Principles of Compiler Design

ALFRED V. AHO

JEFFREY D. ULLMAN

Principles of Compiler Design

ALFRED V. AHO

Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN

Princeton University
Princeton, New Jersey

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts · Menlo Park, California
London · Amsterdam · Don Mills, Ontario · Sydney

This book is in the
ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING

Michael A. Harrison
Consulting Editor

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

Copyright © 1977 by Bell Telephone Laboratories, Incorporated. Philippines copyright 1977 by Bell Telephone Laboratories, Incorporated.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada. Library of Congress Catalog Card No. 77-73953.

ISBN 0-201-00022-9
BCDEFGHIJK-HA-7987

Preface

This book is intended as a text for an introductory course in compiler design at the Junior, Senior, or first-year graduate level. The emphasis is on solving the problems universally encountered in designing a compiler, regardless of the source language or the target machine.

Although few people are likely to implement or even maintain a compiler for a major programming language, many people can profitably use a number of the ideas and techniques disoussed in this book in general software design. For example, the finite-state techniques used to build lexical analyzers have also been used in text editors, bibliographic search systems, and pattern recognition programs. Context-free grammars and syntax-directed translation schemes have been used to build text processors of many sorts, such as the mathematical typesetting system used to produce this book. Techniques of code optimization also have applicability to program verifiers and to programs that produce "structured" programs from unstructured ones.

Use of the Book

We have attempted to cover the major topics in compiler design in depth. Advanced material, however, has been put into separate chapters, so that courses on a variety of levels can be taught from this book. A brief synopsis of the chapters and comments on their appropriateness in a basic course is therefore appropriate.

Chapter 1 introduces the basic structure of a compiler, and is essential to all courses.

Chapter 2 covers basic concepts and terminology in programming languages. In courses we have taught, this material was covered in prerequisite courses, but if that is not the case, this material too is essential.

Chapter 3 covers lexical analysis, finite-state techniques, and the scanner generator. It is one of our favorite chapters, but if time does not permit, all but Sections 3.1 and 3.2 could be skipped.

Chapters 4, 5, and 6 cover parsing. The first of these introduces basic notions and is essential. One may then choose either 5 or 6 if both cannot be covered. Chapter 5 discusses the most common kinds of parsers — operator precedence and recursive descent. Chapter 6 covers LR parsing,

which we believe to be the method of choice.

In Chapter 7 we introduce the principal ideas connected with intermediate-code generation. We use the syntax-directed approach and give translation schemes for the most basic programming language constructs — simple assignments and simple control structures. We regard all this material as essential.

Chapter 8 is a continuation of Chapter 7, covering the translation of additional language constructs such as array and structure references. Chapter 8 may be omitted if time presses.

Chapter 9 covers symbol tables. On the assumption that a course in data structures is a prerequisite to a course on compilers, Section 9.2 may be skipped.

Chapter 10 is on run-time organization. We introduce the subject by discussing the run-time implementation of the programming language C, which is easier to implement than other recursive languages such as ALGOL or PL/I. A possible candidate for omission is Section 10.3 on FORTRAN COMMON and EQUIVALENCE statements.

Chapter 11 discusses error recovery, another essential topic.

Chapters 12, 13, and 14 are on the subject of code optimization. The essentials are introduced in Chapter 12, and a first course will probably not go into the material of 13 and 14.

Finally, Chapter 15 covers object code generation. We have presented only the most universally applicable ideas, and most of what we do cover is appropriate for a first course. If forced to cut, however, we would omit Sections 15.5 and 15.6.

This book also contains sufficient material to make up an advanced course on compiler design. For example, at Princeton and Stevens we taught a graduate course to students who had had an elementary compiler course and a course in automata and language theory. There we covered scanner generators from Chapter 3, LR parsers and parser generators from Chapter 6, code optimization from Chapters 12, 13, and 14, and some topics in code generation from Chapter 15.

The Compiler Project

Appendix B contains a modular implementation project in which the student produces a compiler front end (lexical analyzer, parser, bookkeeping routines, and an intermediate code generator). The intermediate code may be interpreted, giving experience in run-time storage management. Unfortunately, we find that the typical one-semester course does not normally provide enough time for adding object code generation to this project, although such a module can easily be attached if time permits.

