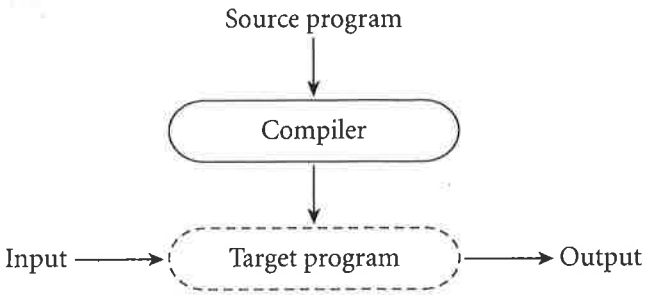
From: Michael L. Scott. Programming Language Pragmatics 4th, Ed, Morgan Kaufmann 2016.

1.4 Compilation and Interpretation

EXAMPLE 1.7
Pure compilation

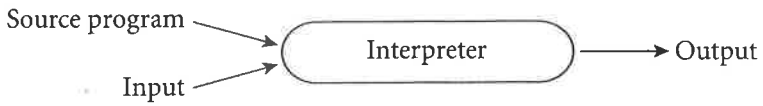
At the highest level of abstraction, the compilation and execution of a program in a high-level language look something like this:



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 1.8
Pure interpretation

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



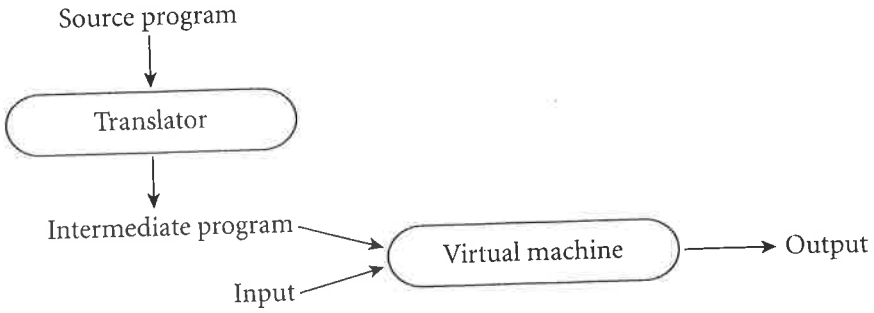
Unlike a compiler, an interpreter stays around for the execution of the application. In fact, the interpreter is the locus of control during that execution. In effect, the interpreter implements a virtual machine whose "machine language" is the high-level programming language. The interpreter reads statements 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 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 1.9
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.

DESIGN & IMPLEMENTATION

1.2 Compiled and interpreted languages

Certain languages (e.g., Smalltalk and Python) are sometimes referred to as "interpreted languages" because most of their semantic error checking must be performed at run time. Certain other languages (e.g., Fortran and C) 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 practice one sees a broad spectrum of implementation strategies:

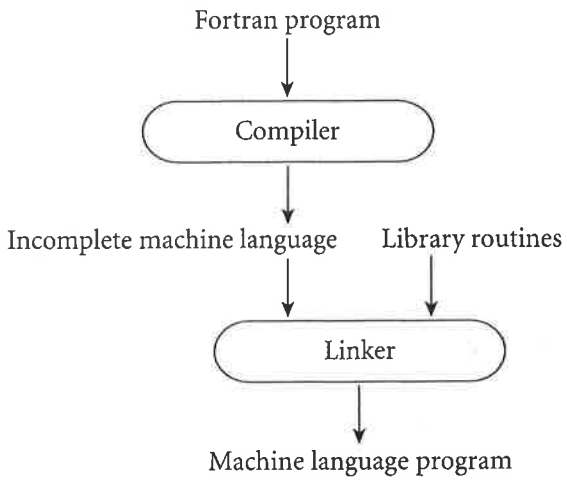
EXAMPLE 1.10
Preprocessing

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.

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 re-read (and then ignore) the comments every time they executed a given part of the program. They had no initial translator.

EXAMPLE 1.11
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:

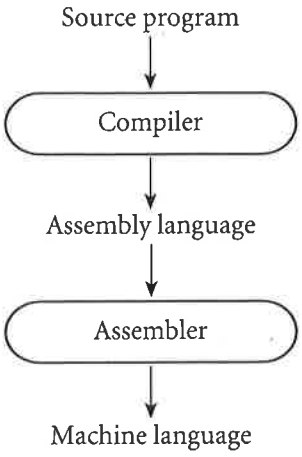


In some sense, one may think of the library routines as extensions to the hardware instruction set. The compiler can then be thought of as generating code for a virtual machine that includes the capabilities of both the hardware and the library.

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.

