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 the computing power of a modern hand-held calculator). 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 MIPS R4000 processor.

EXAMPLE 1.1

GCD program in MIPS machine language

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

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 MIPS assembly language:

EXAMPLE 1.2

GCD program in MIPS assembler

```
     addiu   sp,sp,-32
     sw      ra,20(sp)
     jal     getint
     nop
     jal     getint
     sw      v0,28(sp)
     lw      a0,28(sp)
     move    v1,v0
     beq     a0,v0,D
     slt     at,v1,a0
A:   beq     at,zero,B
     nop
          b       C
          subu    a0,a0,v1
B:        subu    v1,v1,a0
C:        bne     a0,v1,A
          slt     at,v1,a0
D:        jal     putint
          nop
          lw      ra,20(sp)
          addiu   sp,sp,32
          jr      ra
          move    v0,zero
```

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

---

1 Each of the 23 lines of assembly code in the example is encoded in the corresponding 32 bits of the machine language. Note for example that the two `sw` (store word) instructions begin with the same 11 bits (`afa` or `afb`). Those bits encode the operation (`sw`) and the base register (`sp`).

1.1 The Art of Language Design

Today there are thousands of high-level programming languages, and new ones continue to emerge. Human beings use assembly language only for special purpose applications. In a typical undergraduate class, it is not uncommon to find users of scores of different languages. 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 `go to`-based control flow of languages like Fortran, Cobol, and Basic[2] gave way to `while` loops, `case` 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 Smalltalk, C++, Eiffel, and the like.

Special Purposes. Many languages were designed for a specific problem domain. The various Lisp dialects are good for manipulating symbolic data and complex data structures. Snobol and Icon 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, Clu, 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 Turing equivalent—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

2 The name of each of these languages is 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).

large systems. There is no comparison, for example, between early versions of Basic on the one hand and Common Lisp or Ada 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 is due to its very low "learning curve." Logo is popular among elementary-level educators for a similar reason: even a 5-year-old can learn it. Pascal was taught for many years in introductory programming language courses because, at least in comparison to other "serious" languages, it is compact and easy to learn. In recent years Java has come to play a similar role. Though substantially more complex than Pascal, it is much simpler than, say, C++.

Ease of Implementation. In addition to its low learning curve, Basic is successful because it could be implemented easily on tiny machines, with limited resources. Forth has 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 1.12).[3] The Java designers have taken similar steps to make their language available for free to almost anyone who wants it.

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 1970s by Dennis Ritchie and Ken Thompson at Bell Labs,[4] 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 has become available on an enormous variety of additional platforms. Linux, the leading open source operating system, is written in C. As of March 2005, C and its descendants account for 60% of the projects hosted at sourceforge.net.

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

---

3 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.

4 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–) 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.

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 lack 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) are 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. Cobol and PL/I, at least to first approximation, owe their life to IBM. Ada owes its life to the United States Department of Defense: it contains a wealth of excellent features and ideas, but the sheer complexity of implementation would likely have killed it if not for the DoD backing. Similarly, C#, despite its technical merits, would probably not have received the attention it has without the backing of Microsoft. 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.

DESIGN & IMPLEMENTATION

Introduction

Throughout the book, sidebars like this one will highlight the interplay of language design and language implementation. Among other things, we will consider the following.

- Cases (such as those mentioned in this section) in which ease or difficulty of implementation significantly affected the success of a language
- Language features that many designers now believe were mistakes, at least in part because of implementation difficulties
- Potentially useful features omitted from some languages because of concern that they might be too difficult or slow to implement
- Language limitations adopted at least in part out of concern for implementation complexity or cost
- Language features introduced at least in part to facilitate efficient or elegant implementations
- Cases in which a machine architecture makes reasonable features unreasonably expensive
- Various other tradeoffs in which implementation plays a significant role

A complete list of sidebars appears in Appendix B.

Clearly no one 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.

In the early days of computing the implementor's viewpoint was predominant. Programming languages evolved as a means of telling a computer what to do. For programmers, however, a language is more aptly defined as a means of expressing algorithms. Just as natural languages constrain exposition and discourse, so programming languages constrain what can and cannot be expressed, and have both profound and subtle influence over what the programmer can *think*. Donald Knuth has suggested that programming be regarded as the art of telling another human being what one wants the computer to do [Knu84].[5] This definition perhaps strikes the best sort of compromise. It acknowledges that both conceptual clarity and implementation efficiency are fundamental concerns. This book attempts to capture this spirit of compromise by simultaneously considering the conceptual and implementation aspects of each of the topics it covers.

1.2 The Programming Language Spectrum

EXAMPLE 1.3
Classification of programming languages

The many existing languages can be classified into families based on their model of computation. Figure 1.1 shows a common set of families. The top-level division distinguishes between the *declarative* languages, in which the focus is on *what* the computer is to do, and the *imperative* languages, in which the focus is on *how* the computer should do it. ■

Declarative languages are in some sense "higher level"; they are more in tune with the programmer's point of view, and less with the implementor's point of view. Imperative languages predominate, however, mainly for performance reasons. There is a tension in the design of declarative languages between the desire to get away from "irrelevant" implementation details and the need to remain close enough to the details to at least control the outline of an algorithm. The design of efficient algorithms, after all, is what much of computer science is about.

5 Donald E. Knuth (1938–), Professor Emeritus at Stanford University and one of the foremost figures in the design and analysis of algorithms, is also widely known as the inventor of the TEX typesetting system (with which this book was produced) and of the *literate programming* methodology with which TEX was constructed. His multivolume *The Art of Computer Programming* has an honored place on the shelf of most professional computer scientists. He received the ACM Turing Award in 1974.

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets
template-based	XSLT
imperative	
von Neumann	C, Ada, Fortran, …
scripting	Perl, Python, PHP, …
object-oriented	Smalltalk, Eiffel, C++, Java, …

Figure 1.1 Classification of programming languages. Note that the categories are fuzzy and open to debate. In particular, it is possible for a functional language to be object-oriented, and many authors do not consider functional programming to be declarative.

It is not yet clear to what extent, and in what problem domains, we can expect compilers to discover good algorithms for problems stated at a very high level. In any domain in which the compiler cannot find a good algorithm, the programmer needs to be able to specify one explicitly.

Within the declarative and imperative families, there are several important subclasses.

- *Functional* languages employ a computational model based on the recursive definition of functions. They take their inspiration from the *lambda calculus*, a formal computational model developed by Alonzo Church in the 1930s. In essence, a program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement. Languages in this category include Lisp, ML, and Haskell.
- *Dataflow* languages model computation as the flow of information (*tokens*) among primitive functional *nodes*. They provide an inherently parallel model: nodes are triggered by the arrival of input tokens, and can operate concurrently. Id and Val are examples of dataflow languages. Sisal, a descendant of Val, is more often described as a functional language.
- *Logic* or *constraint-based* languages take their inspiration from predicate logic. They model computation as an attempt to find values that satisfy certain specified relationships, using a goal-directed a search through a list of logical rules. Prolog is the best-known logic language. The term can also be applied to the programmable aspects of spreadsheet systems such as Excel, VisiCalc, or Lotus 1-2-3.
- The *von Neumann* languages are the most familiar and successful. They include Fortran, Ada 83, C, and all of the others in which the basic means of computation is the modification of variables.[6] Whereas functional languages

6 John von Neumann (1903–1957) was a mathematician and computer pioneer who helped to develop the concept of *stored program* computing, which underlies most computer hardware. In a stored program computer, both programs and data are represented as bits in memory, which the processor repeatedly fetches, interprets, and updates.

are based on expressions that have values, von Neumann languages are based on statements (assignments in particular) that influence subsequent computation via the *side effect* of changing the value of memory.