Also in Appendix B is a description of a simple language, a "subset" of PASCAL, which can be used in the project if desired. We regret that to make the job simple enough for a term project we have had to take out a

此为试读，需要完整PDF请访问：www.ertongbook.com

number of the elegant features of PASCAL, including data type definitions and block structure (although recursion remains). An SLR(1) grammar for the language is given, along with directions for converting it to operator-precedence or LL(1) form if one of those types of parser is desired.

Exercises

We have traditionally rated exercises with stars. Zero-starred exercises are suitable for elementary courses, singly-starred exercises are intended for more advanced courses, and doubly-starred exercises are food for thought.

Acknowledgments

The manuscript at various stages was read by a number of people who gave us valuable comments. In this regard we owe a debt of gratitude to Brenda Baker, David Copp, Bruce Englar, Hania Gajewska, Sue Graham, Matt Hecht, Ellis Horowitz, Steve Johnson, Randy Katz, Ken Kennedy, Brian Kernighan, Doug McIlroy, Marshall McKusick, Arnaldo Moura, Tom Peterson, Dennis Ritchie, Eric Schmidt, Ravi Sethi, Tom Szymanski, Ken Thompson, and Peter Weinberger.

This book was phototypeset by the authors using the excellent software available on the UNIX operating system. We would like to acknowledge the people who made it possible to do so. Dennis Ritchie and Ken Thompson were the creators and principal implementors of UNIX. Joe Ossanna wrote TROFF, the program which formats text for the phototypesetter. Brian Kernighan and Lorinda Cherry produced EQN, the preprocessor which enables mathematical text to be typeset conveniently. Mike Lesk implemented both the MS macro package, which greatly simplifies the specification of page layouts, and TBL, the preprocessor used to prepare the tables in this book.

The authors would particularly like to thank Carmela Scrocca who so expertly typed the manuscript and prepared it for photocomposition. The authors would also like to acknowledge the support services provided by Bell Laboratories during the preparation of the manuscript.

A. V. A.
J. D. U.

Contents

CHAPTER 1

Introduction to Compilers

The purpose of this book is two-fold. We hope to acquaint the reader with the basic constructs of modern programming languages and to show how they can be efficiently implemented in the machine language of a typical computer. We shall also show how tools can be developed and used to help construct certain translator components. These tools not only facilitate the construction of compilers, but they can also be used in a variety of applications not directly related to compiling.

1.1 Compilers and Translators

A *translator* is a program that takes as input a program written in one programming language (the *source language*) and produces as output a program in another language (the *object* or *target language*). If the source language is a high-level language such as FORTRAN, PL/I, or COBOL, and the object language is a low-level language such as an assembly language or machine language, then such a translator is called a *compiler.*

Executing a program written in a high-level programming language is basically a two-step process, as illustrated in Fig. 1.1. The source program must first be *compiled,* that is, translated into the object program. Then the resulting object program is loaded into memory and executed.

Compilers were once considered almost impossible programs to write. The first FORTRAN compiler, for example, took 18 man-years to implement (Backus et al. [1957]). Today, however, compilers can be built with much less effort. In fact, it is not unreasonable to expect a fairly substantial compiler to be implemented as a student project in a one-semester compiler design course. The principal developments of the past twenty years which led to this improvement are:

此为试读,需要完整PDF请访问: www.ertongbook.com

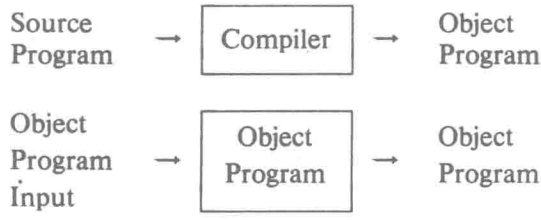

Fig. 1.1. Compilation and execution.

- The understanding of how to organize and modularize the process of compilation,
- The discovery of systematic techniques for handling many of the important tasks that occur during compilation,
- The development of software tools that facilitate the implementation of compilers and compiler components.

These are the developments we shall consider in this book. This chapter provides an overview of the compilation process and introduces the major components of a compiler.

