Chapter 1

Overview of Compilation

CHAPTER OVERVIEW

Compilers are computer programs that translate a program written in one language into a program written in another language. At the same time, a compiler is a large software system, with many internal components and algorithms and complex interactions between them. Thus, the study of compiler construction is an introduction to techniques for the translation and improvement of programs, and a practical exercise in software engineering. This chapter provides a conceptual overview of all the major components of a modern compiler.

Keywords: Compiler, Interpreter, Automatic Translation

1.1 INTRODUCTION

The role of the computer in daily life grows each year. With the rise of the Internet, computers and the software that runs on them provide communications, news, entertainment, and security. Embedded computers have changed the ways that we build automobiles, airplanes, telephones, televisions, and radios. Computation has created entirely new categories of activity, from video games to social networks. Supercomputers predict daily weather and the course of violent storms. Embedded computers synchronize traffic lights and deliver e-mail to your pocket.

All of these computer applications rely on software computer programs that build virtual tools on top of the low-level abstractions provided by the underlying hardware. Almost all of that software is translated by a tool called a *compiler*. A compiler is simply a computer program that translates other computer programs to prepare them for execution. This book presents the fundamental techniques of automatic translation that are used

Compiler
a computer program that translates other computer programs

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00001-3

to build compilers. It describes many of the challenges that arise in compiler construction and the algorithms that compiler writers use to address them.

Conceptual Roadmap

A compiler is a tool that translates software written in one language into another language. To translate text from one language to another, the tool must understand both the form, or syntax, and content, or meaning, of the input language. It needs to understand the rules that govern syntax and meaning in the output language. Finally, it needs a scheme for mapping content from the source language to the target language.

The structure of a typical compiler derives from these simple observations. The compiler has a front end to deal with the source language. It has a back end to deal with the target language. Connecting the front end and the back end, it has a formal structure for representing the program in an intermediate form whose meaning is largely independent of either language. To improve the translation, a compiler often includes an optimizer that analyzes and rewrites that intermediate form.

Overview

Computer programs are simply sequences of abstract operations written in a *programming language*—a formal language designed for expressing computation. Programming languages have rigid properties and meanings—as opposed to natural languages, such as Chinese or Portuguese. Programming languages are designed for expressiveness, conciseness, and clarity. Natural languages allow ambiguity. Programming languages are designed to avoid ambiguity; an ambiguous program has no meaning. Programming languages are designed to specify computations—to record the sequence of actions that perform some task or produce some results.

Programming languages are, in general, designed to allow humans to express computations as sequences of operations. Computer processors, hereafter referred to as processors, microprocessors, or machines, are designed to execute sequences of operations. The operations that a processor implements are, for the most part, at a much lower level of abstraction than those specified in a programming language. For example, a programming language typically includes a concise way to print some number to a file. That single programming language statement must be translated into literally hundreds of machine operations before it can execute.

The tool that performs such translations is called a compiler. The compiler takes as input a program written in some language and produces as its output an equivalent program. In the classic notion of a compiler, the output

program is expressed in the operations available on some specific processor, often called the target machine. Viewed as a black box, a compiler might look like this:

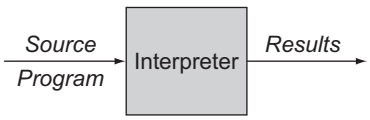
Typical "source" languages might be C, C++, FORTRAN, Java, or ML. The "target" language is usually the instruction set of some processor.

Instruction set
The set of operations supported by a processor; the overall design of an instruction set is often called an *instruction set architecture* or ISA.

Some compilers produce a target program written in a human-oriented programming language rather than the assembly language of some computer. The programs that these compilers produce require further translation before they can execute directly on a computer. Many research compilers produce C programs as their output. Because compilers for C are available on most computers, this makes the target program executable on all those systems, at the cost of an extra compilation for the final target. Compilers that target programming languages rather than the instruction set of a computer are often called *source-to-source translators*.

Many other systems qualify as compilers. For example, a typesetting program that produces PostScript can be considered a compiler. It takes as input a specification for how the document should look on the printed page and it produces as output a PostScript file. PostScript is simply a language for describing images. Because the typesetting program takes an executable specification and produces another executable specification, it is a compiler.

The code that turns PostScript into pixels is typically an *interpreter*, not a compiler. An interpreter takes as input an executable specification and produces as output the result of executing the specification.



Some languages, such as Perl, Scheme, and APL, are more often implemented with interpreters than with compilers.

Some languages adopt translation schemes that include both compilation and interpretation. Java is compiled from source code into a form called *bytecode*, a compact representation intended to decrease download times for Java applications. Java applications execute by running the bytecode on the corresponding Java Virtual Machine (JVM), an interpreter for bytecode. To complicate the picture further, many implementations of the JVM include a

Virtual machine
A virtual machine is a simulator for some processor. It is an *interpreter* for that machine's instruction set.

compiler that executes at runtime, sometimes called a *just-in-time compiler*, or JIT, that translates heavily used bytecode sequences into native code for the underlying computer.

Interpreters and compilers have much in common. They perform many of the same tasks. Both analyze the input program and determine whether or not it is a valid program. Both build an internal model of the structure and meaning of the program. Both determine where to store values during execution. However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the result. This book focuses on the problems that arise in building compilers. However, an implementor of interpreters may find much of the material relevant.

Why Study Compiler Construction?