- *Scripting* languages are a subset of the von Neumann languages. They are distinguished by their emphasis on "gluing together" components that were originally developed as independent programs. Several scripting languages were originally developed for specific purposes: `csh` and `bash`, for example, are the input languages of job control (shell) programs; Awk was intended for text manipulation; PHP and JavaScript are primarily intended for the generation of web pages with dynamic content (with execution on the server and the client, respectively). Other languages, including Perl, Python, Ruby, and Tcl, are more deliberately general purpose. Most place an emphasis on rapid prototyping, with a bias toward ease of expression over speed of execution.
- *Object-oriented* languages are comparatively recent, though their roots can be traced to Simula 67. Most are closely related to the von Neumann languages but have a much more structured and distributed model of both memory and computation. Rather than picture computation as the operation of a monolithic processor on a monolithic memory, object-oriented languages picture it as interactions among semi-independent *objects*, each of which has both its own internal state and subroutines to manage that state. Smalltalk is the purest of the object-oriented languages; C++ and Java are the most widely used. It is also possible to devise object-oriented functional languages (the best known of these is the CLOS [Kee89] extension to Common Lisp), but they tend to have a strong imperative flavor.

One might suspect that concurrent languages also form a separate class (and indeed this book devotes a chapter to the subject), but the distinction between concurrent and sequential execution is mostly orthogonal to the classifications above. Most concurrent programs are currently written using special library packages or compilers in conjunction with a sequential language such as Fortran or C. A few widely used languages, including Java, C#, Ada, and Modula-3, have explicitly concurrent features. Researchers are investigating concurrency in each of the language classes mentioned here.

It should be emphasized that the distinctions among language classes are not clear-cut. The division between the von Neumann and object-oriented languages, for example, is often very fuzzy, and most of the functional and logic languages include some imperative features. The preceding descriptions are meant to capture the general flavor of the classes, without providing formal definitions.

Imperative languages—von Neumann and object-oriented—receive the bulk of the attention in this book. Many issues cut across family lines, however, and the interested reader will discover much that is applicable to alternative computational models in most of the chapters of the book. Chapters 10 through 13 contain additional material on functional, logic, concurrent, and scripting languages.

1.3 Why Study Programming Languages?

Programming languages are central to computer science and to the typical computer science curriculum. Like most car owners, students who have become familiar with one or more high-level languages are generally curious to learn about other languages, and to know what is going on "under the hood." Learning about languages is interesting. It's also practical.

For one thing, a good understanding of language design and implementation can help one choose the most appropriate language for any given task. Most languages are better for some things than for others. No one would be likely to use APL for symbolic computing or string processing, but other choices are not nearly so clear-cut. Should one choose C, C++, or Modula-3 for systems programming? Fortran or Ada for scientific computations? Ada or Modula-2 for embedded systems? Visual Basic or Java for a graphical user interface? This book should help equip you to make such decisions.

Similarly, this book should make it easier to learn new languages. Many languages are closely related. Java and C# are easier to learn if you already know C++. Common Lisp is easier to learn if you already know Scheme. More important, there are basic concepts that underlie all programming languages. Most of these concepts are the subject of chapters in this book: types, control (iteration, selection, recursion, nondeterminacy, concurrency), abstraction, and naming. Thinking in terms of these concepts makes it easier to assimilate the syntax (form) and semantics (meaning) of new languages, compared to picking them up in a vacuum. The situation is analogous to what happens in natural languages: a good knowledge of grammatical forms makes it easier to learn a foreign language.

Whatever language you learn, understanding the decisions that went into its design and implementation will help you use it better. This book should help you

Understand obscure features. The typical C++ programmer rarely uses unions, multiple inheritance, variable numbers of arguments, or the `.*` operator. (If you don't know what these are, don't worry!) Just as it simplifies the assimilation of new languages, an understanding of basic concepts makes it easier to understand these features when you look up the details in the manual.

Choose among alternative ways to express things, based on a knowledge of implementation costs. In C++, for example, programmers may need to avoid unnecessary temporary variables, and use copy constructors whenever possible, to minimize the cost of initialization. In Java they may wish to use `Executor` objects rather than explicit thread creation. With certain (poor) compilers, they may need to adopt special programming idioms to get the fastest code: pointers for array traversal in C; `with` statements to factor out common address calculations in Pascal or Modula-3; `x*x` instead of `x**2` in Basic. In any

language, they need to be able to evaluate the tradeoffs among alternative implementations of abstractions—for example between computation and table lookup for functions like bit set cardinality, which can be implemented either way.

Make good use of debuggers, assemblers, linkers, and related tools. In general, the high-level language programmer should not need to bother with implementation details. There are times, however, when an understanding of those details proves extremely useful. The tenacious bug or unusual system-building problem is sometimes a lot easier to handle if one is willing to peek at the bits.

Simulate useful features in languages that lack them. Certain very useful features are missing in older languages but can be emulated by following a deliberate (if unenforced) programming style. In older dialects of Fortran, for example, programmers familiar with modern control constructs can use comments and self-discipline to write well-structured code. Similarly, in languages with poor abstraction facilities, comments and naming conventions can help imitate modular structure, and the extremely useful *iterators* of Clu, Icon, and C# (which we will study in Section 6.5.3) can be imitated with subroutines and static variables. In Fortran 77 and other languages that lack recursion, an iterative program can be derived via mechanical hand transformations, starting with recursive pseudocode. In languages without named constants or enumeration types, variables that are initialized once and never changed thereafter can make code much more readable and easy to maintain.

Make better use of language technology wherever it appears. Most programmers will never design or implement a conventional programming language, but most will need language technology for other programming tasks. The typical personal computer contains files in dozens of structured formats, encompassing web content, word processing, spreadsheets, presentations, raster and vector graphics, music, video, databases, and a wide variety of other application domains. Each of these structured formats has formal syntax and semantics, which tools must understand. Code to parse, analyze, generate, optimize, and otherwise manipulate structured data can thus be found in almost any sophisticated program, and all of this code is based on language technology. Programmers with a strong grasp of this technology will be in a better position to write well-structured, maintainable tools.

In a similar vein, most tools themselves can be customized, via start-up configuration files, command-line arguments, input commands, or built-in *extension languages* (considered in more detail in Chapter 13). My home directory holds more than 250 separate configuration ("preference") files. My personal configuration files for the `emacs` text editor comprise more than 1200 lines of Lisp code. The user of almost any sophisticated program today will need to make good use of configuration or extension languages. The designers of such a program will need either to adopt (and adapt) some existing extension language, or to invent new notation of their own. Programmers with a strong grasp of language theory will be in a better position to design elegant,

well-structured notation that meets the needs of current users and facilitates future development.

Finally, this book should help prepare you for further study in language design or implementation, should you be so inclined. It will also equip you to understand the interactions of languages with operating systems and architectures, should those areas draw your interest.

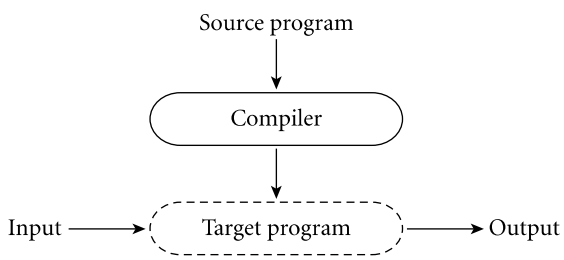
✓ CHECK YOUR UNDERSTANDING

1. What is the difference between machine language and assembly language?
2. In what way(s) are high-level languages an improvement on assembly language? In what circumstances does it still make sense to program in assembler?
3. Why are there so many programming languages?
4. What makes a programming language successful?
5. Name three languages in each of the following categories: von Neumann, functional, object-oriented. Name two logic languages. Name two widely used concurrent languages.
6. What distinguishes declarative languages from imperative languages?
7. What organization spearheaded the development of Ada?
8. What is generally considered the first high-level programming language?
9. What was the first functional language?

