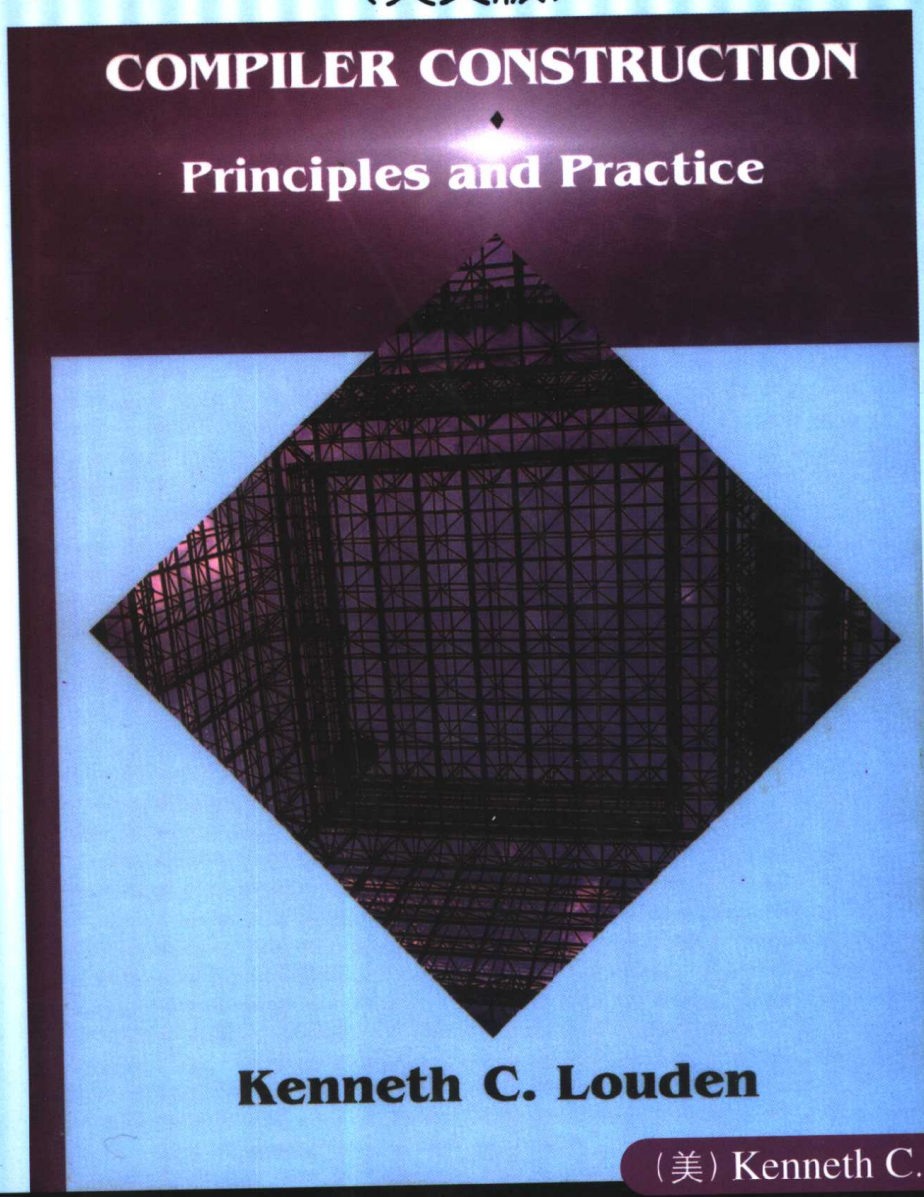
经 典 原 版 书 库

编译原理与实践

（英文版）



（美）Kenneth C. Louden 著

机械工业出版社
China Machine Press

中信出版社
CITIC PUBLISHING HOUSE

THOMSON

经 典 原 版 书 库

编译原理与实践

（英文版）

Compiler Construction Principles and Practice

（美） Kenneth C. Louden 著

机械工业出版社 China Machine Press

中信出版社 CITIC PUBLISHING HOUSE

本书版权登记号：图字：01-2001-5320

图书在版编目（CIP）数据

编译原理与实践/（美）楼登（Louden, K. L.）著. - 北京：机械工业出版社，2002.8
（经典原版书库）
书名原文：Compiler Construction Principles and Practice
ISBN 7-111-10842-6