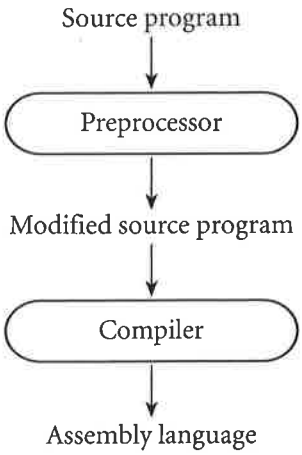
EXAMPLE 1.12
Post-compilation assembly

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 1.13
The C preprocessor

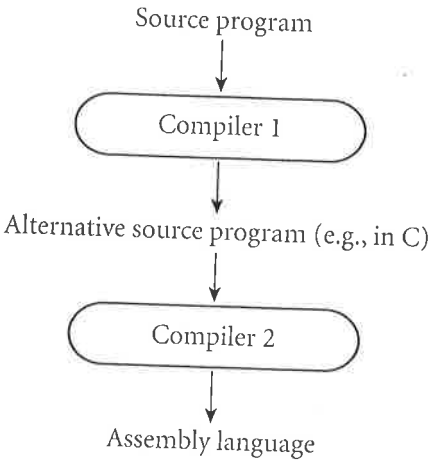
Compilers for C (and for many other languages running under Unix) begin with a preprocessor that removes comments and expands macros. The preprocessor can also be instructed to delete portions of the code itself, providing a *conditional compilation* facility that allows several versions of a program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



EXAMPLE 1.14
Source-to-source translation

A surprising number of compilers generate output in some high-level language—commonly C or some simplified version of the input language. Such *source-to-source* translation is particularly common in research languages and during the early stages of language development. One famous example was AT&T's original compiler for C++. This was indeed a true compiler, though it generated C instead of assembler: it performed a complete analysis of the syntax and semantics of the C++ source program, and with very few excep-

tions 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 1.15
Bootstrapping

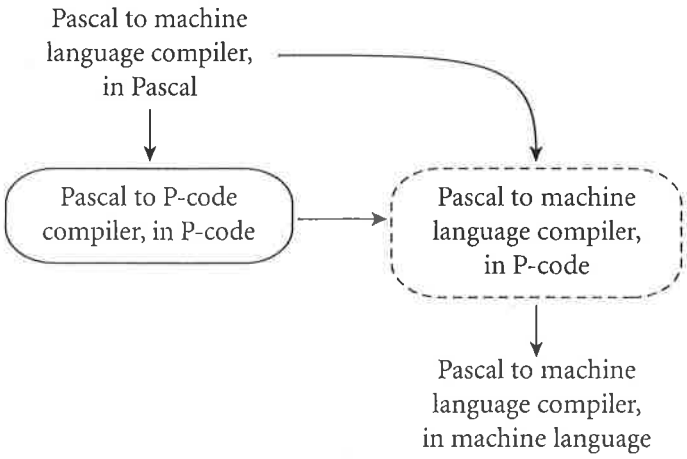
Many compilers are *self-hosting*: they are written in the language they compile—Ada compilers in Ada, C compilers in C. This raises an obvious question: how does one compile the compiler in the first place? The answer is to use a technique known as *bootstrapping*, a term derived from the intentionally ridiculous notion of lifting oneself off the ground by pulling on one's bootstraps. In a nutshell, one starts with a simple implementation—often an interpreter—and uses it to build progressively more sophisticated versions. We can illustrate the idea with an historical example.

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 stack-based language similar to the *bytecode* of modern Java compilers
- 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 bootstrapped—run through itself—to yield a machine-code version of the compiler:



In a more general context, suppose we were building one of the first compilers for a new programming language. Assuming we have a C compiler on our target system, we might start by writing, in a simple subset of C, a compiler for an equally simple subset of our new programming language. Once this compiler was working, we could hand-translate the C code into (the subset of) our new language, and then run the new source through the compiler itself. After that, we could repeatedly extend the compiler to accept a larger subset

DESIGN & IMPLEMENTATION

1.3 The early success of Pascal

The P-code-based implementation of Pascal, and its use of bootstrapping, are 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.[8] 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.

---

8 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, "Core," and Xeon 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.

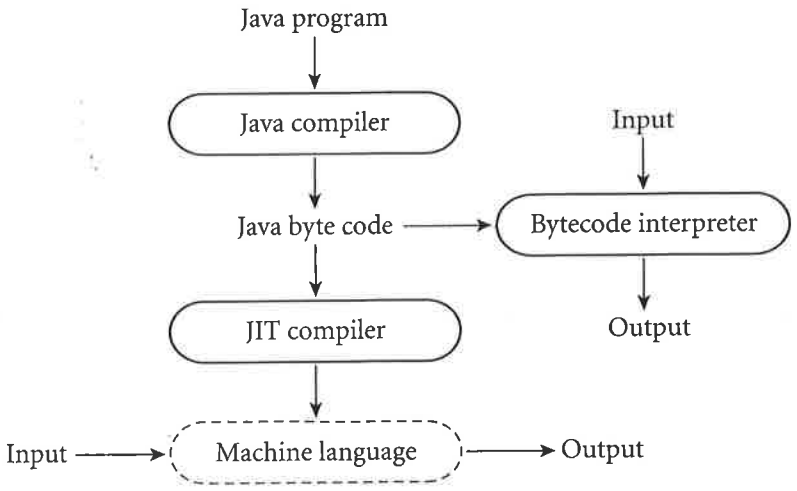
of the new programming language, bootstrap it again, and use the extended language to implement an even larger subset. "Self-hosting" implementations of this sort are actually quite common.

