Jobs left Apple and founded NeXT, he licensed Objective-C and it was used to write the NeXT computer system software. NeXT also released its Objective-C compiler, along with the NeXTstep development environment and a library of utilities. After the NeXT project failed, Apple bought NeXT and used Objective-C to write MAC OS X. Objective-C is the language of all iPhone software, which explains its rapid rise in popularity after the iPhone appeared.

One characteristic that Objective-C inherited from Smalltalk is the dynamic binding of messages to objects. This means that there is no static checking of messages. If a message is sent to an object and the object cannot respond to the message, it is not known until run time, when an exception is raised.

In 2006, Apple announced Objective-C 2.0, which added a form of garbage collection and new syntax for declaring properties. Unfortunately, garbage collection is not supported by the iPhone run-time system.

Objective-C is a strict superset of C, so all of the insecurities of that language are present in Objective-C.

2.16.5 Another Related Language: Delphi

Delphi (Lischner, 2000) is a hybrid language, similar to C++ and Objetive-C in that it was created by adding object-oriented support, among other things, to an existing imperative language, in this case Pascal. Many of the differences between C++ and Delphi are a result of the predecessor languages and the surrounding programming cultures from which they are derived. Because C is a powerful but potentially unsafe language, C++ also fits that description, at least in the areas of array subscript range checking, pointer arithmetic, and its numerous type coercions. Likewise, because Pascal is more elegant and safer than C, Delphi is more elegant and safer than C++. Delphi is also less complex than C++. For example, Delphi does not allow user-defined operator overloading, generic subprograms, and parameterized classes, all of which are part of C++.

Delphi, like Visual C++, provides a graphical user interface (GUI) to the developer and simple ways to create GUI interfaces to applications written in Delphi. Delphi was designed by Anders Hejlsberg, who had previously developed the Turbo Pascal system. Both of these were marketed and distributed by Borland. Hejlsberg was also the lead designer of C#.

2.16.6 A Loosely Related Language: Go

The Go programming language is not directly related to C++, although it is C-based. It is in this section in part because it does not deserve its own section and it does not fit elsewhere.

Go was designed by Rob Pike, Ken Thompson, and Robert Griesemer at Google. Thompson is the designer of the predecessor of C, B, as well as the codesigner with Dennis Ritchie of UNIX. He and Pike were both formerly employed at Bell Labs. The initial design was begun in 2007 and the first implementation was released in late 2009. One of the initial motivations for Go was the slowness of compilation of large C++ programs at Google. One of the characteristics of the initial compiler for Go is that is it extremely fast. The Go language borrows some of its syntax and constructs from C. Some of the new features of Go include the following: (1) Data declarations are syntactically reversed from the other C-based languages; (2) the variables precede the type name; (3) variable declarations can be given a type by inference if the variable is given an initial value; and (4) functions can return multiple values. Go does not support traditional object-oriented programming, as it has no form of inheritance. However, methods can be defined for any type. It also does not have generics. The control statements of Go are similar to those of other C-based languages, although the switch does not include the implicit fall through to the next segment. Go includes a goto statement, pointers, associative arrays, interfaces (though they are different from those of Java and C#), and support for concurrency using its goroutines.

2.17 An Imperative-Based Object-Oriented Language: Java

Java's designers started with C++, removed some constructs, changed some, and added a few others. The resulting language provides much of the power and flexibility of C++, but in a smaller, simpler, and safer language.

2.17.1 Design Process

Java, like many programming languages, was designed for an application for which there appeared to be no satisfactory existing language. In 1990, Sun Microsystems determined there was a need for a programming language for embedded consumer electronic devices, such as toasters, microwave ovens, and interactive TV systems. Reliability was one of the primary goals for such a language. It may not seem that reliability would be an important factor in the software for a microwave oven. If an oven had malfunctioning software, it probably would not pose a grave danger to anyone and most likely would not lead to large legal settlements. However, if the software in a particular model was found to be erroneous after a million units had been manufactured and sold, their recall would entail significant cost. Therefore, reliability *is* an important characteristic of the software in consumer electronic products.

