



# Chapter 1

# Programming

# Languages

## 1.1 Introduction

Computer programs are practical magic. To craft a program is to construct a complex and beautiful incantation; to run a program is to discover with delight that *the magic actually works!* Even if the program does not do exactly what you intended, it does something, which is more than can be said of most other kinds of incantations. The delight of practical magic draws many beginners to computer programming and continues to reward experienced programmers. The author, at least, has not found this pleasure to diminish after 20 years of programming practice.

This book is about programming languages. It gives tutorial introductions to three languages: ML, Java, and Prolog.<sup>1</sup> These languages are very different from each other. Knowing a little about them gives you three widely

---

1. The published definitions of the oldest languages usually gave their names in all upper case: FORTRAN, COBOL, and BASIC. Newer definitions, even for dialects of the older languages, usually give the names in mixed case: Fortran. This book follows the convention of using mixed case for names that are pronounced as words (such as Java and Prolog) and all capital letters only for names that are pronounced as sequences of letters (such as ML, which is pronounced “em ell”).

separated points from which to triangulate the principles of programming languages. There are good, free implementations of these three languages on a variety of platforms, so you can enjoy tinkering with them. Interleaved with the tutorial chapters are philosophical chapters that discuss important concepts of programming languages more abstractly. Although these chapters are abstract, they are not particularly mathematical. Programming languages have, to be sure, an interesting and elegant mathematics, but there is plenty to appreciate about programming languages without going in that direction, where not all readers can follow or care to follow.

Before you read further, be warned: this book assumes that you already know at least one programming language well—at the level normally achieved by a college student after two semesters of study. It does not matter which language you know. But if you have not programmed before, this is not the right place to start learning how.

The rest of this chapter explains what makes programming languages such an interesting subject: the amazing variety, the odd controversies, the intriguing evolution, and the many connections to other branches of computing.

## ■ ■ ■ 1.2 ■ ■ ■ The Amazing Variety

One of the things that makes programming languages so fascinating is their diversity. Here is a quick glance at four very different kinds of languages, including the three core languages of this book.

### Imperative Languages

Here is an example in C, an imperative language. This is a function that computes the factorial of a non-negative integer.

```
int fact(int n) {
    int sofar = 1;
    while (n > 0) sofar *= n--;
    return sofar;
}
```

This example shows two of the hallmarks of an imperative language: assignment and iteration. The C statement `sofar *= n--;` is an assignment to the variable `sofar`. The variable has a current value that is changed each time an assignment is made. The statement also affects `n`, reducing its current value by one. The `while` loop repeats the statement over and over. Eventually, `n`'s current value should

become zero and the loop will stop. Because the values of variables change over time, the order in which the statements of the program are executed is critical.

To most C programmers, these ideas are so elementary that they go unexamined. Of course the order of execution is critical; of course variables have current values that can be changed by assignment. But there are many languages that have no such ideas; no assignment statement, no iteration, and no concept of a changeable “current value” of a variable.

## Functional Languages

Here is the same function implemented in the language ML:

```
fun fact x =
  if x <= 0 then 1 else x * fact(x - 1);
```

This shows two of the hallmarks of functional programs: recursion and single-valued variables. Recursion is as natural to ML programmers as iteration is to C programmers.

Here is the same function in another functional language, Lisp:

```
(defun fact (x)
  (if (<= x 0) 1 (* x (fact (- x 1)))))
```

As you can see, Lisp has a unique syntax. This syntactic difference is superficial. Deep down, the Lisp `fact` function and the ML `fact` function are much more closely related to each other than they are to the C `fact` function: they are both written in a functional style, without assignment or iteration.

These examples look elegant compared with the C version, but the comparison is not really fair. The factorial function is exactly the sort of thing that functional languages do most naturally. There are other kinds of operations (for instance, matrix multiplication) that would show imperative languages to better advantage.

## Logic Programming Languages

If the factorial function is most natural for ML, it is perhaps least natural for Prolog. Nevertheless, here it is in Prolog:

```
fact(X,1) :-
  X == 1.
fact(X,Fact) :-
  X > 1,
  NewX is X - 1,
  fact(NewX,NF),
  Fact is X * NF.
```

The first two lines give a rule that allows the Prolog system to conclude that the factorial of  $X$  is 1 whenever  $X$  is equal to 1. The remaining five lines give a more general way to conclude that the factorial of  $X$  is some value `Fact`. They say, “To prove that the factorial of  $X$  is `Fact`, you must do the following things: prove that  $X$  is greater than one, prove that `NewX` is one less than  $X$ , prove that the factorial of `NewX` is `NF`, and prove that `Fact` is  $X$  times `NF`.” Expressing a program in terms of rules about logical inference is the hallmark of logic programming. It is not particularly well suited to computing mathematical functions, but there are problem domains where it really shines, as you will see starting in Chapter 19.

