Overview

"A good programming language is a conceptual universe for thinking about programming."
Alan Perlis

CHAPTER OUTLINE

1.1	PRINCIPLES	2
1.2	PARADIGMS	3
1.3	SPECIAL TOPICS	5
1.4	A BRIEF HISTORY	6
1.5	ON LANGUAGE DESIGN	11
1.6	COMPILERS AND VIRTUAL MACHINES	18
1.7	SUMMARY	20
EXERCISES		21

Like our natural languages, programming languages facilitate the expression and communication of ideas between people. However, programming languages differ from natural languages in two important ways. First, programming languages also enable the communication of ideas between people and computing machines. Second, programming languages have a narrower expressive domain than our natural languages. That is, they facilitate only the communication of *computational* ideas. Thus, a programming

language must meet different requirements than a natural language. This text explores these requirements and the language design alternatives that they evoke.

In this study, we identify the many similarities between programming languages and natural languages. We also examine the fundamental differences that are imposed by the computational setting in which a program must function. We examine the features of programming languages both abstractly and actively. That is, we combine a conceptually rich treatment of programming language design together with a hands-on laboratory-based study of how these concepts impact language designers and programmers in a wide range of application domains.

This study is important because today's computer science students will be the designers and users of tomorrow's programming languages. To become an informed language designer and user, you will need to understand languages broadly—their features, strengths, and weaknesses across a wide range of programming styles and applications. Knowing one language and application domain does not provide such breadth of understanding. This book will help you to obtain that breadth.

1.1 PRINCIPLES

Language designers have a basic vocabulary about language structure, meaning, and pragmatic concerns that helps them understand how languages work. Their vocabulary falls into three major categories that we call the *principles* of language design.

- Syntax
- Names and types
- Semantics

Many of the concepts in these categories are borrowed from linguistics and mathematics, as we shall learn below. Together, these categories provide an organizational focus for the core Chapters 2, 4, 5, 7, and 9 respectively. Additional depth of study in each category is provided in the companion chapters (3, 6, 8, 10, and 11) as explained below.

Syntax The *syntax* of a language describes what constitutes a structurally correct program. Syntax answers many questions. What is the grammar for writing programs in the language? What is the basic set of words and symbols that programmers use to write structurally correct programs?

We shall see that most of the syntactic structure of modern programming languages is defined using a linguistic formalism called the *context-free grammar*. Other elements of syntax are outside the realm of context-free grammars, and are defined by other means. A careful treatment of programming language syntax appears in Chapter 2.

A study of language syntax raises many questions. How does a compiler analyze the syntax of a program? How are syntax errors detected? How does a context-free grammar facilitate the development of a syntactic analyzer? These deeper questions about syntax are addressed in Chapter 3.

Names and Types The vocabulary of a programming language includes a carefully designed set of rules for naming entities—variables, functions, classes, parameters, and so forth. Names of entities also have other properties during the life of a program, such as their scope, visibility, and binding. The study of names in programming languages and their impact on the syntax and semantics of a program is the subject of Chapter 4.

A language's *types* denote the kinds of values that programs can manipulate: simple types, structured types, and more complex types. Among the simple types are integers, decimal numbers, characters, and boolean values. Structured types include character strings, lists, trees, and hash tables. More complex types include functions and classes. Types are more fully discussed in Chapter 5.

A type system enables the programmer to understand and properly implement operations on values of various types. A carefully specified type system allows the compiler to perform rigorous type checking on a program before run time, thus heading off run-time errors that may occur because of inappropriately typed operands. The full specification and implementation of a type system is the focus of a deeper study in Chapter 6.

Semantics The meaning of a program is defined by its *semantics*. That is, when a program is run, the effect of each statement on the values of the variables in the program is given by the semantics of the language. Thus, when we write a program, we must understand such basic ideas as the exact effect that an assignment has on the program's variables. If we have a semantic model that is independent of any particular platform, we can apply that model to a variety of machines on which that language may be implemented. We study semantics in Chapter 7.

The implementation of run-time semantics is also of interest in a deeper study of semantics. How does an interpreter work, and what is the connection between an interpreter and the specification of a language's semantics? These deeper questions are studied in Chapter 8.

Functions represent the key element of procedural abstraction in any language. An understanding of the semantics of function definition and call is central to any study of programming languages. The implementation of functions also requires an understanding of the static and dynamic elements of memory, including the *run-time stack*. The stack also helps us understand other ideas like the scope of a name and the lifetime of an object. These topics are treated in Chapter 9.

The stack implementation of function call and return is a central topic deserving deeper study. Moreover, strategies for the management of another memory area called the *heap*, are important to the understanding of dynamic objects like arrays. Heap management techniques called "garbage collection" are strongly related to the implementation of these dynamic objects. The stack and the heap are studied in detail in Chapters 10 and 11 respectively.

1.2 PARADIGMS

In general, we think of a "paradigm" as a pattern of thought that guides a collection of related activities. A programming paradigm is a pattern of problem solving thought that underlies a particular genre of programs and languages. Four distinct and fundamental programming paradigms have evolved over the last three decades:

- Imperative programming
- Object-oriented programming
- Functional programming
- Logic programming

Some programming languages are intentionally designed to support more than one paradigm. For instance, C++ is a hybrid imperative and object-oriented language, while

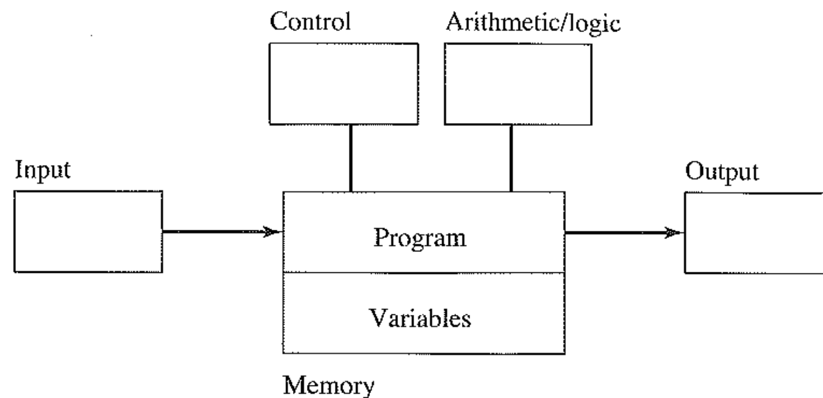


Figure 1.1 **The von Neumann-Eckert Computer Model**

the experimental language Leda [Budd, 1995] is designed to support the imperative, object-oriented, functional, and logic programming paradigms. These languages are reminiscent of earlier efforts (notably, PL/I, Algol 68, and Ada) to design a single language that was more general-purpose than other programming languages of its day. With the exception of C++, these efforts have failed to attract sustained interest.