After considering C and C++, it was decided that neither would be satisfactory for developing software for consumer electronic devices. Although C was relatively small, it did not provide support for object-oriented programming, which they deemed a necessity. C++ supported object-oriented programming, but it was judged to be too large and complex, in part because it also supported procedure-oriented programming. It was also believed that neither C nor C++ provided the necessary level of reliability. So, a new language, later named Java, was designed. Its design was guided by the fundamental goal of providing greater simplicity and reliability than C++ was believed to provide.

Although the initial impetus for Java was consumer electronics, none of the products with which it was used in its early years were ever marketed. Starting in 1993, when the World Wide Web became widely used, and largely because of the new graphical browsers, Java was found to be a useful tool for Web programming. In particular, Java applets, which are relatively small Java programs that are interpreted in Web browsers and whose output can be included in displayed Web documents, quickly became very popular in the middle to late 1990s. In the first few years of Java popularity, the Web was its most common application.

The Java design team was headed by James Gosling, who had previously designed the UNIX emacs editor and the NeWS windowing system.

2.17.2 Language Overview

As we stated previously, Java is based on C++ but it was specifically designed to be smaller, simpler, and more reliable. Like C++, Java has both classes and primitive types. Java arrays are instances of a predefined class, whereas in C++ they are not, although many C++ users build wrapper classes for arrays to add features like index range checking, which is implicit in Java.

Java does not have pointers, but its reference types provide some of the capabilities of pointers. These references are used to point to class instances. All objects are allocated on the heap. References are always implicitly dereferenced, when necessary. So they behave more like ordinary scalar variables.

Java has a primitive Boolean type named **`boolean`**, used mainly for the control expressions of its control statements (such as **`if`** and **`while`**). Unlike C and C++, arithmetic expressions cannot be used for control expressions.

One significant difference between Java and many of its predecessors that support object-oriented programming, including C++, is that it is not possible to write stand-alone subprograms in Java. All Java subprograms are methods and are defined in classes. Furthermore, methods can be called through a class or object only. One consequence of this is that while C++ supports both procedural and object-oriented programming, Java supports object-oriented programming only.

Another important difference between C++ and Java is that C++ supports multiple inheritance directly in its class definitions. Java supports only single

inheritance of classes, although some of the benefits of multiple inheritance can be gained by using its interface construct.

Among the C++ constructs that were not copied into Java are structs and unions.

Java includes a relatively simple form of concurrency control through its **`synchronized`** modifier, which can appear on methods and blocks. In either case, it causes a lock to be attached. The lock ensures mutually exclusive access or execution. In Java, it is relatively easy to create concurrent processes, which in Java are called *threads*.

Java uses implicit storage deallocation for its objects, often called **garbage collection**. This frees the programmer from needing to delete objects explicitly when they are no longer needed. Programs written in languages that do not have garbage collection often suffer from what is sometimes called memory leakage, which means that storage is allocated but never deallocated. This can obviously lead to eventual depletion of all available storage. Object deallocation is discussed in detail in Chapter 6.

Unlike C and C++, Java includes assignment type coercions (implicit type conversions) only if they are widening (from a "smaller" type to a "larger" type). So **`int`** to **`float`** coercions are done across the assignment operator, but **`float`** to **`int`** coercions are not.

2.17.3 Evaluation

The designers of Java did well at trimming the excess and/or unsafe features of C++. For example, the elimination of half of the assignment coercions that are done in C++ was clearly a step toward higher reliability. Index range checking of array accesses also makes the language safer. The addition of concurrency enhances the scope of applications that can be written in the language, as do the class libraries for graphical user interfaces, database access, and networking.

Java's portability, at least in intermediate form, has often been attributed to the design of the language, but it is not. Any language can be translated to an intermediate form and "run" on any platform that has a virtual machine for that intermediate form. The price of this kind of portability is the cost of interpretation, which traditionally has been about an order of magnitude more than execution of machine code. The initial version of the Java interpreter, called the Java Virtual Machine (JVM), indeed was at least 10 times slower than equivalent compiled C programs. However, many Java programs are now translated to machine code before being executed, using Just-in-Time (JIT) compilers. This makes the efficiency of Java programs competitive with that of programs in conventionally compiled languages such as C++.

The use of Java increased faster than that of any other programming language. Initially, this was due to its value in programming dynamic Web documents. Clearly, one of the reasons for Java's rapid rise to prominence is simply that programmers like its design. Some developers thought C++ was simply too

large and complex to be practical and safe. Java offered them an alternative that has much of the power of C++, but in a simpler, safer language. Another reason is that the compiler/interpreter system for Java is free and easily obtained on the Web. Java is now widely used in a variety of different applications areas.

