1 *Introduction*

This chapter presents the basics of compiler history and organization. We begin in Section 1.1 with an overview of compilers and the history of their development. From there, we explain in Section 1.2 what a compiler does and how various compilers can be distinguished from each other: by the kind of machine code they generate and by the format of the target code they generate.

In Section 1.3, we discuss a kind of language processor called an *interpreter* and explain how an interpreter differs from a compiler. Section 1.4 discusses the *syntax* (structure) and *semantics* (meaning) of programs. Next, in Section 1.5, we discuss the tasks that a compiler must perform, primarily *analysis* of the source program and *synthesis* of a target program. That section also covers the parts of a compiler, discussing each in some detail: scanner, parser, type checker, optimizer and code generator.

In Section 1.6, we discuss the mutual interaction of compiler design and programming language design. Similarly, in Section 1.7 the influence of computer architecture on compiler design is covered.

Section 1.8 introduces a number of important compiler variants, including *debugging* and *development compilers*, *optimizing compilers*, and *retargetable compilers*. Finally, in Section 1.9, we consider *program development environments* that integrate a compiler, editor and debugger into a single tool.

1.1 Overview and History of Compilation

Compilers are fundamental to modern computing. They act as *translators*, transforming human-oriented *programming languages* into computer-oriented *machine languages*. To most users, a compiler can be viewed as a "black box" that performs the transformation illustrated in Figure 1.1. A compiler allows virtually all computer users to ignore the machine-dependent details of machine

Reserving space for: UserView

Figure 1.1: A user's view of a compiler.

language. Compilers therefore allow programs and programming expertise to be *machine-independent*. This is a particularly valuable capability in an age in which the number and variety of computers continue to grow explosively.

The term *compiler* was coined in the early 1950s by Grace Murray Hopper. Translation was then viewed as the "compilation" of a sequence of machine-language subprograms selected from a library. Compilation (as we now know it) was called "automatic programming," and there was almost universal skepticism that it would ever be successful. Today, the automatic translation of programming languages is an accomplished fact, but programming language translators are still called "compilers."

Among the first real compilers in the modern sense were the FORTRAN compilers of the late 1950s. They presented the user with a problem-oriented, largely machine-independent source language. They also performed some rather ambitious optimizations to produce efficient machine code, since efficient code was deemed essential for FORTRAN to compete successfully against then-dominant assembly languages. These FORTRAN compilers proved the viability of *high-level* (that is, mostly machine-independent) compiled languages and paved the way for the flood of languages and compilers that was to follow.

In the early days, compilers were *ad hoc* structures; components and techniques were often devised as a compiler was built. This approach to constructing compilers lent an aura of mystery to them, and they were viewed as complex and costly. Today the compilation process is well-understood and compiler construction is routine. Nonetheless, crafting an efficient and reliable compiler is still a complex task. Thus, in this book, we seek first to help you master the fundamentals and then we explore some important innovations.

Compilers normally translate conventional programming languages like Java, C, and C++ into executable machine-language instructions. Compiler technology, however, is far more broadly applicable and has been employed in rather unexpected areas. For example, text-formatting languages like TEX and LATEX [Lam95] are really compilers; they translate text and formatting commands into detailed typesetting commands. PostScript [Ado90], which is generated by text-formatters like LATEX, Microsoft `Word`, and Adobe `FrameMaker`, is really a programming language. It is translated and executed by laser printers and document previewers to produce a readable form of a document. This standardized document representation language allows documents to be freely interchanged, independent of how they were created and how they will be viewed.

Mathmatica [Wol96] is an interactive system that intermixes programming with mathematics, With it, one can solve intricate problems in both symbolic and numeric forms. This system relies heavily on compiler techniques to handle the specification, internal representation, and solution of problems.

Languages like Verilog [Pal96] and VHDL [Coe89] address the creation of VLSI circuits. A **silicon compiler** specifies the layout and composition of a VLSI circuit mask, using standard cell designs. Just as an ordinary compiler must understand and enforce the rules of a particular machine language, a silicon compiler must understand and enforce the design rules that dictate the feasibility of a given circuit.

Compiler technology, in fact, is of value in almost any program that presents a nontrivial text-oriented command set, including the command and scripting languages of operating systems and the query languages of database systems. Thus, while our discussion will focus on traditional compilation tasks, innovative readers will undoubtedly find new and unexpected applications for the techniques presented.

1.2 What Compilers Do

Figure 1.1 represents a compiler as a translator of the programming language being compiled (the *source*) to some machine language (the *target*). This description suggests that all compilers do about the same thing, the only difference being their choice of source and target languages. In fact, the situation is a bit more complicated. While the issue of the accepted source language is indeed simple, there are many alternatives in describing the output of a compiler. These go beyond simply naming a particular target computer. Compilers may be distinguished in two ways:

- By the kind of machine code they generate
- By the format of the target code they generate

These are discussed in the following sections.

1.2.1 Distinguishing Compilers by the Machine Code Generated

Compilers may generate any of three types of code by which they can be differentiated:

- Pure Machine Code
- Augmented Machine Code
- Virtual Machine Code

Pure Machine Code

Compilers may generate code for a particular machine's instruction set, not assuming the existence of any operating system or library routines. Such machine code is often called **pure code** because it includes nothing but instructions that are part of that instruction set. This approach is rare. It is most commonly used in compilers for **system implementation languages**—languages intended for implementing operating systems or embedded applications (like a programmable controller). This form of target code can execute on bare hardware without dependence on any other software.