A compiler is a large, complex program. Compilers often include hundreds of thousands, if not millions, of lines of code, organized into multiple subsystems and components. The various parts of a compiler interact in complex ways. Design decisions made for one part of the compiler have important ramifications for other parts. Thus, the design and implementation of a compiler is a substantial exercise in software engineering.

A good compiler contains a microcosm of computer science. It makes practical use of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), finite automata and push-down automata (scanning and parsing), and fixed-point algorithms (data-flow analysis). It deals with problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy management, and pipeline scheduling. Few software systems bring together as many complex and diverse components. Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.

Compilers play a fundamental role in the central activity of computer science: preparing problems for solution by computer. Most software is compiled, and the correctness of that process and the efficiency of the resulting code have a direct impact on our ability to build large systems. Most students are not satisfied with reading about these ideas; many of the ideas must be implemented to be appreciated. Thus, the study of compiler construction is an important component of a computer science education.

Compilers demonstrate the successful application of theory to practical problems. The tools that automate the production of scanners and parsers apply results from formal language theory. These same tools are used for

text searching, website filtering, word processing, and command-language interpreters. Type checking and static analysis apply results from lattice theory, number theory, and other branches of mathematics to understand and improve programs. Code generators use algorithms for tree-pattern matching, parsing, dynamic programming, and text matching to automate the selection of instructions.

Still, some problems that arise in compiler construction are open problems—that is, the current best solutions have room for improvement. Attempts to design high-level, universal, intermediate representations have foundered on complexity. The dominant method for scheduling instructions is a greedy algorithm with several layers of tie-breaking heuristics. While it is obvious that compilers should use commutativity and associativity to improve the code, most compilers that try to do so simply rearrange the expression into some canonical order.

Building a successful compiler requires expertise in algorithms, engineering, and planning. Good compilers approximate the solutions to hard problems. They emphasize efficiency, in their own implementations and in the code they generate. They have internal data structures and knowledge representations that expose the right level of detail—enough to allow strong optimization, but not enough to force the compiler to wallow in detail. Compiler construction brings together ideas and techniques from across the breadth of computer science and applies them in a constrained setting to solve some truly hard problems.

The Fundamental Principles of Compilation

Compilers are large, complex, carefully engineered objects. While many issues in compiler design are amenable to multiple solutions and interpretations, there are two fundamental principles that a compiler writer must keep in mind at all times. The first principle is inviolable:

The compiler must preserve the meaning of the program being compiled.

Correctness is a fundamental issue in programming. The compiler must preserve correctness by faithfully implementing the "meaning" of its input program. This principle lies at the heart of the social contract between the compiler writer and compiler user. If the compiler can take liberties with meaning, then why not simply generate a `nop` or a `return`? If an incorrect translation is acceptable, why expend the effort to get it right?

The second principle that a compiler must observe is practical:

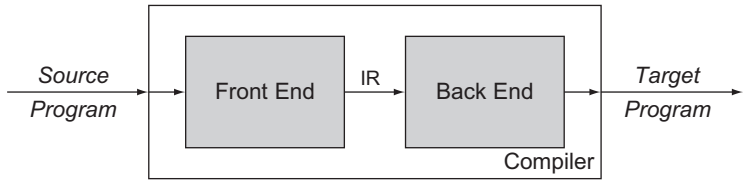
The compiler must improve the input program in some discernible way.

A traditional compiler improves the input program by making it directly executable on some target machine. Other "compilers" improve their input in different ways. For example, `tpic` is a program that takes the specification for a drawing written in the graphics language `pic` and converts it into LaTeX; the "improvement" lies in LaTeX's greater availability and generality. A source-to-source translator for C must produce code that is, in some measure, better than the input program; if it is not, why would anyone invoke it?

1.2 COMPILER STRUCTURE

A compiler is a large, complex software system. The community has been building compilers since 1955, and over the years, we have learned many lessons about how to structure a compiler. Earlier, we depicted a compiler as a simple box that translates a source program into a target program. Reality, of course, is more complex than that simple picture.

As the single-box model suggests, a compiler must both understand the source program that it takes as input and map its functionality to the target machine. The distinct nature of these two tasks suggests a division of labor and leads to a design that decomposes compilation into two major pieces: a *front end* and a *back end*.



The front end focuses on understanding the source-language program. The back end focuses on mapping programs to the target machine. This separation of concerns has several important implications for the design and implementation of compilers.

IR
A compiler uses some set of data structures to represent the code that it processes. That form is called an *intermediate representation*, or IR.

The front end must encode its knowledge of the source program in some structure for later use by the back end. This *intermediate representation* (IR) becomes the compiler's definitive representation for the code it is translating. At each point in compilation, the compiler will have a definitive representation. It may, in fact, use several different IRs as compilation progresses, but at each point, one representation will be the definitive IR. We think of the definitive IR as the version of the program passed between independent phases of the compiler, like the IR passed from the front end to the back end in the preceding drawing.

In a two-phase compiler, the front end must ensure that the source program is well formed, and it must map that code into the IR. The back end must map

MAY YOU STUDY IN INTERESTING TIMES

This is an exciting era in the design and implementation of compilers. In the 1980s, almost all compilers were large, monolithic systems. They took as input one of a handful of languages and produced assembly code for some particular computer. The assembly code was pasted together with the code produced by other compilations—including system libraries and application libraries—to form an executable. The executable was stored on a disk, and at the appropriate time, the final code was moved from the disk to main memory and executed.