1.4 Compilation and Interpretation

EXAMPLE 1.4
Pure compilation

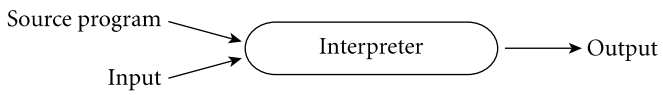
At the highest level of abstraction, the compilation and execution of a program in a high-level language look something like this:



The compiler *translates* the high-level source program into an equivalent target program (typically in machine language) and then goes away. At some arbitrary later time, the user tells the operating system to run the target program. The compiler is the locus of control during compilation; the target program is the locus of control during its own execution. The compiler is itself a machine language program, presumably created by compiling some other high-level program. When written to a file in a format understood by the operating system, machine language is commonly known as *object code*.

EXAMPLE 1.5
Pure interpretation

An alternative style of implementation for high-level languages is known as *interpretation*.



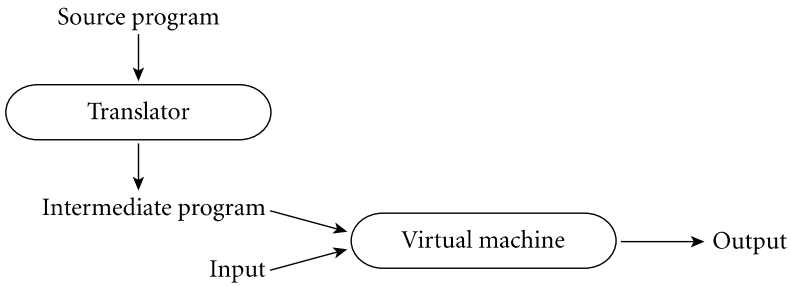
Unlike a compiler, an interpreter stays around for the execution of the application. In fact, the interpreter is the locus of control during that execution. In effect, the interpreter implements a virtual machine whose "machine language" is the high-level programming language. The interpreter reads statements in that language more or less one at a time, executing them as it goes along.

In general, interpretation leads to greater flexibility and better diagnostics (error messages) than does compilation. Because the source code is being executed directly, the interpreter can include an excellent source-level debugger. It can also cope with languages in which fundamental characteristics of the program, such as the sizes and types of variables, or even which names refer to which variables, can depend on the input data. Some language features are almost impossible to implement without interpretation: in Lisp and Prolog, for example, a program can write new pieces of itself and execute them on the fly. (Several scripting languages, including Perl, Tcl, Python, and Ruby, also provide this capability.) Delaying decisions about program implementation until run time is known as *late binding*; we will discuss it at greater length in Section 3.1.

Compilation, by contrast, generally leads to better performance. In general, a decision made at compile time is a decision that does not need to be made at run time. For example, if the compiler can guarantee that variable `x` will always lie at location `49378`, it can generate machine language instructions that access this location whenever the source program refers to `x`. By contrast, an interpreter may need to look `x` up in a table every time it is accessed, in order to find its location. Since the (final version of a) program is compiled only once, but generally executed many times, the savings can be substantial, particularly if the interpreter is doing unnecessary work in every iteration of a loop.

EXAMPLE 1.6
Mixing compilation and interpretation

While the conceptual difference between compilation and interpretation is clear, most language implementations include a mixture of both. They typically look like this:



We generally say that a language is "interpreted" when the initial translator is simple. If the translator is complicated, we say that the language is "compiled." The distinction can be confusing because "simple" and "complicated" are subjective terms, and because it is possible for a compiler (complicated translator) to produce code that is then executed by a complicated virtual machine (interpreter); this is in fact precisely what happens by default in Java. We still say that a language is compiled if the translator analyzes it thoroughly (rather than effecting some "mechanical" transformation) and if the intermediate program does not bear a strong resemblance to the source. These two characteristics—thorough analysis and nontrivial transformation—are the hallmarks of compilation. ■

In practice one sees a broad spectrum of implementation strategies. For example:

EXAMPLE 1.7
Preprocessing

■ Most interpreted languages employ an initial translator (a *preprocessor*) that removes comments and white space, and groups characters together into *tokens*, such as keywords, identifiers, numbers, and symbols. The translator may also expand abbreviations in the style of a macro assembler. Finally, it may identify higher-level syntactic structures, such as loops and subroutines. The goal is to produce an intermediate form that mirrors the structure of the source but can be interpreted more efficiently. ■

DESIGN & IMPLEMENTATION

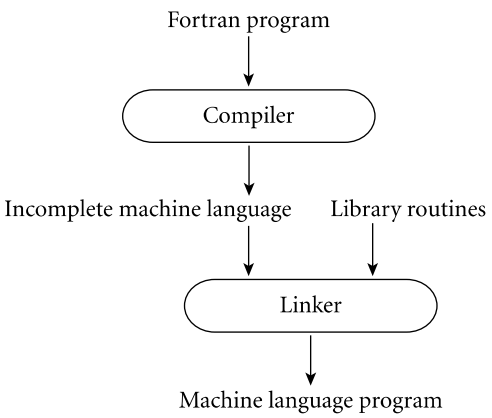
Compiled and interpreted languages

Certain languages (APL and Smalltalk, for example) are sometimes referred to as "interpreted languages" because most of their semantic error checking must be performed at run time. Certain other languages (Fortran and C, for example) are sometimes referred to as "compiled languages" because almost all of their semantic error checking can be performed statically. This terminology isn't strictly correct: interpreters for C and Fortran can be built easily, and a compiler can generate code to perform even the most extensive dynamic semantic checks. That said, language design has a profound effect on "compilability."

In some very early implementations of Basic, the manual actually suggested removing comments from a program in order to improve its performance. These implementations were pure interpreters; they would reread (and then ignore) the comments every time they executed a given part of the program. They had no initial translator.

EXAMPLE 1.8
Library routines and linking

The typical Fortran implementation comes close to pure compilation. The compiler translates Fortran source into machine language. Usually, however, it counts on the existence of a *library* of subroutines that are not part of the original program. Examples include mathematical functions (`sin`, `cos`, `log`, etc.) and I/O. The compiler relies on a separate program, known as a *linker*, to merge the appropriate library routines into the final program:

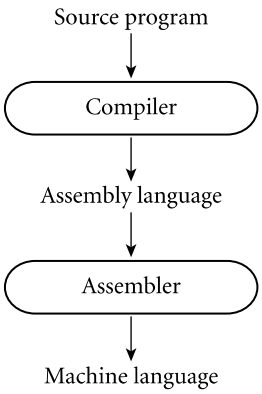


In some sense, one may think of the library routines as extensions to the hardware instruction set. The compiler can then be thought of as generating code for a virtual machine that includes the capabilities of both the hardware and the library.

In a more literal sense, one can find interpretation in the Fortran routines for formatted output. Fortran permits the use of `format` statements that control the alignment of output in columns, the number of significant digits and type of scientific notation for floating-point numbers, inclusion/suppression of leading zeros, and so on. Programs can compute their own formats on the fly. The output library routines include a format interpreter. A similar interpreter can be found in the `printf` routine of C and its descendants. ■

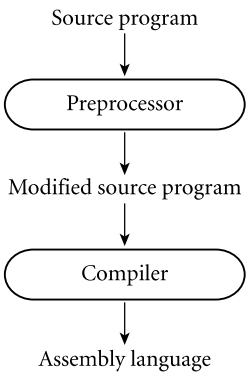
EXAMPLE 1.9
Post-compilation assembly

Many compilers generate assembly language instead of machine language. This convention facilitates debugging, since assembly language is easier for people to read, and isolates the compiler from changes in the format of machine language files that may be mandated by new releases of the operating system (only the assembler must be changed, and it is shared by many compilers).