Imperative Programming *Imperative programming* is the oldest paradigm, as it is grounded in the classic "von Neumann-Eckert" model of computation (see Figure 1.1). In this model, both the program and its variables are stored together, and the program contains a series of commands that perform calculations, assign values to variables, retrieve input, produce output, or redirect control elsewhere in the series.

Procedural abstraction is an essential building block for imperative programming, as are assignments, loops, sequences, conditional statements, and exception handling. The predominant imperative programming languages include Cobol, Fortran, C, Ada, and Perl. The imperative programming paradigm is the subject of Chapter 12.

Object-Oriented Programming *Object-oriented (OO) programming* provides a model in which the program is a collection of objects that interact with each other by passing messages that transform their state. In this sense, message passing allows the data objects to become active rather than passive. This characteristic helps to further distinguish OO programming from imperative programming. Object classification, inheritance, and message passing are fundamental building blocks for OO programming. Major object-oriented languages are Smalltalk, C++, Java, and C#. OO programming is studied in Chapter 13.

Functional Programming *Functional programming* models a computational problem as a collection of mathematical functions, each with an input (domain) and a result (range) spaces. This sets functional programming apart from languages with an assignment statement. For instance, the assignment statement

```
x = x + 1
```

makes no sense either in functional programming or in mathematics.

Functions interact and combine with each other using functional composition, conditionals, and recursion. Major functional programming languages are Lisp, Scheme, Haskell, and ML. Functional programming is discussed and illustrated in Chapter 14.

Logic Programming *Logic (declarative) programming* allows a program to model a problem by declaring what outcome the program should accomplish, rather than how it should be accomplished. Sometimes these languages are called *rule-based* languages, since the program's declarations look more like a set of rules, or constraints on the problem, rather than a sequence of commands to be carried out.

Interpreting a logic program's declarations creates a set of all possible solutions to the problem that it specifies. Logic programming also provides a natural vehicle for expressing nondeterminism, which is appropriate for problems whose specifications are incomplete. The major logic programming language is Prolog, and the logic programming paradigm is covered in Chapter 15.

1.3 SPECIAL TOPICS

Beyond these four paradigms, several key topics in programming language design deserve extensive coverage in a text such as this one. These topics tend to be pervasive, in the sense that they appear in two or more of the above paradigms, rather than just one. Each of the following is briefly introduced below.

- Event-handling
- Concurrency
- Correctness

Event-Handling *Event-handling* occurs with programs that respond to events that are generated in an unpredictable order. In one sense, an event-driven program is just a program whose behavior is fully determined by event-handling concerns. Event-handling is often coupled with the object-oriented paradigm (e.g., Java applets), although it occurs within the imperative paradigm as well (e.g., Tcl/Tk). Events originate from user actions on the screen (mouse clicks or keystrokes, for example), or else from other sources (like readings from sensors on a robot). Major languages that support event-handling include Visual Basic, Java and Tcl/Tk. This topic is treated in Chapter 16.

Concurrency *Concurrent programming* can occur within the imperative, object-oriented, functional, or logic paradigm. Concurrency occurs when the program has a collection of asynchronous elements, which may share information or synchronize with each other from time to time. Concurrency can also occur within an individual process, such as the parallel execution of the different iterations of a loop. Concurrent programming languages include SR [Andrews and Olsson, 1993], Linda [Carriero and Gelenter, 1989], and High Performance Fortran [Adams and others, 1997]. Concurrent programming is treated in Chapter 17.

Correctness *Program correctness* is a subject that, until recently, has had only academic interest. However, newer languages and language features are evolving that support the design of provably correct programs in a variety of application domains. A program is *correct* if it satisfies its formal specification for all its possible inputs.

Proof of correctness is a complex subject, but language tools for formal treatment of correctness by programmers are now becoming available. For instance, the Spark/Ada system [Barnes, 2003] and the Java Modeling Language [Leavens *et al.*, 1998] provide good examples. We introduce the topic of program correctness in Chapter 18.

1.4 A BRIEF HISTORY

The first programming languages were the machine and assembly languages of the earliest computers, beginning in the 1940s. Hundreds of programming languages and dialects have been developed since that time. Most have had a limited life span and utility, while a few have enjoyed widespread success in one or more application domains. Many have played an important role in influencing the design of future languages.

A snapshot of the historical development of several influential programming languages appears in Figure 1.2. While it is surely not complete, Figure 1.2 identifies some of the most influential events and trends. Each arrow in Figure 1.2 indicates a significant design influence from an older language to a successor.

The 1950s marked the beginning of the age of "higher-order languages" (HOLs for short). A HOL distinguishes itself from a machine or assembly language because its programming style is independent of any particular machine architecture. The first higher-order languages were Fortran, Cobol, Algol, and Lisp. Both Fortran and Cobol have survived and evolved greatly since their emergence in the late 1950s. These languages built a large following and carry with them an enormous body of legacy code that today's programmers maintain. On the other hand, Lisp has substantially declined in use and Algol has disappeared altogether.

However, the innovative designs of these early languages have had powerful influence on their successors. For example, Fortran's demonstration that algebraic notation could be translated to efficient code is now taken for granted, as are Cobol's introduction of the record structure, Pascal's design for one-pass compiling, and Algol's demonstration that a linguistic grammar could formally define its syntax.

Perhaps the greatest motivator for the development of programming languages over the last several decades is the rapidly evolving demand for computing power and new applications by large and diverse communities of users. The following user communities can claim a major stake in the programming language landscape:

- Artificial intelligence
- Education
- Science and engineering
- Information systems
- Systems and networks
- World Wide Web

The computational problem domains of these communities are all different, and so are the major programming languages that developed around them. Below we sketch the major computational goals and language designs that have served each of these communities.