Today, compiler technology is being applied in many different settings. As computers find applications in diverse places, compilers must cope with new and different constraints. Speed is no longer the sole criterion for judging the compiled code. Today, code might be judged on how small it is, on how much energy it consumes, on how well it compresses, or on how many page faults it generates when it runs.

At the same time, compilation techniques have escaped from the monolithic systems of the 1980s. They are appearing in many new places. Java compilers take partially compiled programs (in Java "bytecode" format) and translate them into native code for the target machine. In this environment, success requires that the sum of compile time plus runtime must be less than the cost of interpretation. Techniques to analyze whole programs are moving from compile time to link time, where the linker can analyze the assembly code for the entire application and use that knowledge to improve the program. Finally, compilers are being invoked at runtime to generate customized code that capitalizes on facts that cannot be known any earlier. If the compilation time can be kept small and the benefits are large, this strategy can produce noticeable improvements.

the IR program into the instruction set and the finite resources of the target machine. Because the back end only processes IR created by the front end, it can assume that the IR contains no syntactic or semantic errors.

The compiler can make multiple passes over the IR form of the code before emitting the target program. This should lead to better code, as the compiler can, in effect, study the code in one phase and record relevant details. Then, in later phases, it can use these recorded facts to improve the quality of translation. This strategy requires that knowledge derived in the first pass be recorded in the IR, where later passes can find and use it.

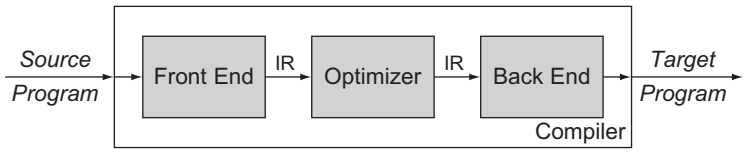
Finally, the two-phase structure may simplify the process of *retargeting* the compiler. We can easily envision constructing multiple back ends for a single front end to produce compilers that accept the same language but target different machines. Similarly, we can envision front ends for different

Retargeting
The task of changing the compiler to generate code for a new processor is often called *retargeting* the compiler.

languages producing the same IR and using a common back end. Both scenarios assume that one IR can serve for several combinations of source and target; in practice, both language-specific and machine-specific details usually find their way into the IR.

Optimizer
The middle section of a compiler, called an *optimizer*, analyzes and transforms the IR to improve it.

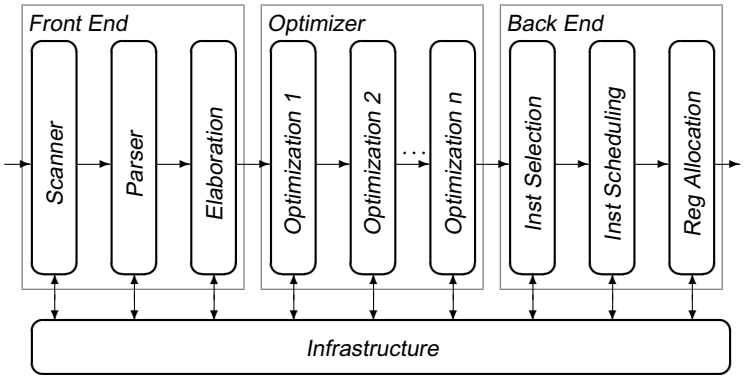
Introducing an IR makes it possible to add more phases to compilation. The compiler writer can insert a third phase between the front end and the back end. This middle section, or *optimizer*, takes an IR program as its input and produces a semantically equivalent IR program as its output. By using the IR as an interface, the compiler writer can insert this third phase with minimal disruption to the front end and back end. This leads to the following compiler structure, termed a *three-phase compiler*.



The optimizer is an IR-to-IR transformer that tries to improve the IR program in some way. (Notice that these transformers are, themselves, compilers according to our definition in Section 1.1.) The optimizer can make one or more passes over the IR, analyze the IR, and rewrite the IR. The optimizer may rewrite the IR in a way that is likely to produce a faster target program from the back end or a smaller target program from the back end. It may have other objectives, such as a program that produces fewer page faults or uses less energy.

Conceptually, the three-phase structure represents the classic optimizing compiler. In practice, each phase is divided internally into a series of passes. The front end consists of two or three passes that handle the details of recognizing valid source-language programs and producing the initial IR form of the program. The middle section contains passes that perform different optimizations. The number and purpose of these passes vary from compiler to compiler. The back end consists of a series of passes, each of which takes the IR program one step closer to the target machine's instruction set. The three phases and their individual passes share a common infrastructure. This structure is shown in Figure 1.1.

In practice, the conceptual division of a compiler into three phases, a front end, a middle section or optimizer, and a back end, is useful. The problems addressed by these phases are different. The front end is concerned with understanding the source program and recording the results of its analysis into IR form. The optimizer section focuses on improving the IR form.



■ **FIGURE 1.1** Structure of a Typical Compiler.

The back end must map the transformed IR program onto the bounded resources of the target machine in a way that leads to efficient use of those resources.

Of these three phases, the optimizer has the murkiest description. The term *optimization* implies that the compiler discovers an optimal solution to some problem. The issues and problems that arise in optimization are so complex and so interrelated that they cannot, in practice, be solved optimally. Furthermore, the actual behavior of the compiled code depends on interactions among all of the techniques applied in the optimizer and the back end. Thus, even if a single technique can be proved optimal, its interactions with other techniques may produce less than optimal results. As a result, a good optimizing compiler can improve the quality of the code, relative to an unoptimized version. However, an optimizing compiler will almost always fail to produce optimal code.