Ⅰ. 编… Ⅱ. 楼… Ⅲ. 编译程序 - 程序设计 - 英文 Ⅳ. TP314

中国版本图书馆CIP数据核字（2002）第063184号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码100037）
责任编辑：华章
北京忠信诚印刷厂印刷·新华书店北京发行所发行
2002年8月第1版第1次印刷
787mm × 1092mm 1/16 · 37印张
印数：0 001-3 000册
定价：58.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

Preface

This book is an introduction to the field of compiler construction. It combines a detailed study of the theory underlying the modern approach to compiler design, together with many practical examples, and a complete description, with source code, of a compiler for a small language. It is specifically designed for use in an introductory course on compiler design or compiler construction at the advanced undergraduate level. However, it will also be of use to professionals joining or beginning a compiler writing project, as it aims to give the reader all the necessary tools and practical experience to design and program an actual compiler.

A great many texts already exist for this field. Why another one? Because virtually all current texts confine themselves to the study of only one of the two important aspects of compiler construction. The first variety of text confines itself to a study of the theory and principles of compiler design, with only brief examples of the application of the theory. The second variety of text concentrates on the practical goal of producing an actual compiler, either for a real programming language or a pared-down version of one, with only small forays into the theory underlying the code to explain its origin and behavior. I have found both approaches lacking. To really understand the practical aspects of compiler design, one needs to have a good understanding of the theory, and to really appreciate the theory, one needs to see it in action in a real or near-real practical setting.

This text undertakes to provide the proper balance between theory and practice, and to provide enough actual implementation detail to give a real flavor for the techniques without overwhelming the reader. In this text, I provide a complete compiler for a small language written in C and developed using the different techniques studied in each chapter. In addition, detailed descriptions of coding techniques for additional language examples are given as the associated topics are studied. Finally, each chapter concludes with an extensive set of exercises, which are divided into two sections. The first contains those of the more pencil-and-paper variety involving little programming. The second contains those involving a significant amount of programming.

In writing such a text one must also take into account the different places that a compiler course occupies in different computer science curricula. In some programs, a course on automata theory is a prerequisite; in others, a course on programming languages is a prerequisite; while in yet others no prerequisites (other than data structures) are assumed. This text makes no assumptions about prerequisites beyond the usual data

structures course and a familiarity with the C language, yet is arranged so that a prerequisite such as an automata theory course can be taken into account. Thus, it should be usable in a wide variety of programs.

A final problem in writing a compiler text is that instructors use many different classroom approaches to the practical application of theory. Some prefer to study the techniques using only a series of separate small examples, each targeting a specific concept. Some give an extensive compiler project, but make it more manageable with the use of Lex and Yacc as tools. Others ask their students to write all the code by hand (using, say, recursive descent for parsing) but may lighten the task by giving students the basic data structures and some sample code. This book should lend itself to all of these scenarios.

Overview and Organization

In most cases each chapter is largely independent of the others, without artificially restricting the material in each. Cross-references in the text allow the reader or instructor to fill in any gaps that might arise even if a particular chapter or section is skipped.

Chapter 1 is a survey of the basic structure of a compiler and the techniques studied in later chapters. It also includes a section on porting and bootstrapping.

Chapter 2 studies the theory of finite automata and regular expressions, and then applies this theory to the construction of a scanner both by hand coding and using the scanner generation tool Lex.

Chapter 3 studies the theory of context-free grammars as it pertains to parsing, with particular emphasis on resolving ambiguity. It gives a detailed description of three common notations for such grammars, BNF, EBNF, and syntax diagrams. It also discusses the Chomsky hierarchy and the limits of the power of context-free grammars, and mentions some of the important computation-theoretic results concerning such grammars. A grammar for the sample language of the text is also provided.

Chapter 4 studies top-down parsing algorithms, in particular the methods of recursive-descent and LL(1) parsing. A recursive-descent parser for the sample language is also presented.

Chapter 5 continues the study of parsing algorithms, studying bottom-up parsing in detail, culminating in LALR(1) parsing tables and the use of the Yacc parser generator tool. A Yacc specification for the sample language is provided.