The most recent version of Java, Java 7, appeared in 2011. Since its beginning, many features have been added to the language, including an enumeration class, generics, and a new iteration construct.

The following is an example of a Java program:

```
// Java Example Program
//  Input: An integer, listlen, where listlen is less
//         than 100, followed by length-integer values
// Output: The number of input data that are greater than
//         the average of all input values
import java.io.*;
class IntSort {
public static void main(String args[]) throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  int listlen,
      counter,
      sum = 0,
      average,
      result = 0;
  int[] intlist = new int[99];
  listlen = Integer.parseInt(in.readLine());
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum  */
    for (counter = 0; counter < listlen; counter++) {
      intlist[counter] =
             Integer.valueOf(in.readLine()).intValue();
      sum += intlist[counter];
    }
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
      System.out.println(
          "\nNumber of values > average is:" + result);
    }  //** end of then clause of if ((listlen > 0) ...
    else System.out.println(
              "Error—input list length is not legal\n");
  }  //** end of method main
}  //** end of class IntSort
```

2.18 Scripting Languages

Scripting languages have evolved over the past 25 years. Early scripting languages were used by putting a list of commands, called a **script**, in a file to be interpreted. The first of these languages, named `sh` (for shell), began as a small collection of commands that were interpreted as calls to system subprograms that performed utility functions, such as file management and simple file filtering. To this were added variables, control flow statements, functions, and various other capabilities, and the result is a complete programming language. One of the most powerful and widely known of these is `ksh` (Bolsky and Korn, 1995), which was developed by David Korn at Bell Laboratories.

Another scripting language is `awk`, developed by Al Aho, Brian Kernighan, and Peter Weinberger at Bell Laboratories (Aho et al., 1988). `awk` began as a report-generation language but later became a more general-purpose language.

2.18.1 Origins and Characteristics of Perl

The Perl language, developed by Larry Wall, was originally a combination of `sh` and `awk`. Perl has grown significantly since its beginnings and is now a powerful, although still somewhat primitive, programming language. Although it is still often called a scripting language, it is actually more similar to a typical imperative language, since it is always compiled, at least into an intermediate language, before it is executed. Furthermore, it has all the constructs to make it applicable to a wide variety of areas of computational problems.

Perl has a number of interesting features, only a few of which are mentioned in this chapter and later discussed in the book.

Variables in Perl are statically typed and implicitly declared. There are three distinctive namespaces for variables, denoted by the first character of the variables' names. All scalar variable names begin with dollar signs (`$`), all array names begin with at signs (`@`), and all hash names (hashes are briefly described below) begin with percent signs (`%`). This convention makes variable names in programs more readable than those of any other programming language.

Perl includes a large number of implicit variables. Some of them are used to store Perl parameters, such as the particular form of newline character or characters that are used in the implementation. Implicit variables are commonly used as default parameters to built-in functions and default operands for some operators. The implicit variables have distinctive—although cryptic—names, such as `$!` and `@_`. The implicit variables' names, like the user-defined variable names, use the three namespaces, so `$!` is a scalar.

Perl's arrays have two characteristics that set them apart from the arrays of the common imperative languages. First, they have dynamic length, meaning that they can grow and shrink as needed during execution. Second, arrays can be sparse, meaning that there can be gaps between the elements. These

gaps do not take space in memory, and the iteration statement used for arrays, **foreach**, iterates over the missing elements.

Perl includes associative arrays, which are called **hashes**. These data structures are indexed by strings and are implicitly controlled hash tables. The Perl system supplies the hash function and increases the size of the structure when necessary.

Perl is a powerful, but somewhat dangerous, language. Its scalar type stores both strings and numbers, which are normally stored in double-precision floating-point form. Depending on the context, numbers may be coerced to strings and vice versa. If a string is used in numeric context and the string cannot be converted to a number, zero is used and there is no warning or error message provided for the user. This effect can lead to errors that are not detected by the compiler or run-time system. Array indexing cannot be checked, because there is no set subscript range for any array. References to nonexistent elements return **undef**, which is interpreted as zero in numeric context. So, there is also no error detection in array element access.

