Compilers

Principles, Techniques, and Tools

ALFRED V. AHO

RAVI SETHI

JEFFREY D. ULLMAN

Compilers

Principles, Techniques, and Tools

ALFRED V. AHO

AT&T Bell Laboratories
Murray Hill, New Jersey

RAVI SETHI

AT&T Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN

Stanford University
Stanford, California

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California
Don Mills, Ontario • Wokingham, England • Amsterdam • Sydney
Singapore • Tokyo • Mexico City • Bogotá • Santiago • San Juan

Mark S. Dalton/Publisher
James T. DeWolf/Sponsoring Editor

Bette J. Aaronson/Production Supervisor
Hugh Crawford/Manufacturing Supervisor
Karen Guardino/Managing Editor

This book is in the Addison-Wesley series in Computer Science
Michael A. Harrison/Consulting Editor

Library of Congress Cataloging in Publication Data

```
Aho, Alfred V.
   Compilers, principles, techniques, and tools.

   Bibliography: p.
   Includes index.
   1. Compiling (Electronic computers)   I. Sethi,
Ravi.  II. Ullman, Jeffrey D., 1942-    .  III. Title.
QA76.76.C65A37  1985    005.4'53     85-15647
ISBN 0-201-10088-6
```

Reproduced by Addison-Wesley from camera-ready copy supplied by the authors.

ABCDEFGHIJK-HA-898765

Preface

This book is a descendant of *Principles of Compiler Design* by Alfred V. Aho and Jeffrey D. Ullman. Like its ancestor, it is intended as a text for a first course in compiler design. The emphasis is on solving problems universally encountered in designing a language translator, regardless of the source or target machine.

Although few people are likely to build or even maintain a compiler for a major programming language, the reader can profitably apply the ideas and techniques discussed in this book to general software design. For example, the string matching techniques for building lexical analyzers have also been used in text editors, information retrieval systems, and pattern recognition programs. Context-free grammars and syntax-directed definitions have been used to build many little languages such as the typesetting and figure drawing systems that produced this book. The techniques of code optimization have been used in program verifiers and in programs that produce "structured" programs from unstructured ones.

Use of the Book

The major topics in compiler design are covered in depth. The first chapter introduces the basic structure of a compiler and is essential to the rest of the book.

Chapter 2 presents a translator from infix to postfix expressions, built using some of the basic techniques described in this book. Many of the remaining chapters amplify the material in Chapter 2.

Chapter 3 covers lexical analysis, regular expressions, finite-state machines, and scanner-generator tools. The material in this chapter is broadly applicable to text-processing.

Chapter 4 covers the major parsing techniques in depth, ranging from the recursive-descent methods that are suitable for hand implementation to the computationally more intensive LR techniques that have been used in parser generators.

Chapter 5 introduces the principal ideas in syntax-directed translation. This chapter is used in the remainder of the book for both specifying and implementing translations.

Chapter 6 presents the main ideas for performing static semantic checking. Type checking and unification are discussed in detail.

Chapter 7 discusses storage organizations used to support the run-time environment of a program.

Chapter 8 begins with a discussion of intermediate languages and then shows how common programming language constructs can be translated into intermediate code.

Chapter 9 covers target code generation. Included are the basic "on-the-fly" code generation methods, as well as optimal methods for generating code for expressions. Peephole optimization and code-generator generators are also covered.

Chapter 10 is a comprehensive treatment of code optimization. Data-flow analysis methods are covered in detail, as well as the principal methods for global optimization.

Chapter 11 discusses some pragmatic issues that arise in implementing a compiler. Software engineering and testing are particularly important in compiler construction.

Chapter 12 presents case studies of compilers that have been constructed using some of the techniques presented in this book.

Appendix A describes a simple language, a "subset" of Pascal, that can be used as the basis of an implementation project.

The authors have taught both introductory and advanced courses, at the undergraduate and graduate levels, from the material in this book at AT&T Bell Laboratories, Columbia, Princeton, and Stanford.

Chapters 1 and 2 along with the early sections of Chapters 3-9 could form the backbone of an introductory course. Chapter 2 shows the implementation of a simple compiler front end and introduces concepts from Chapters 3-8. This organization allows some flexibility in the selection and order of presentation of material from the remaining chapters. Advanced courses might stress type checking in Chapter 6, run-time storage organization in Chapter 7, pattern-directed code generation in Chapter 9, and code optimization in Chapter 10.

Exercises

As before, we rate exercises with stars. Exercises without stars test understanding of definitions, singly-starred exercises are intended for more advanced courses, and doubly-starred exercises are food for thought.

