PART 1

Functions and Foundations

1

Introduction

"The Medium Is the Message"
Marshall McLuhan

1.1 PROGRAMMING LANGUAGES

Programming languages are the medium of expression in the art of computer programming. An ideal programming language will make it easy for programmers to write programs succinctly and clearly. Because programs are meant to be understood, modified, and maintained over their lifetime, a good programming language will help others read programs and understand how they work. Software design and construction are complex tasks. Many software systems consist of interacting parts. These parts, or software components, may interact in complicated ways. To manage complexity, the interfaces and communication between components must be designed carefully. A good language for large-scale programming will help programmers manage the interaction among software components effectively. In evaluating programming languages, we must consider the tasks of designing, implementing, testing, and maintaining software, asking how well each language supports each part of the software life cycle.

There are many difficult trade-offs in programming language design. Some language features make it easy for us to write programs quickly, but may make it harder for us to design testing tools or methods. Some language constructs make it easier for a compiler to optimize programs, but may make programming cumbersome. Because different computing environments and applications require different program characteristics, different programming language designers have chosen different trade-offs. In fact, virtually all successful programming languages were originally designed for one specific use. This is not to say that each language is good for only one purpose. However, focusing on a single application helps language designers make consistent, purposeful decisions. A single application also helps with one of the most difficult parts of language design: leaving good ideas out.

THE AUTHOR

I hope you enjoy using this book. At the beginning of each chapter, I have included pictures of people involved in the development or analysis of programming languages. Some of these people are famous, with major awards and published biographies. Others are less widely recognized. When possible, I have tried to include some personal information based on my encounters with these people. This is to emphasize that programming languages are developed by real human beings. Like most human artifacts, a programming language inevitably reflects some of the personality of its designers.

As a disclaimer, let me point out that I have not made an attempt to be comprehensive in my brief biographical comments. I have tried to liven up the text with a bit of humor when possible, leaving serious biography to more serious biographers. There simply is not space to mention all of the people who have played important roles in the history of programming languages.

Historical and biographical texts on computer science and computer scientists have become increasingly available in recent years. If you like reading about computer pioneers, you might enjoy paging through *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* by Dennis Shasha and Cathy Lazere or other books on the history of computer science.

John Mitchell

Even if you do not use many of the programming languages in this book, you may still be able to put the conceptual framework presented in these languages to good use. When I was a student in the mid-1970s, all "serious" programmers (at my university, anyway) used Fortran. Fortran did not allow recursion, and recursion was generally regarded as too inefficient to be practical for "real programming." However, the instructor of one course I took argued that recursion was still an important idea and explained how recursive techniques could be used in Fortran by managing data in an array. I am glad I took that course and not one that dismissed recursion as an impractical idea. In the 1980s, many people considered object-oriented programming too inefficient and clumsy for real programming. However, students who learned about object-oriented programming in the 1980s were certainly happy to know about

these "futuristic" languages in the 1990s, as object-oriented programming became more widely accepted and used.

Although this is not a book about the history of programming languages, there is some attention to history throughout the book. One reason for discussing historical languages is that this gives us a realistic way to understand programming language trade-offs. For example, programs were different when machines were slow and memory was scarce. The concerns of programming language designers were therefore different in the 1960s from the current concerns. By imaging the state of the art in some bygone era, we can give more serious thought to why language designers made certain decisions. This way of thinking about languages and computing may help us in the future, when computing conditions may change to resemble some past situation. For example, the recent rise in popularity of handheld computing devices and embedded processors has led to renewed interest in programming for devices with limited memory and limited computing power.

When we discuss specific languages in this book, we generally refer to the original or historically important form of a language. For example, "Fortran" means the Fortran of the 1960s and early 1970s. These early languages were called Fortran I, Fortran II, Fortran III, and so on. In recent years, Fortran has evolved to include more modern features, and the distinction between Fortran and other languages has blurred to some extent. Similarly, Lisp generally refers to the Lisps of the 1960s, Smalltalk to the language of the late 1970s and 1980s, and so on.

1.2 GOALS

In this book we are concerned with the basic concepts that appear in modern programming languages, their interaction, and the relationship between programming languages and methods for program development. A recurring theme is the trade-off between language expressiveness and simplicity of implementation. For each programming language feature we consider, we examine the ways that it can be used in programming and the kinds of implementation techniques that may be used to compile and execute it efficiently.

1.2.1 General Goals

In this book we have the following general goals:

- To understand the *design space* of programming languages. This includes concepts and constructs from past programming languages as well as those that may be used more widely in the future. We also try to understand some of the major conflicts and trade-offs between language features, including implementation costs.
- To develop a better understanding of the languages we currently use by comparing them with other languages.
- To understand the programming techniques associated with various language features. The study of programming languages is, in part, the study of conceptual frameworks for problem solving, software construction, and development.

Many of the ideas in this book are common knowledge among professional programmers. The material and ways of thinking presented in this book should be useful to you in future programming and in talking to experienced programmers if you work for a software company or have an interview for a job. By the end of the course, you will be able to evaluate language features, their costs, and how they fit together.