Perl's initial use was as a UNIX utility for processing text files. It was and still is widely used as a UNIX system administration tool. When the World Wide Web appeared, Perl achieved widespread use as a Common Gateway Interface language for use with the Web, although it is now rarely used for that purpose. Perl is used as a general-purpose language for a variety of applications, such as computational biology and artificial intelligence.

The following is an example of a Perl program:

```
# Perl Example Program
# Input:  An integer, $listlen, where $listlen is less
#         than 100, followed by $listlen-integer values.
# Output: The number of input values that are greater than
#        the average of all input values.
($sum, $result) = (0, 0);
$listlen = <STDIN>;
if (($listlen > 0) && ($listlen < 100)) {
# Read input into an array and compute the sum
  for ($counter = 0; $counter < $listlen; $counter++) {
    $intlist[$counter] = <STDIN>;
  } #- end of for (counter ...
# Compute the average
  $average = $sum / $listlen;
# Count the input values that are > average
  foreach $num (@intlist) {
    if ($num > $average) { $result++; }
  } #- end of foreach $num ...
# Print result
  print "Number of values > average is: $result \n";
} #- end of if (($listlen ...
```

```
else {
  print "Error--input list length is not legal \n";
}
```

2.18.2 Origins and Characteristics of JavaScript

Use of the Web exploded in the mid-1990s after the first graphical browsers appeared. The need for computation associated with HTML documents, which by themselves are completely static, quickly became critical. Computation on the server side was made possible with the Common Gateway Interface (CGI), which allowed HTML documents to request the execution of programs on the server, with the results of such computations returned to the browser in the form of HTML documents. Computation on the browser end became available with the advent of Java applets. Both of these approaches have now been replaced for the most part by newer technologies, primarily scripting languages.

JavaScript (Flanagan, 2002) was originally developed by Brendan Eich at Netscape. Its original name was Mocha. It was later renamed LiveScript. In late 1995, LiveScript became a joint venture of Netscape and Sun Microsystems and its name was changed to JavaScript. JavaScript has gone through extensive evolution, moving from version 1.0 to version 1.5 by adding many new features and capabilities. A language standard for JavaScript was developed in the late 1990s by the European Computer Manufacturers Association (ECMA) as ECMA-262. This standard has also been approved by the International Standards Organization (ISO) as ISO-16262. Microsoft's version of JavaScript is named JScript .NET.

Although a JavaScript interpreter could be embedded in many different applications, its most common use is embedded in Web browsers. JavaScript code is embedded in HTML documents and interpreted by the browser when the documents are displayed. The primary uses of JavaScript in Web programming are to validate form input data and create dynamic HTML documents. JavaScript also is now used with the Rails Web development framework.

In spite of its name, JavaScript is related to Java only through the use of similar syntax. Java is strongly typed, but JavaScript is dynamically typed (see Chapter 5). JavaScript's character strings and its arrays have dynamic length. Because of this, array indices are not checked for validity, although this is required in Java. Java fully supports object-oriented programming, but JavaScript supports neither inheritance nor dynamic binding of method calls to methods.

One of the most important uses of JavaScript is for dynamically creating and modifying HTML documents. JavaScript defines an object hierarchy that matches a hierarchical model of an HTML document, which is defined by the Document Object Model. Elements of an HTML document are accessed through these objects, providing the basis for dynamic control of the elements of documents.

Following is a JavaScript script for the problem previously solved in several languages in this chapter. Note that it is assumed that this script will be called from an HTML document and interpreted by a Web browser.

```
// example.js
//   Input: An integer, listLen, where listLen is less
//          than 100, followed by listLen-numeric values
//  Output: The number of input values that are greater
//          than the average of all input values

var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
        "Please type the length of the input list", "");
if ((listLen > 0) && (listLen < 100)) {

// Get the input and compute its sum
   for (counter = 0; counter < listLen; counter++) {
      intList[counter] = prompt (
                     "Please type the next number", "");
      sum += parseInt(intList[counter]);
   }

// Compute the average
   average = sum / listLen;

// Count the input values that are > average
   for (counter = 0; counter < listLen; counter++)
      if (intList[counter] > average) result++;

// Display the results
   document.write("Number of values > average is: ",
                result, "<br />");
} else
   document.write(
       "Error - input list length is not legal <br />");
```