Other Translators

Certain other translators transform a programming language into a simplified language, called *intermediate code,* which can be directly executed using a program called an *interpreter.* We may think of the intermediate code as the machine language of an abstract computer designed to execute the source code. For example, SNOBOL is often interpreted, the intermediate code being a language called Polish postfix notation (see Section 7.4). In some cases, the source language itself can be the intermediate language. For example, most *command languages,* such as JCL, in which one communicates directly with the operating system, are interpreted with no prior translation at all.

Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs. However, the main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

There are several other important types of translators, besides compilers. If the source language is assembly language and the target language is machine language, then the translator is called an *assembler.* The term *preprocessor* is sometimes used for translators that take programs in one

high-level language into equivalent programs in another high-level language. For example, there are many FORTRAN preprocessors that map "structured" versions of FORTRAN into conventional FORTRAN.

1.2 Why do we Need Translators?

The answer to this question is obvious to anyone who has programmed in machine language. With machine language we must communicate directly with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is terribly tedious and fraught with opportunities for mistakes. Perhaps the most serious disadvantage of machine-language coding is that all operations and operands must be specified in a numeric code. Not only is a machine language program cryptic, but it also may be impossible to modify in a convenient manner.

Symbolic Assembly Language

Because of the difficulties with machine language programming, a host of "higher-level" languages have been invented to enable the programmer to code in a way that resembles his own thought processes rather than the elementary steps of the computer. The most immediate step away from machine language is symbolic assembly language. In this language, a programmer uses mnemonic names for both operation codes and data addresses. Thus a programmer could write ADD X, Y in assembly language, instead of something like 0110 001110 010101 in machine language (where 0110 is the hypothetical machine operation code for "add" and 001110 and 010101 are the addresses of X and Y).

A computer, however, cannot execute a program written in assembly language. That program has to be first translated to machine language, which the computer can understand. The program that performs this translation is the assembler.

Macros

Many assembly (and programming) languages provide a "macro" facility whereby a macro statement will translate into a sequence of assembly language statements and perhaps other macro statements before being translated into machine code. Thus, a macro facility is a text replacement capability. There are two aspects to macros: definition and use. To illustrate the utility of macros, consider a situation in which a machine does not have a single machine- or assembly-language statement that adds the contents of one memory address to another, as did our hypothetical assembly

instruction ADD X, Y, above. Instead, suppose the machine has an instruction LOAD, which moves a datum from memory to a register, an instruction ADD, which adds the contents of a memory address to that of a register, and an instruction STORE, which moves data from a register to memory. Using these instructions, we can create, with a *macro definition,* a "two-address add" instruction as follows.

```
MACRO     ADD2    X, Y
          LOAD    Y
          ADD     X
          STORE   Y
ENDMACRO
```

The first statement gives the name ADD2 to the macro and defines its dummy arguments, known as *formal parameters,* X and Y. The next three statements define the macro, that is, they give its translation. We assume that the machine has only one register, so the question of what registers LOAD and STORE refer to needs no elaboration.

Having defined ADD2 in this way, we can then use it as an ordinary assembly language operation code. For example, if the statement ADD2 A, B is encountered somewhere after the definition of ADD2, we have a *macro use.* Here, the macro processor substitutes for ADD2 A, B the three statements which form the definition of ADD2, but with the *actual parameters* A and B replacing the formal parameters X and Y, respectively. That is, ADD2 A, B is translated to

```
LOAD    B
ADD     A
STORE   B
```

High-Level Languages

Symbolic assembly programs are easier to write and understand than machine-language programs primarily because numerical codes for addresses and operators are replaced by more meaningful symbolic codes. Nevertheless, even with macros, there are severe drawbacks to writing in assembly language. The programmer must still know the details of how a specific computer operates. He must also mentally translate complex operations and data structures into sequences of low-level operations which use only the primitive data types that machine language provides. The programmer must also be intimately concerned with how and where data is represented within the machine. Although there are a few situations in

which such detailed knowledge is essential for efficiency, most of the programmer's time is unnecessarily wasted on such intricacies.

To avoid these problems, high-level programming languages were developed. Basically, a high-level programming language allows a programmer to express algorithms in a more natural notation that avoids many of the details of how a specific computer functions. For example, it is much more natural to write the expression A+B than a sequence of assembly language instructions to add A and B. COBOL, FORTRAN, PL/I, ALGOL,† SNOBOL, APL, PASCAL, LISP and C are some of the more common high-level languages, and we assume the reader is familiar with at least one of these languages. References for these languages and others are found in the bibliographic notes of Chapter 2.

