CHAPTER 1

Introduction

1.1. The Role of Programming Languages

The history of programming languages spans a little more than half a century, roughly the lifespan of the electronic computer itself. These artifacts of the information age can be viewed as instances of the youngest form of communication developed by society, communication between man and machine. If this were all there was to programming languages, we would have very little to discuss. After all, every machine language provides pretty much the same features, conditioned only by the architectural variations between different computers. In fact of course, there are many other issues that influence the design of a programming language. These languages must be readable to human beings, in the sense that they must enable a programmer to express clearly and unambiguously the logical structure of a program, at the same time that they enable the underlying software and hardware to implement exactly what the programmer has expressed. But there is more: a programming language tends to embody those principles of design and language structure that were popular at the time that the language was introduced, as well as the particular philosophies and perspectives held by its creators. They are designed to be precise tools but, like many scientific constructs, they are cultural artifacts (more so than their designers would like to admit), influenced by and influencing the information processing subculture in which they play such a central role. One has only to witness the rise of object-oriented technology or the recent "Java" phenomenon to be acutely aware of this.

A programming language, unlike a natural language and more like the language of mathematics, is intended to have a narrow focus but to express the ideas within its field of application very clearly and precisely. Sadly, the former of these two statements is more commonly true than the latter. The meanings of certain constructs in many languages are often not as
clearly defined as is necessary to guarantee that they are fully understood by programmers, and that every implementation is consistent with the specifications. There is, at times, a distinct rift between the intentions of the designer and the behavior produced by a particular implementation. This inconsistency is more likely to be a product of imprecisely worded specifications than of careless implementation. But, that is a long story, to which we shall return later.

1.2. Language Design

The focus of our discussion will be on general purpose, high-level languages, languages such as FORTRAN, Lisp, Algol, PL/I, BASIC, Pascal, C, Modula-2, ML, Ada, Smalltalk, C++, Eiffel, and Java, to name but a few. These are indeed general purpose, in the sense that they can be used interchangeably to tackle almost any programming task. However, this cachet "general purpose" should not be taken to indicate that they are all equally suited for every application, or equally efficient. Each language in the list represents the result of trade-offs between various desirable (and sometimes ill-defined) characteristics: ease of use, speed, flexibility, expressiveness, power, and so on.

For example, BASIC and FORTRAN are in many ways closely related. However, FORTRAN was originally designed for scientific computation, whereas BASIC’s simplified syntax and interpreted environment provided a friendly and flexible introduction to programming. It is much easier for a beginner to write a small program in BASIC than in FORTRAN, on the other hand FORTRAN will always out-perform BASIC in any serious computational task. It is interesting to note that these two elderly languages, in updated garb (FORTRAN-90 and VISUAL BASIC) are still vital participants in the current language scene.

Languages can differ in deeper ways than this. Most programming languages follow the procedural model of computation promoted by John von Neumann in the 1950s. with its attendant data/instruction store and linear sequencing of control. However, other paradigms have gained some popularity and have generated influential languages. The most important class is that of the functional languages, for example, Lisp, Scheme, and ML. In their purest form these transform input data directly into output, eliminating the need for a store (memory). Logic languages, e.g., Prolog, express logical relationships between data items. Because these languages express relationships rather than specify actions, they eliminate the need for explicit sequencing of execution, at least in principle. In practice, some
implicit and explicit sequencing is required in order to maintain an acceptable level of performance. Data flow languages replace the store and sequential execution with a dynamic flow of data through a network of control and computation nodes. Finally, parallel languages replicate the procedural, functional, or logical models (the data flow model is itself a parallel model) to allow parallel access to large data structures and/or multiple streams of control. Other qualities of program organization, such as modularity or conformance to an object model, are to a great extent independent of these considerations, so they can be applied to any of the above paradigms.

Literally hundreds of languages have been designed over the last fifty years. While the tools available for design and implementation have grown more sophisticated with time, the design of a language is still more of an art than a science.

There are two components to the specification of a language, its structural form (syntax) and its meaning (semantics), and to a limited extent these are independent of one another. We see this in natural language, too. The following sentence is grammatically incorrect, but we have very little difficulty understanding it: "Joe, you and me is real knuckleheads!" On the other hand, "A threaded bird clamps the eclectic truth," is grammatically correct, yet we have not the slightest idea what it means.

Of course, syntax and semantics are not completely independent of one another. While, often, we can summarize the intent of an ungrammatical sentence, both in English and in a programming language, this is invariably deduced from contextual clues or from some innate redundancy in the language. Whatever independence there may appear to be, at times, between syntax and semantics, ultimately grammatical structure is the frame upon which meaning is hung.

1.3. Translation

There is a great difference in the way that natural languages and computer languages are understood and used. Once a natural language has been designed, people learn its syntax and meaning with reference to their own native tongue or through communication with others who speak it. Of course, very few of the languages that human beings use to communicate with one another are explicitly designed (Esperanto is one exception; Mathematics is another, although its focus is, like that of programming languages, quite narrow), and the process by which people learn their own native languages is not well understood.
Programming languages, on the other hand, are designed primarily for communication with machines, and machines are not yet capable of learning new languages. So, unless a machine has been designed to understand the language in question (for example, machine code or Lisp, say, for a Lisp machine), it is necessary to write a translator. Translators for programming languages take two forms: interpreters and compilers. An interpreter takes a program in the language and directly executes the actions described by the code, step by step. In other words, the interpreter is a simulator for a machine, a virtual machine, whose native code is the source language. A compiler, on the other hand, translates the program into a program in another language, usually the machine code for the target computer. The translated program is then "interpreted" directly by the hardware itself.