Augmented Machine Code

Far more often, compilers generate code for a machine architecture that is **augmented** with operating system routines and run-time language support routines. The execution of a program generated by such a compiler requires that a particular operating system be present on the target machine and a collection of language-specific run-time support routines (I/O, storage allocation, mathematical functions, and so on) be available to the program. The combination of the target machine's instruction set and these operating system and language support routines can be thought of as defining a **virtual machine**. A virtual machine is a machine that exists only as a hardware/software combination.

The degree to which the virtual machine matches the actual hardware can vary greatly. Some common compilers translate almost entirely to hardware instructions; for example, most FORTRAN compilers use software support only for I/O and mathematical functions. Other compilers assume a wide range of virtual instructions. These may include data transfer instructions (for example, to move bit fields), procedure call instructions (to pass parameters, save registers, allocate stack space, and so on), and dynamic storage instructions (to provide for heap allocation).

Virtual Machine Code

The third type of code generated is composed entirely of virtual instructions. This approach is particularly attractive as a technique for producing a **transportable compiler**, a compiler that can be run easily on a variety of computers. Transportability is enhanced because moving the compiler entails only writing a simulator for the virtual machine used by the compiler. This applies, however, only if the compiler **bootstraps**—compiles itself—or is written in an available language. If this virtual machine is kept simple and clean, the interpreter can be quite easy to write. Examples of this approach are early Pascal compilers and the Java compiler included in the Java Development Kit [Sun98]. Pascal uses P-code [Han85], while Java uses **Java virtual machine** (JVM) code. Both represent virtual stack machines. A decent simulator for P-code or JVM code can easily be written in a few weeks. Execution speed is roughly five to ten times slower than that of compiled code. Alternatively, the virtual machine code can be either translated into C code or expanded to machine code directly. This approach made Pascal and Java available for almost any platform. It was instrumental in Pascal's success in the 1970s and has been an important factor in Java's growing popularity.

As can be seen from the preceding discussion, virtual instructions serve a variety of purposes. They simplify the job of a compiler by providing primitives suitable for the particular language being translated (such as procedure calls and string manipulation). They also contribute to compiler transportability. Further, they may allow for a significant *decrease* in the size of generated code—instructions can be designed to meet the needs of a particular programming language (for example, JVM code for Java). Using this approach, one can realize as much as a two-thirds reduction in generated program size. When a program is transmitted over a slow communications path (e.g., a Java applet sent from a slow server), size can be a crucial factor.

When an entirely virtual instruction set is used as the target language, the instruction set must be interpreted (simulated) in software. In a **just-in-time** (JIT) approach, virtual instructions can be translated to target code just as they are about to be executed or when they have been interpreted often enough to merit translation into target code.

If a virtual instruction set is used often enough, it is even possible to develop special microprocessors (such as the PicoJava processor by Sun Microsystems) that directly implement the virtual instruction set in hardware.

To summarize, almost all compilers, to a greater or lesser extent, generate code for a virtual machine, some of whose operations must be interpreted in software or firmware. We consider them compilers because they make use of a distinct translation phase that precedes execution.

1.2.2 Target Code Formats

Another way that compilers differ from one another is in the format of the target code they generate. Target formats may be categorized as follows:

- Assembly language
- Relocatable binary
- Memory-image

Assembly Language (Symbolic) Format

The generation of assembly code simplifies and modularizes translation. A number of code generation decisions (how to compute addresses, whether to use absolute or relative addressing, and so on) can be left for the assembler. This approach is common among compilers that were developed either as instructional projects or to support experimental programming language designs.

Generating assembler code is also useful for **cross-compilation** (running a compiler on one computer, with its target language being the machine language of a second computer). This is because a symbolic form is produced that is easily transferred between different computers. This approach also makes it easier to check the correctness of a compiler, since its output can be observed.

Often C, rather than a specific assembly language, is generated, with C's being used as a “universal assembly language.” C is far more platform-independent than any particular assembly language. However, some aspects of a program (such as the run-time representations of program and data) are inaccessible using C code, while they are readily accessible in assembly language.

Most production-quality compilers do not generate assembly language; direct generation of target code (in relocatable or binary format, discussed next) is more efficient. However, the compiler writer still needs a way to check the correctness of generated code. Thus it is wise to design a compiler so that it optionally will produce **pseudoassembly language**, that is, a listing of what the assembly language would look like if it were produced.

Relocatable Binary Format

Target code also may be generated in a **binary format**. In this case, external references and local instruction and data addresses are not yet bound. Instead, addresses are assigned relative either to the beginning of the module or to symbolically named locations. (This latter alternative makes it easy to group together code sequences or data areas.) This format is the typical output of an assembler, so this approach simply eliminates one step in the process of preparing a program for execution. A linkage step is required to add any support libraries and other separately compiled routines referenced from within a compiled program and to produce an **absolute binary program** format that is executable.

Both relocatable binary and assembly language formats allow **modular compilation**, the breaking up of a large program into separately compiled pieces. They also allow **cross-language references**, calls of assembler routines or subprograms in other high-level languages. Further, they support subroutine libraries, for example, I/O, storage allocation, and math routines.

Memory-Image (Absolute Binary) Form

The compiled output may be loaded into the compiler's address space and immediately executed, instead of being left in a file as with the first two approaches. This process is usually much faster than going through the intermediate step of link/editing. It also allows a program to be prepared and executed in a single step. However, the ability to interface with external, library, and precompiled routines may be limited. Further, the program must be recompiled for each execution unless some means is provided for storing the memory image. Memory-image compilers are useful for student and debugging use, where frequent changes are the rule and compilation costs far exceed execution costs. It also can be useful to *not* save absolutes after compilation (for example, in order to save file space or to guarantee the use of only the most current library routines and class definitions).