Chapter 6 is a comprehensive account of static semantic analysis, focusing on attribute grammars and syntax tree traversals. It gives extensive coverage to the construction of symbol tables and static type checking, the two primary examples of semantic analysis. A hash table implementation for a symbol table is also given and is used to implement a semantic analyzer for the sample language.

Chapter 7 discusses the common forms of runtime environments, from the fully static environment of Fortran through the many varieties of stack-based environments to the fully dynamic environments of Lisp-like languages. It also provides an implementation for a heap of dynamically allocated storage.

Chapter 8 discusses code generation both for intermediate code such as three-address code and P-code and for executable object code for a simple von Neumann

architecture, for which a simulator is given. A complete code generator for the sample language is given. The chapter concludes with an introduction to code optimization techniques.

Three appendices augment the text. The first contains a detailed description of a language suitable for a class project, together with a list of partial projects that can be used as assignments. The remaining appendices give line-numbered listings of the source code for the sample compiler and the machine simulator, respectively.

Use as a Text

This text can be used in a one-semester or two-semester introductory compiler course, either with or without the use of Lex and Yacc compiler construction tools. If an automata theory course is a prerequisite, then Sections 2.2., 2.3, and 2.4 in Chapter 2 and Sections 3.2 and 3.6 in Chapter 3 can be skipped or quickly reviewed. In a one-semester course this still makes for an extremely fast-paced course, if scanning, parsing, semantic analysis, and code generation are all to be covered.

One reasonable alternative is, after an overview of scanning, to simply provide a scanner and move quickly to parsing. (Even with standard techniques and the use of C, input routines can be subtly different for different operating systems and platforms.) Another alternative is to use Lex and Yacc to automate the construction of a scanner and a parser (I do find, however, that in doing this there is a risk that, in a first course, students may fail to understand the actual algorithms being used). If an instructor wishes to use only Lex and Yacc, then further material may be skipped: all sections of Chapter 4 except 4.4, and Section 2.5 of Chapter 2.

If an instructor wishes to concentrate on hand coding, then the sections on Lex and Yacc may be skipped (2.6, 5.5, 5.6, and 5.7). Indeed, it would be possible to skip all of Chapter 5 if bottom-up parsing is ignored.

Similar shortcuts may be taken with the later chapters, if necessary, in either a tools-based course or a hand-coding style course. For instance, not all the different styles of attribute analysis need to be studied (Section 6.2). Also, it is not essential to study in detail all the different runtime environments cataloged in Chapter 7. If the students are to take a further course that will cover code generation in detail, then Chapter 8 may be skipped.

In a two-quarter or two-semester course it should be possible to cover the entire book.

Internet Availability of Resources

All the code in Appendices B and C is available on the Web at locations pointed to from my home page (http://www.mathcs.sjsu.edu/faculty/louden/). Additional resources, such as errata lists and solutions to some of the exercises, may also be available from me. Please check my home page or contact me by e-mail at louden@cs.sjsu.edu.

Acknowledgments

My interest in compilers began in 1984 with a summer course taught by Alan Demers. His insight and approach to the field have significantly influenced my own views.

Indeed, the basic organization of the sample compiler in this text was suggested by that course, and the machine simulator of Appendix C is a descendant of the one he provided.

More directly, I would like to thank my colleagues Bill Giles and Sam Khuri at San Jose State for encouraging me in this project, reading and commenting on most of the text, and for using preliminary drafts in their classes. I would also like to thank the students at San Jose State University in both my own and other classes who provided useful input. Further, I would like to thank Mary T. Stone of PWS for gathering a great deal of information on compiler tools and for coordinating the very useful review process.

The following reviewers contributed many excellent suggestions, for which I am grateful:

Jeff Jenness
Arkansas State University

Joe Lambert
Penn State University

Joan Lukas
University of Masschusetts, Boston

Jerry Potter
Kent State University

Samuel A. Rebelsky
Dartmouth College

Of course I alone am responsible for any shortcomings of the text. I have tried to make this book as error-free as possible. Undoubtedly errors remain, and I would be happy to hear from any readers willing to point them out to me.