The middle section can be a single monolithic pass that applies one or more optimizations to improve the code, or it can be structured as a series of smaller passes with each pass reading and writing IR. The monolithic structure may be more efficient. The multipass structure may lend itself to a less complex implementation and a simpler approach to debugging the compiler. It also creates the flexibility to employ different sets of optimization in different situations. The choice between these two approaches depends on the constraints under which the compiler is built and operates.

1.3 OVERVIEW OF TRANSLATION

To translate code written in a programming language into code suitable for execution on some target machine, a compiler runs through many steps.

NOTATION

Compiler books are, in essence, about notation. After all, a compiler translates a program written in one notation into an equivalent program written in another notation. A number of notational issues will arise in your reading of this book. In some cases, these issues will directly affect your understanding of the material.

Expressing Algorithms We have tried to keep the algorithms concise. Algorithms are written at a relatively high level, assuming that the reader can supply implementation details. They are written in a *slanted, sans-serif font*. Indentation is both deliberate and significant; it matters most in an *if-then-else* construct. Indented code after a *then* or an *else* forms a block. In the following code fragment

```
if Action [s,word] = "shift sᵢ" then
   push word
   push sᵢ
   word ← NextWord()
else if ···
```

all the statements between the *then* and the *else* are part of the *then* clause of the *if-then-else* construct. When a clause in an *if-then-else* construct contains just one statement, we write the keyword *then* or *else* on the same line as the statement.

Writing Code In some examples, we show actual program text written in some language chosen to demonstrate a particular point. Actual program text is written in a `monospace` font.

Arithmetic Operators Finally, we have forsaken the traditional use of * for × and of / for ÷, except in actual program text. The meaning should be clear to the reader.

To make this abstract process more concrete, consider the steps needed to generate executable code for the following expression:

```
a ← a × 2 × b × c × d
```

where a, b, c, and d are variables, ← indicates an assignment, and × is the operator for multiplication. In the following subsections, we will trace the path that a compiler takes to turn this simple expression into executable code.

1.3.1 The Front End

Before the compiler can translate an expression into executable target-machine code, it must understand both its form, or *syntax*, and its meaning,

or *semantics*. The front end determines if the input code is well formed, in terms of both syntax and semantics. If it finds that the code is valid, it creates a representation of the code in the compiler's intermediate representation; if not, it reports back to the user with diagnostic error messages to identify the problems with the code.

Checking Syntax

To check the syntax of the input program, the compiler must compare the program's structure against a definition for the language. This requires an appropriate formal definition, an efficient mechanism for testing whether or not the input meets that definition, and a plan for how to proceed on an illegal input.

Mathematically, the source language is a set, usually infinite, of strings defined by some finite set of rules, called a *grammar*. Two separate passes in the front end, called the scanner and the parser, determine whether or not the input code is, in fact, a member of the set of valid programs defined by the grammar.

Programming language grammars usually refer to words based on their parts of speech, sometimes called syntactic categories. Basing the grammar rules on parts of speech lets a single rule describe many sentences. For example, in English, many sentences have the form

Sentence → *Subject* `verb` *Object* `endmark`

where `verb` and `endmark` are parts of speech, and *Sentence*, *Subject*, and *Object* are syntactic variables. *Sentence* represents any string with the form described by this rule. The symbol "→" reads "derives" and means that an instance of the right-hand side can be abstracted to the syntactic variable on the left-hand side.

Consider a sentence like "Compilers are engineered objects." The first step in understanding the syntax of this sentence is to identify distinct words in the input program and to classify each word with a part of speech. In a compiler, this task falls to a pass called the *scanner*. The scanner takes a stream of characters and converts it to a stream of classified words—that is, pairs of the form (*p*,*s*), where *p* is the word's *part of speech* and *s* is its spelling. A scanner would convert the example sentence into the following stream of classified words:

Scanner
the compiler pass that converts a string of characters into a stream of words

(`noun`,"Compilers"), (`verb`,"are"), (`adjective`,"engineered"), (`noun`,"objects"), (`endmark`,".")

In practice, the actual spelling of the words might be stored in a hash table and represented in the pairs with an integer index to simplify equality tests. Chapter 2 explores the theory and practice of scanner construction.

In the next step, the compiler tries to match the stream of categorized words against the rules that specify syntax for the input language. For example, a working knowledge of English might include the following grammatical rules:

1	*Sentence*	→	*Subject* `verb` *Object* `endmark`
2	*Subject*	→	`noun`
3	*Subject*	→	*Modifier* `noun`
4	*Object*	→	`noun`
5	*Object*	→	*Modifier* `noun`
6	*Modifier*	→	`adjective`
	...		

By inspection, we can discover the following *derivation* for our example sentence:

Rule	Prototype Sentence
—	*Sentence*
1	*Subject* `verb` *Object* `endmark`
2	`noun verb` *Object* `endmark`
5	`noun verb` *Modifier* `noun endmark`
6	`noun verb adjective noun endmark`

The derivation starts with the syntactic variable *Sentence*. At each step, it rewrites one term in the prototype sentence, replacing the term with a right-hand side that can be derived from that rule. The first step uses Rule 1 to replace *Sentence*. The second uses Rule 2 to replace *Subject*. The third replaces *Object* using Rule 5, while the final step rewrites *Modifier* with `adjective` according to Rule 6. At this point, the prototype sentence generated by the derivation matches the stream of categorized words produced by the scanner.