EXAMPLE 1.16
Compiling interpreted languages

One will sometimes find compilers for languages (e.g., Lisp, Prolog, Smalltalk) 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 1.17
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 language implementations (e.g., for 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 Java *bytecode*. Bytecode 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 modern implementations obtain significantly better performance with a *just-in-time* compiler that translates bytecode into machine language immediately before each execution of the program:



C#, similarly, is intended for just-in-time translation. The main C# compiler produces *Common Intermediate Language* (CIL), which is then translated into machine language immediately prior to execution. CIL is deliberately language independent, so it can be used for code produced by a variety of front-end compilers. We will explore the Java and C# implementations in detail in Section 16.1.

EXAMPLE 1.18
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 C-5.4.1.

As some of these examples make clear, a compiler does not necessarily translate from a high-level programming language into machine language. Some compilers, in fact, accept inputs that we might not immediately think of as programs at all. Text formatters like TEX, for example, compile high-level document descriptions into commands for a laser printer or phototypesetter. (Many laser printers themselves contain pre-installed interpreters for the Postscript page-description language.) Query language processors for database systems translate 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 augmented with 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 1.14). 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.

Modern environments provide more integrated tools. When an invalid address error occurs in an integrated development environment (IDE), 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. 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.

The editor for an IDE may incorporate knowledge of language syntax, providing templates for all the standard control structures, and checking syntax as it is typed in. Internally, the IDE is likely to maintain not only a program's source and object code, but also a partially compiled internal representation. When the source is edited, the internal representation will be updated automatically—often incrementally (without reparsing large portions of the source). In some cases, structural changes to the program may be implemented first in the internal representation, and then automatically reflected in the source.

IDEs are fundamental to Smalltalk—it is nearly impossible to separate the language from its graphical environment—and have been routinely used for Common Lisp since the 1980s. With the ubiquity of graphical interfaces, integrated environments have largely displaced command-line tools for many languages and systems. Popular open-source IDEs include Eclipse and NetBeans. Commercial systems include the Visual Studio environment from Microsoft and the XCode environment from Apple. Much of the appearance of integration can also be achieved within sophisticated editors such as `emacs`.

✓ CHECK YOUR UNDERSTANDING

11. Explain the distinction between interpretation and compilation. What are the comparative advantages and disadvantages of the two approaches?
12. Is Java compiled or interpreted (or both)? How do you know?
13. What is the difference between a compiler and a preprocessor?
14. What was the intermediate form employed by the original AT&T C++ compiler?

DESIGN & IMPLEMENTATION

1.4 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.) made it difficult to port the language to systems accessed through a textual interface, or to graphical systems with a different "look and feel."

15. What is P-code?
16. What is bootstrapping?
17. What is a just-in-time compiler?
18. Name two languages in which a program can write new pieces of itself "on the fly."
19. Briefly describe three "unconventional" compilers—compilers whose purpose is not to prepare a high-level program for execution on a general-purpose processor.
20. List six kinds of tools that commonly support the work of a compiler within a larger programming environment.
21. Explain how an integrated development environment (IDE) differs from a collection of command-line tools.

1.6 An Overview of Compilation

Compilers are among the most well-studied computer programs. We will consider them repeatedly throughout the rest of the book, and in chapters 2, 4, 15, and 17 in particular. The remainder of this section provides an introductory overview.

EXAMPLE 1.19
Phases of compilation and interpretation

In a typical compiler, compilation proceeds through a series of well-defined *phases*, shown in Figure 1.3. 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.

An interpreter (Figure 1.4) shares the compiler's front-end structure, but "executes" (interprets) the intermediate form directly, rather than translating it into machine language. The execution typically takes the form of a set of mutually recursive subroutines that traverse ("walk") the syntax tree, "executing" its nodes in program order. Many compiler and interpreter phases 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