Acknowledgments

At various stages in the writing of this book, a number of people have given us invaluable comments on the manuscript. In this regard we owe a debt of gratitude to Bill Appelbe, Jon Bentley, Rodney Farrow, Stu Feldman, Charles Fischer, Chris Fraser, Dave Hanson, Robert Henry, Gerard Holzmann, Steve Johnson, Brian Kernighan, Ken Kubota, Dave MacQueen, Dianne Maki, Doug McIlroy, Charles McLaughlin, John Mitchell, Elliott Organick, Rob Pike, Kari-Jouko Räihä, Dennis Ritchie, Bjarne Stroustrup, Tom Szymanski, Peter Weinberger, and Reinhard Wilhelm.

This book was phototypeset by the authors using the excellent software available on the UNIX system. The typesetting command read

```
pic files | tbl | eqn | troff -ms
```

pic is Brian Kernighan's language for typesetting figures; we owe Brian a special debt of gratitude for accommodating our special and extensive figure-drawing needs so cheerfully. tbl is Mike Lesk's language for laying out tables. eqn is Brian Kernighan and Lorinda Cherry's language for typesetting mathematics. troff is Joe Ossana's program for formatting text for a phototypesetter, which in our case was a Mergenthaler Linotron 202/N. The ms package of troff macros was written by Mike Lesk. In addition, we managed the text using make due to Stu Feldman. Cross references within the text were maintained using awk created by Al Aho, Brian Kernighan, and Peter Weinberger, and sed created by Lee McMahon.

The authors would particularly like to acknowledge Patricia Solomon for helping prepare the manuscript for photocomposition. Her cheerfulness and expert typing were greatly appreciated. J. D. Ullman was supported by an Einstein Fellowship of the Israeli Academy of Arts and Sciences during part of the time in which this book was written. Finally, the authors would like to thank AT&T Bell Laboratories for its support during the preparation of the manuscript.

A. V. A.
R. S.
J. D. U.

Contents

CHAPTER 1

Introduction to Compiling

The principles and techniques of compiler writing are so pervasive that the ideas found in this book will be used many times in the career of a computer scientist. Compiler writing spans programming languages, machine architecture, language theory, algorithms, and software engineering. Fortunately, a few basic compiler-writing techniques can be used to construct translators for a wide variety of languages and machines. In this chapter, we introduce the subject of compiling by describing the components of a compiler, the environment in which compilers do their job, and some software tools that make it easier to build compilers.

1.1 COMPILERS

Simply stated, a compiler is a program that reads a program written in one language – the *source* language – and translates it into an equivalent program in another language – the *target* language (see Fig. 1.1). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

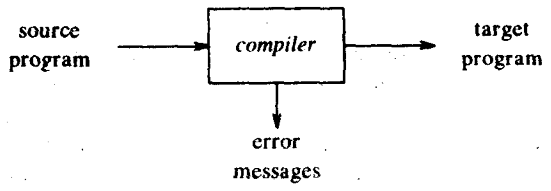

Fig. 1.1. A compiler.

At first glance, the variety of compilers may appear overwhelming. There are thousands of source languages, ranging from traditional programming languages such as Fortran and Pascal to specialized languages that have arisen in virtually every area of computer application. Target languages are equally as varied; a target language may be another programming language, or the machine language of any computer between a microprocessor and a supercomputer. Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same. By understanding these tasks, we can construct compilers for a wide variety of source languages and target machines using the same basic techniques.

Our knowledge about how to organize and write compilers has increased vastly since the first compilers started to appear in the early 1950's. It is difficult to give an exact date for the first compiler because initially a great deal of experimentation and implementation was done independently by several groups. Much of the early work on compiling dealt with the translation of arithmetic formulas into machine code.

Throughout the 1950's, compilers were considered notoriously difficult programs to write. The first Fortran compiler, for example, took 18 staff-years to implement (Backus et al. [1957]). We have since discovered systematic techniques for handling many of the important tasks that occur during compilation. Good implementation languages, programming environments, and software tools have also been developed. With these advances, a substantial compiler can be implemented even as a student project in a one-semester compiler-design course.

The Analysis-Synthesis Model of Compilation

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent parts and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialized techniques. We shall consider analysis informally in Section 1.2 and outline the way target code is synthesized in a standard compiler in Section 1.3.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation. For example, a syntax tree for an assignment statement is shown in Fig. 1.2.

