国外著名高等院校
信息科学与技术优秀教材

编译原理 技术与工具

Compilers:
Principles, Techniques, and Tools

Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

英文版

人民邮电出版社
www.pptph.com.cn

ADDISON WESLEY
Pearson Education
出 版 集 团

国外著名高等院校信息科学与技术优秀教材

编译原理 技术与工具

(英文版)

Compilers: Principles, Techniques, and Tools

Alfred V. Aho

Ravi Sethi

Jeffrey D. Ullman

人民邮电出版社 Pearson Education 出版集团

国外著名高等院校信息科学与技术优秀教材

编译原理 技术与工具

(英文版)

◆ 著　　Alfred V. Aho　Ravi Sethi　Jeffrey D. Ullman

责任编辑　李　际

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号

邮编　100061　　电子函件　315@ pptph.com.cn

网址　http://www.pptph.com.cn

读者热线　010-67129212　010-67129211(传真)

北京汉魂图文设计有限公司制作

北京朝阳展望印刷厂印刷

新华书店总店北京发行所经销

◆ 开本:720 × 980　1/16

印张:50.75

字数:1 009 千字　　2002 年 2 月第 1 版

印数:1 - 4 000 册　　2002 年 2 月北京第 1 次印刷

著作权合同登记　图字:01 - 2001 - 4831 号

ISBN 7-115-09916-2/TP·2649

定价:63.00 元

图书在版编目（CIP）数据

编译原理技术与工具/（美）阿霍（Aho,A.V.），（美）塞西（Sethi，R.），（美）厄尔曼（Ullman，J.D.）著. —北京：人民邮电出版社，2002.2
国外著名高等院校信息科学与技术优秀教材
ISBN 7-115-09916-2

Ⅰ.编… Ⅱ.①阿… ②塞… ③厄… Ⅲ.编译程序—程序设计—高等学校—教材—英文 Ⅳ.TP314

中国版本图书馆 CIP 数据核字（2001）第 089601 号

内 容 提 要

作为编译器设计的教程，本书重点主要放在解决在设计语言翻译器过程中所普遍面对的一些问题上而并不考虑源语言或者目标机器。本书共 12 章：第一章介绍了编译器的基本结构；第二章给出了一个将前缀表达式转换成后缀表达式的编译器，主要使用本书的一些基本技巧来构建；第三章阐述了词法分析、正则表达式、有限自动机和扫描生成器工具，这章中的技术广泛应用于文本处理；第四章详细阐述了主要的分析技术，从适合手工实现的递归下降算法到在分析生成器中使用的 LR 算法；第五章介绍了语法制导翻译中的主要思想，本书的其它部分都用本章来说明和实现翻译；第六章提出了完成静态语义检查的主要思想，并对类型检查和类型的统一进行了详细的讨论；第七章讨论了支持应用程序运行时环境的存储组织；第八章从中间语言的讨论开始，说明了编程语言结构如何被翻译成中间代码；第九章阐述了目标代码的生成，包含基本的 on-the-fly 代码生成方法、为表达式生成代码的优化方法、Peephole 优化和代码生成器；第十章是代码优化的总述。除了关于数据流分析方法的详细说明，还有关于如何进行全局优化的基本方法；第十一章讨论了在编译器实现过程中可能会产生的一些实际问题；第十二章提出一些使用本书中的技术构建的一些编译器的学习用例。

本书可作为高校计算机专业本科和研究生编译原理的教科书，也可供从事计算机软件开发的人员参考。

出版说明

2001年，教育部印发了《关于“十五”期间普通高等教育教材建设与改革的意见》。该文件明确指出，“九五”期间原国家教委在“抓好重点教材，全面提高质量”方针指导下，调动了各方面的积极性，产生了一大批具有改革特色的新教材。然而随着科学技术的飞速发展，目前高校教材建设工作仍滞后于教学改革的实践，一些教材内容陈旧，不能满足按新的专业目录修订的教学计划和课程设置的需要。为此该文件明确强调，要加强国外教材的引进工作。当前，引进的重点是信息科学与技术和生物科学与技术两大学科的教材。要根据专业（课程）建设的需要，通过深入调查、专家论证，引进国外优秀教材。要注意引进教材的系统配套，加强对引进教材的宣传，促进引进教材的使用和推广。