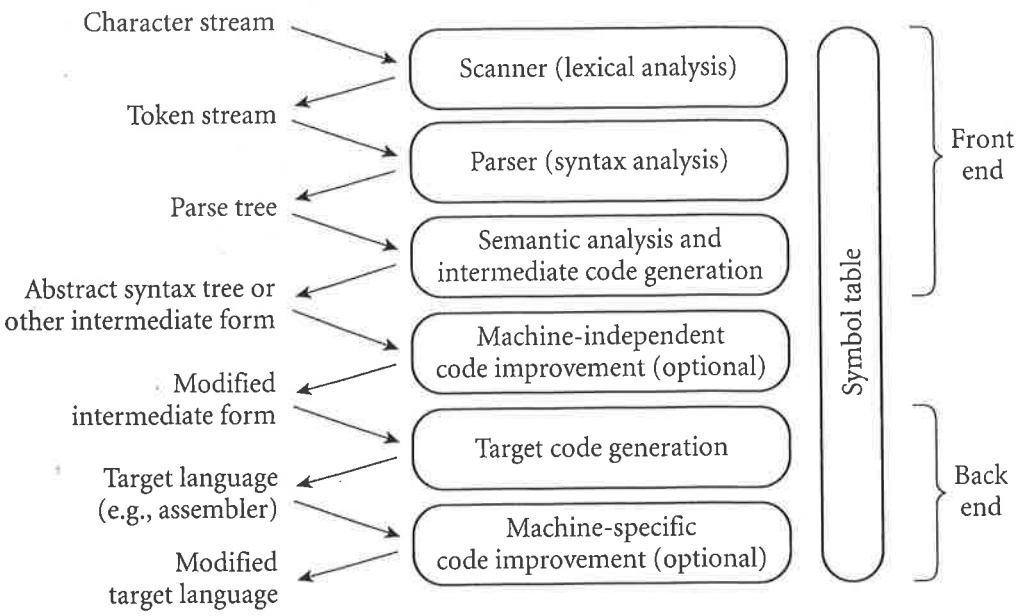


Figure 1.3 Phases of compilation. Phases are listed on the right and the forms in which information is passed between phases are listed on the left. The symbol table serves throughout compilation as a repository for information about identifiers.

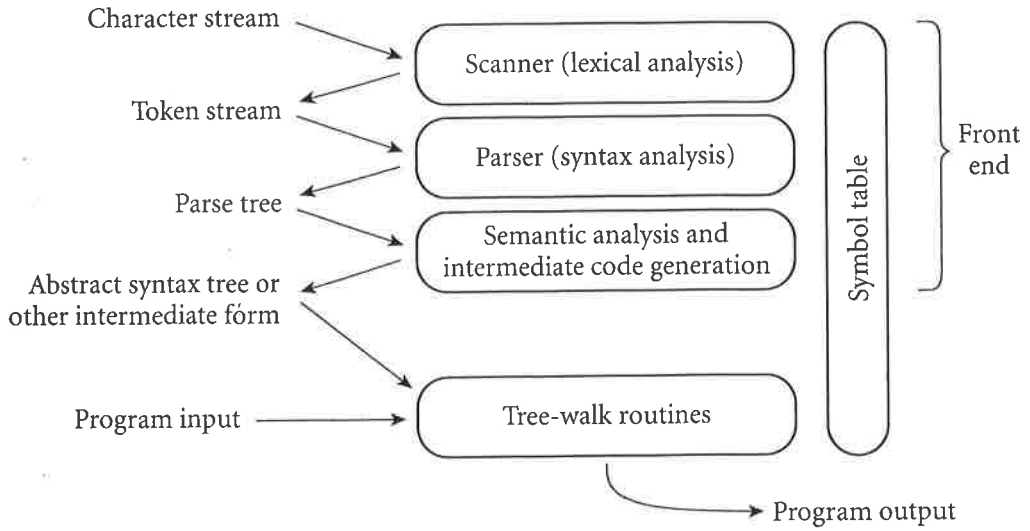


Figure 1.4 Phases of interpretation. The front end is essentially the same as that of a compiler. The final phase "executes" the intermediate form, typically using a set of mutually recursive subroutines that walk the syntax tree.

for more than one machine (target language), and so that the back end may be shared by compilers for more than one source language. In some implementations the front end and the back end may be separated by a "middle end" that is responsible for language- and machine-independent code improvement. 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 1.20
GCD program in C

Consider the greatest common divisor (GCD) problem introduced at the beginning of this chapter, and shown as a function in Figure 1.2. Hypothesizing trivial I/O routines and recasting the function as a stand-alone program, our code might look like this in C:

```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

EXAMPLE 1.21
GCD program tokens

Scanning and parsing serve to recognize the structure of the program, without regard to its meaning. The scanner reads characters ('i', 'n', 't', ' ', 'm', 'a', 'i', 'n', '(', ')', etc.) and groups them into *tokens*, which are the smallest meaningful units of the program. In our example, the tokens are

