Compiler Design Theory

Compiler Design Theory

PHILIP M. LEWIS II
DANIEL J. ROSENKRANTZ
RICHARD E. STEARNS
General Electric Company

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

 Printed in the United States of America. Published simultaneously in Canada. Library of Congress Catalog Card No. 75-9012.

ISBN 0-201-14455-7
ABCDEFGHIJ-HA-798765

编译程序设计理论

该书由IBM公司发起，组织了14名专业人员编纂了12本一套的丛书，取名"系统程序设计丛书"。这里介绍的"编译程序设计理论"是该丛书之一，它取材于大学使用过多年的教材，内容包括设计编译程序或其它语言加工程序所必须的某些基本数学理论以及如何实际运用这些实际理论。作者以"自动机"和形式语言理论作为数学概念的基础，并着重介绍了自动翻译程序设计的综合过程。作者认为：只要掌握了上述两种概念即可设计出教学用或应用的编译程序。并以此概念设计了两个商用编译程序为范例。所以此书内容是理论和实践并重，有较好的参考价值，可供那些对于程序设计语言以及数学方法较为熟习的计算机软件工作者研读。全书共15章：①引论，②有限状态计算机，③有限状态计算机的实现，④ MINI-BASIC 词法框，⑤下推计算机，⑥与上下文无关的语法，⑦句法制导处理，⑧由顶向下的处理，⑨表征文法的由顶向下的处理，⑩ MINI-BASIC句法框，⑪由底向上的处理，⑫替换-认别处理，⑬替换-归约处理，⑭ MINI-BASIC编译程序的代码产生程序，⑮目标代码优化概论。附录A：MINI-BASIC 语言手册；附录B：关系式；附录C：语法的翻译。

THE SYSTEMS PROGRAMMING SERIES

*The Program Development Process Part I—The Individual Programmer	Joel D. Aron
The Program Development Process Part II—The Programming Team	Joel D. Aron
*The Design and Structure of Programming Languages	John E. Nicholls
Mathematical Background of Programming	Frank Beckman
Structured Programming	Harlan D. Mills Richard C. Linger
*An Introduction to Database Systems	C. J. Date
Compiler Engineering	Patricia Goldberg
Interactive Computer Graphics	Andries Van Dam
*Sorting and Sort Systems	Harold Lorin
*Compiler Design Theory	Philip M. Lewis II Daniel J. Rosenkrantz Richard E. Stearns
Recursive Programming Techniques*	William Burge
Conceptual Structures: Information Processing in Mind and Machines	John F. Sowa

*Published

Foreword

The field of systems programming primarily grew out of the efforts of many programmers and managers whose creative energy went into producing practical, utilitarian systems programs needed by the rapidly growing computer industry. Programming was practiced as an art where each programmer invented his own solutions to problems with little guidance beyond that provided by his immediate associates. In 1968, the late Ascher Opler, then at IBM, recognized that it was necessary to bring programming knowledge together in a form that would be accessible to all systems programmers. Surveying the state of the art, he decided that enough useful material existed to justify a significant codification effort. On his recommendation, IBM decided to sponsor The Systems Programming Series as a long term project to collect, organize, and publish those principles and techniques that would have lasting value throughout the industry.

The Series consists of an open-ended collection of text-reference books. The contents of each book represent the individual author's view of the subject area and do not necessarily reflect the views of the IBM Corporation. Each is organized for course use but is detailed enough for reference. Further, the Series is organized in three levels: broad introductory material in the foundation volumes, more specialized material in the software volumes, and very specialized theory in the computer science volumes. As such, the Series meets the needs of the novice, the experienced programmer, and the computer scientist.

Taken together, the Series is a record of the state of the art in systems programming that can form the technological base for the systems programming discipline.

The Editorial Board

Preface

This book is intended to be a text for a one- or two-semester course in compiler design at the senior or first-year-graduate level. It covers the basic mathematical theory underlying the design of compilers and other language processors and shows how to use that theory in practical design situations.