The derivation proves that the sentence "Compilers are engineered objects." belongs to the language described by Rules 1 through 6. The sentence is grammatically correct. The process of automatically finding derivations is called *parsing*. Chapter 3 presents the techniques that compilers use to parse the input program.

Parser
the compiler pass that determines if the input stream is a sentence in the source language

A grammatically correct sentence can be meaningless. For example, the sentence "Rocks are green vegetables" has the same parts of speech in the same order as "Compilers are engineered objects," but has no rational meaning. To understand the difference between these two sentences requires contextual knowledge about software systems, rocks, and vegetables.

Type checking
the compiler pass that checks for type-consistent uses of names in the input program

The semantic models that compilers use to reason about programming languages are simpler than the models needed to understand natural language. A compiler builds mathematical models that detect specific kinds of inconsistency in a program. Compilers check for consistency of type; for example, the expression

```
a ← a × 2 × b × c × d
```

might be syntactically well-formed, but if `b` and `d` are character strings, the sentence might still be invalid. Compilers also check for consistency of number in specific situations; for example, an array reference should have the same number of dimensions as the array's declared rank and a procedure call should specify the same number of arguments as the procedure's definition. Chapter 4 explores some of the issues that arise in compiler-based type checking and semantic elaboration.

Intermediate Representations

The final issue handled in the front end of a compiler is the generation of an IR form of the code. Compilers use a variety of different kinds of IR, depending on the source language, the target language, and the specific transformations that the compiler applies. Some IRs represent the program as a graph. Others resemble a sequential assembly code program. The code in the margin shows how our example expression might look in a low-level, sequential IR. Chapter 5 presents an overview of the variety of kinds of IRs that compilers use.

```
t0 ← a × 2
t1 ← t0 × b
t2 ← t1 × c
t3 ← t2 × d
a  ← t3
```

For every source-language construct, the compiler needs a strategy for how it will implement that construct in the IR form of the code. Specific choices affect the compiler's ability to transform and improve the code. Thus, we spend two chapters on the issues that arise in generation of IR for source-code constructs. Procedure linkages are, at once, a source of inefficiency in the final code and the fundamental glue that pieces together different source files into an application. Thus, we devote Chapter 6 to the issues that surround procedure calls. Chapter 7 presents implementation strategies for most other programming language constructs.

TERMINOLOGY

A careful reader will notice that we use the word *code* in many places where either *program* or *procedure* might naturally fit. Compilers can be invoked to translate fragments of code that range from a single reference through an entire system of programs. Rather than specify some scope of compilation, we will continue to use the ambiguous, but more general, term, *code*.

1.3.2 The Optimizer

When the front end emits IR for the input program, it handles the statements one at a time, in the order that they are encountered. Thus, the initial IR program contains general implementation strategies that will work in any surrounding context that the compiler might generate. At runtime, the code will execute in a more constrained and predictable context. The optimizer analyzes the IR form of the code to discover facts about that context and uses that contextual knowledge to rewrite the code so that it computes the same answer in a more efficient way.

Efficiency can have many meanings. The classic notion of optimization is to reduce the application's running time. In other contexts, the optimizer might try to reduce the size of the compiled code, or other properties such as the energy that the processor consumes evaluating the code. All of these strategies target efficiency.

Returning to our example, consider it in the context shown in Figure 1.2a. The statement occurs inside a loop. Of the values that it uses, only `a` and `d` change inside the loop. The values of `2`, `b`, and `c` are invariant in the loop. If the optimizer discovers this fact, it can rewrite the code as shown in Figure 1.2b. In this version, the number of multiplications has been reduced from $4 \cdot n$ to $2 \cdot n + 2$. For $n > 1$, the rewritten loop should execute faster. This kind of optimization is discussed in Chapters 8, 9, and 10.

Analysis

Data-flow analysis
a form of compile-time reasoning about the runtime flow of values

Most optimizations consist of an analysis and a transformation. The analysis determines where the compiler can safely and profitably apply the technique. Compilers use several kinds of analysis to support transformations. *Data-flow analysis* reasons, at compile time, about the flow of values at runtime. Data-flow analyzers typically solve a system of simultaneous set equations that are derived from the structure of the code being translated. *Dependence analysis* uses number-theoretic tests to reason about the values that can be

```
b ← ...
c ← ...
a ← 1
for i = 1 to n
  read d
  a ← a × 2 × b × c × d
  end
```

(a) Original Code in Context

```
b ← ...
c ← ...
a ← 1
t ← 2 × b × c
for i = 1 to n
  read d
  a ← a × d × t
  end
```

(b) Improved Code

■ **FIGURE 1.2** Context Makes a Difference.

assumed by subscript expressions. It is used to disambiguate references to array elements. Chapter 9 presents a detailed look at data-flow analysis and its application, along with the construction of static-single-assignment form, an IR that encodes information about the flow of both values and control directly in the IR.

Transformation

To improve the code, the compiler must go beyond analyzing it. The compiler must use the results of analysis to rewrite the code into a more efficient form. Myriad transformations have been invented to improve the time or space requirements of executable code. Some, such as discovering loop-invariant computations and moving them to less frequently executed locations, improve the running time of the program. Others make the code more compact. Transformations vary in their effect, the scope over which they operate, and the analysis required to support them. The literature on transformations is rich; the subject is large enough and deep enough to merit one or more separate books. Chapter 10 covers the subject of scalar transformations—that is, transformations intended to improve the performance of code on a single processor. It presents a taxonomy for organizing the subject and populates that taxonomy with examples.