1.2.2 Specific Themes

Here are some specific themes that are addressed repeatedly in the text:

- *Computability:* Some problems cannot be solved by computer. The undecidability of the halting problem implies that programming language compilers and interpreters cannot do everything that we might wish they could do.
- *Static analysis:* There is a difference between compile time and run time. At compile time, the program is known but the input is not. At run time, the program and the input are both available to the run-time system. Although a program designer or implementer would like to find errors at compile time, many will not surface until run time. Methods that detect program errors at compile time are usually conservative, which means that when they say a program does not have a certain kind of error this statement is correct. However, compile-time error-detection methods will usually say that some programs contain errors even if errors may not actually occur when the program is run.
- *Expressiveness versus efficiency:* There are many situations in which it would be convenient to have a programming language implementation do something automatically. An example discussed in Chapter 3 is memory management: The Lisp run-time system uses garbage collection to detect memory locations no longer needed by the program. When something is done automatically, there is a cost. Although an automatic method may save the programmer from thinking about something, the implementation of the language may run more slowly. In some cases, the automatic method may make it easier to write programs and make programming less prone to error. In other cases, the resulting slowdown in program execution may make the automatic method infeasible.

1.3 PROGRAMMING LANGUAGE HISTORY

Hundreds of programming languages have been designed and implemented over the last 50 years. As many as 50 of these programming languages contained new concepts, useful refinements, or innovations worthy of mention. Because there are far too many programming languages to survey, however, we concentrate on six programming languages: Lisp, ML, C, C++, Smalltalk, and Java. Together, these languages contain most of the important language features that have been invented since higher-level programming languages emerged from the primordial swamp of assembly language programming around 1960.

The history of modern programming languages begins around 1958–1960 with the development of Algol, Cobol, Fortran, and Lisp. The main body of this book

covers Lisp, with a shorter discussion of Algol and subsequent related languages. A brief account of some earlier languages is given here for those who may be curious about programming language prehistory.

In the 1950s, a number of languages were developed to simplify the process of writing sequences of computer instructions. In this decade, computers were very primitive by modern standards. Most programming was done with the native machine language of the underlying hardware. This was acceptable because programs were small and efficiency was extremely important. The two most important programming language developments of the 1950s were Fortan and Cobol.

Fortran was developed at IBM around 1954–1956 by a team led by John Backus. The main innovation of Fortran (a contraction of formula translator) was that it became possible to use ordinary mathematical notation in expressions. For example, the Fortran expression for adding the value of i to twice the value of j is i + 2*j. Before the development of Fortran, it might have been necessary to place i in a register, place j in a register, multiply j times 2 and then add the result to i. Fortran allowed programmers to think more naturally about numerical calculation by using symbolic names for variables and leaving some details of evaluation order to the compiler. Fortran also had subroutines (a form of procedure or function), arrays, formatted input and output, and declarations that gave programmers explicit control over the placement of variables and arrays in memory. However, that was about it. To give you some idea of the limitations of Fortran, many early Fortran compilers stored numbers $1, 2, 3 \ldots$ in memory locations, and programmers could change the values of numbers if they were not careful! In addition, it was not possible for a Fortran subroutine to call itself, as this required memory management techniques that had not been invented yet (see Chapter 7).

Cobol is a programming language designed for business applications. Like Fortran programs, many Cobol programs are still in use today, although current versions of Fortran and Cobol differ substantially from forms of these languages of the 1950s. The primary designer of Cobol was Grace Murray Hopper, an important computer pioneer. The syntax of Cobol was intended to resemble that of common English. It has been suggested in jest that if object-oriented Cobol were a standard today, we would use "add 1 to Cobol giving Cobol" instead of "C++".

The earliest languages covered in any detail in this book are Lisp and Algol, which both came out around 1960. These languages have stack memory management and recursive functions or procedures. Lisp provides higher-order functions (still not available in many current languages) and garbage collection, whereas the Algol family of languages provides better type systems and data structuring. The main innovations of the 1970s were methods for organizing data, such as records (or structs), abstract data types, and early forms of objects. Objects became mainstream in the 1980s, and the 1990s brought increasing interest in network-centric computing, interoperability, and security and correctness issues associated with active content on the Internet. The 21st century promises greater diversity of computing devices, cheaper and more powerful hardware, and increasing interest in correctness, security, and interoperability.

1.4 ORGANIZATION: CONCEPTS AND LANGUAGES

There are many important language concepts and many programming languages. The most natural way to summarize the field is to use a two-dimensional matrix, with languages along one axis and concepts along the other. Here is a partial sketch of such a matrix:

Language	Expressions	Functions	Heap storage	Exceptions	Modules	Objects	Threads
Lisp	x	x	x				
C	x	x	x				
Algol 60	x	x					
Algol 68	x	x	x				x
Pascal	x	x	x				
Modula-2	x	x	x		x		
Modula-3	x	x	x	x	x	x	
ML	x	x	x	x	x		
Simula	x	x	x			x	x
Smalltalk	x	x	x	x		x	x
C++	x	x	x	x	x	x	
Objective C	x	x	x			x	
Java	x	x	x	x	x	x	x