```
int     main    (       )       {       int     i       =
getint  (       )       ,       j       =       getint  (
)       ;       while   (       i       !=      j       )
{       if      (       i       >       j       )       i
=       i       -       j       ;       else    j       =
j       -       i       ;       }       putint  (       i
)       ;       }
```

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 like white space. The scanner also typically removes comments 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 1.22
Context-free grammar and parsing

Parsing organizes tokens into a *parse tree* that represents higher-level constructs (statements, expressions, subroutines, and so on) in terms of their constituents. Each construct is a node in the tree; its constituents are its children. The root of the tree is simply "*program*"; the leaves, from left to right, are the tokens received from the scanner. Taken as a whole, the tree shows how the tokens fit

together to make a valid program. The structure relies on a set of potentially recursive rules known as a *context-free grammar*. Each rule has an arrow sign (⟶) with the construct name on the left and a possible expansion on the right.[9] In C, for example, a `while` loop consists of the keyword `while` followed by a parenthesized Boolean expression and a statement:

iteration-statement ⟶ `while (` *expression* `)` *statement*

The statement, in turn, is often a list enclosed in braces:

statement ⟶ *compound-statement*
compound-statement ⟶ `{` *block-item-list_opt* `}`

where

block-item-list_opt ⟶ *block-item-list*

or

block-item-list_opt ⟶ ϵ

and

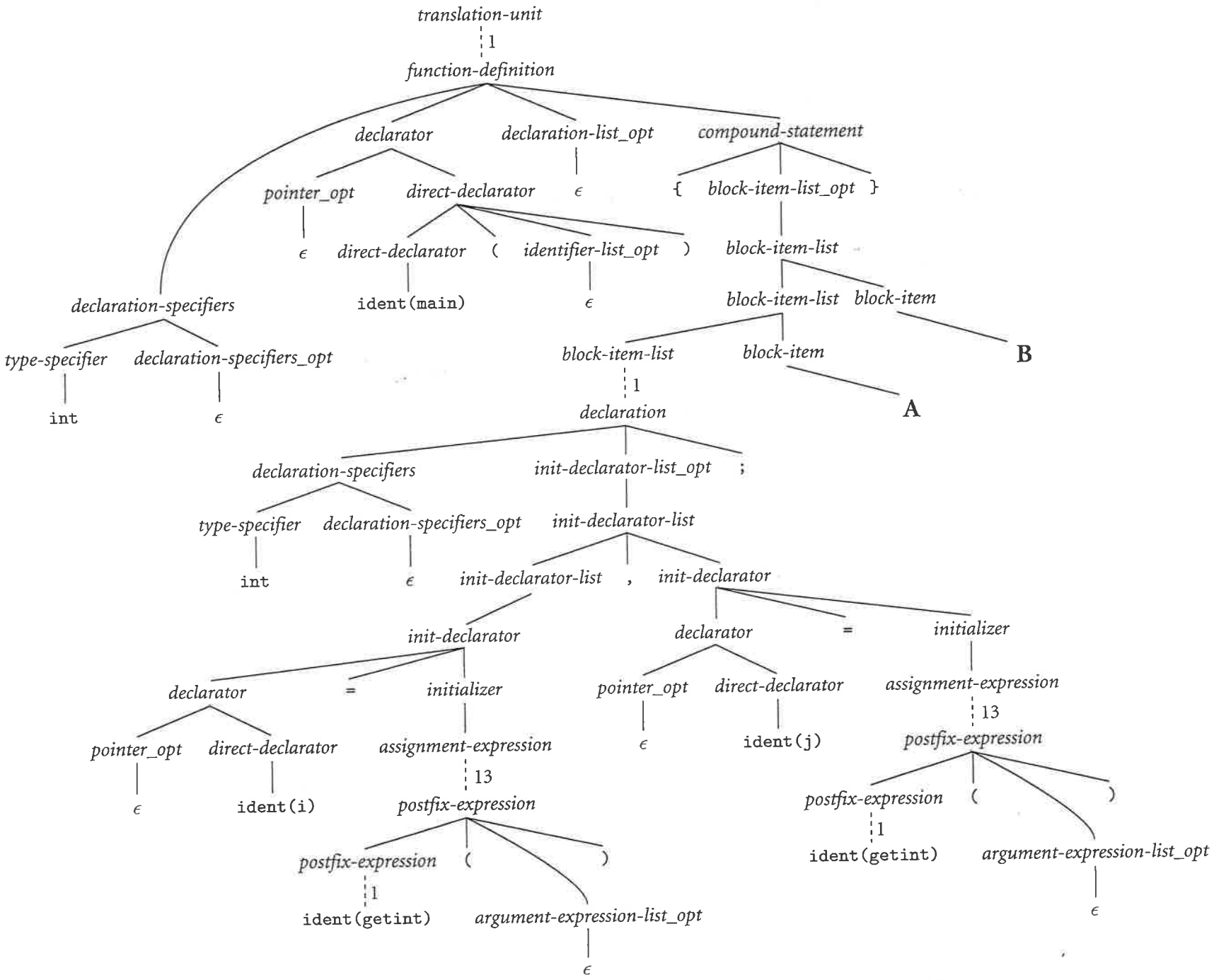
block-item-list ⟶ *block-item*
block-item-list ⟶ *block-item-list block-item*
block-item ⟶ *declaration*
block-item ⟶ *statement*

Here ϵ represents the empty string; it indicates that *block-item-list_opt* 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 *syntax analysis*. There are many possible grammars for C (an infinite number, in fact); the fragment shown above is taken from the sample grammar contained in the official language definition [Int99]. A full parse tree for our GCD program (based on a full grammar not shown here) appears in Figure 1.5. While the size of the tree may seem daunting, its details aren't particularly important at this point in the text. What *is* important is that (1) each individual branching point represents the application of a single grammar rule, and (2) the resulting complexity is more a reflection of the grammar than it is of the input program. Much of the bulk stems from (a) the use of such artificial "constructs" as *block_item-list* and *block_item-list_opt* to generate lists of arbitrary

EXAMPLE 1.23
GCD program parse tree

9 Theorists also study *context-sensitive* grammars, in which the allowable expansions of a construct (the applicable rules) depend on the context in which the construct appears (i.e., on constructs to the left and right). Context sensitivity is important for natural languages like English, but it is almost never used in programming language design.