2.18.3 Origins and Characteristics of PHP

PHP (Converse and Park, 2000) was developed by Rasmus Lerdorf, a member of the Apache Group, in 1994. His initial motivation was to provide a tool to help track visitors to his personal Web site. In 1995, he developed a package called Personal Home Page Tools, which became the first publicly distributed version of PHP. Originally, PHP was an abbreviation for Personal Home Page. Later, its user community began using the recursive name PHP: Hypertext

Preprocessor, which subsequently forced the original name into obscurity. PHP is now developed, distributed, and supported as an open-source product. PHP processors are resident on most Web servers.

PHP is an HTML-embedded server-side scripting language specifically designed for Web applications. PHP code is interpreted on the Web server when an HTML document in which it is embedded has been requested by a browser. PHP code usually produces HTML code as output, which replaces the PHP code in the HTML document. Therefore, a Web browser never sees PHP code.

PHP is similar to JavaScript, in its syntactic appearance, the dynamic nature of its strings and arrays, and its use of dynamic typing. PHP's arrays are a combination of JavaScript's arrays and Perl's hashes.

The original version of PHP did not support object-oriented programming, but that support was added in the second release. However, PHP does not support abstract classes or interfaces, destructors, or access controls for class members.

PHP allows simple access to HTML form data, so form processing is easy with PHP. PHP provides support for many different database management systems. This makes it a useful language for building programs that need Web access to databases.

2.18.4 Origins and Characteristics of Python

Python (Lutz and Ascher, 2004) is a relatively recent object-oriented interpreted scripting language. Its initial design was by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands in the early 1990s. Its development is now being done by the Python Software Foundation. Python is being used for the same kinds of applications as Perl: system administration, CGI programming, and other relatively small computing tasks. Python is an open-source system and is available for most common computing platforms. The Python implementation is available at `www.python.org`, which also has extensive information regarding Python.

Python's syntax is not based directly on any commonly used language. It is type checked, but dynamically typed. Instead of arrays, Python includes three kinds of data structures: lists; immutable lists, which are called **tuples**; and hashes, which are called **dictionaries**. There is a collection of list methods, such as `append`, `insert`, `remove`, and `sort`, as well as a collection of methods for dictionaries, such as `keys`, `values`, `copy`, and `has_key`. Python also supports list comprehensions, which originated with the Haskell language. List comprehensions are discussed in Section 15.8.

Python is object oriented, includes the pattern-matching capabilities of Perl, and has exception handling. Garbage collection is used to reclaim objects when they are no longer needed.

Support for CGI programming, and form processing in particular, is provided by the `cgi` module. Modules that support cookies, networking, and database access are also available.

Python includes support for concurrency with its threads, as well as support for network programming with its sockets. It also has more support for functional programming than other nonfunctional programming languages.

One of the more interesting features of Python is that it can be easily extended by any user. The modules that support the extensions can be written in any compiled language. Extensions can add functions, variables, and object types. These extensions are implemented as additions to the Python interpreter.

2.18.5 Origins and Characteristics of Ruby

Ruby (Thomas et al., 2005) was designed by Yukihiro Matsumoto (aka Matz) in the early 1990s and released in 1996. Since then it has continually evolved. The motivation for Ruby was dissatisfaction of its designer with Perl and Python. Although both Perl and Python support object-oriented programming,[14] neither is a pure object-oriented language, at least in the sense that each has primitive (nonobject) types and each supports functions.

The primary characterizing feature of Ruby is that it is a pure object-oriented language, just as is Smalltalk. Every data value is an object and all operations are via method calls. The operators in Ruby are only syntactic mechanisms to specify method calls for the corresponding operations. Because they are methods, they can be redefined. All classes, predefined or user defined, can be subclassed.

Both classes and objects in Ruby are dynamic in the sense that methods can be dynamically added to either. This means that both classes and objects can have different sets of methods at different times during execution. So, different instantiations of the same class can behave differently. Collections of methods, data, and constants can be included in the definition of a class.

The syntax of Ruby is related to that of Eiffel and Ada. There is no need to declare variables, because dynamic typing is used. The scope of a variable is specified in its name: A variable whose name begins with a letter has local scope; one that begins with `@` is an instance variable; one that begins with `$` has global scope. A number of features of Perl are present in Ruby, including implicit variables with silly names, such as `$_`.