Artificial Intelligence The artificial intelligence programming community has been active since the early 1960s. This community is concerned about developing

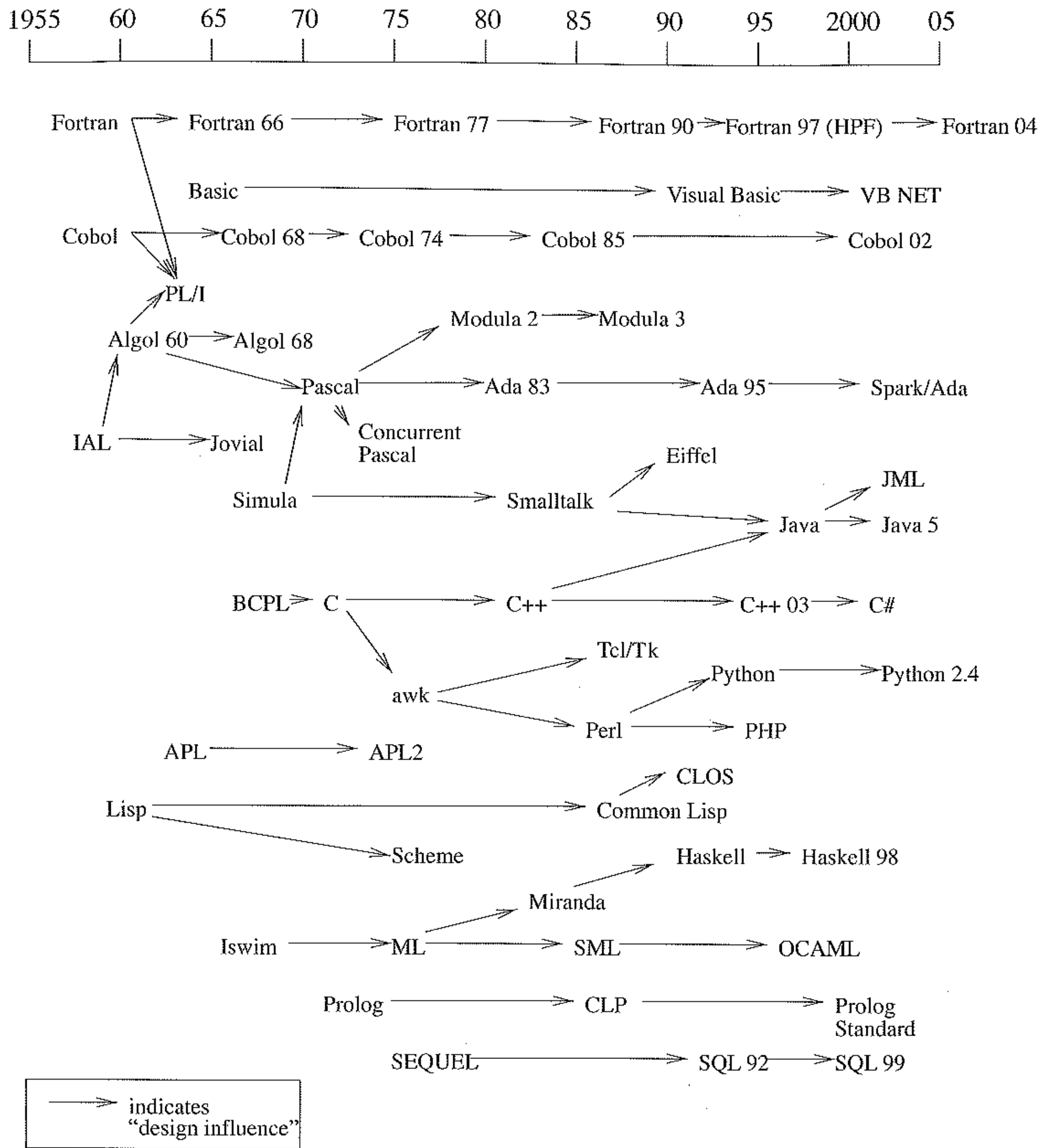


Figure 1.2 A Snapshot of Programming Language History

programs that model human intelligent behavior, logical deduction, and cognition. Symbol manipulation, functional expressions, and the design of logical proof systems have been central goals in this ongoing effort.

The paradigms of *functional programming* and *logic programming* have evolved largely through the efforts of artificial intelligence programmers. Prominent functional

programming languages over the years include Lisp, Scheme, ML, and Haskell. The prominent logic programming languages include Prolog and CLP.

The first AI language, Lisp (an acronym for "*Lis*t *P*rocessor"), was designed by John McCarthy in 1960. Figure 1.2 suggests that Lisp was dominant in early years and has become less dominant in recent years. However, Lisp's core features have motivated the development of more recent languages such as Scheme, ML, and Haskell. The strong relationship between Lisp and the *lambda calculus* (a formalism for modeling the nature of mathematical functions) provides a firm mathematical basis for the later evolution of these successors. The lambda calculus and its relationship with functional languages are explained more fully in Chapter 14.

In the logic programming area, only one language, Prolog, has been the major player, and Prolog has had little influence on the design of languages in other application areas.

Education In the 1960s and 1970s, several key languages were designed with a primary goal of teaching students about programming. For example, Basic was designed in the 1960s by John Kemeny to facilitate the learning of programming through *time sharing*, an architecture in which a single computer is directly connected to several terminals at one time. Each terminal user shares time on the computer by receiving a small "time slice" of computational power on a regular basis. Basic has enjoyed great popularity over the years, especially as a teaching language in secondary schools and college-level science programs.

The language Pascal, a derivative of Algol, was designed in the 1970s for the purpose of teaching programming. Pascal served for several years as the main teaching language in college-level computer science curricula.

During the last decade, these languages have been largely replaced in educational programs by such "industrial strength" languages as C, C++, and Java. This change has both benefits and liabilities. On the one hand, learning an industrial strength language provides graduates with a programming tool that they can use immediately when they enter the computing profession. On the other hand, such a language is inherently more complex and cumbersome to learn as a first language in undergraduate coursework.

The recent emergence of Python may provide a vehicle through which introductory computer science courses can return to simplicity and concentrate again on teaching first principles. For example, Python has a more transparent syntax and semantics, which makes it more amenable to mastery by a novice than any of its industrial strength alternatives. Moreover, introductory courses using Python seem to introduce a richer variety of computer science topics courses using C, C++, or Java.

Science and Engineering The scientific and engineering programming community played a major role in the early history of computing, and it continues to play a major role today. The first programs were written in the 1940s to predict the trajectories of ballistics during World War II, using the well-worn physics formulae that characterize bodies in motion. These programs were first written in machine and assembly language by specially trained mathematicians.