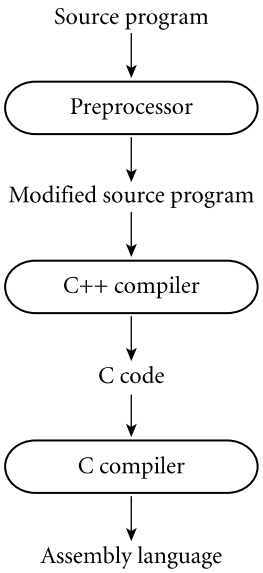
EXAMPLE 1.10
The C preprocessor

Compilers for C (and for many other languages running under Unix) begin with a preprocessor that removes comments and expands macros. The preprocessor can also be instructed to delete portions of the code itself, providing a *conditional compilation* facility that allows several versions of a program to be built from the same source.



EXAMPLE 1.11
Source-to-source translation (C++)

C++ implementations based on the early AT&T compiler actually generated an intermediate program in C, instead of in assembly language. This C++ compiler was indeed a true compiler: it performed a complete analysis of the syntax and semantics of the C++ source program, and with very few exceptions generated all of the error messages that a programmer would see prior to running the program. In fact, programmers were generally unaware that the C compiler was being used behind the scenes. The C++ compiler did not invoke the C compiler unless it had generated C code that would pass through the second round of compilation without producing any error messages.



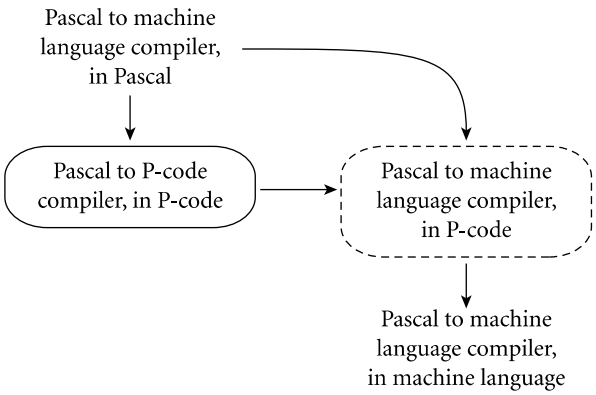
Occasionally one would hear the C++ compiler referred to as a preprocessor, presumably because it generated high-level output that was in turn compiled. I consider this a misuse of the term: compilers attempt to "understand" their source; preprocessors do not. Preprocessors perform transformations based on simple pattern matching, and may well produce output that will generate error messages when run through a subsequent stage of translation. ■

EXAMPLE 1.12
Bootstrapping

■ Many early Pascal compilers were built around a set of tools distributed by Niklaus Wirth. These included the following.

- A Pascal compiler, written in Pascal, that would generate output in *P-code*, a simple stack-based language
- The same compiler, already translated into P-code
- A P-code interpreter, written in Pascal

To get Pascal up and running on a local machine, the user of the tool set needed only to translate the P-code interpreter (by hand) into some locally available language. This translation was not a difficult task; the interpreter was small. By running the P-code version of the compiler on top of the P-code interpreter, one could then compile arbitrary Pascal programs into P-code, which could in turn be run on the interpreter. To get a faster implementation, one could modify the Pascal version of the Pascal compiler to generate a locally available variety of assembly or machine language, instead of generating P-code (a somewhat more difficult task). This compiler could then be "run through itself" in a process known as *bootstrapping*, a term derived from the intentionally ridiculous notion of lifting oneself off the ground by pulling on one's bootstraps.



At this point, the P-code interpreter and the P-code version of the Pascal compiler could simply be thrown away. More often, however, programmers would choose to keep these tools around. The P-code version of a program tends to be significantly smaller than its machine language counterpart. On a circa 1970 machine, the savings in memory and disk requirements could really be important. Moreover, as noted near the beginning of this section, an interpreter will often provide better run-time diagnostics than will the output of a compiler. Finally, an interpreter allows a program to be rerun immediately after modification, without waiting for recompilation—a feature that can be particularly valuable during program development. Some of the best programming environments for imperative languages include both a compiler and an interpreter. ■

DESIGN & IMPLEMENTATION

The early success of Pascal

The P-code based implementation of Pascal is largely responsible for the language's remarkable success in academic circles in the 1970s. No single hardware platform or operating system of that era dominated the computer landscape the way the x86, Linux, and Windows do today.[7] Wirth's toolkit made it possible to get an implementation of Pascal up and running on almost any platform in a week or so. It was one of the first great successes in system portability.

7 Throughout this book we will use the term "x86" to refer to the instruction set architecture of the Intel 8086 and its descendants, including the various Pentium processors. Intel calls this architecture the IA-32, but x86 is a more generic term that encompasses the offerings of competitors such as AMD as well.

EXAMPLE 1.13
Compiling interpreted languages

One will sometimes find compilers for languages (e.g., Lisp, Prolog, Smalltalk, etc.) that permit a lot of late binding and are traditionally interpreted. These compilers must be prepared, in the general case, to generate code that performs much of the work of an interpreter, or that makes calls into a library that does that work instead. In important special cases, however, the compiler can generate code that makes reasonable assumptions about decisions that won't be finalized until run time. If these assumptions prove to be valid the code will run very fast. If the assumptions are not correct, a dynamic check will discover the inconsistency, and revert to the interpreter. ■

EXAMPLE 1.14
Dynamic and just-in-time compilation

In some cases a programming system may deliberately delay compilation until the last possible moment. One example occurs in implementations of Lisp or Prolog that invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set. Another example occurs in implementations of Java. The Java language definition defines a machine-independent intermediate form known as *byte code*. Byte code is the standard format for distribution of Java programs; it allows programs to be transferred easily over the Internet and then run on any platform. The first Java implementations were based on byte-code interpreters, but more recent (faster) implementations employ a *just-in-time* compiler that translates byte code into machine language immediately before each execution of the program. C#, similarly, is intended for just-in-time translation. The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution. CIL is deliberately language independent, so it can be used for code produced by a variety of front-end compilers. ■

EXAMPLE 1.15
Microcode (firmware)

On some machines (particularly those designed before the mid-1980s), the assembly-level instruction set is not actually implemented in hardware but in fact runs on an interpreter. The interpreter is written in low-level instructions called *microcode* (or *firmware*), which is stored in read-only memory and executed by the hardware. Microcode and microprogramming are considered further in Section 5.4.1. ■

As some of these examples make clear, a compiler does not necessarily translate from a high-level language into machine language. It is not uncommon for compilers, especially prototypes, to generate C as output. A little farther afield, text formatters like TEX and `troff` are actually compilers, translating high-level document descriptions into commands for a laser printer or phototypesetter. (Many laser printers themselves incorporate interpreters for the Postscript page-description language.) Query language processors for database systems are also compilers, translating languages like SQL into primitive operations on files. There are even compilers that translate logic-level circuit specifications into photographic masks for computer chips. Though the focus in this book is on imperative programming languages, the term "compilation" applies whenever we translate automatically from one nontrivial language to another, with full analysis of the meaning of the input.

1.5 Programming Environments

Compilers and interpreters do not exist in isolation. Programmers are assisted in their work by a host of other tools. Assemblers, debuggers, preprocessors, and linkers were mentioned earlier. Editors are familiar to every programmer. They may be assisted by cross-referencing facilities that allow the programmer to find the point at which an object is defined, given a point at which it is used. Pretty printers help enforce formatting conventions. Style checkers enforce syntactic or semantic conventions that may be tighter than those enforced by the compiler (see Exploration 1.11). Configuration management tools help keep track of dependences among the (many versions of) separately compiled modules in a large software system. Perusal tools exist not only for text but also for intermediate languages that may be stored in binary. Profilers and other performance analysis tools often work in conjunction with debuggers to help identify the pieces of a program that consume the bulk of its computation time.