The applicable mathematical concepts come from automata and formal language theory. We have developed these concepts in a rigorous but nonformal style to make them understandable to a wide range of readers, including those who are not mathematically oriented. We believe that automata and formal language concepts constitute an excellent basis both for teaching compiler design and for designing real compilers. We ourselves have designed two commercial compilers based on this theory.

In our selection and presentation of material we emphasize "translation," in contrast to just "parsing." The formal concept of a syntax-directed attributed translation is used to specify the input-output performance of various language processors.

Another concept we emphasize is that of an "automaton." We use such automata as finite-state machines and pushdown machines as basic building blocks for compilers. We emphasize synthesis procedures for designing an automaton to perform a specified translation.

The material in this book constitutes an essentially complete design theory for the lexical and syntax portions of a compiler. The use of attributed translations allows us to include in the syntax-box design a good deal of what is often characterized as "code generation" or "semantics." The book also includes additional material on code generation and a brief survey of code optimization.

The subject of run-time implementation is not discussed. Although this topic is of considerable importance in deciding what code a compiler should generate, we believe it is not a part of "compiler design theory" but rather a separate subject that should be treated in a course dealing specifically with programming language structures.

Because the automata theory in this text has been selected for its relevance to compiler design, certain basic automata-theory concepts have been bypassed; consequently, this book cannot be used as the exclusive text in an automata theory course. However, students who have taken a course using this book should find a subsequent course in automata-theory greatly simplified. Conversely, some training in basic automata theory permits students to cover the material in this book more rapidly; thus the text can be used either prior to or after taking an introductory course in automata theory.

Compiler Design Theory is completely self-contained and assumes only the familiarity with programming languages and the mathematical sophistication commonly found in juniors or seniors.

After an introductory chapter, Chapters 2, 3, and 4 cover finite-state machines and other topics relevant to lexical processing. Chapters 5 and 6 introduce pushdown machines and context-free grammars.

If students have had an introductory course in automata theory, much of the material in Chapters 2 through 6 can be omitted and the remainder covered very rapidly emphasizing the applications to compilers. In any case, Sections 2.7 through 2.11 can be omitted.

Chapter 7 introduces the ideas of translations and attributed translations; this material should be mastered before progressing.

Chapters 8, 9, and 10 cover top-down processing while Chapters 11, 12, and 13 deal with bottom-up processing. These portions of the book are independent and the instructor can elect to cover either or both of them. In any case, Sections 8.7, 10.5, 12.6, and 13.6 can be omitted.

To demonstrate the theoretical concepts in a "real" design situation, the book includes the design of a compiler for a subset of BASIC. The language was selected to be sufficiently complex to illustrate the concepts presented in the book and to have a trivial run-time implementation (since, as we have said, we believe the implementation of language features to be a separate subject). However the language has a variety of syntactic and semantic features including a syntactically recursive control structure (a FOR loop). In Chapter 4, we design a lexical box for the compiler and in Chapters 10 and 12 we design syntax boxes that operate top-down and bottom-up, respectively. In Chapter 14 we design a code generator. Either this design or some extension of it can be implemented in a laboratory portion of the course.

Chapter 15 contains a brief survey of code optimization.

The book also contains three appendices: Appendix A is a language manual for MINI-BASIC; Appendix B discusses those aspects of mathematical relations needed for various test and design procedures; Appendix C presents several methods for transforming a given programming-language grammar into one of the special forms presented in the text.

The material in this book has been taught for several years in one-semester first-year graduate courses at Rensselaer Polytechnic Institute in Troy, N.Y. and at the State University of New York at Albany. It has also been taught as a one-semester undergraduate elective at Union College in Schenectady, N.Y. We would like to thank the students at these institutions whose occasional bewildered looks have motivated us to do several rewrites of the material.

We would like to express our appreciation to the following people who read early versions of the manuscript and made helpful comments: John Hutchison, Michael Hammer, Stephen Morse, John Johnston, Donna Phillips, Daniel Berry, Alyce Orne, Gary Fisher, Walter Stone, James Roberts, and Robert Blean.