A major driving force behind scientific and engineering applications throughout their history is the need to obtain as much processing power as possible. The processing power of today's supercomputers is measured in *teraflops* (trillions of floating point operations

per second), and the current leader runs at a speed of 280 teraflops under the standard performance benchmark called LINPAK (see **www.top500.org** for more information). Many of today's scientific and engineering applications are models of complex natural systems in fields like bioinformatics and the earth and atmospheric sciences.

The first scientific programming language, Fortran I, was designed by John Backus at IBM in 1954 [Backus and *et al.*, 1954]. The acronym "Fortran" is an abbreviation for "*For*mula *Tran*slator." Fortran is probably the most widely used scientific programming language today.

Early versions of Fortran had many problems, however. The most difficult problem was that of consistency—the same Fortran program ran differently on different machines, or else would not run at all. These problems gave rise to several new efforts. One such effort produced the language Algol, short for "*Algo*rithmic *L*anguage," which was designed by an international committee in 1959. Algol's principal design goal was to provide a better-defined language than Fortran for both the computation and the publication of scientific and mathematical algorithms.

Algol was originally named the "*I*nternational *A*lgebraic *L*anguage" (IAL). The language Jovial was designed by Jules Schwartz in the 1960s to refine and augment the features of IAL. This acronym stands for "*J*ules' *O*wn *V*ersion of the *I*nternational *A*lgebraic *L*anguage." *Jovial* was widely used in US Department of Defense applications.

Another interesting language called APL (short for "*A* *P*rogramming *L*anguage") [Iverson, 1962] was designed by Kenneth Iverson in the 1960s to facilitate the rapid programming of matrix algebraic and other mathematical computations. APL had an extended character set that included single-symbol matrix operators that could replace the tedium of writing `for` loops in most cases. The proliferation of such special symbols required the design of a specialized keyboard to facilitate the typing of APL programs. APL programs were known for their brevity; a matrix computation that required an explicit `for` loop in a conventional language needed only a single symbol in APL. APL's brevity was also its curse in many people's eyes. That is, most APL programs were so terse that they defied understanding by anyone but the most skilled technicians. The cost of supporting a specialized character set and stylized keyboard also contributed to APL's demise.

To this day, scientific computing remains a central activity in the history of programming and programming languages. Its problem domain is primarily concerned with performing complex calculations very fast and very accurately. The calculations are defined by mathematical models that represent scientific phenomena. They are primarily implemented using the *imperative programming* paradigm. Modern programming languages that are widely used in the scientific programming arena include Fortran 90 [Chamberland, 1995], C [Kernighan and Ritchie, 1988], C++ [Stroustrup, 1997], and High Performance Fortran [Adams and others, 1997].

The more complex the scientific phenomena become, the greater the need for sophisticated, highly parallel computers and programming languages. Thus, *concurrent programming* is strongly motivated by the needs of such scientific applications as modeling weather systems and ocean flow. Some languages, like High Performance Fortran, support concurrent programming by adding features to a widely used base language (e.g., Fortran). Others, like SR and Occam, are designed specifically to support concurrent programming. General purpose languages, like Java, support concurrency as just one of their many design goals.

Information Systems Programs designed for use by institutions to manage their information systems are probably the most prolific in the world. Corporations realized in the 1950s that the use of computers could greatly reduce their record-keeping tedium and improve the accuracy and reliability of what they could accomplish. Information systems found in corporations include the payroll system, the accounting system, the online sales and marketing systems, the inventory and manufacturing systems, and so forth. Such systems are characterized by the need to process large amounts of data (often organized into so-called databases), but require relatively simple transformations on the data as it is being processed.

Traditionally, information systems have been developed in programming languages like Cobol and SQL. Cobol was first designed in the late 1950s by a group of industry representatives who wanted to develop a language that would be portable across a variety of different machine architectures. Cobol stands for "*Co*mmon *B*usiness *O*riented *L*anguage," uses English as the basis for its syntax, and supports an *imperative* programming style.

Cobol programs are constructed out of clauses, sentences, and paragraphs, and generally tend to be more wordy than comparable programs in other languages. The aim here was to define a language that would be easy for programmers to assimilate. Whether or not that aim was ever reached is still open for discussion. Nevertheless, Cobol quickly became, and still remains the most widely used programming language for information systems applications.

By contrast, SQL [Pratt, 1990] emerged in the 1980s as a *declarative* programming tool for database specification, report generation, and information retrieval. SQL stands for "*S*tructured *Q*uery *L*anguage" and is the predominant language used for specifying and retrieving information from relational databases. The relational database model is widely used, in part because of its strong mathematical underpinnings in relational algebra.

More recently, businesses have developed a wide range of electronic commerce applications. These applications often use a "client-server" model for program design, where the program interacts with users at remote sites and provides simultaneous access to a shared database. A good example of this model is an online book ordering system, in which the database reflects the company's inventory of books and the interaction helps the user through the database search, book selection, and ordering process. *Event-driven programming* is essential in these applications, and programmers combine languages like Java, Perl, Python and SQL to implement them.

Systems and Networks Systems programmers design and maintain the basic software that runs systems—operating system components, network software, programming language compilers and debuggers, virtual machines and interpreters, and real time and embedded systems (in cell phones, ATMs, aircraft, etc.). These types of software are closely tied with the architectures of specific machines, like the Intel/AMD x86 and the Apple/Motorola/IBM PowerPC.

Most of these programs are written in C, which allows programmers to get very close to the machine language level. Systems programming is typically done using the *imperative* design paradigm. However, systems programmers must also deal with *concurrent* and *event-driven* programming, and they also have special concerns for program *correctness* as well.

Thus, the primary example of a systems programming language is C, designed in the early 1970s in part to support the coding of the Unix operating system. In fact, about 95 percent of the code of the Unix system is written in C. C++ was designed by Bjarne Stroustrup in the 1980s as an extension of C to provide new features that would support object-oriented programming.

The programming language Ada is named after Ada Lovelace, who is believed to have been the first computer programmer. In the early 1800s, she worked with the computer inventor Charles Babbage. The development of Ada was funded by the US Department of Defense, whose original goal was to have a single language that would support all DoD applications, especially command and control and embedded systems applications. While Ada never achieved this particular goal, its design has some notable features. Today, Ada provides a robust host upon which the Spark compiler provides tools to support program correctness.

Scripting languages are widely used today for a variety of systems tasks. For example, an awk program can be designed quickly to check a password file in a Unix machine for consistency. Some of the primary scripting languages are awk [Kernighan and Pike, 1984], Perl [Wall *et al.*, 1996b], Tcl/Tk [Ousterhout, 1994], and Python [Lutz, 2001]. We treat scripting languages in Chapter 12, where Perl programming is explored in some detail.