In older programming environments, tools may be executed individually, at the explicit request of the user. If a running program terminates abnormally with a "bus error" (invalid address) message, for example, the user may choose to invoke a debugger to examine the "core" file dumped by the operating system. He or she may then attempt to identify the program bug by setting breakpoints, enabling tracing, and so on, and running the program again under the control of the debugger. Once the bug is found, the user will invoke the editor to make an appropriate change. He or she will then recompile the modified program, possibly with the help of a configuration manager.

More recent programming environments provide much more integrated tools. When an invalid address error occurs in an integrated environment, a new window is likely to appear on the user's screen, with the line of source code at which the error occurred highlighted. Breakpoints and tracing can then be set in this window without explicitly invoking a debugger. Changes to the source can be made without explicitly invoking an editor. The editor may also incorporate knowledge of the language syntax, providing templates for all the standard control structures, and checking syntax as it is typed in. If the user asks to rerun the program after making changes, a new version may be built without explicitly invoking the compiler or configuration manager.

DESIGN & IMPLEMENTATION

Powerful development environments

Sophisticated development environments can be a two-edged sword. The quality of the Common Lisp environment has arguably contributed to its widespread acceptance. On the other hand, the particularity of the graphical environment for Smalltalk (with its insistence on specific fonts, window styles, etc.) has made it difficult to port the language to systems accessed through a textual interface, or to graphical systems with a different "look and feel."

Integrated environments have been developed for a variety of languages and systems. They are fundamental to Smalltalk—it is nearly impossible to separate the language from its graphical environment—and are widely used with Common Lisp. They are common on personal computers; examples include the Visual Studio environment from Microsoft and the Project Builder environment from Apple. Several similar commercial and open source environments are available for Unix, and much of the appearance of integration can be achieved within sophisticated editors such as `emacs`.

CHECK YOUR UNDERSTANDING

10. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?
11. Is Java compiled or interpreted (or both)? How do you know?
12. What is the difference between a compiler and a preprocessor?
13. What was the intermediate form employed by the original AT&T C++ compiler?
14. What is P-code?
15. What is bootstrapping?
16. What is a just-in-time compiler?
17. Name two languages in which a program can write new pieces of itself "on-the-fly."
18. Briefly describe three "unconventional" compilers—compilers whose purpose is not to prepare a high-level program for execution on a microprocessor.
19. Describe six kinds of tools that commonly support the work of a compiler within a larger programming environment.

1.6 An Overview of Compilation

EXAMPLE 1.16
Phases of compilation

Compilers are among the most well-studied types of computer programs. In a typical compiler, compilation proceeds through a series of well-defined *phases*, shown in Figure 1.2. Each phase discovers information of use to later phases, or transforms the program into a form that is more useful to the subsequent phase. ■

The first few phases (up through semantic analysis) serve to figure out the meaning of the source program. They are sometimes called the *front end* of the compiler. The last few phases serve to construct an equivalent target program. They are sometimes called the *back end* of the compiler. Many compiler phases

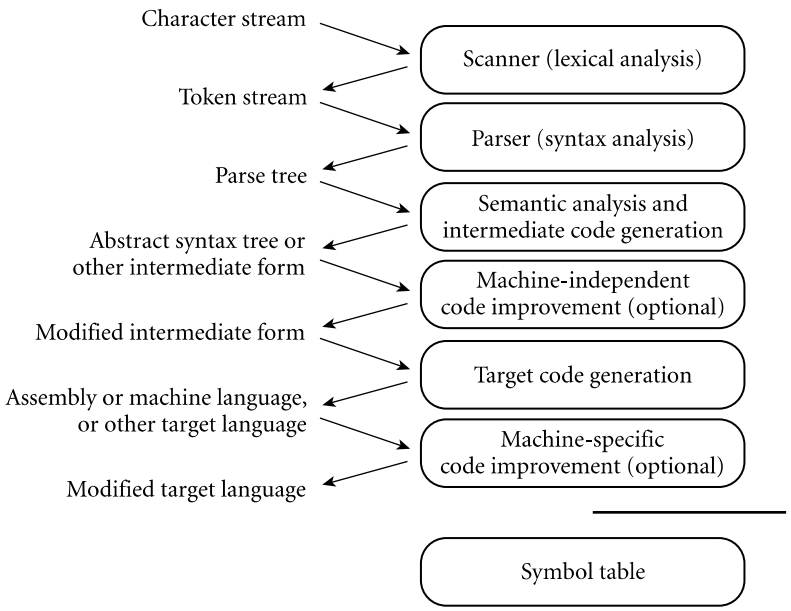


Figure 1.2 Phases of compilation. Phases are listed on the right and the forms in which information is passed between phases are listed on the left. The symbol table serves throughout compilation as a repository for information about identifiers.

can be created automatically from a formal description of the source and/or target languages.

One will sometimes hear compilation described as a series of *passes*. A pass is a phase or set of phases that is serialized with respect to the rest of compilation: it does not start until previous phases have completed, and it finishes before any subsequent phases start. If desired, a pass may be written as a separate program, reading its input from a file and writing its output to a file. Compilers are commonly divided into passes so that the front end may be shared by compilers for more than one machine (target language), and so that the back end may be shared by compilers for more than one source language. Prior to the dramatic increases in memory sizes of the mid- to late 1980s, compilers were also sometimes divided into passes to minimize memory usage: as each pass completed, the next could reuse its code space.

1.6.1 Lexical and Syntax Analysis

EXAMPLE 1.17
GCD program in Pascal

Consider the greatest common divisor (GCD) program introduced at the beginning of this chapter. Written in Pascal, the program might look like this:[8]

8 We use Pascal for this example because its lexical and syntactic structure is significantly simpler than that of most modern imperative languages.

```
program gcd(input, output);
var i, j : integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
end.
```

EXAMPLE 1.18
GCD program tokens

Scanning and parsing serve to recognize the structure of the program, without regard to its meaning. The scanner reads characters ('p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', 'g', 'c', 'd', etc.) and groups them into *tokens*, which are the smallest meaningful units of the program. In our example, the tokens are

```
program  gcd      (        input    ,        output   )        ;
var      i        ,        j        :        integer  ;        begin
read     (        i        ,        j        )        ;        while
i        <>       j        do       if       i        >        j
then     i        :=       i        -        j        else     j
:=       j        -        i        ;        writeln  (        i
)        end      .
```

Scanning is also known as *lexical analysis*. The principal purpose of the scanner is to simplify the task of the parser by reducing the size of the input (there are many more characters than tokens) and by removing extraneous characters. The scanner also typically removes comments, produces a listing if desired, and tags tokens with line and column numbers to make it easier to generate good diagnostics in later phases. One could design a parser to take characters instead of tokens as input—dispensing with the scanner—but the result would be awkward and slow.

