# Programming Language Pragmatics

## FOURTH EDITION

Michael L. Scott

*Department of Computer Science*
*University of Rochester*

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MK MORGAN KAUFMANN

# Contents

# Introduction

**The first electronic computers were monstrous contraptions,** filling several rooms, consuming as much electricity as a good-size factory, and costing millions of 1940s dollars (but with much less computing power than even the simplest modern cell phone). The programmers who used these machines believed that the computer's time was more valuable than theirs. They programmed in machine language. Machine language is the sequence of bits that directly controls a processor, causing it to add, compare, move data from one place to another, and so forth at appropriate times. Specifying programs at this level of detail is an enormously tedious task. The following program calculates the greatest common divisor (GCD) of two integers, using Euclid's algorithm. It is written in machine language, expressed here as hexadecimal (base 16) numbers, for the x86 instruction set.

EXAMPLE 1.1
GCD program in x86 machine language

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

As people began to write larger programs, it quickly became apparent that a less error-prone notation was required. Assembly languages were invented to allow operations to be expressed with mnemonic abbreviations. Our GCD program looks like this in x86 assembly language:

EXAMPLE 1.2
GCD program in x86 assembler

```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    andl    $-16, %esp
    call    getint
    movl    %eax, %ebx
    call    getint
    cmpl    %eax, %ebx
    je      C
A:  cmpl    %eax, %ebx
    jle     D
    subl    %eax, %ebx
B:  cmpl    %eax, %ebx
    jne     A
C:  movl    %ebx, (%esp)
    call    putint
    movl    -4(%ebp), %ebx
    leave
    ret
D:  subl    %ebx, %eax
    jmp     B
```

Assembly languages were originally designed with a one-to-one correspondence between mnemonics and machine language instructions, as shown in this example.[1] Translating from mnemonics to machine language became the job of a systems program known as an *assembler*. Assemblers were eventually augmented with elaborate "macro expansion" facilities to permit programmers to define parameterized abbreviations for common sequences of instructions. The correspondence between assembly language and machine language remained obvious and explicit, however. Programming continued to be a machine-centered enterprise: each different kind of computer had to be programmed in its own assembly language, and programmers thought in terms of the instructions that the machine would actually execute.

As computers evolved, and as competing designs developed, it became increasingly frustrating to have to rewrite programs for every new machine. It also became increasingly difficult for human beings to keep track of the wealth of detail in large assembly language programs. People began to wish for a machine-independent language, particularly one in which numerical computations (the most common type of program in those days) could be expressed in something more closely resembling mathematical formulae. These wishes led in the mid-1950s to the development of the original dialect of Fortran, the first arguably high-level programming language. Other high-level languages soon followed, notably Lisp and Algol.

Translating from a high-level language to assembly or machine language is the job of a systems program known as a *compiler*.[2] Compilers are substantially more complicated than assemblers because the one-to-one correspondence between source and target operations no longer exists when the source is a high-level language. Fortran was slow to catch on at first, because human programmers, with some effort, could almost always write assembly language programs that would run faster than what a compiler could produce. Over time, however, the performance gap has narrowed, and eventually reversed. Increases in hardware complexity (due to pipelining, multiple functional units, etc.) and continuing improvements in compiler technology have led to a situation in which a state-of-the-art compiler will usually generate better code than a human being will. Even in cases in which human beings can do better, increases in computer speed and program size have made it increasingly important to economize on programmer effort, not only in the original construction of programs, but in subsequent

---

1 The 22 lines of assembly code in the example are encoded in varying numbers of bytes in machine language. The three `cmp` (compare) instructions, for example, all happen to have the same register operands, and are encoded in the two-byte sequence (`39 c3`). The four `mov` (move) instructions have different operands and lengths, and begin with `89` or `8b`. The chosen syntax is that of the GNU `gcc` compiler suite, in which results overwrite the last operand, not the first.

2 High-level languages may also be *interpreted* directly, without the translation step. We will return to this option in Section 1.4. It is the principal way in which scripting languages like Python and JavaScript are implemented.

program *maintenance*—enhancement and correction. Labor costs now heavily outweigh the cost of computing hardware.

## 1.1 The Art of Language Design

Today there are thousands of high-level programming languages, and new ones continue to emerge. Why are there so many? There are several possible answers:

*Evolution.* Computer science is a young discipline; we're constantly finding better ways to do things. The late 1960s and early 1970s saw a revolution in "structured programming," in which the `goto`-based control flow of languages like Fortran, Cobol, and Basic[3] gave way to `while` loops, `case` (`switch`) statements, and similar higher-level constructs. In the late 1980s the nested block structure of languages like Algol, Pascal, and Ada began to give way to the object-oriented structure of languages like Smalltalk, C++, Eiffel, and—a decade later—Java and C#. More recently, scripting languages like Python and Ruby have begun to displace more traditional compiled languages, at least for rapid development.

*Special Purposes.* Some languages were designed for a specific problem domain. The various Lisp dialects are good for manipulating symbolic data and complex data structures. Icon and Awk are good for manipulating character strings. C is good for low-level systems programming. Prolog is good for reasoning about logical relationships among data. Each of these languages can be used successfully for a wider range of tasks, but the emphasis is clearly on the specialty.

*Personal Preference.* Different people like different things. Much of the parochialism of programming is simply a matter of taste. Some people love the terseness of C; some hate it. Some people find it natural to think recursively; others prefer iteration. Some people like to work with pointers; others prefer the implicit dereferencing of Lisp, Java, and ML. The strength and variety of personal preference make it unlikely that anyone will ever develop a universally acceptable programming language.