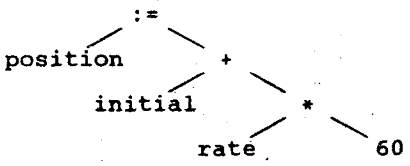

Fig. 1.2. Syntax tree for `position := initial + rate * 60`.

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

1. *Structure editors.* A structure editor takes as input as sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. Thus, the structure editor can perform additional tasks that are useful in the preparation of programs. For example, it can check that the input is correctly formed, can supply keywords automatically (e.g., when the user types `while`, the editor supplies the matching **do** and reminds the user that a conditional must come between them), and can jump from a `begin` or left parenthesis to its matching **end** or right parenthesis. Further, the output of such an editor is often similar to the output of the analysis phase of a compiler.

2. *Pretty printers.* A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font, and statements may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

3. *Static checkers.* A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program. The analysis portion is often similar to that found in optimizing compilers of the type discussed in Chapter 10. For example, a static checker may detect that parts of the source program can never be executed, or that a certain variable might be used before being defined. In addition, it can catch logical errors such as trying to use a real variable as a pointer, employing the type-checking techniques discussed in Chapter 6.

4. *Interpreters.* Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program. For an assignment statement, for example, an interpreter might build a tree like Fig. 1.2, and then carry out the operations at the nodes as it "walks" the tree. At the root it would discover it had an assignment to perform, so it would call a routine to evaluate the expression on the right, and then store the resulting value in the location associated with the identifier `position`. At the right child of the root, the routine would discover it had to compute the sum of two expressions. It would call itself recursively to compute the value of the expression `rate * 60`. It would then add that value to the value of the variable `initial`.

   Interpreters are frequently used to execute command languages, since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler. Similarly, some "very high-level" languages, like APL, are normally interpreted because there are many things about the data, such as the size and shape of arrays, that cannot be deduced at compile time.

Traditionally, we think of a compiler as a program that translates a source language like Fortran into the assembly or machine language of some computer. However, there are seemingly unrelated places where compiler technology is regularly used. The analysis portion in each of the following examples is similar to that of a conventional compiler.

1. *Text formatters.* A text formatter takes input that is a stream of characters, most of which is text to be typeset, but some of which includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts. We mention some of the analysis done by text formatters in the next section.

2. *Silicon compilers.* A silicon compiler has a source language that is similar or identical to a conventional programming language. However, the variables of the language represent, not locations in memory, but, logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language. See Johnson [1983], Ullman [1984], or Trickey [1985] for a discussion of silicon compilation.

3. *Query interpreters.* A query interpreter translates a predicate containing relational and boolean operators into commands to search a database for records satisfying that predicate. (See Ullman [1982] or Date [1986].)

The Context of a Compiler

In addition to a compiler, several other programs may be required to create an executable target program. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

Figure 1.3 shows a typical "compilation." The target program created by the compiler may require further processing before it can be run. The compiler in Fig. 1.3 creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.

We shall consider the components of a compiler in the next two sections; the remaining programs in Fig. 1.3 are discussed in Section 1.4.

1.2 ANALYSIS OF THE SOURCE PROGRAM

In this section, we introduce analysis and illustrate its use in some text-formatting languages. The subject is treated in more detail in Chapters 2-4 and 6. In compiling, analysis consists of three phases:

1. *Linear analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into *tokens* that are sequences of characters having a collective meaning.

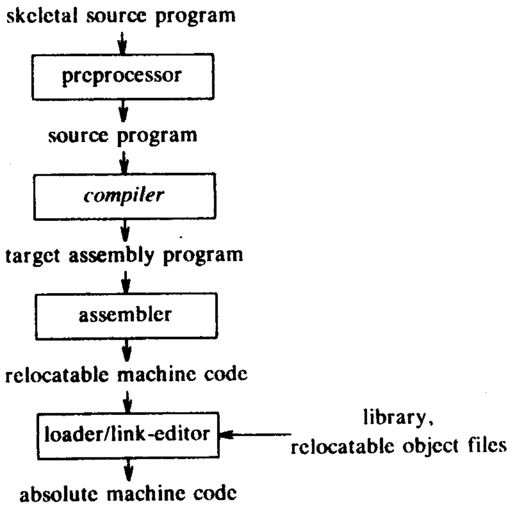

Fig. 1.3. A language-processing system.

2. *Hierarchical analysis*, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.