Java is designed to use and share classes defined and implemented at a variety of organizations. Rather than use a fixed copy of a class (which may be outdated), the JVM supports **dynamic linking** of externally defined classes. That is, when a class is first referenced, a class definition may be remotely fetched, checked, and loaded during program execution. In this way, "foreign code" can be guaranteed to be up-to-date and secure.

The code format alternatives and the target code alternatives discussed here show that compilers can differ quite substantially while still performing the same sort of translation task.

1.3 Interpreters

Another kind of language processor is the **interpreter**. An interpreter differs from a compiler in that it executes programs without explicitly performing a translation. Figure 1.2 illustrates schematically how interpreters work.

Interpreters behave differently than a compiler. To an interpreter, a program is merely data that can be arbitrarily manipulated, just like any other data. The locus of control during execution resides in the interpreter, *not* in the user program (that is, the user program is passive rather than active).

Interpreters provide a number of capabilities not found in compilers, as follows.

- Programs may be modified as execution proceeds. This provides a straightforward interactive debugging capability. Depending on program structure, program modifications may require reparsing or repeated semantic analysis.

- Languages in which the type of object a variable denotes may change dynamically (e.g., Lisp and Scheme) are easily supported in an interpreter. Since the user program is continuously reexamined as execution proceeds, symbols need not have a fixed meaning (for example, a symbol may denote an integer scalar at one point and a Boolean array at a later point). Such **fluid bindings** are obviously much more troublesome for compilers, since dynamic changes in the meaning of a symbol make direct translation into machine code impossible.

Reserving space for: Idealized

Figure 1.2: An idealized interpreter.

- Interpreters can provide better diagnostics. Since source text analysis is intermixed with program execution, especially good diagnostics (re-creation of source lines in error, use of variable names in error messages, and so on) are produced more easily than they are by compilers.

- Interpreters provide a significant degree of machine independence, since no machine code is generated. All operations are performed within the interpreter. To move an interpreter to a new machine, you need only recompile the interpreter on the new machine.

However, direct interpretation of source programs can involve large overheads, as follows.

- As execution proceeds, program text must be continuously reexamined, with identifier bindings, types, and operations sometimes recomputed at each reference. For very dynamic languages, this can represent a 100 to 1 (or worse) factor in execution speed over compiled code. For more static languages (such as C or Java), the speed degradation is closer to a factor of 5 or 10 to 1.

- Substantial space overhead may be involved. The interpreter and all support routines must usually be kept available. Source text is often not as compact as if it were compiled (for example, symbol tables are present and program text may be stored in a format designed for easy access and modification rather than for space minimization). This size penalty may lead to restrictions in, for example, the size of programs and the number of variables or procedures. Programs beyond these built-in limits cannot be handled by the interpreter.

- Many languages (for example, C, C++, and Java) have both interpreters (for debugging and program development) and compilers (for production work).

In summary, all language processing involves interpretation at some level. Interpreters directly interpret source programs or some syntactically transformed versions of them. They may exploit the availability of a source representation to allow program text to be changed as it is executed and debugged. While a compiler has distinct translation and execution phases, some form of "interpretation" is still involved. The translation phase may generate a virtual machine language that is interpreted by software or a real machine language that is interpreted by a particular computer, either in firmware or hardware.

1.4 Syntax and Semantics of Programming Languages

A complete definition of a programming language must include the specification of its *syntax* (structure) and its *semantics* (meaning).

Syntax typically means context-free syntax because of the almost universal use of **context-free grammars** (CFGs) as a syntactic specification mechanism. Syntax defines the sequences of symbols that are legal; syntactic legality is independent of any notion of what the symbols mean. For example, a context-free syntax might say that a = b + c is syntactically legal, while b + c = a is not. Not all program structure can be described by context-free syntax, however. For example, CFGs cannot specify type compatibility and scoping rules (for instance, that a = b + c is illegal if any of the variables is undeclared or if b or c is of type Boolean).

Because of the limitations of CFGs, the semantic component of a programming language is commonly divided into two classes:

- Static semantics
- Run-time semantics

1.4.1 Static Semantics

The **static semantics** of a language is a set of restrictions that determine which syntactically legal programs are actually valid. Typical static semantic rules require that all identifiers be declared, that operators and operands be type-compatible, and that procedures be called with the proper number of parameters. The common thread through all of these rules is that they cannot be expressed with a CFG. Static semantics thus *augment* context-free specifications and complete the definition of valid programs.

Static semantics can be specified formally or informally. The prose feature descriptions found in most programming language manuals are informal specifications. They tend to be relatively compact and easy to read, but often they are imprecise. Formal specifications might be expressed using any of a variety of notations. For example, **attribute grammars** [Knu68] are a popular method of formally specifying static semantics. They formalize the semantic checks commonly found in compilers. The following rewriting rule, called a *production*, specifies that an expression, denoted by E, can be rewritten into an expression, E, plus a term, T.

$$\mathsf{E} \quad \rightarrow \quad \mathsf{E} + \mathsf{T}$$

In an attribute grammar, this production might be augmented with a type attribute for E and T and a predicate testing for type compatibility, such as

```
E_result  →  E_v1 + T_v2
                 if   $v1.type = \mathsf{numeric}$ and $v2.type = \mathsf{numeric}$
                 then $result.type \leftarrow \mathsf{numeric}$
                 else call ERROR( )
```

Attribute grammars are a reasonable blend of formality and readability, but they can still be rather verbose. A number of compiler writing systems employ attribute grammars and provide automatic evaluation of attribute values [RT88].

1.4.2 Run-time Semantics

Run-time, or **execution semantics** are used to specify what a program computes. These semantics are often specified very informally in a language manual or report. Alternatively, a more formal *operational*, or *interpreter*, model can be used. In such a model, a program "state" is defined and program execution is described in terms of changes to that state. For example, the semantics of the statement a = 1 is that the state component corresponding to a is changed to 1.