Of course, some languages are more successful than others. Of the many that have been designed, only a few dozen are widely used. What makes a language successful? Again there are several answers:

*Expressive Power.* One commonly hears arguments that one language is more "powerful" than another, though in a formal mathematical sense they are all

---

3 The names of these languages are sometimes written entirely in uppercase letters and sometimes in mixed case. For consistency's sake, I adopt the convention in this book of using mixed case for languages whose names are pronounced as words (e.g., Fortran, Cobol, Basic), and uppercase for those pronounced as a series of letters (e.g., APL, PL/I, ML).

*Turing complete*—each can be used, if awkwardly, to implement arbitrary algorithms. Still, language features clearly have a huge impact on the programmer's ability to write clear, concise, and maintainable code, especially for very large systems. There is no comparison, for example, between early versions of Basic on the one hand, and C++ on the other. The factors that contribute to expressive power—abstraction facilities in particular—are a major focus of this book.

*Ease of Use for the Novice.* While it is easy to pick on Basic, one cannot deny its success. Part of that success was due to its very low "learning curve." Pascal was taught for many years in introductory programming language courses because, at least in comparison to other "serious" languages, it was compact and easy to learn. Shortly after the turn of the century, Java came to play a similar role; though substantially more complex than Pascal, it is simpler than, say, C++. In a renewed quest for simplicity, some introductory courses in recent years have turned to scripting languages like Python.

*Ease of Implementation.* In addition to its low learning curve, Basic was successful because it could be implemented easily on tiny machines, with limited resources. Forth had a small but dedicated following for similar reasons. Arguably the single most important factor in the success of Pascal was that its designer, Niklaus Wirth, developed a simple, portable implementation of the language, and shipped it free to universities all over the world (see Example 1.15).[4] The Java and Python designers took similar steps to make their language available for free to almost anyone who wants it.

*Standardization.* Almost every widely used language has an official international standard or (in the case of several scripting languages) a single canonical implementation; and in the latter case the canonical implementation is almost invariably written in a language that has a standard. Standardization—of both the language and a broad set of libraries—is the only truly effective way to ensure the portability of code across platforms. The relatively impoverished standard for Pascal, which was missing several features considered essential by many programmers (separate compilation, strings, static initialization, random-access I/O), was at least partially responsible for the language's drop from favor in the 1980s. Many of these features were implemented in different ways by different vendors.

*Open Source.* Most programming languages today have at least one open-source compiler or interpreter, but some languages—C in particular—are much more closely associated than others with freely distributed, peer-reviewed, community-supported computing. C was originally developed in the early

4 Niklaus Wirth (1934–), Professor Emeritus of Informatics at ETH in Zürich, Switzerland, is responsible for a long line of influential languages, including Euler, Algol W, Pascal, Modula, Modula-2, and Oberon. Among other things, his languages introduced the notions of enumeration, subrange, and set types, and unified the concepts of records (structs) and variants (unions). He received the annual ACM Turing Award, computing's highest honor, in 1984.

1970s by Dennis Ritchie and Ken Thompson at Bell Labs,[5] in conjunction with the design of the original Unix operating system. Over the years Unix evolved into the world's most portable operating system—the OS of choice for academic computer science—and C was closely associated with it. With the standardization of C, the language became available on an enormous variety of additional platforms. Linux, the leading open-source operating system, is written in C. As of June 2015, C and its descendants account for well over half of a variety of language-related on-line content, including web page references, book sales, employment listings, and open-source repository updates.

*Excellent Compilers.* Fortran owes much of its success to extremely good compilers. In part this is a matter of historical accident. Fortran has been around longer than anything else, and companies have invested huge amounts of time and money in making compilers that generate very fast code. It is also a matter of language design, however: Fortran dialects prior to Fortran 90 lacked recursion and pointers, features that greatly complicate the task of generating fast code (at least for programs that can be written in a reasonable fashion without them!). In a similar vein, some languages (e.g., Common Lisp) have been successful in part because they have compilers and supporting tools that do an unusually good job of helping the programmer manage very large projects.

*Economics, Patronage, and Inertia.* Finally, there are factors other than technical merit that greatly influence success. The backing of a powerful sponsor is one. PL/I, at least to first approximation, owed its life to IBM. Cobol and Ada owe their life to the U. S. Department of Defense. C# owes its life to Microsoft. In recent years, Objective-C has enjoyed an enormous surge in popularity as the official language for iPhone and iPad apps. At the other end of the life cycle, some languages remain widely used long after "better" alternatives are available, because of a huge base of installed software and programmer expertise, which would cost too much to replace. Much of the world's financial infrastructure, for example, still functions primarily in Cobol.

Clearly no single factor determines whether a language is "good." As we study programming languages, we shall need to consider issues from several points of view. In particular, we shall need to consider the viewpoints of both the programmer and the language implementor. Sometimes these points of view will be in harmony, as in the desire for execution speed. Often, however, there will be conflicts and tradeoffs, as the conceptual appeal of a feature is balanced against the cost of its implementation. The tradeoff becomes particularly thorny when the implementation imposes costs not only on programs that use the feature, but also on programs that do not.

---

**5** Ken Thompson (1943–) led the team that developed Unix. He also designed the B programming language, a child of BCPL and the parent of C. Dennis Ritchie (1941–2011) was the principal force behind the development of C itself. Thompson and Ritchie together formed the core of an incredibly productive and influential group. They shared the ACM Turing Award in 1983.