Finally, I would like to thank my wife Margreth for her understanding, patience, and support, and our son Andrew for encouraging me to finish this book.

K.C.L.

Contents

Chapter 1

Introduction

Compilers are computer programs that translate one language to another. A compiler takes as its input a program written in its **source language** and produces an equivalent program written in its **target language**. Usually, the source language is a **high-level language**, such as C or C++, and the target language is **object code** (sometimes also called **machine code**) for the target machine, that is, code written in the machine instructions of the computer on which it is to be executed. We can view this process schematically as follows:

A compiler is a fairly complex program that can be anywhere from 10,000 to 1,000,000 lines of code. Writing such a program, or even understanding it, is not a simple task, and most computer scientists and professionals will never write a complete compiler. Nevertheless, compilers are used in almost all forms of computing, and anyone professionally involved with computers should know the basic organization and operation of a compiler. In addition, a frequent task in computer applications is the development of command interpreters and interface programs, which are smaller than compilers but which use the same techniques. A knowledge of these techniques is, therefore, of significant practical use.

It is the purpose of this text not only to provide such basic knowledge but also to give the reader all the necessary tools and practical experience to design and pro-

gram an actual compiler. To accomplish this, it is necessary to study the theoretical techniques, mainly from automata theory, that make compiler construction a manageable task. In covering this theory, we do not assume that the reader has previous knowledge of automata theory. Indeed, the viewpoint taken here is different from that in a standard automata theory text, in that it is aimed specifically at the compilation process. Nevertheless, a reader who has studied automata theory will find the theoretical material more familiar and will be able to proceed more quickly through those sections. In particular, Sections 2.2, 2.3, 2.4, and 3.2 may be skipped or skimmed by a reader with a good background in automata theory. In any case, the reader should be familiar with basic data structures and discrete mathematics. Some knowledge of machine architecture and assembly language is also essential, particularly for the chapter on code generation.

The study of the practical coding techniques themselves requires careful planning, since even with a good theoretical foundation the details of the code can be complex and overwhelming. This text contains a series of simple examples of programming language constructs that are used to elaborate the discussion of the techniques. The language we use for this discussion is called TINY. We also provide (in Appendix A) a more extensive example, consisting of a small but sufficiently complex subset of C, which we call C-Minus, which is suitable for a class project. In addition there are numerous exercises; these include simple paper-and-pencil exercises, extensions of code in the text, and more involved coding exercises.

In general, there is significant interaction between the structure of a compiler and the design of the programming language being compiled. In this text we will only incidentally study language design issues. Other texts are available that more fully treat programming language concepts and design issues. (See the Notes and References section at the end of this chapter.)

We begin with a brief look at the history and the raison d'être of compilers, together with a description of programs related to compilers. Then, we examine the structure of a compiler and the various translation processes and associated data structures and tour this structure using a simple concrete example. Finally, we give an overview of other issues of compiler structure, including bootstrapping and porting, concluding with a description of the principal language examples used in the remainder of the book.

1.1 WHY COMPILERS? A BRIEF HISTORY

With the advent of the stored-program computer pioneered by John von Neumann in the late 1940s, it became necessary to write sequences of codes, or programs, that would cause these computers to perform the desired computations. Initially, these programs were written in **machine language**—numeric codes that represented the actual machine operations to be performed. For example,

```
C7 06 0000 0002
```

represents the instruction to move the number 2 to the location 0000 (in hexadecimal) on the Intel 8x86 processors used in IBM PCs. Of course, writing such codes is extremely time consuming and tedious, and this form of coding was soon replaced by

assembly language, in which instructions and memory locations are given symbolic forms. For example, the assembly language instruction

```
MOV X , 2
```

is equivalent to the previous machine instruction (assuming the symbolic memory location X is 0000). An **assembler** translates the symbolic codes and memory locations of assembly language into the corresponding numeric codes of machine language.