1.3.3 The Back End

The compiler's back end traverses the IR form of the code and emits code for the target machine. It selects target-machine operations to implement each IR operation. It chooses an order in which the operations will execute efficiently. It decides which values will reside in registers and which values will reside in memory and inserts code to enforce those decisions.

ABOUT ILOC

Throughout the book, low-level examples are written in a notation that we call ILOC—an acronym derived from "intermediate language for an optimizing compiler." Over the years, this notation has undergone many changes. The version used in this book is described in detail in Appendix A.

Think of ILOC as the assembly language for a simple RISC machine. It has a standard set of operations. Most operations take arguments that are registers. The memory operations, `loads` and `stores`, transfer values between memory and the registers. To simplify the exposition in the text, most examples assume that all data consists of integers.

Each operation has a set of operands and a target. The operation is written in five parts: an operation name, a list of operands, a separator, a list of targets, and an optional comment. Thus, to add registers 1 and 2, leaving the result in register 3, the programmer would write

```
add r1,r2 ⇒ r3  // example instruction
```

The separator, ⇒, precedes the target list. It is a visual reminder that information flows from left to right. In particular, it disambiguates cases where a person reading the assembly-level text can easily confuse operands and targets. (See `loadAI` and `storeAI` in the following table.)

The example in Figure 1.3 only uses four ILOC operations:

ILOC Operation		Meaning
`loadAI`	$r_1, c_2 \Rightarrow r_3$	Memory(r_1+c_2) → r_3
`loadI`	$c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
`mult`	$r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
`storeAI`	$r_1 \Rightarrow r_2, c_3$	$r_1 \rightarrow$ Memory(r_2+c_3)

Appendix A contains a more detailed description of ILOC. The examples consistently use r_{arp} as a register that contains the start of data storage for the current procedure, also known as the *activation record pointer*.

Instruction Selection

$t_0 \leftarrow a \times 2$
$t_1 \leftarrow t_0 \times b$
$t_2 \leftarrow t_1 \times c$
$t_3 \leftarrow t_2 \times d$
$a \leftarrow t_3$

The first stage of code generation rewrites the IR operations into target machine operations, a process called *instruction selection*. Instruction selection maps each IR operation, in its context, into one or more target machine operations. Consider rewriting our example expression, $a \leftarrow a \times 2 \times b \times c \times d$, into code for the ILOC virtual machine to illustrate the process. (We will use ILOC throughout the book.) The IR form of the expression is repeated in the margin. The compiler might choose the operations shown in Figure 1.3. This code assumes that `a`, `b`, `c`, and `d`

```
loadAI  rarp, @a ⇒ ra       // load 'a'
loadI   2        ⇒ r2       // constant 2 into r2
loadAI  rarp, @b ⇒ rb       // load 'b'
loadAI  rarp, @c ⇒ rc       // load 'c'
loadAI  rarp, @d ⇒ rd       // load 'd'
mult    ra, r2   ⇒ ra       // ra ← a × 2
mult    ra, rb   ⇒ ra       // ra ← (a × 2) × b
mult    ra, rc   ⇒ ra       // ra ← (a × 2 × b) × c
mult    ra, rd   ⇒ ra       // ra ← (a × 2 × b × c) × d
storeAI ra       ⇒ rarp,@a  // write ra back to 'a'
```

■ **FIGURE 1.3** ILOC Code for a ← a × 2 × b × c × d.

are located at offsets @a, @b, @c, and @d from an address contained in the register r_{arp}.

The compiler has chosen a straightforward sequence of operations. It loads all of the relevant values into registers, performs the multiplications in order, and stores the result to the memory location for a. It assumes an unlimited supply of registers and names them with symbolic names such as r_a to hold a and r_{arp} to hold the address where the data storage for our named values begins. Implicitly, the instruction selector relies on the register allocator to map these symbolic register names, or *virtual registers*, to the actual registers of the target machine.

Virtual register
a symbolic register name that the compiler uses to indicate that a value can be stored in a register

The instruction selector can take advantage of special operations on the target machine. For example, if an immediate-multiply operation (multI) is available, it might replace the operation mult $r_a, r_2 \Rightarrow r_a$ with multI $r_a, 2 \Rightarrow r_a$, eliminating the need for the operation loadI $2 \Rightarrow r_2$ and reducing the demand for registers. If addition is faster than multiplication, it might replace mult $r_a, r_2 \Rightarrow r_a$ with add $r_a, r_a \Rightarrow r_a$, avoiding both the loadI and its use of r_2, as well as replacing the mult with a faster add. Chapter 11 presents two different techniques for instruction selection that use pattern matching to choose efficient implementations for IR operations.

Register Allocation

During instruction selection, the compiler deliberately ignored the fact that the target machine has a limited set of registers. Instead, it used virtual registers and assumed that "enough" registers existed. In practice, the earlier stages of compilation may create more demand for registers than the hardware can support. The register allocator must map those virtual registers

onto actual target-machine registers. Thus, the register allocator decides, at each point in the code, which values should reside in the target-machine registers. It then rewrites the code to reflect its decisions. For example, a register allocator might minimize register use by rewriting the code from Figure 1.3 as follows:

```
loadAI  r_arp, @a ⇒ r1          // load 'a'
add     r1, r1    ⇒ r1          // r1 ← a × 2
loadAI  r_arp, @b ⇒ r2          // load 'b'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2) × b
loadAI  r_arp, @c ⇒ r2          // load 'c'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2 × b) × c
loadAI  r_arp, @d ⇒ r2          // load 'd'
mult    r1, r2    ⇒ r1          // r1 ← (a × 2 × b × c) × d
storeAI r1        ⇒ r_arp, @a   // write r_a back to 'a'
```

This sequence uses three registers instead of six.