邓小平同志早在1977年就明确指出：“要引进外国教材，吸收外国教材中有益的东西。”随着我国加入WTO，信息产业的国际竞争将日趋激烈，我们必须尽快培养出大批具有国际竞争能力的高水平信息技术人才。教材是一个很关键的问题，国外的一些优秀教材不但内容新，而且还提供了很多新的研究方法和思考方式。引进国外原版教材，可以促进我国教学水平的提高，提高学生的英语水平和学习能力，保证我们培养出的学生具有国际水准。

为了贯彻中央“科教兴国”的方针，配合国内高等教育教材建设的需要，人民邮电出版社约请有关专家反复论证，与国外知名的教材出版公司合作，陆续引进一些信息科学与技术优秀教材。第一批教材针对计算机专业的主干核心课程，是国外著名高等院校所采用的教材，教材的作者都是在相关领域享有盛名的专家教授。这些教材内容新，反映了计算机科学技术的最新发展，对全面提高我国信息科学与技术的教学水平必将起到巨大的推动作用。

出版国外著名高等院校信息科学与技术优秀教材的工作将是一个长期的、坚持不懈的过程，我社网站（www.pptph.com.cn）上介绍了我们首批陆续推出的图书的详细情况，后续教材的引进和出版情况我们会及时在网上发布，敬请关注。希望广大教师和学生将使用中的意见和建议及时反馈给我们，我们将根据您的反馈不断改进我们的工作，推出更多更好的引进版信息科学与技术教材。

人民邮电出版社

2001年12月

序　言

Alfred V. Aho 和 Jeffrey D. Ullman 是计算机科学领域人员十分熟悉的作者。Alfred V. Aho 负责 AT&T 贝尔实验室计算机原理研究部，Jeffrey D. Ullman 现为美国斯坦福大学计算机科学系的教授。早在 1977 年，他们合写了《Principles of Compiler Design》一书，1986 年 Alfred V. Aho、Ravi Sethi 和 Jeffrey D. Ullman 等三人又合写了《Compilers：Principles, Techniques, and Tools》，对编译器设计原理进行了科学的概括并做了系统的阐述。这两本书将庞大复杂的编译系统用极为简单、清晰的结构介绍给所有从事计算机科学的人员。因此，它们成为国外著名高校编译原理的教科书。这两本书是国外编译原理方面的教材中对我国计算机界影响最大的两本著作。从 20 世纪 80 年代中期至今，我国陆续出版了一些理论性强、质量高的编译原理教材，它们从体系结构和内容上主要得益于这两本书。

本书共 12 章：

第一章介绍了编译器的基本结构，它是本书的其余部分的基础。

第二章给出了一个将前缀表达式转换成后缀表达式的编译器，主要使用本书的一些基本技巧来构建。其余章节的一些部分详细叙述了这个内容。

第三章阐述了词法分析、正则表达式、有限自动机和扫描生成器工具。这章中的技术广泛应用于文本处理。

第四章详细阐述了主要的分析技术，从适合手工实现的递归下降算法到在分析生成器中使用的 LR 算法。

第五章介绍了语法制导翻译中的主要思想。本书的其它部分都用本章来说明和实现翻译。

第六章提出了完成静态语义检查的主要思想。并对类型检查和类型的统一进行了详细的讨论。

第七章讨论了支持应用程序运行时环境的存储组织。

第八章从中间语言的讨论开始，接着说明了编程语言结构如何被翻译成中间代码。

第九章阐述了目标代码的生成。包含基本的 on-the-fly 代码生成方法，还有为表达式生成代码的优化方法。本章对 Peephole 优化和代码生成器也分别进行了讲述。

第十章是代码优化的总述。除了关于数据流分析方法的详细说明，还有关于如何进行全局优化的基本方法。

第十一章讨论了在编译器实现过程中可能会产生的一些实际问题。软件工程

和测试在编译器构建中显得特别重要。

第十二章提出一些使用本书中的技术构建的一些编译器的学习用例。

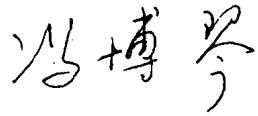
由于它是一本原理性著作，因此本书主要着眼于设计编译器中共性的问题，而不拘泥在某种语言和目标机器上。全书 12 章内容全面地论述了编译器设计中的有关问题，重点是语法制导翻译、类型检查、运行时存储组织、代码生成和代码优化。