The process of creating a translator involves a double challenge. It requires an unambiguous description of the language itself, and also equally precise information about the target machine (or language). Of course, this raises the question as to what form language specifications should take. In the 1950s there was no universally accepted notation for describing languages. Some of the earliest languages (e.g., FORTRAN) were implemented as they were designed; their designs and, specifically, their meanings were embodied in their implementations. In the 1960s, notation for expressing syntactical structure was developed, and more recently the same has been done for semantics.

A clear and unambiguous notation is only half the battle. A good notation may enable one to write a precise description of a language, but the accuracy of the actual translation will still be dependent on the implementor’s complete understanding of the specifications and the care taken in converting them into a translator. Ideally, one would like to have tools that can take specifications of the language and the target environment and generate a translator automatically. Parts of this process are well understood. For example, there are tools which will take a syntax description and turn it into a parser (syntax checker) for the language. These have been available for almost twenty five years and are commonly used in the implementation of translators. They allow the inclusion of actions, driven by the parsing process, which perform code generation and other tasks associated with the translation. However, there are other steps in the process that are more resistant to efficient implementation.

1.4. Semantic Specification

Fully automatic generation of a translator from syntactic and semantic
specifications is still more the subject of research projects than of practical software tools. This state of affairs is a direct result of the complexity of semantic specification. While fully automatic systems do exist, the performance of the code they generate is not usually acceptable for practical applications. In fact, there is not yet agreement on the form that semantic descriptions should take. There are a number of rival schools of thought on this matter; they can be summed up in three different styles of specification: **axiomatic**, **operational**, and **denotational** semantics. Ignoring axiomatic semantics, which is really more suited to the analysis of the behavior of programs than to language specification, the other two approaches embody different views of the translation process. Operational semantics describes the behavior of a language in terms of its implementation on an abstract machine. Denotational semantics, at least in its most popular form, describes a language by means of its translation into an abstract functional language. One could contrast these two approaches, perhaps a little naively, as follows: an operational description is an abstract interpreter, whereas a denotational description is an abstract compiler.

Without judging the relative merits of these, we use the denotational approach in the chapters that follow, for the simple reason that it is easy to generate a compiler from a denotational description. All that is required is that one implement the abstract functional descriptions specified by the semantics, using a real functional language. For example, if one were to use Lisp (as we do in this book), one would create a compiler that translates the language in question into Lisp. Actually, the restriction to functional languages, although natural, is not necessary. Almost any general purpose language that allows the passing of functions as parameters could be used (the language C, for example).

From a practical point of view, the compilers so produced are inefficient and difficult to optimize. This is, to some extent, the reason why automation of this process has met with limited success. However, for our purposes this is not a severe limitation. We will be studying the behavior of programming languages and their components. We are interested in observing how behavior matches specification, and analyzing the effect of different design decisions. The efficiency of the resulting programs is not important to us.

### 1.5. The Structure of Procedural Languages

Every natural language has a structure that it has acquired from the environment in which it is spoken and from the mental landscape of its
speakers. Much, but not all, of this structure is shared by every known language. Because these structural forms are not uniformly distributed across all languages, it is possible to group natural languages according to their similarities. Thus, we talk of the Indo-European family of languages, the Austro-Asiatic, the Sino-Tibetan, and so on, and within each of these families we identify subfamilies, according to further similarities in their structural form (Germanic, Indo-Iranian, Italic, Slavic are all subfamilies of the Indo-European family). The same is true of programming languages, albeit on a much simpler level. The largest group is that of the procedural languages (sometimes called imperative languages). Because of its influential role, this is the group on which we will focus in this book.

The most significant component of any procedural language is the assignment statement. Assignment is the defining feature of this family. All other constructs are provided to exert control over this statement. It is assignment that marks the progress of program execution and records the computations done. Beyond this, branching and looping mechanisms are common to all procedural languages, as is some form of subprogram construct (subroutines or procedures). The subtler issues of block structure, scope, naming, parameter passing, and exception handling are addressed by almost all languages in the procedural family. However, each language provides its own interpretation of these mechanisms.

While the syntax may vary from one procedural language to another, the similarities are more striking than the differences. There are variations, too, in the semantics of these constructs, but again the range of differences is relatively small. So, it is reasonable for us to discuss this family as a group, without becoming mired in the inconsequential details of individual languages. This way, we can understand the concepts common to the family, at the same time that we investigate the semantic differences that lead to subtle variations in behavior.

Procedural languages are our primary focus, but we cannot neglect the family of functional languages. The functional model provides a higher level view of the computing process, one that is closer to the language of mathematics (that of function definition and composition). While functional languages do not enjoy quite the popularity of the procedural family, they present an elegant programming model that avoids the state-based approach of the procedural languages. This model is useful, at times, even to procedural programmers. The value of both of these approaches is shown by the fact that most procedural languages are sufficiently powerful to support the functional model, and on the other hand some functional languages (notably Lisp) provide support for the procedural model.

So, the goal of this book is to investigate the unifying principles and variations in semantics that characterize these two language families.