As is the case with Python, any user can extend and/or modify Ruby. Ruby is culturally interesting because it is the first programming language designed in Japan that has achieved relatively widespread use in the United States.

2.18.6 Origins and Characteristics of Lua

Lua[15] was designed in the early 1990s by Roberto Ierusalimschy, Waldemar Celes, and Luis Henrique de Figueiredo at the Pontifical University of Rio de Janeiro in Brazil. It is a scripting language that supports procedural and

14. Actully, Python's support for object-oriented programming is partial.

15. The name Lua is derived from the Portuguese word for moon.

functional programming with extensibility as one of its primary goals. Among the languages that influenced its design are Scheme, Icon, and Python.

Lua is similar to JavaScript in that it does not support object-oriented programming but it was clearly influenced by it. Both have objects that play the role of both classes and objects and both have prototype inheritance rather than class inheritance. However, in Lua, the language can be extended to support object-oriented programming.

As in Scheme, Lua's functions are first-class values. Also, Lua supports closures. These capabilities allow it to be used for functional programming. Also like Scheme, Lua has only a single data structure, although in Lua's case, it is the table. Lua's tables extend PHP's associate arrays, which subsume the arrays of traditional imperative languages. References to table elements can take the form of references to traditional arrays, associative arrays, or records. Because functions are first-class values, they can be stored in tables, and such tables can serve as namespaces.

Lua uses garbage collection for its objects, which are all heap allocated. It uses dynamic typing, as do most of the other scripting languages.

Lua is a relatively small and simple language, having only 21 reserved words. The design philosophy of the language was to provide the bare essentials and relatively simple ways to extend the language to allow it to fit a variety of application areas. Much of its extensibility derives from its table data structure, which can be customized using Lua's metatable concept.

Lua can conveniently be used as a scripting language extension to other languages. Like early implementations of Java, Lua is translated to an intermediate code and interpreted. It easily can be embedded simply in other systems, in part because of the small size of its interpreter, which is only about 150K bytes.

During 2006 and 2007, the popularity of Lua grew rapidly, in part due to its use in the gaming industry. The sequence of scripting languages that have appeared over the past 20 years has already produced several widely used languages. Lua, the latest arrival among them, is quickly becoming one.

2.19 The Flagship .NET Language: C#

C#, along with the new development platform .NET,[16] was announced by Microsoft in 2000. In January 2002, production versions of both were released.

2.19.1 Design Process

C# is based on C++ and Java but includes some ideas from Delphi and Visual BASIC. Its lead designer, Anders Hejlsberg, also designed Turbo Pascal and Delphi, which explains the Delphi parts of the heritage of C#.

16. The .NET development system is briefly discussed in Chapter 1.

The purpose of C# is to provide a language for component-based software development, specifically for such development in the .NET Framework. In this environment, components from a variety of languages can be easily combined to form systems. All of the .NET languages, which include C#, Visual Basic .NET, Managed C++, F#, and JScript .NET,[17] use the Common Type System (CTS). The CTS provides a common class library. All types in all five .NET languages inherit from a single class root, `System.Object`. Compilers that conform to the CTS specification create objects that can be combined into software systems. All .NET languages are compiled into the same intermediate form, Intermediate Language (IL).[18] Unlike Java, however, the IL is never interpreted. A Just-in-Time compiler is used to translate IL into machine code before it is executed.

2.19.2 Language Overview

Many believe that one of Java's most important advances over C++ lies in the fact that it excludes some of C++'s features. For example, C++ supports multiple inheritance, pointers, structs, **`enum`** types, operator overloading, and a goto statement, but Java includes none of these. The designers of C# obviously disagreed with this wholesale removal of features, because all of these except multiple inheritance have been included in the new language.

To the credit of C#'s designers, however, in several cases, the C# version of a C++ feature has been improved. For example, the **`enum`** types of C# are safer than those of C++, because they are never implicitly converted to integers. This allows them to be more type safe. The **`struct`** type was changed significantly, resulting in a truly useful construct, whereas in C++ it serves virtually no purpose. C#'s structs are discussed in Chapter 12. C# takes a stab at improving the **`switch`** statement that is used in C, C++, and Java. C#'s switch is discussed in Chapter 8.