World Wide Web The most dynamic area for new programming applications is the Internet, which is the enabling vehicle for electronic commerce and a wide range of applications in academia, government, and industry. The notion of Web-centric computing, and hence Web-centric programming, is motivated by an interactive model, in which a program remains continuously active waiting for the next event to occur, responding to that event, and returning to its continuously active state.

Programming languages that support Web-centric computing use *event-driven* programming, which encourages system-user interaction. Web-centric computing also uses the *object-oriented* paradigm, since various entities that appear on the user's screen are most naturally modeled as objects that send and receive messages. Programming languages that support Web-centric computing include Perl, PHP [Hughes, 2001], Visual Basic, Java, and Python.

1.5 ON LANGUAGE DESIGN

Programming language design is an enormous challenge. Language designers are the people who create a language medium that enables programmers to solve complex problems. To achieve this goal, designers must work within several practical constraints and adopt specific goals which combine to provide focus to this challenge. This section provides an overview of these design constraints and goals.

1.5.1 Design Constraints

The following elements of computational settings provide major constraints for language designers.

- Architecture
- Technical setting
- Standards
- Legacy systems

Architecture Programming languages are designed for computers. This fact is both a blessing and a curse to language designers. It is a blessing because a well-designed and implemented language can greatly enhance the utility of the computer in an application domain. It is a curse because most computer designs over the past several decades have been bound by the architecture ideas of the classic von Neumann-Eckert model discussed above. Many languages, like Fortran, Cobol and C, are well-matched with that architecture, while others, like Lisp, are not.

For a few years, it became attractive to consider the idea of computer architecture as a by-product of language design, rather than as a precursor. In the 1960s, Burroughs designed the B5500, which had a stack architecture particularly suited to running Algol programs. Another effort produced the genre of Lisp machines that emerged in the early 1980s. These machines were configured so that Lisp programs would run efficiently on them, and they enjoyed a degree of success for a few years. However, Lisp machine architectures were eclipsed in the late 1980s by the advent of Reduced Instruction Set Computer (RISC) architectures, on which Lisp programs could be implemented efficiently.

So as we consider the virtues of various language design choices, we are always constrained by the need to implement the language efficiently and effectively within the constraints imposed by today's variations of the classical von Neumann model. The notion that a good language design can lead to a radically new and commercially viable computer architecture is probably not in the cards.

Technical Setting Not only are language designs constrained by the limits of computer architectures, they must also satisfy other constraints imposed by the technical setting in which they are used: the application area, the operating system, Integrated Development Environment (IDE), the network, and the other preferences of a particular programming community. For example, Fortran is implemented on certain platforms by different compilers to suit the needs of scientific programmers. These programmers work in various professions that use their own software design styles, tools, and (above all) their own natural languages for communication among themselves. This larger picture of the complex setting for language design is summarized in Figure 1.3.

Some languages are intentionally more *general-purpose* in their design, aiming to serve the interests of a wide range of applications. For instance, Ada [The Department of Defense, 1983] was designed to be useful for all applications supported by the Defense Department, while Cobol [Brown, 1977] was designed to support all business-oriented applications. While Cobol was moderately successful in its goal, Ada has been far less successful.

Other languages are designed to be more *special-purpose* in nature. For instance, Prolog [Clocksin and Mellish, 1997] was designed to serve the narrow interests of natural language processing, theorem proving, and expert systems. C was designed primarily to support the interests of systems programming, although it has since been adopted by a broader range of applications. And Spark/Ada and JML were designed, respectively, to support the formal proof of correctness of Ada and Java programs.

Standards When a programming language receives wide enough usage among programmers, the process of *standardization* usually begins. That is, an effort is made to define a machine-independent standard definition of the language to which all of its

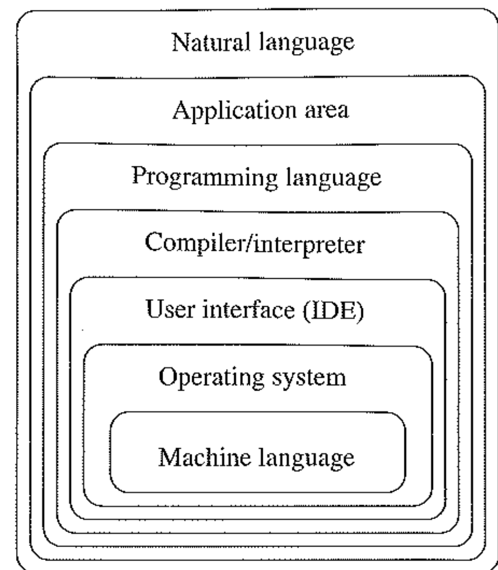


Figure 1.3 Levels of Abstraction in Computing

implementors must adhere. Language standardization generally stabilizes the language across different platforms and programming groups, making program portability feasible.

The two major organizations that oversee and maintain standards for programming languages are the American National Standards Institute (ANSI) and the International Standards Organization (ISO). Several languages have been standardized over the years since the language standardization process began. Some of these, along with their most recent dates of standardization, are:

ANSI/ISO Cobol (2002)
ISO Fortran (2004)
ISO Haskell (1998)
ISO Prolog (2000)
ANSI/ISO C (1999)
ANSI/ISO C++ (2003)
ANSI/ISO Ada (2005)
ANSI Smalltalk (2002)
ISO Pascal (1990)

The language standardization process is complex and time-consuming, with a long period of community involvement and usually a voluminous definition of the standard language as its outcome.

Standardization of programming languages has been accompanied by the standardization of character sets (e.g., the ASCII and UNICODE sets) and libraries (e.g., the C++ Standard Template Library) that directly support programming activities.

The value of standardization to the community is that software and hardware designers play a role in the process and commit their implementations of compilers and

interpreters to conform to the standard. Such conformity is essential to maintain portability of programs across different compilers and hardware platforms.

Some have argued language standardization is a negative influence because it inhibits innovation in language design. That is, standard versions of languages tend to last for long periods of time, thus perpetuating the life of the poor features alongside that of the most valuable features. However, ISO and ANSI standards are reviewed every five years, which provides a modest buffer against prolonged obsolescence.

More information about specific language standards and the standardization process itself can be found at the websites **www.ansi.org** and **www.iso.org**.

Legacy Systems It is well-known that the great majority of a programmer's time is spent maintaining *legacy systems*. Such systems are those software artifacts that were designed and implemented by former programming staff, but are maintained and updated by current staff. The largest body of code for legacy systems is probably written in Cobol, the most dominant programming language for information systems during the last four decades.