A variety of formal approaches to defining the run-time semantics of programming languages have been developed. Three of them, natural semantics, axiomatic semantics and denotational semantics, are described below.

Natural Semantics

Natural semantics [NN92] (sometimes called **structured operational semantics**) formalizes the operational approach. Given assertions known to be true before the evaluations of a construct, we can infer assertions that will hold after the construct's evaluation. Natural semantics has been used to define the semantics of a variety of languages, including standard ML [MTH90].

Axiomatic Definitions

Axiomatic definitions [Gri81] can be used to model execution at a more abstract level than operational models. They are based on formally specified *relations*, or *predicates*, that relate program variables. Statements are defined by how they modify these relations.

As an example of axiomatic definitions, the axiom defining $var \leftarrow exp$, states that a predicate involving *var* is **true** after statement execution if, and only if, the predicate obtained by replacing all occurrences of *var* by *exp* is **true** beforehand. Thus, for $y > 3$ to be **true** after execution of the statement $y \leftarrow x + 1$, the predicate $x + 1 > 3$ would have to be **true** before the statement is executed. Similarly, $y = 21$ is **true** after execution of $x \leftarrow 1$ if $y = 21$ is **true** before its execution—this is a roundabout way of saying that changing x doesn't affect y. However, if x is an **alias** (an alternative name) for y, the axiom is invalid. This is one reason why aliasing is discouraged (or forbidden) in modern language designs.

The axiomatic approach is good for deriving proofs of program correctness because it avoids implementation details and concentrates on how relations among variables are changed by statement execution. Although axioms can formalize important properties of the semantics of a programming language, it is difficult to use them to define most programming languages completely. For example, they do not do a good job of modeling implementation considerations such as running out of memory.

Denotational Models

Denotational models [Sch86] are more mathematical in form than operational models. Yet they still present notions of memory access and update that are central to procedural languages. They rely on notation and terminology drawn from mathematics, so they are often fairly compact, especially in comparison with operational definitions.

A denotational definition may be viewed as a syntax-directed definition that specifies the meaning of a construct in terms of the meaning of its immediate constituents. For example, to define addition, we might use the following rule:

$$E[T1 + T2]m = E[T1]m + E[T2]m$$

This definition says that the value obtained by adding two subexpressions, $T1$ and $T2$, in the context of a memory state m is defined to be the sum of the arithmetic values obtained by evaluating $T1$ in the context of m (denoted $E[T1]m$) and $T2$ in the context of m (denoted $E[T2]m$).

Denotational techniques are quite popular and form the basis for rigorous definitions of programming languages. Research has shown it is possible to convert denotational representations *automatically* to equivalent representations that are directly executable [Set83, Wan82, App85].

Again, our concern for precise semantic specification is motivated by the fact that writing a complete and accurate compiler for a programming language requires that the language itself be well-defined. While this assertion may seem self-evident, many languages are defined by imprecise language reference manuals. Such a manual typically includes a formal syntax specification but otherwise is written in an informal prose style. The resulting definition inevitably is ambiguous or incomplete on certain points. For example, in Java all functions must return via a `return expr` statement, where `expr` is assignable to the function's return type. Thus

```
public static int subr(int b) {
   if (b != 0)
      return b+100;
}
```

is illegal, since if `b` is equal to zero, `subr` will not return a value. But what about this:

```
public static int subr(int b) {
   if (b != 0)
      return b+100;
   else if (10*b == 0)
           return 1;
}
```

In this case, a proper return is always executed, since the `else` part is reached only if `b` equals zero; this implies that `10*b` is also equal to zero. Is the compiler expected to duplicate this rather involved chain of reasoning? Although the Java reference manual doesn't explicitly say so, there is an implicit "all paths reachable" assumption that allows a compiler to assume that both legs of an conditional are executable even if a detailed program analysis shows this to be untrue. Thus a compiler may reject `subr` as semantically illegal and in so doing trade simplicity for accuracy in its analysis. Indeed, the general problem of deciding whether a particular statement in a program is reachable is *undecidable* (this is a variant of the famous *halting problem* [HU79]). We certainly can't ask our Java compiler literally to do the impossible!

In practice, a trusted **reference compiler** can serve as a *de facto* language definition. That is, a programming language is, in effect, defined by what a compiler chooses to accept and how it chooses to translate language constructs. In fact, the operational and natural semantic approaches introduced previously take this view. A standard interpreter is defined for a language, and the meaning of a program is precisely whatever the interpreter says. An early (and very elegant) example of an operational definition is the seminal Lisp interpreter [McC65]. There, all of Lisp was defined in terms of the actions of a Lisp interpreter, assuming only seven primitive functions and the notions of argument binding and function call.

Of course, a reference compiler or interpreter is no substitute for a clear and precise semantic definition. Nonetheless, it is very useful to have a reference against which to test a compiler that is under development.

Reserving space for: SyntaxDirected

Figure 1.3: A syntax-directed compiler.

1.5 Organization of a Compiler

Any compiler must perform two major tasks:

1. *Analysis* of the source program being compiled
2. *Synthesis* of a target program that, when executed, will correctly perform the computations described by the source program

Almost all modern compilers are **syntax-directed**. That is, the compilation process is driven by the syntactic structure of the source program, as recognized by the parser. The parser builds this structure out of **tokens**, the elementary symbols used to define a programming language syntax. Recognition of syntactic structure is a major part of the **syntax analysis** task.

Semantic analysis examines the meaning (semantics) of the program on the basis of its syntactic structure. It plays a dual role. It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). It also begins the *synthesis phase*.