EXAMPLE 1.19
Context-free grammar and parsing

Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents. The ways in which these constituents combine are defined by a set of potentially recursive rules known as a *context-free grammar*. For example, we know that a Pascal program consists of the keyword `program`, followed by an identifier (the program name), a parenthesized list of files, a semicolon, a series of definitions, and the main `begin` ... `end` block, terminated by a period:

program ⟶ `PROGRAM id (` id *more_ids* `) ;` *block* `.`

where

block ⟶ *labels constants types variables subroutines* `BEGIN` *stmt*
more_stmts `END`

and

more_ids ⟶ `,` `id` *more_ids*

or

more_ids $\longrightarrow$ ϵ

Here ϵ represents the empty string; it indicates that *more_ids* can simply be deleted. Many more grammar rules are needed, of course, to explain the full structure of a program. ■

A context-free grammar is said to define the *syntax* of the language; parsing is therefore known as *syntactic analysis*. There are many possible grammars for Pascal (an infinite number, in fact); the fragment shown above is based loosely on the "circles-and-arrows" syntax diagrams found in the original Pascal text [JW91]. A full parse tree for our GCD program (based on a full grammar not shown here) appears in Figure 1.3. Much of the complexity of this figure stems from (1) the use of such artificial "constructs" as *more_stmts* and *more_exprs* to represent lists of arbitrary length and (2) the use of the equally artificial *term*, *factor*, and so on, to capture precedence and associativity in arithmetic expressions. Grammars and parse trees will be covered in more detail in Chapter 2. ■

EXAMPLE 1.20
GCD program parse tree

In the process of scanning and parsing, the compiler checks to see that all of the program's tokens are well formed and that the sequence of tokens conforms to the syntax defined by the context-free grammar. Any malformed tokens (e.g., `123abc` or `$@foo` in Pascal) should cause the scanner to produce an error message. Any syntactically invalid token sequence (e.g., `A := B C D` in Pascal) should lead to an error message from the parser.

1.6.2 Semantic Analysis and Intermediate Code Generation

Semantic analysis is the discovery of *meaning* in a program. The semantic analysis phase of compilation recognizes when multiple occurrences of the same identifier are meant to refer to the same program entity, and ensures that the uses are consistent. In most languages the semantic analyzer tracks the *types* of both identifiers and expressions, both to verify consistent usage and to guide the generation of code in later phases.

To assist in its work, the semantic analyzer typically builds and maintains a *symbol table* data structure that maps each identifier to the information known about it. Among other things, this information includes the identifier's type, internal structure (if any), and scope (the portion of the program in which it is valid).

Using the symbol table, the semantic analyzer enforces a large variety of rules that are not captured by the hierarchical structure of the context-free grammar and the parse tree. For example, it checks to make sure that

- Every identifier is declared before it is used.
- No identifier is used in an inappropriate context (calling an integer as a subroutine, adding a string to an integer, referencing a field of the wrong type of record, etc.).
- Subroutine calls provide the correct number and types of arguments.

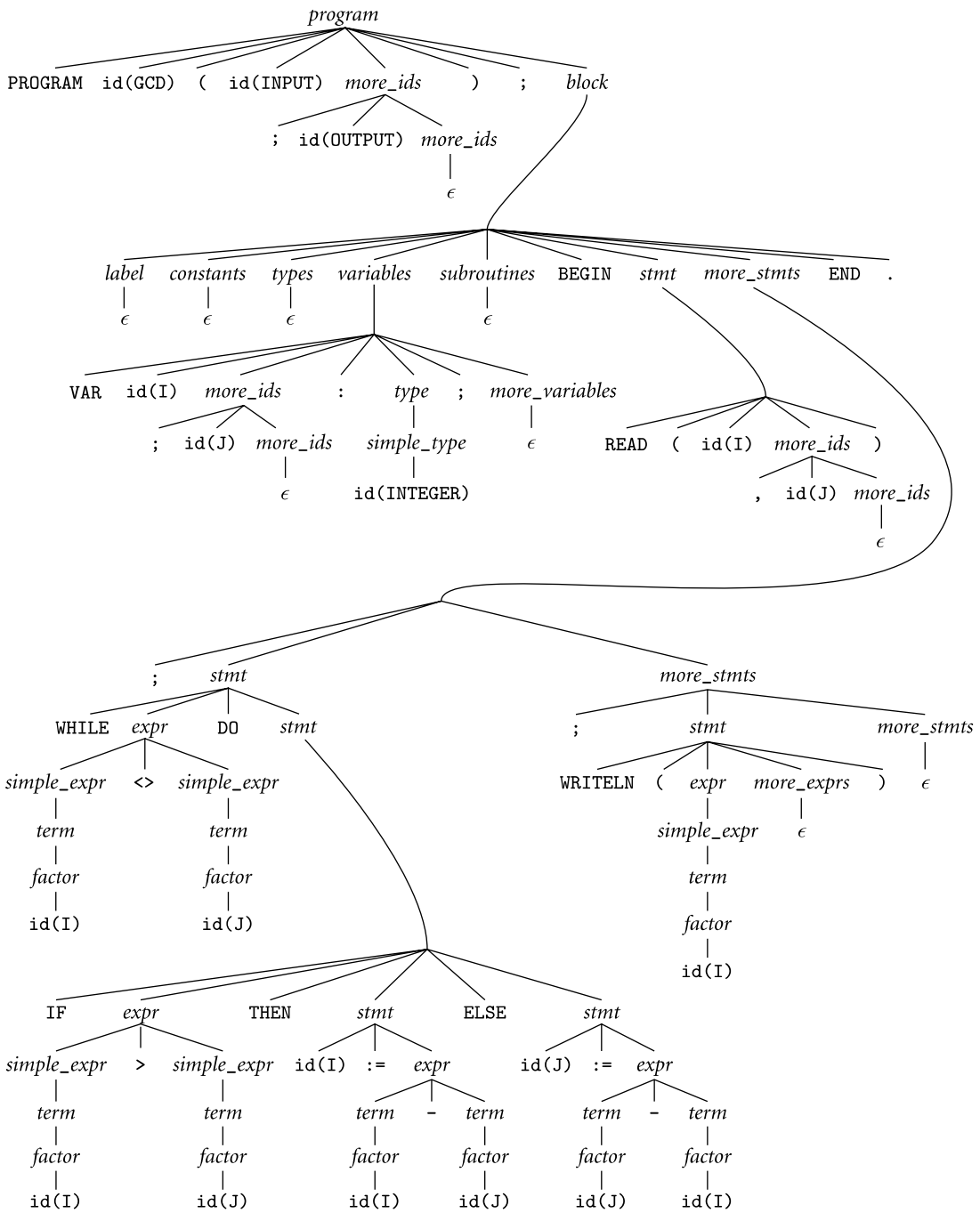


Figure 1.3 **Parse tree for the GCD program.** The symbol ϵ represents the empty string. The remarkable level of complexity in this figure is an artifact of having to fit the (much simpler) source code into the hierarchical structure of a context-free grammar.

- Labels on the arms of a `case` statement are distinct constants.
- Every function contains at least one statement that specifies a return value.

In many compilers, the work of the semantic analyzer takes the form of *semantic action routines*, invoked by the parser when it realizes that it has reached a particular point within a production.

Of course, not all semantic rules can be checked at compile time. Those that can are referred to as the *static semantics* of the language. Those that must be checked at run time are referred to as the *dynamic semantics* of the language. Examples of rules that must often be checked at run time include

- Variables are never used in an expression unless they have been given a value.[9]
- Pointers are never dereferenced unless they refer to a valid object.
- Array subscript expressions lie within the bounds of the array.
- Arithmetic operations do not overflow.

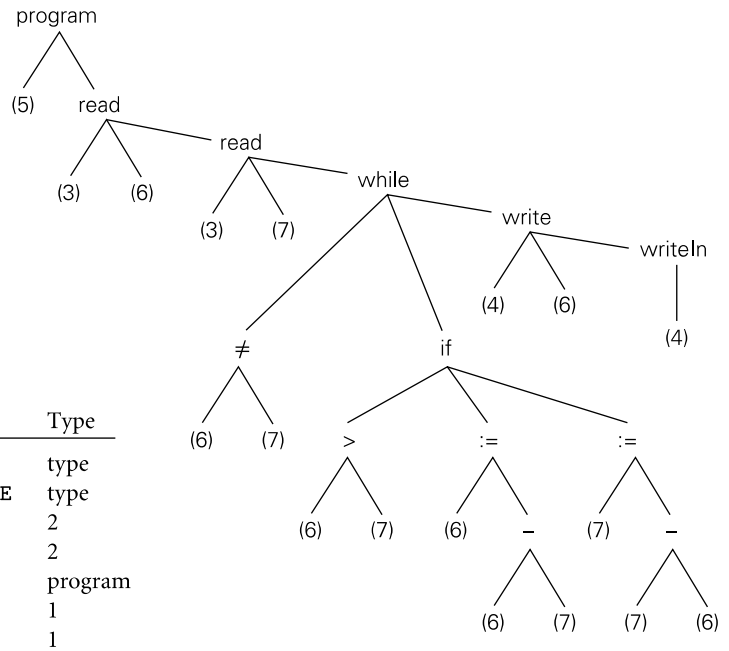
When it cannot enforce rules statically, a compiler will often produce code to perform appropriate checks at run time, aborting the program or generating an *exception* if one of the checks then fails. (Exceptions will be discussed in Section 8.5.) Some rules, unfortunately, may be unacceptably expensive or impossible to enforce, and the language implementation may simply fail to check them. In Ada, a program that breaks such a rule is said to be *erroneous*; in C its behavior is said to be *undefined*.