translation-unit
1
function-definition
declarator
declaration-list_opt
compound-statement
pointer_opt
direct-declarator
ϵ
{ block-item-list_opt }
ϵ
direct-declarator (identifier-list_opt)
block-item-list
declaration-specifiers
ident(main)
ϵ
block-item-list block-item
type-specifier
declaration-specifiers_opt
block-item-list
block-item
B
int
ϵ
1
A
declaration
declaration-specifiers
init-declarator-list_opt ;
type-specifier
declaration-specifiers_opt
init-declarator-list
int
ϵ
init-declarator-list , init-declarator
init-declarator
declarator = initializer
declarator
= initializer
pointer_opt
direct-declarator
assignment-expression
pointer_opt
direct-declarator
assignment-expression
ϵ
ident(i)
13
ϵ
ident(j)
13
postfix-expression
postfix-expression
postfix-expression ( )
1
ident(getint)
argument-expression-list_opt
ident(getint)
argument-expression-list_opt
ϵ
ϵ

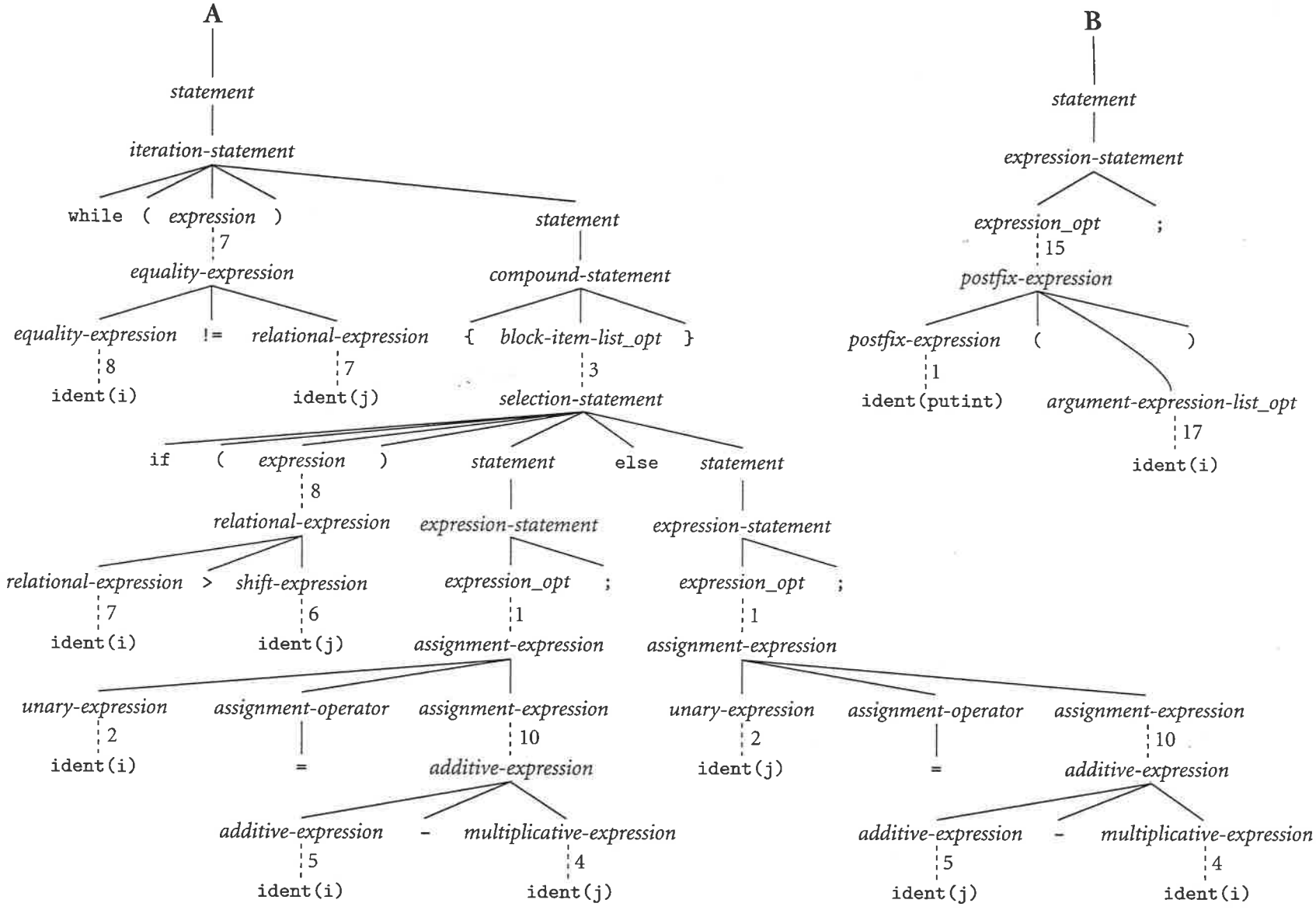


Figure 1.5 Parse tree for the GCD program. The symbol ϵ represents the empty string. Dotted lines indicate a chain of one-for-one replacements, elided to save space; the adjacent number indicates the number of omitted nodes. While the details of the tree aren't important to the current chapter, the sheer *amount* of detail is: it comes from having to fit the (much simpler) source code into the hierarchical structure of a context-free grammar.

length, and (b) the use of the equally artificial *assignment-expression*, *additive-expression*, *multiplicative-expression*, and so on, to capture precedence and associativity in arithmetic expressions. We shall see in the following subsection that much of this complexity can be discarded once parsing is complete.

In the process of scanning and parsing, the compiler or interpreter 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 C) should cause the scanner to produce an error message. Any syntactically invalid token sequence (e.g., `A = X Y Z` in C) 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. Among other things, the semantic analyzer 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 it also tracks the *types* of both identifiers and expressions, both to verify consistent usage and to guide the generation of code in the back end of a compiler.

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. In C, 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 `struct`, etc.).
- Subroutine calls provide the correct number and types of arguments.
- Labels on the arms of a `switch` statement are distinct constants.
- Any function with a non-`void` return type `returns` a value explicitly.

In many front ends, 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 grammar rule.