Minimizing register use may be counterproductive. If, for example, any of the named values, `a`, `b`, `c`, or `d`, are already in registers, the code should reference those registers directly. If all are in registers, the sequence could be implemented so that it required no additional registers. Alternatively, if some nearby expression also computed `a × 2`, it might be better to preserve that value in a register than to recompute it later. This optimization would increase demand for registers but eliminate a later instruction. Chapter 13 explores the problems that arise in register allocation and the techniques that compiler writers use to solve them.

Instruction Scheduling

To produce code that executes quickly, the code generator may need to reorder operations to reflect the target machine's specific performance constraints. The execution time of the different operations can vary. Memory access operations can take tens or hundreds of cycles, while some arithmetic operations, particularly division, take several cycles. The impact of these longer latency operations on the performance of compiled code can be dramatic.

Assume, for the moment, that a `loadAI` or `storeAI` operation requires three cycles, a `mult` requires two cycles, and all other operations require one cycle. The following table shows how the previous code fragment performs under these assumptions. The **Start** column shows the cycle in which each operation begins execution and the **End** column shows the cycle in which it completes.

Start	End			
1	3	loadAI	r_{arp}, @a ⇒ r_1	// load 'a'
4	4	add	r_1, r_1 ⇒ r_1	// $r_1 \leftarrow a \times 2$
5	7	loadAI	r_{arp}, @b ⇒ r_2	// load 'b'
8	9	mult	r_1, r_2 ⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$
10	12	loadAI	r_{arp}, @c ⇒ r_2	// load 'c'
13	14	mult	r_1, r_2 ⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$
15	17	loadAI	r_{arp}, @d ⇒ r_2	// load 'd'
18	19	mult	r_1, r_2 ⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
20	22	storeAI	r_1 ⇒ r_{arp}, @a	// write r_a back to 'a'

This nine-operation sequence takes 22 cycles to execute. Minimizing register use did not lead to rapid execution.

Many processors have a property by which they can initiate new operations while a long-latency operation executes. As long as the results of a long-latency operation are not referenced until the operation completes, execution proceeds normally. If, however, some intervening operation tries to read the result of the long-latency operation prematurely, the processor delays the operation that needs the value until the long-latency operation completes. An operation cannot begin to execute until its operands are ready, and its results are not ready until the operation terminates.

The instruction scheduler reorders the operations in the code. It attempts to minimize the number of cycles wasted waiting for operands. Of course, the scheduler must ensure that the new sequence produces the same result as the original. In many cases, the scheduler can drastically improve the performance of "naive" code. For our example, a good scheduler might produce the following sequence:

Start	End			
1	3	loadAI	r_{arp}, @a ⇒ r_1	// load 'a'
2	4	loadAI	r_{arp}, @b ⇒ r_2	// load 'b'
3	5	loadAI	r_{arp}, @c ⇒ r_3	// load 'c'
4	4	add	r_1, r_1 ⇒ r_1	// $r_1 \leftarrow a \times 2$
5	6	mult	r_1, r_2 ⇒ r_1	// $r_1 \leftarrow (a \times 2) \times b$
6	8	loadAI	r_{arp}, @d ⇒ r_2	// load 'd'
7	8	mult	r_1, r_3 ⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b) \times c$
9	10	mult	r_1, r_2 ⇒ r_1	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
11	13	storeAI	r_1 ⇒ r_{arp}, @a	// write r_a back to 'a'

COMPILER CONSTRUCTION IS ENGINEERING

A typical compiler has a series of passes that, together, translate code from some source language into some target language. Along the way, the compiler uses dozens of algorithms and data structures. The compiler writer must select, for each step in the process, an appropriate solution.

A successful compiler executes an unimaginable number of times. Consider the total number of times that GCC compiler has run. Over GCC's lifetime, even small inefficiencies add up to a significant amount of time. The savings from good design and implementation accumulate over time. Thus, the compiler writer must pay attention to compile time costs, such as the asymptotic complexity of algorithms, the actual running time of the implementation, and the space used by data structures. The compiler writer should have in mind a budget for how much time the compiler will spend on its various tasks.

For example, scanning and parsing are two problems for which efficient algorithms abound. Scanners recognize and classify words in time proportional to the number of characters in the input program. For a typical programming language, a parser can build derivations in time proportional to the length of the derivation. (The restricted structure of programming languages makes efficient parsing possible.) Because efficient and effective techniques exist for scanning and parsing, the compiler writer should expect to spend just a small fraction of compile time on these tasks.

By contrast, optimization and code generation contain several problems that require more time. Many of the algorithms that we will examine for program analysis and optimization will have complexities greater than $\mathbf{O}(n)$. Thus, algorithm choice in the optimizer and code generator has a larger impact on compile time than it does in the compiler's front end. The compiler writer may need to trade precision of analysis and effectiveness of optimization against increases in compile time. He or she should make such decisions consciously and carefully.