Although C++ includes function pointers, they share the lack of safety that is inherent in C++'s pointers to variables. C# includes a new type, delegates, which are both object-oriented and type-safe method references to subprograms. Delegates are used for implementing event handlers, controlling the execution of threads, and callbacks.[19] Callbacks are implemented in Java with interfaces; in C++, method pointers are used.

In C#, methods can take a variable number of parameters, as long as they are all the same type. This is specified by the use of a formal parameter of array type, preceded by the **`params`** reserved word.

Both C++ and Java use two distinct typing systems: one for primitives and one for objects. In addition to being confusing, this leads to a frequent need to

17. Many other languages have been modified to be .NET languages.

18. Initially, IL was called MSIL (Microsoft Intermediate Language), but apparently many people thought that name was too long.

19. When an object calls a method of another object and needs to be notified when that method has completed its task, the called method calls its caller back. This is known as a callback.

convert values between the two systems—for example, to put a primitive value into a collection that stores objects. C# makes the conversion between values of the two typing systems partially implicit through the implicit boxing and unboxing operations, which are discussed in detail in Chapter 12.[20]

Among the other features of C# are rectangular arrays, which are not supported in most programming languages, and a **foreach** statement, which is an iterator for arrays and collection objects. A similar **foreach** statement is found in Perl, PHP, and Java 5.0. Also, C# includes properties, which are an alternative to public data members. Properties are specified as data members with get and set methods, which are implicitly called when references and assignments are made to the associated data members.

C# has evolved continuously and quickly from its initial release in 2002. The most recent version is C# 2010. C# 2010 adds a form of dynamic typing, implicit typing, and anonymous types (see Chapter 6).

2.19.3 Evaluation

C# was meant to be an improvement over both C++ and Java as a general-purpose programming language. Although it can be argued that some of its features are a step backward, C# clearly includes some constructs that move it beyond its predecessors. Some of its features will surely be adopted by programming languages of the near future. Some already do.

The following is an example of a C# program:

```
// C# Example Program
// Input:  An integer, listlen, where listlen is less than
//         100, followed by listlen-integer values.
// Output: The number of input values that are greater
//         than the average of all input values.
using System;
public class Ch2example {
  static void Main() {
    int[] intlist;
    int listlen,
        counter,
        sum = 0,
        average,
        result = 0;
    intList = new int[99];
    listlen = Int32.Parse(Console.readLine());
    if ((listlen > 0) && (listlen < 100)) {
// Read input into an array and compute the sum
      for (counter = 0; counter < listlen; counter++) {
```

20. This feature was added to Java in Java 5.0.

```
        intList[counter] =
                        Int32.Parse(Console.readLine());
        sum += intList[counter];
      } //- end of for (counter ...
// Compute the average
      average = sum / listlen;
// Count the input values that are > average
      foreach (int num in intList)
        if (num > average) result++;
// Print result
      Console.WriteLine(
         "Number of values > average is:" + result);
    } //- end of if ((listlen ...
    else
      Console.WriteLine(
         "Error--input list length is not legal");
  } //- end of method Main
} //- end of class Ch2example
```

2.20 Markup/Programming Hybrid Languages

A markup/programming hybrid language is a markup language in which some of the elements can specify programming actions, such as control flow and computation. The following subsections introduce two such hybrid languages, XSLT and JSP.

2.20.1 XSLT

eXtensible Markup Language (XML) is a metamarkup language. Such a language is used to define markup languages. XML-derived markup languages are used to define data documents, which are called XML documents. Although XML documents are human readable, they are processed by computers. This processing sometimes consists only of transformations to forms that can be effectively displayed or printed. In many cases, such transformations are to HTML, which can be displayed by a Web browser. In other cases, the data in the document is processed, just as with other forms of data files.

The transformation of XML documents to HTML documents is specified in another markup language, eXtensible Stylesheet Language Transformations (XSLT) (`www.w3.org/TR/XSLT`). XSLT can specify programming-like operations. Therefore, XSLT is a markup/programming hybrid language. XSLT was defined by the World Wide Web Consortium (W3C) in the late 1990s.