EXAMPLE 1.21
GCD program abstract syntax tree

A parse tree is sometimes known as a *concrete syntax tree*, because it demonstrates, completely and concretely, how a particular sequence of tokens can be derived under the rules of the context-free grammar. Once we know that a token sequence is valid, however, much of the information in the parse tree is irrelevant to further phases of compilation. In the process of checking static semantic rules, the semantic analyzer typically transforms the parse tree into an *abstract syntax tree* (otherwise known as an *AST*, or simply a *syntax tree*) by removing most of the "artificial" nodes in the tree's interior. The semantic analyzer also *annotates* the remaining nodes with useful information, such as pointers from identifiers to their symbol table entries. The annotations attached to a particular node are known as its *attributes*. A syntax tree for our GCD program is shown in Figure 1.4. ■

In many compilers, the annotated syntax tree constitutes the intermediate form that is passed from the front end to the back end. In other compilers, semantic analysis ends with a traversal of the tree that generates some other intermediate form. Often this alternative form resembles assembly language for an extremely simple idealized machine. In a suite of related compilers, the front ends

9 As we shall see in Section 6.1.3, Java and C# actually do enforce initialization at compile time, but only by adopting a conservative set of rules for "definite assignment," which outlaw programs for which correctness is difficult or impossible to verify at compile time.



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Figure 1.4 Syntax tree and symbol table for the GCD program. Unlike Figure 1.3, the syntax tree retains just the essential structure of the program, omitting detail that was needed only to drive the parsing algorithm.

for several languages and the back ends for several machines would share a common intermediate form.

1.6.3 Target Code Generation

The code generation phase of a compiler translates the intermediate form into the target language. Given the information contained in the syntax tree, generating correct code is usually not a difficult task (generating *good* code is harder, as we shall see in Section 1.6.4). To generate assembly or machine language, the code generator traverses the symbol table to assign locations to variables, and then traverses the syntax tree, generating loads and stores for variable references, interspersed with appropriate arithmetic operations, tests, and branches. Naive code for our GCD example appears in Figure 1.5, in MIPS assembly language. It was generated automatically by a simple pedagogical compiler.

EXAMPLE 1.22
GCD program assembly code

The assembly language mnemonics may appear a bit cryptic, but the comments on each line (not generated by the compiler!) should make the correspondence between Figures 1.4 and 1.5 generally apparent. A few hints: `sp`, `ra`, `at`, `a0`, `v0`, and `t0`–`t9` are registers (special storage locations, limited in number, that can be accessed very quickly). `28(sp)` refers to the memory location 28 bytes beyond

```
    addiu   sp,sp,-32       # reserve room for local variables
    sw      ra,20(sp)       # save return address
    jal     getint          # read
    nop
    sw      v0,28(sp)       # store i
    jal     getint          # read
    nop
    sw      v0,24(sp)       # store j
    lw      t6,28(sp)       # load i
    lw      t7,24(sp)       # load j
    nop
    beq     t6,t7,D         # branch if i = j
    nop
A:  lw      t8,28(sp)       # load i
    lw      t9,24(sp)       # load j
    nop
    slt     at,t9,t8        # determine whether j < i
    beq     at,zero,B       # branch if not
    nop
    lw      t0,28(sp)       # load i
    lw      t1,24(sp)       # load j
    nop
    subu    t2,t0,t1        # t2 := i - j
    sw      t2,28(sp)       # store i
    b       C
    nop
B:  lw      t3,24(sp)       # load j
    lw      t4,28(sp)       # load i
    nop
    subu    t5,t3,t4        # t5 := j - i
    sw      t5,24(sp)       # store j
C:  lw      t6,28(sp)       # load i
    lw      t7,24(sp)       # load j
    nop
    bne     t6,t7,A         # branch if i <> j
    nop
D:  lw      a0,28(sp)       # load i
    jal     putint          # writeln
    nop
    move    v0,zero         # exit status for program
    b       E               # branch to E
    nop
    b       E               # branch to E
    nop
E:  lw      ra,20(sp)       # retrieve return address
    addiu   sp,sp,32        # deallocate space for local variables
    jr      ra              # return to operating system
    nop
```

Figure 1.5 Naive MIPS assembly language for the GCD program.

the location whose address is in register `sp`. `Jal` is a subroutine call ("jump and link"); the first argument is passed in register `a0`, and the return value comes back in register `v0`. `Nop` is a "no-op"; it does no useful work but delays the program for one time cycle, allowing a two-cycle load or branch instruction to complete (branch and load delays were a common feature in early RISC machines; we will consider them in Section 5.5.1). Arithmetic operations generally operate on the second and third arguments, and put their result in the first. ■

Often a code generator will save the symbol table for later use by a symbolic debugger—for example, by including it as comments or some other nonexecutable part of the target code.

1.6.4 Code Improvement

Code improvement is often referred to as *optimization*, though it seldom makes anything optimal in any absolute sense. It is an optional phase of compilation whose goal is to transform a program into a new version that computes the same result more efficiently—more quickly or using less memory, or both.

Some improvements are machine independent. These can be performed as transformations on the intermediate form. Other improvements require an understanding of the target machine (or of whatever will execute the program in the target language). These must be performed as transformations on the target program. Thus code improvement often appears as two additional phases of compilation, one immediately after semantic analysis and intermediate code generation, the other immediately after target code generation.

EXAMPLE 1.23
GCD program optimization

Applying a good code improver to the code in Figure 1.5 produces the code shown in Example 1.2 (page 3). Comparing the two programs, we can see that the improved version is quite a lot shorter. Conspicuously absent are most of the loads and stores. The machine-independent code improver is able to verify that `i` and `j` can be kept in registers throughout the execution of the main loop (this would not have been the case if, for example, the loop contained a call to a subroutine that might reuse those registers, or that might try to modify `i` or `j`). The machine-specific code improver is then able to assign `i` and `j` to actual registers of the target machine. In our example the machine-specific improver is also able to *schedule* (reorder) instructions to eliminate several of the no-ops. Careful examination of the instructions following the loads and branches will reveal that they can be executed safely even when the load or branch has not yet completed. For modern microprocessor architectures, particularly those with so-called *superscalar* RISC instruction sets (ones in which separate functional units can execute multiple instructions simultaneously), compilers can usually generate better code than can human assembly language programmers. ■

CHECK YOUR UNDERSTANDING

20. List the principal phases of compilation, and describe the work performed by each.
21. Describe the form in which a program is passed from the scanner to the parser; from the parser to the semantic analyzer; from the semantic analyzer to the intermediate code generator.
22. What distinguishes the front end of a compiler from the back end?
23. What is the difference between a phase and a pass of compilation? Under what circumstances does it make sense for a compiler to have multiple passes?
24. What is the purpose of the compiler's symbol table?
25. What is the difference between static and dynamic semantics?
26. On modern machines, do assembly language programmers still tend to write better code than a good compiler can? Why or why not?