Of course, not all semantic rules can be checked at compile time (or in the front end of an interpreter). Those that can are referred to as the *static semantics* of the language. Those that must be checked at run time (or in the later phases of an interpreter) are referred to as the *dynamic semantics* of the language. C has very little in the way of dynamic checks (its designers opted for performance over safety). Examples of rules that other languages enforce at run time include:

- Variables are never used in an expression unless they have been given a value.[10]
- 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 9.4.) 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*.

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 1.6.

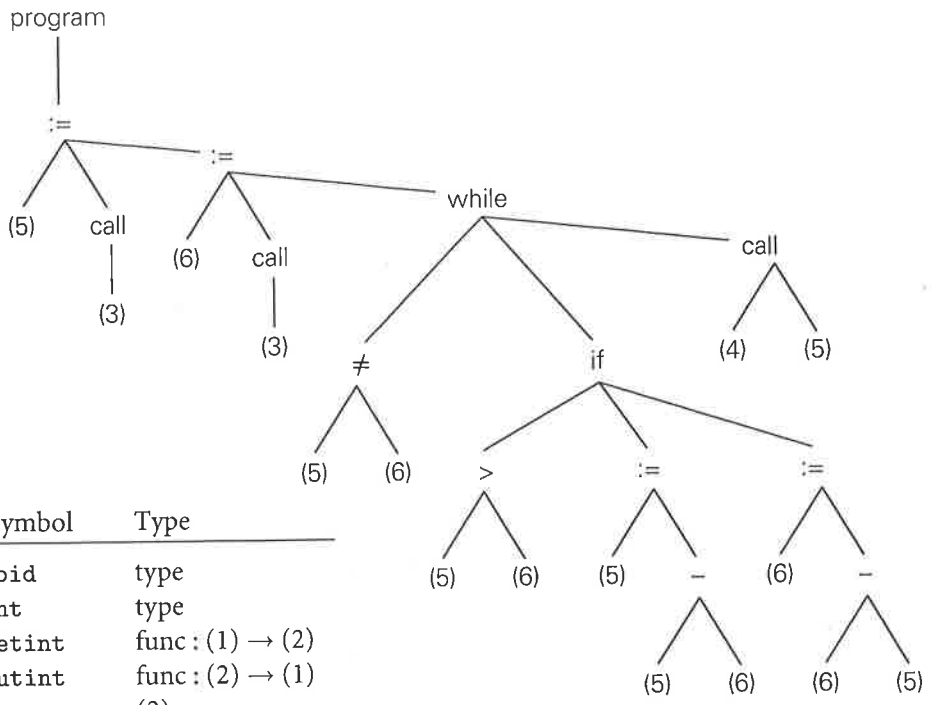
EXAMPLE 1.24
GCD program abstract syntax tree

Many interpreters use an annotated syntax tree to represent the running program: "execution" then amounts to tree traversal. In our GCD program, an interpreter would start at the root of Figure 1.6 and visit, in order, the statements on the main spine of the tree. At the first ":=" node, the interpreter would notice that the right child is a call: it would therefore call the `getint` routine (found in slot 3 of the symbol table) and assign the result into `i` (found in slot 5 of the symbol table). At the second ":=" node the interpreter would similarly assign the result of `getint` into `j`. At the while node it would repeatedly evaluate the left ("≠") child and, if the result was true, recursively walk the tree under the right (if) child. Finally, once the while node's left child evaluated to false, the interpreter would move on to the final call node, and output its result.

EXAMPLE 1.25
Interpreting the syntax tree

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 (typically single pass) that generates some other intermediate form. One common such form consists of a *control flow graph* whose nodes resemble fragments of assembly language for a simple

10 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," outlawing programs for which correctness is difficult or impossible to verify at compile time.



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Figure 1.6 Syntax tree and symbol table for the GCD program. Note the contrast to Figure 1.5: the syntax tree retains just the essential structure of the program, omitting details that were needed only to drive the parsing algorithm.

idealized machine. We will consider this option further in Chapter 15, where a control flow graph for our GCD program appears in Figure 15.3. In a suite of related compilers, the front ends 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 intermediate representation of the program, generating loads and stores for variable references, interspersed with appropriate arithmetic operations, tests, and branches. Naive code for our GCD example appears in Figure 1.7, in x86 assembly language. It was generated automatically by a simple pedagogical compiler.

EXAMPLE 1.26
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 correspon-