In the synthesis phase, either source language constructs are translated into some **intermediate representation** (IR) of the program or target code may be directly generated. If an IR is generated, it then serves as input to a *code generator* component that actually produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated. A common organization of all of these compiler components is depicted schematically in Figure 1.3. The following subsections describe the components of a compiler:

- Scanner
- Parser
- Type checker
- Optimizer
- Code generator

Chapter Chapter:global:two presents a simple compiler to provide concrete examples of many of the concepts introduced in this overview.

1.5.1 The Scanner

The **scanner** begins the analysis of the source program by reading the input text—character by character—and grouping individual characters into tokens—identifiers, integers, reserved words, delimiters, and so on. This is the first of several steps that produce successively higher-level representations of the input. The tokens are encoded (often as integers) and are fed to the parser for syntactic analysis. When necessary, the actual character string comprising the token is also passed along for use by the semantic phases. The scanner does the following.

- It puts the program into a compact and uniform format (a stream of tokens).
- It eliminates unneeded information (such as comments).
- It processes compiler control directives (for example, turn the listing on or off and include source text from a file).
- It sometimes enters preliminary information into symbol tables (for example, to register the presence of a particular label or identifier).
- It optionally formats and lists the source program.

The main action of building tokens is often driven by token descriptions. *Regular expression* notation (discussed in Chapter Chapter:global:three) is an effective approach to describing tokens. Regular expressions are a formal notation sufficiently powerful to describe the variety of tokens required by modern programming languages. In addition, they can be used as a specification for the automatic generation of finite automata (discussed in Chapter Chapter:global:three) that recognize **regular sets**, that is, the sets that regular expressions define. Recognition of regular sets is the basis of the **scanner generator**. A scanner generator is a program that actually produces a working scanner when given only a specification of the tokens it is to recognize. Scanner generators are a valuable compiler-building tool.

1.5.2 The Parser

The **parser**, when given a formal syntax specification, typically as a CFG, reads tokens and groups them into phrases as specified by the *productions* of the CFG being used. (Grammars are discussed in Chapters Chapter:global:two and Chapter:global:four; parsing is discussed in Chapters Chapter:global:five and Chapter:global:six.) Parsers are typically driven by tables created from the CFG by a *parser generator*.

The parser verifies correct syntax. If a syntax error is found, it issues a suitable error message. Also, it may be able to repair the error (to form a syntactically valid program) or to recover from the error (to allow parsing to be resumed). In many cases, syntactic error recovery or repair can be done automatically by consulting *error-repair* tables created by a parser/repair generator.

As syntactic structure is recognized, the parser usually builds an **abstract syntax tree** (AST). An abstract syntax tree is a concise representation of program structure, that is used to guide semantic processing Abstract syntax trees are discussed in Chapter Chapter:global:two, seven.

1.5.3 The Type Checker (Semantic Analysis)

The type checker checks the **static semantics** of each AST node. That is, it verifies that the construct the node represents is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on). If the construct is semantically correct, the type checker "decorates" the AST node by adding type information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be *translated*. That is, IR code that correctly implements the construct the AST represents is generated. Translation involves capturing the run-time meaning of a construct.

For example, an AST for a while loop contains two subtrees, one representing the loop's expression and the other representing the loop's body. *Nothing* in the AST captures the notion that a while loop loops! This meaning is captured when a while loop's AST is translated to IR form. In the IR, the notion of testing the value of the loop control expression and conditionally executing the loop body is made explicit.

The translator is largely dictated by the semantics of the source language. Little of the nature of the target machine needs to be made evident. As a convenience during translation, some general aspects of the target machine may be exploited (for example, that the machine is byte-addressable and that it has a runtime stack). However, detailed information on the nature of the target machine (operations available, addressing, register characteristics, and so on) is reserved for the code-generation phase.

In simple, nonoptimizing compilers, the translator may generate target code directly without using an explicit intermediate representation. This simplifies a compiler's design by removing an entire phase. However, it also makes retargeting the compiler to another machine much more difficult. Most compilers implemented as instructional projects generate target code directly from the AST, without using an IR.

More elaborate compilers may first generate a high-level IR (that is source language-oriented) and then subsequently translate it into a low-level IR (that is target machine-oriented). This approach allows a cleaner separation of source and target dependencies.

Symbol Tables

A **symbol table** is a mechanism that allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is declared or used, a symbol table provides access to the information collected about it. Symbol tables are used extensively during type checking, but they can also be used by other compiler phases to enter, share, and later retrieve information about variables, procedures, labels, and so on. Compilers may choose to use other structures to share information between compiler phases. For example, a program representation such as an AST may be expanded and refined to provide detailed information needed by optimizers, code generators, linkers, loaders, and debuggers.

1.5.4 The Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the **optimizer**. This phase can be complex, often involving numerous subphases, some of which may need to be applied more than once. Most compilers allow optimizations to be turned off so as to speed translation. Nonetheless, a carefully designed optimizer can significantly speed program execution by simplifying, moving, or eliminating unneeded computations.

If both a high-level and low-level IR are used, optimizations may be performed in stages. For example, a simple subroutine call may be expanded into the subroutine's body, with actual parameters substituted for formal parameters. This is a high-level optimization. Alternatively, a value already loaded from memory may be reused. This is a low-level optimization.

Optimization can also be done *after* code generation. An example is **peephole optimization**. Peephole optimization examines generated code a few instructions at a time (in effect, through a “peephole”). Common peephole optimizations include eliminating multiplications by one or additions of zero, eliminating a load of a value into a register when the value is already in another register, and replacing a sequence of instructions by a single instruction with the same effect.

A peephole optimizer does not offer the payoff of a full-scale optimizer. However, it can significantly improve code and is often useful for "cleaning up" after earlier compiler phases.