We are also grateful to the management of the General Electric Research and Development Center and especially to Richard L. Shuey and James L. Lawson who established the free and stimulating environment in which our work was done and provided the time and opportunity to write this book.

Schenectady, N.Y.
December 1975

P.M.L.
D.J.R.
R.E.S.

Contents

CHAPTER 3
IMPLEMENTING FINITE STATE MACHINES

CHAPTER 4
MINI-BASIC LEXICAL BOX

CHAPTER 5
PUSHDOWN MACHINES

CHAPTER 6
CONTEXT-FREE GRAMMARS

CHAPTER 14
A CODE GENERATOR FOR THE MINI-BASIC COMPILER

CHAPTER 15
A SURVEY OF OBJECT CODE OPTIMIZATION

APPENDIX A
MINI-BASIC LANGUAGE MANUAL

APPENDIX B
RELATIONS

APPENDIX C
GRAMMATICAL TRANSFORMATIONS

1
Introduction

1.1 LANGUAGE PROCESSORS

There is a natural communication gap between man and machine. Computer hardware operates at a very atomic level in terms of bits and registers, whereas people tend to express themselves in terms of natural languages such as English or in mathematical notation. This communication gap is usually bridged by means of an artificial language which allows the human to express himself with a well-defined set of words, sentences, and formulas that can be "understood" by a computer. To achieve this communication, the human is supplied with a user manual which explains the constructs and meanings allowed by the language, and the computer is supplied with software by which it can take a stream of bits representing the commands or programs written in the language by the human and translate this input into the internal bit patterns required to carry out the human's intent.

Existing computer languages vary widely in complexity, including, for example:

the instruction set of a particular computer, the machine language, which is interpreted by the hardware or micro-programs of the machine itself;

assembly languages, the "low-level" languages that largely mirror the instruction set of a particular computer;

control-card and command languages that are used to communicate with an operating system;

"high-level" languages, such as FORTRAN, PL/I, LISP, etc., which have a complicated structure and do not depend on the instruction set or operating system of any particular machine.

We use the term "language processor" to describe the computer programs that enable the computer to "understand" the commands and inputs supplied by the human. Broadly speaking, there are two types of such language-processing programs: interpreters and translators.

An *interpreter* is a program that accepts as input a program written in a computer language called the *source language* and performs the computations implied by the program.

A *translator* is a program that accepts as input a program written in a *source language* and produces as output another version of that program written in another language called the *object language*. Usually the object language is the machine language of some computer, in which case the program can be immediately executed on that computer. Translators are rather arbitrarily divided into *assemblers* and *compilers* which translate low-level and high-level languages, respectively.

The common mathematical foundation of all language processing is the theory of automata and formal languages. Since the main concern of this book is the design of compilers, we present those portions of this theory that are most relevant to compiler design and show practical methods whereby this mathematics may be applied. Although the theory is presented in the context of compilers, it can be used in the design of any language processor.

1.2 A NAIVE COMPILER MODEL

The job of a compiler is to translate the bit patterns that represent a program written in some computer language into a sequence of machine instructions that carry out the programmer's intent. This task is sufficiently complex that understanding or designing a compiler as a single entity is both difficult and cumbersome. Therefore, it is desirable to consider the compilation process as an interconnection of smaller processes whose tasks can be more easily described.

The selection of these subprocesses for any particular compiler may depend on the details of the language being processed and, in any case, can best be done when taking available design theory into account. Hence, we do not want to endorse a specific set of subprocesses. On the other hand, it is impossible to describe a compiler design theory without having some ideas as to a possible internal organization of a compiler. Therefore, in order to establish a frame of reference, we introduce a rather naive but specific model.

In this model, the compiling job is done by a serial connection of three boxes which we call the *lexical box*, the *syntax box*, and the *code generator*. These three boxes have access to a common set of tables where long-term or