## Object-Oriented Languages

In Java, the factorial function looks almost the same as in C. But Java is an object-oriented language, which means that in addition to being imperative, it also makes it easier to solve programming problems using objects. An object is a little bundle of data that knows how to do things to itself. For example, this is a Java definition for objects that hold an integer value and know how to report both that value and its factorial:

```
public class MyInt {
    private int value;
    public MyInt(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public MyInt getFact() {
        return new MyInt(fact(value));
    }
    private int fact(int n) {
        int sofar = 1;
        while (n > 1) sofar *= n--;
        return sofar;
    }
}
```

This object-oriented example looks wordy in comparison with the others, but again, the comparison is not really fair. The object-oriented style helps keep large programs organized; it does not show to advantage on very small examples.

You have now seen examples from four language families: imperative languages (like C), functional languages (like ML), logic programming languages (like Prolog), and object-oriented languages (like Java). With a little effort, you can fit any language into one of these four categories. But these four categories are not well

defined, and the best way to categorize a language is not always clear. There are plenty of languages that straddle the boundaries. There are, in fact, many categories of languages, not just these four. Languages have been variously categorized as applicative, concurrent, constraint, declarative, definitional, procedural, scripting, single-assignment, and so forth.

Some programming languages are so unique that assigning them to a language family would be pointless. Take Forth, for example. Here is a factorial function in Forth:

```
: FACTORIAL
  1 SWAP BEGIN ?DUP WHILE TUCK * SWAP 1- REPEAT ;
```

Forth is a stack-oriented language, similar to the page-oriented graphical language PostScript. The Forth word `SWAP` exchanges the two top elements of the stack. Forth could be called an imperative language, but it has little in common with most other imperative languages.

Consider APL. Here is an APL expression to compute the factorial of  $X$ :

$$\times / \iota X$$

APL is famous for using a large character set that includes many symbols not present on ordinary keyboards. The expression above works by expanding  $X$  into the vector of the numbers 1 through  $X$  and then multiplying them all together. (In practice, you would not write the expression that way in APL, since you could just write  $! X$  for the factorial of  $X$ .) APL could be called a functional language, but it has little in common with most other functional languages.

## 1.3 The Odd Controversies

There are some fields of study that seem inherently controversial. Evolution, the big bang, human sexuality—any subject that makes regular appearances in the Science section of the *New York Times* is bound to stir up an argument. But programming languages? Our field of study rarely makes it into the popular press, and comparatively few people know or care about it. Nevertheless, and oddly enough, the study of programming languages is full of heated debates.

First of all, there are partisans for every language, eagerly defending the virtues of their favorite against all others. Some partisans of ML are at daggers drawn with partisans of Haskell. Partisans of Forth will be sorry, but not surprised, to find they are holding yet another book that neglects their favorite language. Some partisans of Prolog cannot understand why logic programming has not been universally

adopted. Some partisans of Fortran are confident that theirs is the not only the first but also the most important high-level language.

Among the proponents of a particular language are heated debates of another sort. The standards for programming languages are often developed by international committees. Who are the stakeholders, and who gets to participate in such decisions? What will and will not be in the next official version of the language? The development of language standards can be astonishingly slow, complicated, and rancorous.

Of more interest to us is the frequent disagreement about basic definitions. We have already used at least one heavily debated term: *object oriented*. Exactly what are the properties a language must have to be considered object oriented? This book dodges the question, giving only an informal description of object-oriented languages. It generally avoids giving strict definitions for such terms for two reasons. First, it would not be completely honest to pretend there is a definition, when in fact there are many competing and conflicting definitions. Second, strict definitions wouldn't be useful; the only use for a strict definition of "object oriented" is fueling partisan debates (my language *is* and your language *isn't!*). In the case of this book, informal descriptions are more useful. Some languages are more object oriented than others; such things are left to the reader's judgment.<sup>2</sup>

## 1.4 The Intriguing Evolution

People invent new languages all the time. Well, perhaps not *completely* new—language designers build on ideas from languages that have come before. But the designer of a new language has a free hand, since there are no issues of compatibility with existing code. Some new languages are widely used, while others languish. Used or unused, new languages can provide the ideas from which the next generation of languages develops.