Assembly language greatly improved the speed and accuracy with which programs could be written, and it is still in use today, especially when extreme speed or conciseness of code is needed. However, assembly language has a number of defects: it is still not easy to write and it is difficult to read and understand. Moreover, assembly language is extremely dependent on the particular machine for which it was written, so code written for one computer must be completely rewritten for another machine. Clearly, the next major step in programming technology was to write the operations of a program in a concise form more nearly resembling mathematical notation or natural language, in a way that was independent of any one particular machine and yet capable of itself being translated by a program into executable code. For example, the previous assembly language code can be written in a concise, machine-independent form as

```
X = 2
```

At first, it was feared that this might not be possible, or if it was, then the object code would be so inefficient as to be useless.

The development of the FORTRAN language and its compiler by a team at IBM led by John Backus between 1954 and 1957 showed that both these fears were unfounded. Nevertheless, the success of this project came about only with a great deal of effort, since most of the processes involved in translating programming languages were not well understood at the time.

At about the same time that the first compiler was under development, Noam Chomsky began his study of the structure of natural language. His findings eventually made the construction of compilers considerably easier and even capable of partial automation. Chomsky's study led to the classification of languages according to the complexity of their **grammars** (the rules specifying their structure) and the power of the algorithms needed to recognize them. The **Chomsky hierarchy**, as it is now called, consists of four levels of grammars, called the type 0, type 1, type 2, and type 3 grammars, each of which is a specialization of its predecessor. The type 2, or **context-free, grammars** proved to be the most useful for programming languages, and today they are the standard way to represent the structure of programming languages. The study of the **parsing problem** (the determination of efficient algorithms for the recognition of context-free languages) was pursued in the 1960s and 1970s and led to a fairly complete solution of this problem, which today has become a standard part of compiler theory. Context-free languages and parsing algorithms are studied in Chapters 3, 4, and 5.

Closely related to context-free grammars are **finite automata** and **regular expressions**, which correspond to Chomsky's type 3 grammars. Begun at about the same time as Chomsky's work, their study led to symbolic methods for expressing the structure of the words, or tokens, of a programming language. Chapter 2 discusses finite automata and regular expressions.

Much more complex has been the development of methods for generating efficient object code, which began with the first compilers and continues to this day. These techniques are usually misnamed **optimization techniques**, but they really should be called **code improvement techniques**, since they almost never result in truly optimal object code but only improve its efficiency. Chapter 8 describes the basics of these techniques.

As the parsing problem became well understood, a great deal of work was devoted to developing programs that would automate this part of compiler development. These programs were originally called compiler-compilers, but are more aptly referred to as **parser generators**, since they automate only one part of the compilation process. The best-known of these programs is Yacc (yet another compiler-compiler) written by Steve Johnson in 1975 for the Unix system. Yacc is studied in Chapter 5. Similarly, the study of finite automata led to the development of another tool called a **scanner generator**, of which Lex (developed for the Unix system by Mike Lesk about the same time as Yacc) is the best known. Lex is studied in Chapter 2.

During the late 1970s and early 1980s, a number of projects focused on automating the generation of other parts of a compiler, including code generation. These attempts have been less successful, possibly because of the complex nature of the operations and our less than perfect understanding of them. We do not study them in detail in this text.

More recent advances in compiler design have included the following. First, compilers have included the application of more sophisticated algorithms for inferring and/or simplifying the information contained in a program, and these have gone hand in hand with the development of more sophisticated programming languages that allow this kind of analysis. Typical of these is the unification algorithm of Hindley-Milner type checking, used in the compilation of functional languages. Second, compilers have become more and more a part of a window-based **interactive development environment**, or IDE, that includes editors, linkers, debuggers, and project managers. So far there has been little standardization of such IDEs, but the development of standard windowing environments is leading in that direction. The study of such topics is beyond the scope of this text (but see the next section for a brief description of some of the components of an IDE). For pointers to the literature, see the Notes and References section at the end of the chapter. Despite the amount of research activity in recent years, however, the basics of compiler design have not changed much in the last 20 years, and they have increasingly become a part of the standard core of the computer science curriculum.

1.2 PROGRAMS RELATED TO COMPILERS

In this section, we briefly describe other programs that are related to or used together with compilers and that often come together with compilers in a complete language development environment. (We have already mentioned some of these.)