1.5.5 The Code Generator

The IR code produced by the translator is mapped into target machine code by the **code generator**. This phase requires detailed information about the target machine and includes machine-specific optimization such as register allocation and code scheduling. Normally, code generators are hand-coded and can be quite complex, since generation of good target code requires consideration of many special cases.

The notion of *automatic construction* of code generators has been actively studied. The basic approach is to match a low-level IR to target-instruction templates, with the code generator automatically choosing instructions that best match IR instructions. This approach localizes the target-machine dependences of a compiler and, at least in principle, makes it easy to **retarget** a compiler to a new target machine. Automatic retargeting is an especially desirable goal, since a great deal of work is usually needed to move a compiler to a new machine. The ability to retarget by simply changing the set of target machine templates and generating (from the templates) a new code generator is compelling.

A well-known compiler using these techniques is the GNU C compiler [Sta89], `gcc`. `gcc` is a heavily optimizing compiler that has machine description files for more than ten popular computer architectures and at least two language front ends (C and C++).

1.5.6 Compiler Writing Tools

Finally, note that in discussing compiler design and construction, we often talk of *compiler writing tools*. These are often packaged as *compiler generators* or *compiler-compilers*. Such packages usually include scanner and parser generators. Some also include symbol table routines, attribute grammar evaluators, and code-generation tools. More advanced packages may aid in error repair generation.

These sorts of generators greatly aid in building pieces of compilers, but much of the effort in building a compiler lies in writing and debugging the semantic phases. These routines are numerous (a type checker and translator are needed for each distinct AST node) and are usually hand-coded.

1.6 Compiler Design and Programming Language Design

Our primary interest is the design and implementation of compilers for modern programming languages. An interesting aspect of this study is how programming language design and compiler design influence one another. Obviously, programming language design influences, and indeed often dictates, compiler design. Many clever and sometimes subtle compiler techniques arise from the need to cope with some programming language construct. A good example of this is the **closure** mechanism that was invented to handle formal procedures. A closure is a special run-time representation for a function. It consists of a pointer to the function's body *and* to its execution environment.

The state of the art in compiler design also strongly affects programming language design, if only because a programming language that cannot be compiled effectively will usually remain unused! Most successful programming language designers (such as the Java development team) have extensive compiler design backgrounds.

A programming language that is easy to compile has many advantages, as follows.

- It often is easier to learn, read, and understand. If a feature is hard to compile, it may well be hard to understand.
- It will have quality compilers on a wide variety of machines. This fact is often crucial to a language's success. For example, C, C++, and FORTRAN are widely available and very popular; Ada and Modula-3 have limited availability and are far less popular.
- Often better code will be generated. Poor quality code can be fatal in major applications.
- Fewer compiler bugs will occur. If a language can't be understood, how can it be effectively compiled?
- The compiler will be smaller, cheaper, faster, more reliable, and more widely used.
- Compiler diagnostics and program development tools will often be better.

Throughout our discussion of compiler design, we draw ideas, solutions, and shortcomings from many languages. Our primary focus is on Java and C, but we also consider Ada, C++, SmallTalk, ML, Pascal, and FORTRAN. We concentrate on Java and C because they are representative of the issues posed by modern language designs. We consider other languages so as to identify alternative design approaches that present new challenges to compilation.

1.7 Architectural Influences of Computer Design

Advances in computer architecture and microprocessor fabrication have spearheaded the computer revolution. At one time, a computer offering one megaflop performance (1,000,000 floating-point operations per second) was considered advanced. Now computers offering in excess of 10 *teraflops* (10,000,000,000,000 floating-point operations per second) are under development.

Compiler designers are responsible for making this vast computing capability available to programmers. Although compilers are rarely visibly to the end users of application programs, they are an essential "enabling technology." The problems encountered in efficiently harnessing the capability of a modern microprocessor are numerous, as follows.

- Instruction sets for some popular microprocessors, particularly the x86 series, are highly nonuniform. Some operations must be done in registers, while others can be done in memory. Often a number of register classes exist, each suitable for only a particular class of operations.
- High-level programming language operations are not always easy to support. Heap operations can take hundreds or thousands of machine instructions to implement. Exceptions and program threads are far more expensive and complex to implement than most users suspect.
- Essential architectural features such as hardware caches and distributed processors and memory are difficult to present to programmers in an architecturally independent manner. Yet misuse of these features can impose immense performance penalties.

Data and program integrity have been undervalued, and speed has been overemphasized. As a result, programming errors can go undetected because of a fear that extra checking will slow down execution unacceptably. A major complexity in implementing Java is efficiently enforcing the run-time integrity constraints it imposes.

1.8 Compiler Variants

Compilers come in many forms, including the following:

- Debugging, or development compilers
- Optimizing compilers
- Retargetable compilers

These are discussed in the following sections.

1.8.1 Debugging (Development) Compilers

A **debugging**, or **development** compiler such as CodeCenter [KLP88] or Borland C++ [Sch97] is specially designed to aid in the development and debugging of programs. It carefully scrutinizes programs and details programmer errors. It also often can tolerate or repair minor errors (for example, insert a missing comma or parenthesis). Some program errors can be detected only at run-time. Such errors include invalid subscripts, misuse of pointers, and illegal file manipulations. A debugging compiler may include the checking of code that can detect run-time errors and initiate a symbolic debugger. Although debugging compilers are particularly useful in instructional environments, diagnostic techniques are of value in all compilers. In the past, development compilers were used only in the initial stages of program development. When a program neared completion, a "production compiler," which increased compilation and execution speed by ignoring diagnostic concerns, was used. This strategy has been likened by Tony Hoare to wearing a life jacket in sailing classes held on dry land but abandoning the jacket when at sea! Indeed, it is becoming increasingly clear that for almost all programs, correctness rather than speed is the paramount concern. Java, for example, mandates run-time checks that C and C++ ignore.