3. *Semantic analysis*, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

Lexical Analysis

In a compiler, linear analysis is called *lexical analysis* or *scanning*. For example, in lexical analysis the characters in the assignment statement

```
position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`.
2. The assignment symbol `:=`.
3. The identifier `initial`.
4. The plus sign.
5. The identifier `rate`.
6. The multiplication sign.
7. The number `60`.

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Analysis

Hierarchical analysis is called *parsing* or *syntax analysis*. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree such as the one shown in Fig. 1.4.

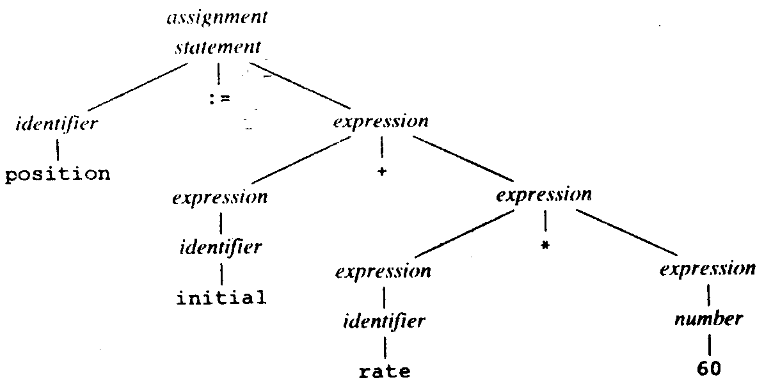

Fig. 1.4. Parse tree for `position := initial + rate * 60`.

In the expression `initial + rate * 60`, the phrase `rate * 60` is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed before addition. Because the expression `initial + rate` is followed by a `*`, it is not grouped into a single phrase by itself in Fig. 1.4.

The hierarchical structure of a program is usually expressed by recursive rules. For example, we might have the following rules as part of the definition of expressions:

1. Any *identifier* is an expression.
2. Any *number* is an expression.
3. If $expression_1$ and $expression_2$ are expressions, then so are

$$expression_1 \;+\; expression_2$$
$$expression_1 \;*\; expression_2$$
$$(\; expression_1 \;)$$

Rules (1) and (2) are (nonrecursive) basis rules, while (3) defines expressions in terms of operators applied to other expressions. Thus, by rule (1), `initial` and `rate` are expressions. By rule (2), `60` is an expression, while by rule (3), we can first infer that `rate * 60` is an expression and finally that `initial + rate * 60` is an expression.

Similarly, many languages define statements recursively by rules such as:

1. If *identifier*$_1$ is an identifier, and *expression*$_2$ is an expression, then

   identifier$_1$:= *expression*$_2$

   is a statement.

2. If *expression*$_1$ is an expression and *statement*$_2$ is a statement, then

   while (*expression*$_1$) **do** *statement*$_2$
   if (*expression*$_1$) **then** *statement*$_2$

   are statements.

The division between lexical and syntactic analysis is somewhat arbitrary. We usually choose a division that simplifies the overall task of analysis. One factor in determining the division is whether a source language construct is inherently recursive or not. Lexical constructs do not require recursion, while syntactic constructs often do. Context-free grammars are a formalization of recursive rules that can be used to guide syntactic analysis. They are introduced in Chapter 2 and studied extensively in Chapter 4.

For example, recursion is not required to recognize identifiers, which are typically strings of letters and digits beginning with a letter. We would normally recognize identifiers by a simple scan of the input stream, waiting until a character that was neither a letter nor a digit was found, and then grouping all the letters and digits found up to that point into an identifier token. The characters so grouped are recorded in a table, called a symbol table, and removed from the input so that processing of the next token can begin.

On the other hand, this kind of linear scan is not powerful enough to analyze expressions or statements. For example, we cannot properly match parentheses in expressions, or **begin** and **end** in statements, without putting some kind of hierarchical or nesting structure on the input.

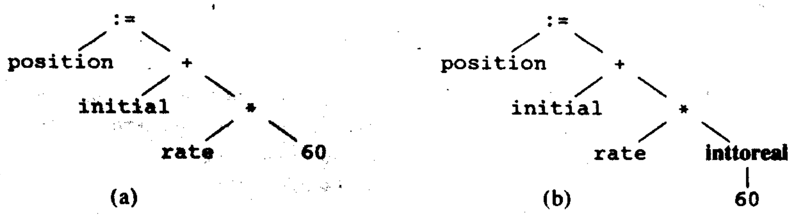

Fig. 1.5. Semantic analysis inserts a conversion from integer to real.

The parse tree in Fig. 1.4 describes the syntactic structure of the input. A more common internal representation of this syntactic structure is given by the syntax tree in Fig. 1.5(a). A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator. The construction of trees such as the one in Fig. 1.5(a) is discussed in Section 5.2.