Although this matrix lists only a fraction of the languages and concepts that might be covered in a basic text or course on the programming languages, one general characteristic should be clear. There are some basic language concepts, such as expressions, functions, local variables, and stack storage allocation that are present in many languages. For these concepts, it makes more sense to discuss the concept in general than to go through a long list of similar languages. On the other hand, for concepts such as objects and threads, there are relatively few languages that exhibit these concepts in interesting ways. Therefore, we can study most of the interesting aspects of objects by comparing a few languages. Another factor that is not clear from the matrix is that, for some concepts, there is considerable variation from language to language. For example, it is more interesting to compare the way objects have been integrated into languages than it is to compare integer expressions. This is another reason why competing object-oriented languages are compared, but basic concepts related to expressions, statements, functions, and so on, are covered only once, in a concept-oriented way.

Most courses and texts on programming languages use some combination of language-based and concept-based presentation. In this book a concept-oriented organization is followed for most concepts, with a language-based organization used to compare object-oriented features.

The text is divided into four parts:

Part 1: Functions and Foundations (Chapters 1–4)
Part 2: Procedures, Types, Memory Management, and Control (5–8)
Part 3: Modularity, Abstraction and Object-Oriented Programming (9–13)

Part 4: Concurrency and Logic Programming (14 and 15)

In Part 1 a short study of Lisp is presented, followed by a discussion of compiler structure, parsing, lambda calculus, and denotational semantics. A short chapter provides a brief discussion of computability and the limits of compile-time program analysis and optimization. For C programmers, the discussion of Lisp should provide a good chance to think differently about programming and programming languages.

In Part 2, we progress through the main concepts associated with the conventional languages that are descended in some way from the Algol family. These concepts include type systems and type checking, functions and stack storage allocation, and control mechanisms such as exceptions and continuations. After some of the history of the Algol family of languages is summarized, the ML programming language is used as the main example, with some discussion and comparisons using C syntax.

Part 3 is an investigation of program-structuring mechanisms. The important language advances of the 1970s were abstract data types and program modules. In the late 1980s, object-oriented concepts attained widespread acceptance. Because object-oriented programming is currently the most prominent programming paradigm, in most of Part 3 we focus on object-oriented concepts and languages, comparing Smalltalk, C++, and Java.

Part 4 contains chapters on language mechanisms for concurrent and distributed programs and on logic programming.

Because of space limitations, a number of interesting topics are not covered. Although scripting languages and other "special-purpose" languages are not covered explicitly in detail, an attempt has been made to integrate some relevant language concepts into the exercises.

2

Computability

Some mathematical functions are computable and some are not. In all general-purpose programming languages, it is possible to write a program for each function that is computable in principle. However, the limits of computability also limit the kinds of things that programming language implementations can do. This chapter contains a brief overview of computability so that we can discuss limitations that involve computability in other chapters of the book.

2.1 PARTIAL FUNCTIONS AND COMPUTABILITY

From a mathematical point of view, a program defines a function. The output of a program is computed as a function of the program inputs and the state of the machine before the program starts. In practice, there is a lot more to a program than the function it computes. However, as a starting point in the study of programming languages, it is useful to understand some basic facts about computable functions.

The fact that not all functions are computable has important ramifications for programming language tools and implementations. Some kinds of programming constructs, however useful they might be, cannot be added to real programming languages because they cannot be implemented on real computers.

2.1.1 Expressions, Errors, and Nontermination

In mathematics, an expression may have a defined value or it may not. For example, the expression $3 + 2$ has a defined value, but the expression $3/0$ does not. The reason that $3/0$ does not have a value is that division by zero is not defined: division is defined to be the inverse of multiplication, but multiplication by zero cannot be inverted. There is nothing to try to do when we see the expression $3/0$; a mathematician would just say that this operation is undefined, and that would be the end of the discussion.

In computation, there are two different reasons why an expression might not have a value:

ALAN TURING

Alan Turing was a British mathematician. He is known for his early work on computability and his work for British Intelligence on code breaking during the Second World War. Among computer scientists, he is best known for the invention of the Turing machine. This is not a piece of hardware, but an idealized computing device. A Turing machine consists of an infinite tape, a tape read–write head, and a finite-state controller. In each computation step, the machine reads a tape symbol and the finite-state controller decides whether to write a different symbol on the current tape square and then whether to move the read–write head one square left or right. The importance of this idealized computer is that it is both very simple and very powerful.

Turing was a broad-minded individual with interests ranging from relativity theory and mathematical logic to number theory and the engineering design of mechanical computers. There are numerous published biographies of Alan Turing, some emphasizing his wartime work and others calling attention to his sexuality and its impact on his professional career.

The ACM Turing Award is the highest scientific honor in computer science, equivalent to a Nobel Prize in other fields.

- *Error termination:* Evaluation of the expression cannot proceed because of a conflict between operator and operand.
- *Nontermination:* Evaluation of the expression proceeds indefinitely.

An example of the first kind is division by zero. There is nothing to compute in this case, except possibly to stop the computation in a way that indicates that it could not proceed any further. This may halt execution of the entire program, abort one