In order to support the maintenance of legacy code, updated and improved versions of old languages must be *backward compatible* with their predecessors. That is, old programs must continue to compile and run when new compilers are developed for the updated version. Thus, all syntactic and semantic features, even the ones that are less desirable from an aesthetic point of view, cannot be thrown out without disrupting the integrity of legacy code.

For this reason, older programming languages have become overburdened with features as new versions emerge; languages rarely become more compact as they evolve. This is especially true for Fortran, Cobol, and C++, which was designed as a true extension of C in order to maintain backward compatibility with legacy code.

The design of Java, although a lot of its features are reminiscent of C++, departed from this tradition. As a central theme, Java designers wanted to free their new language from having to support the less desirable features of C++, so they simply cut them out. The result was a more streamlined language, at least temporarily. That is, recent versions of Java have added many new features without removing a comparably large set of obsolete features. Perhaps it is inevitable that, as any language matures, it naturally becomes more feature-burdened in order to address the increasing demands of its application domain.

1.5.2 Outcomes and Goals

In light of these requirements, we are led to ask two important questions:

1 How does a programming language emerge and become successful?
2 What key characteristics make an ideal programming language?

Looking briefly at the past, we first observe that some successful programming languages were designed by individuals, others were designed by industry-wide committees, and still others were the product of strong advocacy by their corporate sponsors. For instance, Lisp and C++ were designed primarily by individuals (John McCarthy and Bjarne Stroustrup, respectively), while the languages Algol, Cobol, and Ada were

designed by committees.[1] PL/I, Java, and C# are the products of their corporate sponsors (IBM, Sun, and Microsoft, respectively). So it's not clear that the design process—individual, committee, or corporate sponsorship—has much overarching influence on the success of a language design.

Since this study aims to prepare readers to evaluate and compare programming languages in general, it is important to have a small set of key characteristics by which you can do this. We shall call these *design goals*, since they have served as effective measures of successful language designs over the years:

- Simplicity and readability
- Clarity about binding
- Reliability
- Support
- Abstraction
- Orthogonality
- Efficient implementation

Simplicity and Readability Programs should be easy to write. They should also be intelligible and easy to read by the average programmer. An ideal programming language should thus support writability and readability. Moreover, it should be easy to learn and to teach.

Some languages, like Basic, Algol, and Pascal, were intentionally designed to facilitate clarity of expression. Basic, for instance, had a very small instruction set. Algol 60 had a "publication language" which provided a standard format for typesetting programs that appeared in published journal articles. Pascal was explicitly designed as a teaching language, with features that facilitated the use of structured programming principles.

Other languages were designed to minimize either the total number of keystrokes needed to express an algorithm or the amount of storage that the compiler would require. Surely, the designers of APL and C valued these economies.

Clarity About Binding A language element is *bound* to a property at the time that property is defined for it. A good language design should be very clear about when the principal binding time for each element to its properties occurs. Here are the major binding times.

- Language definition time: When the language is defined, basic data types are bound to special tags, called *reserved words*, that represent them. For example, integers are bound to the identifier `int`, and real numbers are bound to `float` in the language C.
- Language implementation time: When the compiler or interpreter for the language is written, values are bound to machine representations. For example, the size of an `int` value in C is determined at language implementation time.

1. In the case of Ada, the design process also had an element of competition—several competing designs were evaluated, and Ada emerged as the most suitable language to meet the Defense Department's needs.

- Program writing time: When programs are written in some languages, variable names are bound to types, which remain associated with those names throughout the run of the program. For instance, a variable can be bound to its type at the time it is declared, as in the declaration

```
int x;
```

  which binds the variable `x` to the type `int`.
- Compile time: When programs are compiled, program statements and expressions are bound to equivalent machine language instruction sequences.
- Program load time: When the machine code is loaded, the static variables are assigned to fixed memory addresses, the run-time stack is allocated to a block of memory, and so is the machine code itself.
- Program run time: When programs are running, variables are bound to values, as in the execution of the assignment `x = 3`.

Sometimes, an element can be bound to a property at any one of a number of alternative times in this continuum. For example, the association of a value with a constant may be done either at program compile/load time or at the beginning of run time. When such choices are possible, the notion of *early binding* means simply that an element is bound to a property as early as possible (rather than later) in this time continuum. *Late binding* means to delay binding until the last possible opportunity.

As we shall see in Chapter 4, early binding leads to better error detection and is usually less costly. However, late binding leads to greater programming flexibility (as illustrated in Chapter 14). In general, a language design must take all these alternatives into account, and decisions about binding are ultimately made by the language designer.

Reliability Does the program behave the same way every time it is run with the same input data? Does it behave the same way when it is run on different platforms? Can its behavior be independently specified in a way that would encourage its formal (or informal) verification?

Especially pertinent to these questions is the need to design appropriate exception handling mechanisms into the language. Moreover, languages that restrict aliasing and memory leaks, support strong typing, have well-defined syntax and semantics, and support program verification and validation would have an edge in this category.

Support A good programming language should be easily accessable by someone who wants to learn it and install it on his/her own computer. Ideally, its compilers should be in the public domain, rather than being the property of a corporation and costly to obtain. The language should be implemented on multiple platforms. Courses, textbooks, tutorials, and a wide base of people familiar with the language are all assets that help preserve and extend the vitality of a language.

Questions related to cost may be of more concern to individual programmers and students, rather than to corporate or government employees whose software costs are generally covered by their jobs. The history of programming languages has seen success on both sides. For instance, C, C++, and Java are nonproprietary languages, available in the public domain for a wide variety of platforms. On the other hand, C# and Eiffel are

vendor-supported languages whose use is constrained by their cost and the platforms/ IDEs on which they are implemented.

Abstraction Abstraction is a fundamental aspect of the program design process. Programmers spend a lot of time building abstractions, both data abstractions and procedural abstractions, to exploit the reuse of code and avoid reinventing it. A good programming language supports data and procedural abstraction so well that it is a preferred design tool in most applications.

Libraries that accompany modern programming languages attest to the accumulated experience of programmers in building abstractions. For example, Java's class libraries contain implementations of basic data structures (e.g., vectors and stacks) that, in earlier languages, had to be explicitly designed by programmers themselves. How often have we reinvented a sorting algorithm or a linked list data structure that has probably been implemented thousands of times before?