Ways of detecting and characterizing errors in heavily used application programs are of great interest. Tools such as `purify` [HJ92] can add initialization and array bounds checks to already compiled programs, thereby allowing illegal operations to be detected even when source files are not available.

1.8.2 Optimizing Compilers

An **optimizing compiler** is specially designed to produce efficient target code at the cost of increased compiler complexity and possibly increased compilation times. In practice, all production quality compilers—those whose output will be used in everyday work—make some effort to generate good target code. For example, no add instruction would normally be generated for the expression `i+0`.

The term *optimizing compiler* is actually a misnomer. This is because no compiler, no matter how sophisticated, can produce *optimal* code for all programs. The reason for this is twofold. First, theoretical computer science has shown that even so simple a question as whether two programs are equivalent is **undecidable**, that is, it cannot be solved by *any* computer program. Thus finding the simplest (and most efficient) translation of a program can't always be done. Second, many program optimizations require time proportional to an exponential function of the size of the program being compiled. Thus optimal code, even when theoretically possible, often is infeasible in practice.

Optimizing compilers actually use a wide variety of transformations that improve a program's performance. The complexity of an optimizing compiler arises from the need to employ a variety of transforms, some of which interfere with each other. For example, keeping frequently used variables in registers reduces their access time but makes procedure and function calls more expensive. This is because registers need to be saved across calls. Many optimizing compilers provide a number of levels of optimization, each providing increasingly greater code improvements at increasingly greater costs. The choice of which improvements are most effective (and least expensive) is a matter of judgment and experience. In later chapters, we suggest possible optimizations, with the emphasis on those that are both simple and effective. Discussion of a comprehensive optimizing compiler is beyond the scope of this book. However, compilers that produce high-quality code at reasonable cost are an achievable goal.

1.8.3 Retargetable Compilers

Compilers are designed for a particular programming language (the source language) and a particular target computer (the computer for which it will generate code). Because of the wide variety of programming languages and computers that exist, a large number of similar, but not identical, compilers must be written. While this situation has decided benefits for those of us in the compiler writing business, it does make for a lot of duplication of effort and for a wide variance in compiler quality. As a result, a new kind of compiler, the **retargetable compiler**, has become important.

A retargetable compiler is one whose target machine can be changed without its machine-independent components having to be rewritten. A retargetable compiler is more difficult to write than an ordinary compiler because target machine dependencies must be carefully localized. It also is often difficult for a retargetable compiler to generate code that is as efficient as that of an ordinary compiler because special cases and machine idiosyncrasies are harder to exploit. Nonetheless, because a retargetable compiler allows development costs to be shared and provides for uniformity across computers, it is an important innovation. While discussing the fundamentals of compilation, we concentrate on compilers targeted to a single machine. In later chapters, the techniques needed to provide retargetability will be considered.

1.9 Program Development Environment

In practice, a compiler is but one tool used in the edit-compile-test cycle. That is, a user first edits a program, then compiles it, and finally tests its performance. Since program bugs will inevitably be discovered and corrected, this cycle is repeated many times. A popular programming tool, the **program development environment** (PDE), has been designed to integrate this cycle within a single tool. A PDE allows programs to be built incrementally, with program checking and testing fully integrated. PDEs may be viewed as the next stage in the evolution of compilers.

We focus on the traditional **batch compilation** approach in which an entire source file is translated. Many of the techniques we develop can be reformulated into **incremental** form to support PDEs. Thus a parser can reparse only portions of a program that have been changed [WG97], and a type checker can analyze only portions of an abstract syntax tree that have been changed.

In this book, we concentrate on the translation of C, C++, and Java. We use the JVM as our target, but we also address popular microprocessor architectures, particularly RISC processors such as the MIPS [KHH91] and Sparc [WG94]. At the code-generation stage, a variety of current techniques designed to exploit fully a processor's capabilities will be explored. Like so much else in compiler design, experience is the best guide, so we start with the translation of a very simple language and work our way up to ever more challenging translation tasks.

Exercises

1. The model of compilation we introduced is essentially batch-oriented. In particular, it assumes that an entire source program has been written and that the program will be fully compiled before the programmer can execute the program or make any changes. An interesting and important alternative is an **interactive compiler**. An interactive compiler, usually part of an integrated program development environment, allows a programmer to interactively create and modify a program, fixing errors as they are detected. It also allows a program to be tested before it is fully written, thereby providing for stepwise implementation and testing.

   Redesign the compiler structure of Figure 1.3 to allow incremental compilation. (The key idea is to allow individual phases of a compiler to be run or rerun without necessarily doing a full compilation.)

2. Most programming languages, such as C and C++, are compiled directly into the machine language of a "real" microprocessor (for example, an Intel x86 or DEC alpha). Java takes a different approach. It is commonly compiled into the machine language of the **Java virtual machine** (JVM). The JVM isn't implemented in its own microprocessor but rather is interpreted on some existing processor. This allows Java to be run on a wide variety of machines, thereby making it highly "platform-independent."

   Explain why building an interpreter for a virtual machine like the JVM is easier and faster than building a complete Java compiler. What are the disadvantages of this virtual machine approach to compiler implementation?

3. C compilers are almost always written in C. This raises something of a "chicken and egg" problem—how was the *first* C compiler for a particular system created? If you need to create the first compiler for language X on system Y, one approach is to create a **cross-compiler**. A cross-compiler runs on system Z but generates code for system Y.