```
    pushl   %ebp                # \
    movl    %esp, %ebp          #  ) reserve space for local variables
    subl    $16, %esp           # /
    call    getint              # read
    movl    %eax, -8(%ebp)      # store i
    call    getint              # read
    movl    %eax, -12(%ebp)     # store j
A:  movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    je      D                   # jump if i == j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    cmpl    %ebx, %edi          # compare
    jle     B                   # jump if i < j
    movl    -8(%ebp), %edi      # load i
    movl    -12(%ebp), %ebx     # load j
    subl    %ebx, %edi          # i = i - j
    movl    %edi, -8(%ebp)      # store i
    jmp     C
B:  movl    -12(%ebp), %edi     # load j
    movl    -8(%ebp), %ebx      # load i
    subl    %ebx, %edi          # j = j - i
    movl    %edi, -12(%ebp)     # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx      # load i
    push    %ebx                # push i (pass to putint)
    call    putint              # write
    addl    $4, %esp            # pop i
    leave                       # deallocate space for local variables
    mov     $0, %eax            # exit status for program
    ret                         # return to operating system
```

Figure 1.7 Naive x86 assembly language for the GCD program.

dence between Figures 1.6 and 1.7 generally apparent. A few hints: `esp`, `ebp`, `eax`, `ebx`, and `edi` are registers (special storage locations, limited in number, that can be accessed very quickly). `-8(%ebp)` refers to the memory location 8 bytes before the location whose address is in register `ebp`; in this program, `ebp` serves as a *base* from which we can find variables `i` and `j`. The argument to a subroutine `call` instruction is passed by pushing it onto a stack, for which `esp` is the top-of-stack pointer. The return value comes back in register `eax`. Arithmetic operations overwrite their second argument with the result of the operation.[11] ■

11 As noted in footnote 1, these are GNU assembler conventions; Microsoft and Intel assemblers specify arguments in the opposite order.

Often a code generator will save the symbol table for later use by a symbolic debugger, by including it in a 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 twice in the list of compiler phases: once immediately after semantic analysis and intermediate code generation, and again immediately after target code generation.

EXAMPLE 1.27
GCD program optimization

Applying a good code improver to the code in Figure 1.7 produces the code shown in Example 1.2. 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. For modern microprocessors, with complex internal behavior, compilers can usually generate better code than can human assembly language programmers. ■

CHECK YOUR UNDERSTANDING

22. List the principal phases of compilation, and describe the work performed by each.
23. List the phases that are also executed as part of interpretation.
24. 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.
25. What distinguishes the front end of a compiler from the back end?
26. 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?
27. What is the purpose of the compiler's symbol table?
28. What is the difference between static and dynamic semantics?

29. 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 tied 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.

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—target code generation and machine-specific code improvement—are known as the *back end*. They serve to build a target program—preferably a fast one—whose semantics match those of the source. Between the front end and the back end, a good compiler performs extensive machine-independent code improvement; the phases of this "middle end" typically comprise the bulk of the code of the compiler, and account for most of its execution time.

Chapters 3, 6, 7, 8, 9, and 10 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 15 through 17 discuss compiler back ends, including assemblers and linkers, run-time systems, and code improvement techniques. Additional language paradigms are covered in Chapters 11 through 14. 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 & Implementation" sidebars; Appendix C contains a list of numbered examples.