Orthogonality A language is said to be *orthogonal* if its statements and features are built upon a small, mutually independent set of primitive operations. The more orthogonal a language, the fewer exceptional rules are needed for writing correct programs. Thus, programs in an orthogonal language often tend to be simpler and clearer than those in a non-orthogonal language.

As an example of orthogonality, consider the passing of arguments in a function call. A fully orthogonal language allows any type of object, including a function definition, to be passed as an argument. We shall see examples of this in our discussion of functional programming in Chapter 14.

Other languages restrict the types of objects that can be passed in a call. For example, most imperative languages do not allow function definitions to be passed as arguments, and therefore are not orthogonal in this regard.

Orthogonality tends to correlate with conceptual simplicity, since the programmer doesn't need to keep a lot of exceptional rules in her head. Alan Perlis put it this way:

> It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

On the other hand, non-orthogonality often correlates with efficiency because its exceptional rules eliminate programming options that would be time- or space-consuming.

Efficient Implementation A language's features and constructs should permit a practical and efficient implementation on contemporary platforms.

For a counterexample, Algol 68 was an elegant language design, but its specifications were so complex that it was (nearly) impossible to implement effectively. Early versions of Ada were criticized for their inefficient run-time characteristics since Ada was designed in part to support programs that run in "real time." Programs embedded in systems like airplanes had to respond immediately to a sudden change in input values, like wind speed. Ada programs stood at the intersection of the sensors that provided the readings and the mechanisms that were to respond to them. Early implementations of Ada fell far short of these ambitious performance goals. The harshest critics of Ada's performance were known to utter, "Well, there's 'real time' and then there's 'Ada time'!"

Initial implementations of Java had been criticized on this same basis, although recent refinements to Sun's Java compiling system have improved its run-time performance.

1.6 COMPILERS AND VIRTUAL MACHINES

An implementation of a programming language requires that programs in the language be analyzed, and then translated into a form that can be either:

1. Run by a computer (i.e., a "real machine"), or
2. Run by an interpreter (i.e., a piece of software that simulates a "virtual machine" and runs on a real machine).

Translation of the first kind is often called *compiling*, while translation of the second is called *interpreting*.

Compilers The *compiling* process translates a source program into the language of a computer. Later, the resulting *machine code* can be run on that computer. This process is pictured in Figure 1.4. For example, Fortran, Cobol, C, and C++ are typical compiled languages.

The five stages of the compiling process itself are lexical analysis, syntactical analysis, type checking, code optimization, and code generation. The first three stages are concerned with finding and reporting errors to the programmer. The last two stages are concerned with generating efficient machine code to run on the target computer.

A compiled program's machine code combines with its input to run in a separate step that follows compilation. Run-time errors are generally traceable to the source program through the use of a debugger.

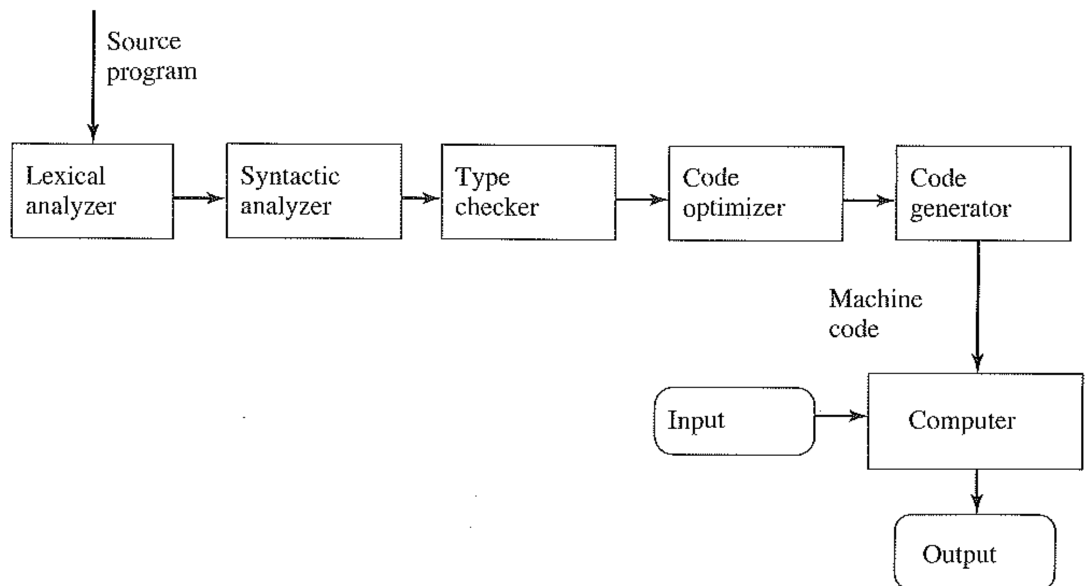
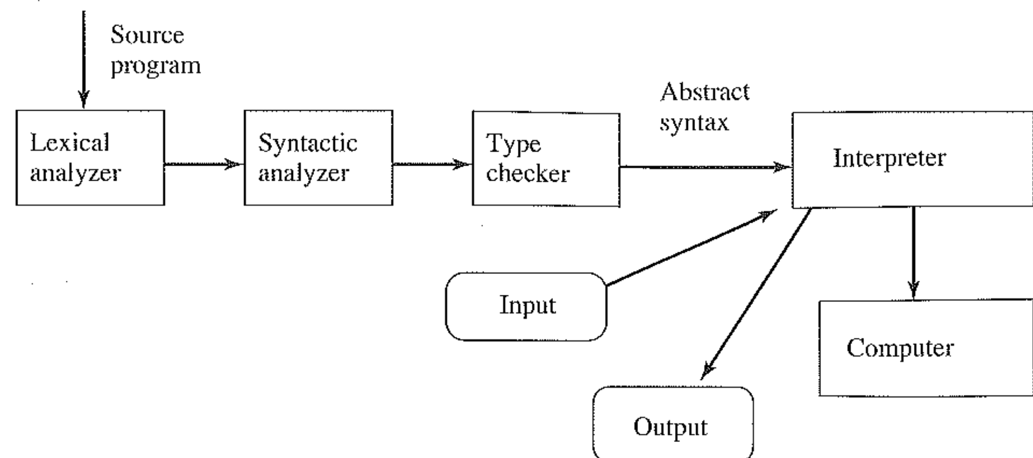


Figure 1.4 The Compile-and-Run Process



| **Figure 1.5** **Virtual Machines and Interpreters**

We shall have some opportunities to explore the first three stages of compiling in Chapters 3 and 6. However, the subjects of code generation and optimization are typically covered in a compiler course and will not be addressed in this text.