INTERPRETERS

An interpreter is a language translator like a compiler. It differs from a compiler in that it executes the source program immediately rather than generating object code that is executed after translation is complete. In principle, any programming language can be either interpreted or compiled, but an interpreter may be preferred to a compiler depending on the language in use and the situation under which translation occurs. For example, BASIC is a language that is more usually interpreted than

compiled. Similarly, functional languages such as LISP tend to be interpreted. Interpreters are also often used in educational and software development situations, where programs are likely to be translated and retranslated many times. On the other hand, a compiler is to be preferred if speed of execution is a primary consideration, since compiled object code is invariably faster than interpreted source code, sometimes by a factor of 10 or more. Interpreters, however, share many of their operations with compilers, and there can even be translators that are hybrids, lying somewhere between interpreters and compilers. We will discuss interpreters intermittently, but our main focus in this text will be on compilation.

ASSEMBLERS

An assembler is a translator for the assembly language of a particular computer. As we have already noted, assembly language is a symbolic form of the machine language of the computer and is particularly easy to translate. Sometimes, a compiler will generate assembly language as its target language and then rely on an assembler to finish the translation into object code.

LINKERS

Both compilers and assemblers often rely on a program called a linker, which collects code separately compiled or assembled in different object files into a file that is directly executable. In this sense, a distinction can be made between object code—machine code that has not yet been linked—and executable machine code. A linker also connects an object program to the code for standard library functions and to resources supplied by the operating system of the computer, such as memory allocators and input and output devices. It is interesting to note that linkers now perform the task that was originally one of the principal activities of a compiler (hence the use of the word *compile*—to construct by collecting from different sources). We will not study the linking process in this text, since it is extremely dependent on the details of the operating system and processor. We will also not always make a clear distinction between unlinked object code and executable code, since this distinction will not be important for our study of compilation techniques.

LOADERS

Often a compiler, assembler, or linker will produce code that is not yet completely fixed and ready to execute, but whose principal memory references are all made relative to an undetermined starting location that can be anywhere in memory. Such code is said to be **relocatable**, and a loader will resolve all relocatable addresses relative to a given base, or starting, address. The use of a loader makes executable code more flexible, but the loading process often occurs behind the scenes (as part of the operating environment) or in conjunction with linking. Rarely is a loader an actual separate program.

PREPROCESSORS

A preprocessor is a separate program that is called by the compiler before actual translation begins. Such a preprocessor can delete comments, include other files, and perform **macro** substitutions (a macro is a shorthand description of a repeated sequence of text). Preprocessors can be required by the language (as in C) or can be later add-ons that provide additional facilities (such as the Ratfor preprocessor for FORTRAN).

EDITORS

Compilers usually accept source programs written using any editor that will produce a standard file, such as an ASCII file. More recently, compilers have been bundled together with editors and other programs into an interactive development environment, or IDE. In such a case, an editor, while still producing standard files, may be oriented toward the format or structure of the programming language in question. Such editors are called **structure based** and already include some of the operations of a compiler, so that, for example, the programmer may be informed of errors as the program is written rather than when it is compiled. The compiler and its companion programs can also be called from within the editor, so that the programmer can execute the program without leaving the editor.

DEBUGGERS

A debugger is a program that can be used to determine execution errors in a compiled program. It is also often packaged with a compiler in an IDE. Running a program with a debugger differs from straight execution in that the debugger keeps track of most or all of the source code information, such as line numbers and names of variables and procedures. It can also halt execution at prespecified locations called **breakpoints** as well as provide information on what functions have been called and what the current values of variables are. To perform these functions, the debugger must be supplied with appropriate symbolic information by the compiler, and this can sometimes be difficult, especially in a compiler that tries to optimize the object code. Thus, debugging becomes a compiler question, which, however, is beyond the scope of this book.

PROFILERS

A profiler is a program that collects statistics on the behavior of an object program during execution. Typical statistics that may be of interest to the programmer are the number of times each procedure is called and the percentage of execution time spent in each procedure. Such statistics can be extremely useful in helping the programmer to improve the execution speed of the program. Sometimes the compiler will even use the output of the profiler to automatically improve the object code without intervention by the programmer.