1.7 Summary and Concluding Remarks

In this chapter we introduced the study of programming language design and implementation. We considered why there are so many languages, what makes them successful or unsuccessful, how they may be categorized for study, and what benefits the reader is likely to gain from that study. We noted that language design and language implementation are intimately related to one another. Obviously an implementation must conform to the rules of the language. At the same time, a language designer must consider how easy or difficult it will be to implement various features, and what sort of performance is likely to result for programs that use those features.

Language implementations are commonly differentiated into those based on interpretation and those based on compilation. We noted, however, that the difference between these approaches is fuzzy, and that most implementations include a bit of each. As a general rule, we say that a language is compiled if execution is preceded by a translation step that (1) fully analyzes both the structure (syntax) and meaning (semantics) of the program and (2) produces an equivalent program in a significantly different form. The bulk of the implementation material in this book pertains to compilation.

Compilers are generally structured as a series of *phases*. The first few phases—scanning, parsing, and semantic analysis—serve to analyze the source program. Collectively these phases are known as the compiler's *front end*. The final few phases—intermediate code generation, code improvement, and target code generation—are known as the *back end*. They serve to build a tar-

get program—preferably a fast one—whose semantics match those of the source.

Chapters 3, 6, 7, 8, and 9 form the core of the rest of this book. They cover fundamental issues of language design, both from the point of view of the programmer and from the point of view of the language implementor. To support the discussion of implementations, Chapters 2 and 4 describe compiler front ends in more detail than has been possible in this introduction. Chapter 5 provides an overview of assembly-level architecture. Chapters 14 and 15 discuss compiler back ends, including assemblers and linkers. Additional language paradigms are covered in Chapters 10 through 13. Appendix A lists the principal programming languages mentioned in the text, together with a genealogical chart and bibliographic references. Appendix B contains a list of "Design and Implementation" sidebars. Appendix C contains a list of numbered examples.

1.8 Exercises

1.1 Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.

(a) A lexical error, detected by the scanner

(b) A syntax error, detected by the parser

(c) A static semantic error, detected by semantic analysis

(d) A dynamic semantic error, detected by code generated by the compiler

(e) An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)

1.2 Algol family languages are typically compiled, while Lisp family languages, in which many issues cannot be settled until run time, are typically interpreted. Is interpretation simply what one "has to do" when compilation is infeasible, or are there actually some *advantages* to interpreting a language, even when a compiler is available?

1.3 The `gcd` program of Example 1.17 might also be written

```
program gcd(input, output);
var i, j : integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i mod j
        else j := j mod i;
    writeln(i)
end.
```

Does this program compute the same result? If not, can you fix it? Under what circumstances would you expect one or the other to be faster?

1.4 In your local implementation of C, what is the limit on the size of integers? What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another? How do the answers to these questions differ for Java? For Ada? For Pascal? For Scheme? (You may need to find a manual.)

1.5 The Unix `make` utility allows the programmer to specify *dependences* among the separately compiled pieces of a program. If file *A* depends on file *B* and file *B* is modified, `make` deduces that *A* must be recompiled, in case any of the changes to *B* would affect the code produced for *A*. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?

1.6 Why is it difficult to tell whether a program is correct? How do you go about finding bugs in your code? What kinds of bugs are revealed by testing? What kinds of bugs are not? (For more formal notions of program correctness, see the bibliographic notes at the end of Chapter 4.)

1.9 Explorations

1.7 (a) What was the first programming language you learned? If you chose it, why did you do so? If it was chosen for you by others, why do you think they chose it? What parts of the language did you find the most difficult to learn?

(b) For the language with which you are most familiar (this may or may not be the first one you learned), list three things you wish had been differently designed. Why do you think they were designed the way they were? How would you fix them if you had the chance to do it over? Would there be any negative consequences—for example, in terms of compiler complexity or program execution speed?

1.8 Get together with a classmate whose principal programming experience is with a language in a different category of Figure 1.1. (If your experience is mostly in C, for example, you might search out someone with experience in Lisp.) Compare notes. What are the easiest and most difficult aspects of programming, in each of your experiences? Pick some simple problem (e.g., sorting, or identification of connected components in a graph) and solve it using each of your favorite languages. Which solution is more elegant (do the two of you agree)? Which is faster? Why?

1.9 (a) If you have access to a Unix system, compile a simple program with the `-S` command-line flag. Add comments to the resulting assembly language file to explain the purpose of each instruction.

(b) Now use the `-o` command-line flag to generate a *relocatable object file*. Using appropriate local tools (look in particular for `nm`, `objdump`, or a symbolic debugger like `gdb` or `dbx`), identify the machine language corresponding to each line of assembler.

(c) Using `nm`, `objdump`, or a similar tool, identify the *undefined external symbols* in your object file. Now run the compiler to completion, to produce an *executable* file. Finally, run `nm` or `objdump` again to see what has happened to the symbols in part (b). Where did they come from, and how did the linker resolve them?

(d) Run the compiler to completion one more time, using the `-v` command-line flag. You should see messages describing the various subprograms invoked during the compilation process (some compilers use a different letter for this option; check the `man` page). The subprograms may include a preprocessor, separate passes of the compiler itself (often two), probably an assembler, and the linker. If possible, run these subprograms yourself, individually. Which of them produce the files described in the previous subquestions? Explain the purpose of the various command-line flags with which the subprograms were invoked.

1.10 Write a program that commits a dynamic semantic error (e.g., division by zero, access off the end of an array, dereference of a `nil` pointer). What happens when you run this program? Does the compiler give you options to control what happens? Devise an experiment to evaluate the cost of run-time semantic checks. If possible, try this exercise with more than one language or compiler.

1.11 C has a reputation for being a relatively "unsafe" high-level language. In particular, it allows the programmer to mix operands of different sizes and types in many more ways than do its "safer" cousins. The Unix `lint` utility can be used to search for potentially unsafe constructs in C programs. In effect, many of the rules that are enforced by the compiler in other languages are optional in C and are enforced (if desired) by a separate program. What do you think of this approach? Is it a good idea? Why or why not?

1.12 Using an Internet search engine or magazine indexing service, read up on the history of Java and C#, including the conflict between Sun and Microsoft over Java standardization. Some have claimed that C# is, at least in part, Microsoft's attempt to kill Java. Defend or refute this claim.

1.10 Bibliographic Notes

The compiler-oriented chapters of this book attempt to convey a sense of what the compiler does, rather than explaining how to build one. A much greater level of detail can be found in other texts. Leading options include the work of Cooper and Torczon [CT04], Grune et al. [GBJL01], and Appel [App97]. The older texts by Aho, Sethi, and Ullman [ASU86] and Fischer and LeBlanc [FL88] were for many years the standards in the field, but have grown somewhat dated. High-quality texts on programming language design include those of Louden [Lou03], Sebesta [Seb04], and Sethi [Set96].

Some of the best information on the history of programming languages can be found in the proceedings of conferences sponsored by the Association for Computing Machinery in 1978 and 1993 [Wex78, Ass93]. Another excellent reference is Horowitz's 1987 text [Hor87]. A broader range of historical material can be found in the quarterly *IEEE Annals of the History of Computing*. Given the importance of personal taste in programming language design, it is inevitable that some language comparisons should be marked by strongly worded opinions. Examples include the writings of Dijkstra [Dij82], Hoare [Hoa81], Kernighan [Ker81], and Wirth [Wir85a].

Most personal computer software development now takes place in integrated programming environments. Influential precursors to these environments include the Genera Common Lisp environment from Symbolics Corp. [WMWM87] and the Smalltalk [Gol84], Interlisp [TM81], and Cedar [SZBH86] environments at the Xerox Palo Alto Research Center.

This page intentionally left blank