Virtual Machines and Interpreters Other languages are implemented using an *interpretive* process, as shown in Figure 1.5. Here, the source program is translated to an intermediary abstract form, which is then interpretively executed. Lisp and Prolog, for instance, are often implemented using interpreters (although compilers for these languages also exist).

As Figure 1.5 suggests, the first three stages of a compiler also occur in an interpreter. However, the abstract representation of the program that comes out of these three stages becomes the subject of execution by an interpreter. The interpreter itself is a program that executes the steps of the abstract program while running on a real machine. The interpreter is usually written in a language distinct from the language being interpreted.

Sometimes a language is designed so that the compiler is written only once, targeting the code for an abstract *virtual machine*, and then that virtual machine is implemented by an interpreter on each of the different real machines of the day. This is the case for Java, whose abstract machine was called the Java Virtual Machine (JVM) [Lindholm and Yellin, 1997]. When making this choice, the Java language designers gave up a bit of efficiency (since interpreted code generally requires more resources than machine code) in favor of flexibility and portability. That is, any change in the Java language specification can be implemented by altering a single compiler rather than a family of compilers.

A major advantage of compiling over interpreting is that the run-time performance of a compiled program is usually more efficient (in time and space) than its interpreted performance. On the other hand, the quality of interaction with the system that the programmer enjoys can be better with an interpreter than with a compiler. Since program development is interactive, individual program segments (functions) can be tested as they

are designed, rather than waiting for the entire program in which they are to be embedded to be complete.[2]

Later versions of Java have gained back some of this efficiency loss by embedding a just-in-time (JIT) compiler into the JVM. This feature enables the JVM byte code to be translated on-the-fly into the native machine code of the host machine before it is executed.

The virtual machine concept is valuable for other reasons that can offset its inherent loss of efficiency. For example, it is expedient to implement a language and its interpreter using an existing virtual machine for the purpose of design or experimentation with the language itself. Similarly, it is useful to study language design foundations and paradigms by using an available *interpreter* that facilitates experimentation and evaluation of programs in that language.

The virtual machine concept has an immediate practical value in this study. That is, you will have access to an interpreter for a small C subset called *Clite*. The use of *Clite* facilitates much of our work because it eliminates machine-specific details that can often hide the principles and other ideas being taught. For instance, in the study of language syntax, you can exercise the *Clite* interpreter to explore the syntactic structure of different language elements, like arithmetic expressions and loops.

1.7 SUMMARY

The study of programming languages includes principles, paradigms, and special topics. The principles are studied both conceptually and in a hands-on way, via the Clite interpreter.

Mastery of one or more new paradigms—imperative, object-oriented, functional, or logic programming—is also important to this study. This activity helps us to appreciate a broader range of computing applications and discover approaches to problem solving with which we are not yet familiar.

Investigation of one or more special topics—event-handling, concurrency, or correctness—allows us to look carefully at three particular language design features and the programming challenges that surround their effective utilization.

In all, we hope that this study will help broaden your view and your technical skills across the wide landscape of programming languages. In particular, you should expect to acquire:

- An appreciation for the use of hands-on tools to examine the principles of language design.
- An appreciation for the value of different programming paradigms that are particularly powerful in specific application domains.
- Laboratory experiences with new languages and design tools, both for testing the principles and for mastering new problem solving techniques.

To support this study, you may occasionally visit the book's website. All the programs that appear in the book can be downloaded from that website, along with other pedagogical and learning aids.

2. However, the development of modern debuggers and IDEs for compiled languages have substantially neutralized this advantage in recent years.

EXERCISES

1.1 An online Web search on "programming languages" will yield links to major information sources for all the major programming languages, past and present. For each of the following languages, use the Web to learn something about it. Write, in your own words, a brief (one paragraph) summary of its distinguishing features, as well as its historical relationship with other languages that preceded or followed it.

(a) Eiffel
(b) Perl
(c) Python

1.2 Give an example statement in C, C++, or Java that is particularly unreadable. Rewrite that statement in a more readable style. For instance, have you ever seen the expression `A[i++]` in a C/C++ program?

1.3 Unreadable code is not the exclusive province of C, C++, and Java. Consider the following strongly held opinions about the weaknesses of particular languages over the last four or more decades:

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration. E. Dijkstra

> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. E. Dijkstra

> APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums. E. Dijkstra

> There does not now, nor will there ever exist, a programming language in which it is the least bit hard to write bad programs. L. Flon

(a) Dijkstra seems not to have much regard for Basic, Cobol, or APL. However, he *did* have a high regard for Algol and its successors. Do enough reading on the Web to determine what general features Algol possessed that would make it superior to languages like Basic, Cobol, and APL.
(b) What does Flon mean by this last statement? Are programming languages inherently flawed? Or is he suggesting that programmers are inherently inept? Or is there a middle-ground interpretation? Explain.

1.4 Give a feature of C, C++, or Java that illustrates orthogonality. Give a feature different from the one discussed in the text that illustrates non-orthogonality.

1.5 Two different implementations of a language are *incompatible* if there are programs that run differently (give different results) under one implementation than under the other. After reading on the Web and in other sources about early versions of Fortran, can you determine whether or not Fortran had incompatible versions? In what specific form (statement type, data type, etc.) did this incompatibility appear? What can be done to eliminate incompatibilities between two different implementations of a language?

1.6 The standardization effort for the language C began in 1982 with an ANSI working group, and the first C standard was completed in 1989. This work was later accepted as an ISO standard in 1990

and has continued until the present day. Read enough on the Web to determine what significant changes have been made to the C standard since 1990.

1.7 Find the C++ standard on the Web. What is meant by *nonconformant* when the standard discusses a language feature supported by a particular compiler? For the C++ compiler on your computer, are there nonconformant features?

1.8 After learning what you can from the Java website **java.sun.com** and other sources, what can you say about the status of the Java standardization effort by ANSI and ISO at this time?

1.9 Find the Python 2.4 version on the Web. What new features does this version add to Python 2.3? What old features of Python are eliminated by the newer version, if any?

1.10 Compare two languages that you know using the goals for language design outlined in Section 1.5.2. For each goal, determine which of the two languages meets the goal better and justify your conclusion with an example. For instance, in comparing C and Java you could conclude that C has more efficient implementations because it compiles to native code rather than an interpreter.