A high-level programming language makes the programming task simpler, but it also introduces some problems. The most obvious is that we need a program to translate the high-level language into a language the machine can understand. In a sense, this program, the compiler, is completely analogous to the assembler for an assembly language.

A compiler, however, is a substantially more complex program to write than an assembler. Some compilers even make use of an assembler as an appendage, with the compiler producing assembly code, which is then assembled and loaded before being executed in the resulting machine-language form.

Before discussing compilers in detail, however, we should know the types of constructs typically found in high-level programming languages. The form and meaning of the constructs in a programming language have a strong impact on the overall design of a compiler for that language. Chapter 2 of this book reviews the main concepts concerning programming languages.

1.3 The Structure of a Compiler

A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of subprocesses called *phases,* as shown in Fig. 1.2. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

† Throughout this book, ALGOL refers to ALGOL 60 rather than ALGOL 68.

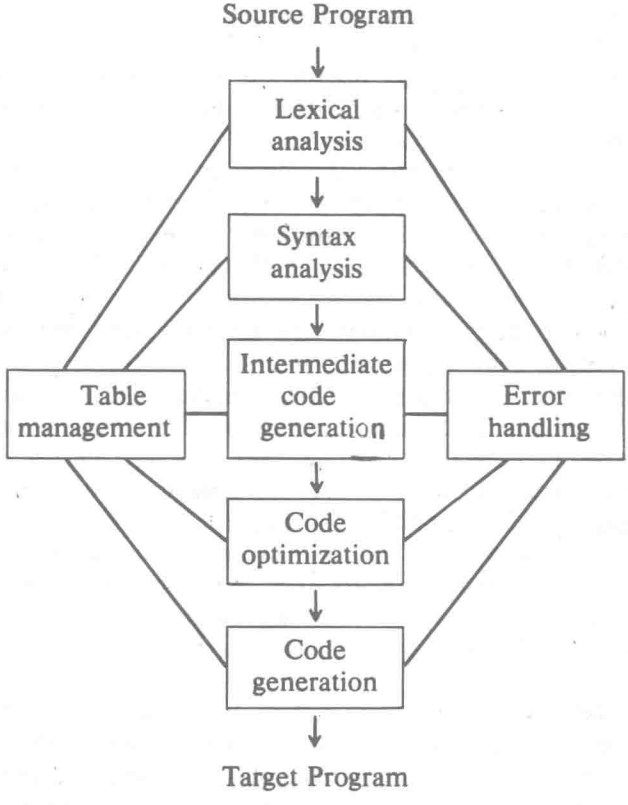

Fig. 1.2. Phases of a compiler.

The first phase, called the *lexical analyzer,* or *scanner,* separates characters of the source language into groups that logically belong together; these groups are called *tokens.* The usual tokens are keywords, such as DO or IF, identifiers, such as X or NUM, operator symbols such as <= or +, and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the *syntax analyzer,* or *parser.* The tokens in this stream can be represented by codes which we may regard as integers. Thus, DO might be represented by 1, + by 2, and "identifier" by 3. In the case of a token like "identifier," a second quantity, telling which of those identifiers used by the program is represented by this instance of token "identifier," is passed along with the integer code for "identifier."

The syntax analyzer groups tokens together into syntactic structures. For example, the three tokens representing A+B might be grouped into a

syntactic structure called an *expression*. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

The *intermediate code generator* uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and a small number of operands. These instructions can be viewed as simple macros like the macro ADD2 discussed in Section 1.2. The primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space.

The final phase, *code generation,* produces the object code by deciding on the memory locations for data, selecting code to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically.

The *table-management,* or *bookkeeping,* portion of the compiler keeps track of the names used by the program and records essential information about each, such as its type (integer, real, etc.). The data structure used to record this information is called a *symbol table.*

The *error handler* is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on flawed programs, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routines interact with all phases of the compiler.

Passes

In an implementation of a compiler, portions of one or more phases are combined into a module called a *pass.* A pass reads the source program or the output of the previous pass, makes the transformations specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

此为试读,需要完整PDF请访问: www.ertongbook.com