Uwe Wolter

Compiler Construction
Principles and Practice

Kenneth C. Louden
San Jose State University

PWS Publishing Company
ITP
An International Thomson Publishing Company
Boston • Albany • Bonn • Cincinnati • Detroit • London • Madrid • Melbourne • Mexico City • New York
Pacific Grove • Paris • San Francisco • Singapore • Tokyo • Toronto • Washington

PWS PUBLISHING COMPANY
20 Park Plaza, Boston, MA 02116-4324

I(T)P™
International Thomson Publishing
The trademark ITP is used under license.

This book is printed on recycled, acid-free paper.

For more information, contact:

PWS Publishing Company
20 Park Plaza
Boston, MA 02116

International Thomson Publishing Europe
Berkshire House 168-173
High Holborn
London WC1V 7AA
England

Thomas Nelson Australia
102 Dodds Street
South Melbourne, 3205
Victoria, Australia

Nelson Canada
1120 Birchmont Road
Scarborough, Ontario
Canada M1K 5G4

International Thomson Editores
Campos Eliseos 385, Piso 7
Col. Polanco
11560 Mexico D.F., Mexico

International Thomson Publishing GmbH
Königswinterer Strasse 418
53227 Bonn, Germany

International Thomson Publishing Asia
221 Henderson Road
#05-10 Henderson Building
Singapore 0315

International Thomson Publishing Japan
Hirakawacho Kyowa Building, 31
2-2-1 Hirakawacho
Chiyoda-ku, Tokyo 102
Japan

Library of Congress Cataloging-in-Publication Data

Louden, Kenneth C.
Compiler construction : principles and practice / Kenneth C. Louden.
p. cm.
Includes bibliographical references and index.
ISBN 0-534-93972-4
1. Compilers (Computer programs) I. Title.
QA76.76.C65L68 1997
005.4'53—dc21 96–49664
CIP

Sponsoring Editor: *David Dietz*
Marketing Manager: *Nathan Wilbur*
Editorial Assistant: *Susan Garland*
Production Coordinator: *Pamela Rockwell*
Manufacturing Coordinator: *Andrew Christensen*

Compositor: *Better Graphics*
Cover/Text Designer: *Pamela Rockwell*
Cover Printer: *Phoenix Color Corporation*
Text Printer and Binder: *Phoenix Color Corporation*

Printed and bound in the United States of America.
01 — 10 9 8 7 6

For Andrew

Contents

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

Jerry Potter
Kent State University

Joe Lambert
Penn State University

Samuel A. Rebelsky
Dartmouth College

Joan Lukas
University of Masschusetts, Boston

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.

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-