   Explain how, starting with a compiler for language X that runs on system Z, you might use cross-compilation to create a compiler for language X, written in X, that runs on system Y and generates code for system Y.

   What extra problems arise if system Y is "bare"—that is, has no operating system or compilers for any language? (Recall that Unix is written in C and thus must be compiled before its facilities can be used.)

4. Cross-compilation assumes that a compiler for language X exists on some machine. When the first compiler for a new language is created, this assumption doesn't hold. In this situation, a **bootstrapping** approach can be taken. First, a subset of language X is chosen that is sufficient to implement a simple compiler. Next, a simple compiler for the X subset is written in any available language. This compiler must be correct, but it should not be any more elaborate than is necessary, since it will soon be discarded. Next, the subset compiler for X is rewritten in the X subset and then compiled

using the subset compiler previously created. Finally, the X subset, and its compiler, can be enhanced until a complete compiler for X, written in X, is available.

Assume you are bootstrapping C++ or Java (or some comparable language). Outline a suitable subset language. What language features must be in the language? What other features are desirable?

5. To allow the creation of camera-ready documents, languages like TEX and LATEX have been created. These languages can be thought of as varieties of programming language whose output controls laser printers or phototypesetters. Source language commands control details like spacing, font choice, point size, and special symbols. Using the syntax-directed compiler structure of Figure 1.3, suggest the kind of processing that might occur in each compiler phase if TEX or LATEX input was being translated.

   An alternative to "programming" documents is to use a sophisticated editor such as that provided in Microsoft `Word` or Adobe `FrameMaker` to interactively enter and edit the document. (Editing operations allow the choice of fonts, selection of point size, inclusion of special symbols, and so on.) This approach to document preparation is called **WYSIWYG**—what you see is what you get—because the exact form of the document is always visible.

   What are the relative advantages and disadvantages of the two approaches? Do analogues exist for ordinary programming languages?

6. Although compilers are designed to translate a particular language, they often allow calls to subprograms that are coded in some other language (typically, FORTRAN, C, or assembler). Why are such "foreign calls" allowed? In what ways do they complicate compilation?

7. Most C compilers (including the GNU `gcc` compiler) allow a user to examine the machine instructions generated for a given source program. Run the following program through such a C compiler and examine the instructions generated for the `for` loop. Next, recompile the program, enabling optimization, and reexamine the instructions generated for the `for` loop. What improvements have been made? Assuming that the program spends all of its time in the `for` loop, estimate the speedup obtained. Execute and time the two versions of the program and see how accurate your estimate is.

```
int proc(int a[]) {
    int sum = 0, i;
    for (i=0; i < 1000000; i++)
       sum += a[i];
    return sum;
}
```

8. C is sometimes called the "universal assembly language" in light of its ability to be very efficiently implemented on a wide variety of computer architectures. In light of this characterization, some compiler writers have chosen to generate C code, rather than a particular machine language, as their output. What are the advantages to this approach to compilation? Are there any disadvantages?

9. Many computer systems provide an interactive debugger (for example, `gdb` or `dbx`) to assist users in diagnosing and correcting run-time errors. Although a debugger is run long after a compiler has done its job, the two tools still must cooperate. What information (beyond the translation of a program) must a compiler supply to support effective run-time debugging?

10. Assume you have a source program *P*. It is possible to transform *P* into an equivalent program *P′* by reformatting *P* (by adding or deleting spaces, tabs, and line breaks), systematically renaming its variables (for example, changing all occurrences of `sum` to `total`), and reordering the definition of variables and subroutines.

    Although *P* and *P′* are equivalent, they may well look very different. How could a compiler be modified to compare two programs and determine if they are equivalent (or very similar)? In what circumstances would such a tool be useful?

Bibliography

[Ado90] Adobe Systems Incorporated. *PostScript Language Reference Manual, 2nd Edition*. Addison-Wesley, Reading, Mass., 1990.

[App85] Andrew W. Appel. Semantics-directed code generation. *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana*, pages 315–324, 1985.

[Coe89] David R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, Boston, Mass., 1989.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, N.Y., 1981.

[Han85] Per Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice-Hall, 1985.

[HJ92] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Winter Usenix Conference Proceedings*, pages 125–136, 1992.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[KHH91] Gerry Kane, Joe Heinrich, and Joseph Heinrich. *Mips Risc Architecture, Second Edition*. Prentice Hall, 1991.

[KLP88] S. Kaufer, R. Lopez, and S. Pratap. Saber-c an interpreter-based programming environment for the c language. *Summer Usenix Conference Proceedings*, pages 161–171, 1988.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, 1968.

[Lam95] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, 1995.

[McC65] John McCarthy. *Lisp 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, and London, England, 1965.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[NN92] H.R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley and Sons, New York, N.Y., 1992.

[Pal96] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Sun Microsystems Press, 1996.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual, Third Edition*. Springer-Verlag, New York, N.Y., 1988.

[Sch86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sch97] Herbert Schildt. *Borland C++ : The Complete Reference*. Osborne McGraw-Hill, 1997.

[Set83] Ravi Sethi. Control flow aspects of semantics-directed compiling. *ACM Transactions on Programming Languages and Systems*, 5(4), 1983.

[Sta89] Richard M. Stallman. *Using and Porting GNU CC*. Free Siftware Foundation, Inc., 1989.

[Sun98] Sun Microsystems. Jdk 1.1.6 release notes., 1998.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[WG94] David L. Weaver and Tom Germond. *The Sparc Architecture Manual, Version 9*. Prentice Hall, 1994.

[WG97] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 31–43, June 1997.

[Wol96] Stephen Wolfram. *The Mathematica Book, Third Edition*. Cambridge University Press, 1996.