An XSLT processor is a program that takes as input an XML data document and an XSLT document (which is also in the form of an XML document). In this processing, the XML data document is transformed to another XML

document,[21] using the transformations described in the XSLT document. The XSLT document specifies transformations by defining templates, which are data patterns that could be found by the XSLT processor in the XML input file. Associated with each template in the XSLT document are its transformation instructions, which specify how the matching data is to be transformed before being put in the output document. So, the templates (and their associated processing) act as subprograms, which are "executed" when the XSLT processor finds a pattern match in the data of the XML document.

XSLT also has programming constructs at a lower level. For example, a looping construct is included, which allows repeated parts of the XML document to be selected. There is also a sort process. These lower-level constructs are specified with XSLT tags, such as `<for-each>`.

2.20.2 JSP

The "core" part of the Java Server Pages Standard Tag Library (JSTL) is another markup/programming hybrid language, although its form and purpose are different from those of XSLT. Before discussing JSTL, it is necessary to introduce the ideas of servlets and Java Server Pages (JSP). A **servlet** is an instance of a Java class that resides on and is executed on a Web server system. The execution of a servlet is requested by a markup document being displayed by a Web browser. The servlet's output, which is in the form of an HTML document, is returned to the requesting browser. A program that runs in the Web server process, called a **servlet container**, controls the execution of servlets. Servlets are commonly used for form processing and for database access.

JSP is a collection of technologies designed to support dynamic Web documents and provide other processing needs of Web documents. When a JSP document, which is often a mixture of HTML and Java, is requested by a browser, the JSP processor program, which resides on a Web server system, converts the document to a servlet. The document's embedded Java code is copied to the servlet. The plain HTML is copied into Java print statements that output it as is. The JSTL markup in the JSP document is processed, as discussed in the following paragraph. The servlet produced by the JSP processor is run by the servlet container.

The JSTL defines a collection of XML action elements that control the processing of the JSP document on the Web server. These elements have the same form as other elements of HTML and XML. One of the most commonly used JSTL control action elements is `if`, which specifies a Boolean expression as an attribute.[22] The content of the `if` element (the text between the opening tag (`<if>`) and its closing tag (`</if>`)) is HTML code that will be included in the output document only if the Boolean expression evaluates to true. The `if` element is related to the C/C++ `#if` preprocessor command. The JSP

21. The output document of the XSLT processor could also be in HTML or plain text.
22. An attribute in HTML, which is embedded in the opening tag of an element, provides further information about that element.

container processes the JSTL parts of JSP documents in a way that is similar to how the C/C++ preprocessor processes C and C++ programs. The preprocessor commands are instructions for the preprocessor to specify how the output file is to be constructed from the input file. Similarly, JSTL control action elements are instructions for the JSP processor to specify how to build the XML output file from the XML input file.

One common use of the `if` element is for the validation of form data submitted by a browser user. Form data is accessible by the JSP processor and can be tested with the `if` element to ensure that it is sensible data. If not, the `if` element can insert an error message for the user in the output document.

For multiple selection control, JSTL has `choose`, `when`, and `otherwise` elements. JSTL also includes a `forEach` element, which iterates over collections, which typically are form values from a client. The `forEach` element can include `begin`, `end`, and `step` attributes to control its iterations.

SUMMARY

We have investigated the development and the development environments of a number of programming languages. This chapter gives the reader a good perspective on current issues in language design. We have set the stage for an in-depth discussion of the important features of contemporary languages.

BIBLIOGRAPHIC NOTES

Perhaps the most important source of historical information about the development of early programming languages is *History of Programming Languages*, edited by Richard Wexelblat (1981). It contains the developmental background and environment of 13 important programming languages, as told by the designers themselves. A similar work resulted from a second "history" conference, published as a special issue of *ACM SIGPLAN Notices* (ACM, 1993a). In this work, the history and evolution of 13 more programming languages are discussed.

The paper "Early Development of Programming Languages" (Knuth and Pardo, 1977), which is part of the *Encyclopedia of Computer Science and Technology*, is an excellent 85-page work that details the development of languages up to and including Fortran. The paper includes example programs to demonstrate the features of many of those languages.

Another book of great interest is *Programming Languages: History and Fundamentals*, by Jean Sammet (1969). It is a 785-page work filled with details of 80 programming languages of the 1950s and 1960s. Sammet has also published several updates to her book, such as *Roster of Programming Languages for 1974–75* (1976).