PROJECT MANAGERS

Modern software projects are usually so large that they are undertaken by groups of programmers rather than a single programmer. In such cases, it is important that the files being worked on by different people are coordinated, and this is the job of a project manager program. For example, a project manager should coordinate the merging of separate versions of the same file produced by different programmers. It should also maintain a history of changes to each of a group of files, so that coherent versions of a program under development can be maintained (this is something that can also be useful to the one-programmer project). A project manager can be written in a language-independent way, but when it is bundled together with a compiler, it can maintain information on the specific compiler and linker operations needed to build a complete executable program. Two popular project manager programs on Unix systems are **sccs (source code control system)** and **rcs (revision control system)**.

1.3 THE TRANSLATION PROCESS

A compiler consists internally of a number of steps, or **phases**, that perform distinct logical operations. It is helpful to think of these phases as separate pieces within the compiler, and they may indeed be written as separately coded operations although in practice they are often grouped together. The phases of a compiler are shown in Figure 1.1, together with three auxiliary components that interact with some or all of

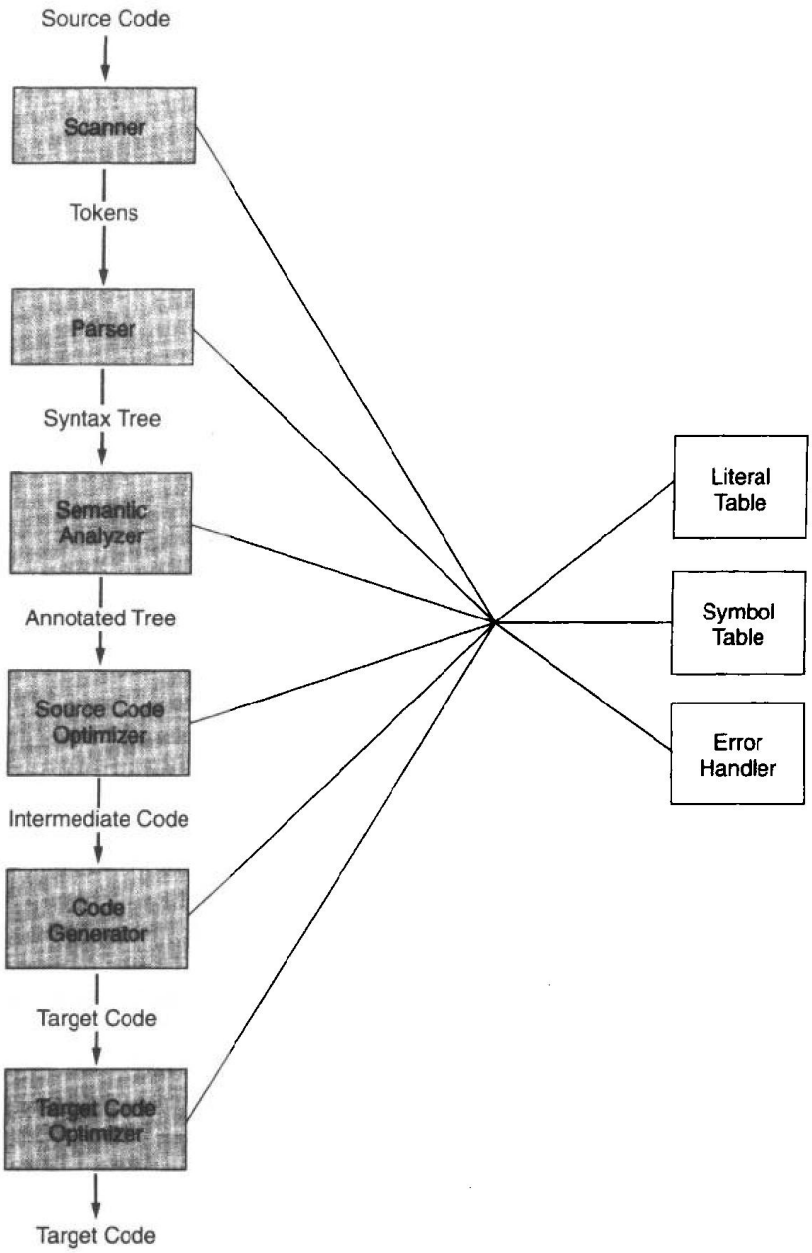


Figure 1.1
The phases of a compiler