较国内几乎所有的编译原理教材，本书有两个明显的特点，其一是增加了对编译开发技术的实际应用。学生在学习编译原理课程时，往往会有这样的困惑："有机会开发编译系统的人很少，学了编译有何用？"师生都不善于把这里学到的思想、技术应用于实际，本书注意到了这一点，给出了许多启示；其二是本书的例子和习题非常丰富：每章约有 30～40 个例子和习题，其中第四章竟有 52 个例子，65 个习题之多！本书还根据习题难易程度划分了层次，便于读者选用。本书中既有偏理论、概念性的，亦有实践性很强的要求用某种算法写编译器一部分的习题。这样丰富的教学资料为教学提供了充分操作的空间。

人民邮电出版社独具慧眼，引进这本很有价值的书。我相信该书在国内的出版，会对计算机科学的编译原理及教学产生积极的推动作用。

本书可作为高校计算机专业本科和研究生编译原理的教科书，也可供从事计算机软件开发的人员参考。

由于本人学识浅薄，时间仓促，未能仔细阅读，因此以上陋见难免会有差错，恳望同行不吝指正。



冯博琴 教授
西安交通大学计算机教学实验中心主任
教育部高校计算机科学与技术教学指导委员会副主任委员

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.

An introductory compiler course might cover material from the following sections of this book:

introduction	Chapter 1 and Sections 2.1-2.5
lexical analysis	2.6, 3.1-3.4
symbol tables	2.7, 7.6
parsing	2.4, 4.1-4.4
syntax-directed translation	2.5, 5.1-5.5
type checking	6.1-6.2
run-time organization	7.1-7.3
intermediate code generation	8.1-8.3
code generation	9.1-9.4
code optimization	10.1-10.2

Information needed for a programming project like the one in Appendix A is introduced in Chapter 2.

A course stressing tools in compiler construction might include the discussion of lexical analyzer generators in Sections 3.5, of parser generators in Sections 4.8 and 4.9, of code-generator generators in Section 9.12, and material on techniques for compiler construction from Chapter 11.

An advanced course might stress the algorithms used in lexical analyzer generators and parser generators discussed in Chapters 3 and 4, the material

on type equivalence, overloading, polymorphism, and unification in Chapter 6, the material on run-time storage organization in Chapter 7, the pattern-directed code generation methods discussed in Chapter 9, and material on code optimization from 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, Nelson Beebe, Jon Bentley, Lois Bogess, Rodney Farrow, Stu Feldman, Charles Fischer, Chris Fraser, Art Gittelman, Eric Grosse, Dave Hanson, Fritz Henglein, Robert Henry, Gerard Holzmann, Steve Johnson, Brian Kernighan, Ken Kubota, Daniel Lehmann, Dave MacQueen, Dianne Maki, Alan Martin, Doug McIlroy, Charles McLaughlin, John Mitchell, Elliott Organick, Robert Paige, Phil Pfeiffer, Rob Pike, Kari-Jouko Räihä, Dennis Ritchie, Sriram Sankar, Paul Stoecker, Bjarne Stroustrup, Tom Szymanski, Kim Tracy, Peter Weinberger, Jennifer Widom, 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.

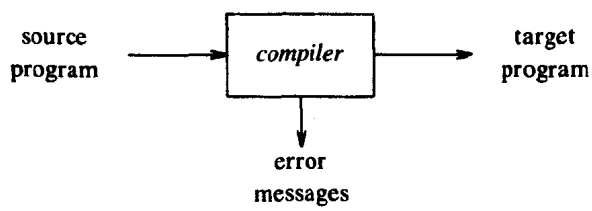


Fig. 1.1. A compiler.

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 pieces 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.

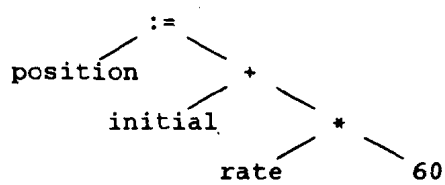


Fig. 1.2. Syntax tree for `position := initial + rate * 60`.