A good deal of language invention is slow and incremental. Almost all languages, even relatively new ones, evolve multiple dialects. The venerable language Fortran is mentioned above; in fact, there is no such thing as Fortran. There are the initial designs and implementations of Fortran at IBM, which date from the mid-

---

2. In mathematical circles, a rancorous yet obscure argument is sometimes called a "frog-mouse battle." Albert Einstein famously referred to a hot debate on the foundations of mathematics, involving the mathematicians David Hilbert and L. E. J. Brouwer, as a "Frosch-Mäuse-Krieg" (that is, frog-mouse battle). The term comes from an ancient and often-retold fable; it appears in the Hellenistic parody of *The Iliad* called *Batrachomyomachy* (again, frog-mouse battle). The odd controversies surrounding programming languages often have a frog-mouse-battle quality.

1950s. There is a sequence of standards: Fortran II, Fortran III, Fortran IV, Fortran 66, Fortran 77, Fortran 90, Fortran 95, and, perhaps, Fortran 2000.<sup>3</sup> New standards supplant old ones slowly, if ever; many Fortran programmers still work in Fortran 77. For each dialect, different platforms may have different implementations, each interpreting the standard for that dialect differently. In addition, there are many special-purpose dialects. There are, for example, a dozen or more dialects of Fortran that add constructs for parallel programming.

Whether suddenly or gradually, *programming languages change*. They change quickly compared with natural languages. If you continue to program computers, you will almost certainly have to repeatedly learn new dialects and new languages. It would be easier to have just one favorite programming language forever, as comfortable as old clothes. Since that is not feasible, at least we can enjoy watching the story of programming languages unfold.

## 1.5 The Many Connections

This book is about languages, not about how to write beautiful programs. However, it will often have to address questions of programming style, because languages are never neutral on the subject. Each language favors a programming style—a particular approach to algorithmic problem-solving. The connection between programming languages and programming practice affects them both.

In one direction, languages guide programmers toward particular programming styles. Object-oriented languages like Java guide programmers toward a style that uses objects; functional languages like ML guide programmers toward a style that uses many small functions; logic languages like Prolog guide programmers toward a style that expresses problems as searches in a logically defined space of solutions. Writing in a style that is unnatural for the language you are using is always possible, but rarely a good idea. You can write imperative programs with loops and assignment in ML (although this book will not show you how!). You can write Java programs that do not create any objects and just pack a lot of ordinary imperative code into a single, large class definition. In Java, or any language that supports recursion, you can write function-oriented programs that avoid iteration and assignment. It isn't natural, and it isn't usually wise, but it can be done. These are exceptions that prove the rule. When you code in a style that is unnatural for the

---

3. Standards committees usually name standards optimistically. The Fortran 90 standard was revised as Fortran 82, 8X, and 88, before being released as Fortran 90 (in 1991!).

language, when you fight the paradigm, you can feel the language working against you.

In the other direction, programming practice often guides programmers toward new programming-language ideas. For example, recursion and conditional expressions were introduced in Lisp because Lisp's designer, John McCarthy, found that he needed them in the artificial intelligence applications he was developing. Classes and objects were introduced in Simula because Simula's designers, Kristen Nygaard and Ole-Johan Dahl, needed them for the large simulations they were implementing. Chapter 24 has more of these kinds of stories from the history of programming languages.

Programming languages are connected not just to programming practice, but also to many other branches of computer science. Language evolution drives and is driven by hardware evolution. The theory of formal languages and automata, one of the more mathematical parts of computer science, has many applications in the definition and implementation of programming languages. Operating systems interact closely with language systems. Every application area—artificial intelligence, networking, database management, business applications, numeric processing, and so forth—contributes its own point of view to the problem of programming-language design.

## 1.6 **A Word about Application Programming Interfaces**

Today's commercial programming languages are supported by large standardized libraries of predefined code. Such a library is called an API (Application Programming Interface). An API might include code to implement basic data structures (stacks, queues, hash tables, and so on), two-dimensional and three-dimensional graphics, graphical user interfaces, network and file input/output, encryption and security, and many other services. Understanding what is in an API and how to use it can be a big part of a working programmer's expertise. The printed specification of a language often takes up much less space on a shelf than the documentation of its API.

We mention APIs here in order to dismiss them. They are very important, but they are not the subject of this book. The tutorial chapters of this book present basic ML, Java, and Prolog programming—enough to enable you to solve simple exercises. But to develop full-scale applications, you would need a knowledge of the APIs as well. That, this book does not cover.



## 1.7 Conclusion

This chapter explained some of the reasons the author loves programming languages. Many people skip the introductory chapter of a book such as this one, expecting that it will be vague and boring—full of generalizations about a subject of which they do not yet know the particulars. This one was no exception, but at least it had the virtue of being short.

This chapter also introduced the organization of the book: a mixture of tutorial and theoretical chapters. All the chapters of this book, except the first one and the last one, have exercises at the end. Many have a section that suggests further reading. This chapter does not. The next chapter is the first theoretical chapter. It discusses the definition of programming-language syntax. (The tutorial chapters begin with ML in Chapter 5.)