This version of the code requires just 13 cycles to execute. The code uses one more register than the minimal number. It starts an operation in every cycle except 8, 10, and 12. Other equivalent schedules are possible, as are equal-length schedules that use more registers. Chapter 12 presents several scheduling techniques that are in widespread use.

Interactions Among Code-Generation Components

Most of the truly hard problems that occur in compilation arise during code generation. To make matters more complex, these problems interact. For

example, instruction scheduling moves `load` operations away from the arithmetic operations that depend on them. This can increase the period over which the values are needed and, correspondingly, increase the number of registers needed during that period. Similarly, the assignment of particular values to specific registers can constrain instruction scheduling by creating a "false" dependence between two operations. (The second operation cannot be scheduled until the first completes, even though the values in the common register are independent. Renaming the values can eliminate this false dependence, at the cost of using more registers.)

1.4 SUMMARY AND PERSPECTIVE

Compiler construction is a complex task. A good compiler combines ideas from formal language theory, from the study of algorithms, from artificial intelligence, from systems design, from computer architecture, and from the theory of programming languages and applies them to the problem of translating a program. A compiler brings together greedy algorithms, heuristic techniques, graph algorithms, dynamic programming, DFAs and NFAs, fixed-point algorithms, synchronization and locality, allocation and naming, and pipeline management. Many of the problems that confront the compiler are too hard for it to solve optimally; thus, compilers use approximations, heuristics, and rules of thumb. This produces complex interactions that can lead to surprising results—both good and bad.

To place this activity in an orderly framework, most compilers are organized into three major phases: a front end, an optimizer, and a back end. Each phase has a different set of problems to tackle, and the approaches used to solve those problems differ, too. The front end focuses on translating source code into some IR. Front ends rely on results from formal language theory and type theory, with a healthy dose of algorithms and data structures. The middle section, or optimizer, translates one IR program into another, with the goal of producing an IR program that executes efficiently. Optimizers analyze programs to derive knowledge about their runtime behavior and then use that knowledge to transform the code and improve its behavior. The back end maps an IR program to the instruction set of a specific processor. A back end approximates the answers to hard problems in allocation and scheduling, and the quality of its approximation has a direct impact on the speed and size of the compiled code.

This book explores each of these phases. Chapters 2 through 4 deal with the algorithms used in a compiler's front end. Chapters 5 through 7 describe background material for the discussion of optimization and code generation. Chapter 8 provides an introduction to code optimization. Chapters 9 and 10

provide more detailed treatment of analysis and optimization for the interested reader. Finally, Chapters 11 through 13 cover the techniques used by back ends for instruction selection, scheduling, and register allocation.

CHAPTER NOTES

The first compilers appeared in the 1950s. These early systems showed surprising sophistication. The original FORTRAN compiler was a multipass system that included a distinct scanner, parser, and register allocator, along with some optimizations [26, 27]. The Alpha system, built by Ershov and his colleagues, performed local optimization [139] and used graph coloring to reduce the amount of memory needed for data items [140, 141].

Knuth provides some interesting recollections of compiler construction in the early 1960s [227]. Randell and Russell describe early implementation efforts for Algol 60 [293]. Allen describes the history of compiler development inside IBM with an emphasis on the interplay of theory and practice [14].

Many influential compilers were built in the 1960s and 1970s. These include the classic optimizing compiler FORTRAN H [252, 307], the Bliss-11 and Bliss-32 compilers [72, 356], and the portable BCPL compiler [300]. These compilers produced high-quality code for a variety of CISC machines. Compilers for students, on the other hand, focused on rapid compilation, good diagnostic messages, and error correction [97, 146].

The advent of RISC architecture in the 1980s led to another generation of compilers; these focused on strong optimization and code generation [24, 81, 89, 204]. These compilers featured full-blown optimizers structured as shown in Figure 1.1. Modern RISC compilers still follow this model.

During the 1990s, compiler-construction research focused on reacting to the rapid changes taking place in microprocessor architecture. The decade began with Intel's *i*860 processor challenging compiler writers to manage pipelines and memory latencies directly. At its end, compilers confronted challenges that ranged from multiple functional units to long memory latencies to parallel code generation. The structure and organization of 1980s RISC compilers proved flexible enough for these new challenges, so researchers built new passes to insert into the optimizers and code generators of their compilers.

While Java systems use a mix of compilation and interpretation [63, 279], Java is not the first language to employ such a mix. Lisp systems have long included both native-code compilers and virtual-machine implementation

schemes [266, 324]. The Smalltalk-80 system used a bytecode distribution and a virtual machine [233]; several implementations added just-in-time compilers [126].

EXERCISES

1. Consider a simple Web browser that takes as input a textual string in HTML format and displays the specified graphics on the screen. Is the display process one of compilation or interpretation?

2. In designing a compiler, you will face many tradeoffs. What are the five qualities that you, as a user, consider most important in a compiler that you purchase? Does that list change when you are the compiler writer? What does your list tell you about a compiler that you would implement?

3. Compilers are used in many different circumstances. What differences might you expect in compilers designed for the following applications?
   a. A just-in-time compiler used to translate user interface code downloaded over a network
   b. A compiler that targets the embedded processor used in a cellular telephone
   c. A compiler used in an introductory programming course at a high school
   d. A compiler used to build wind-tunnel simulations that run on a massively parallel processor (where all processors are identical)
   e. A compiler that targets numerically intensive programs to a large number of diverse machines