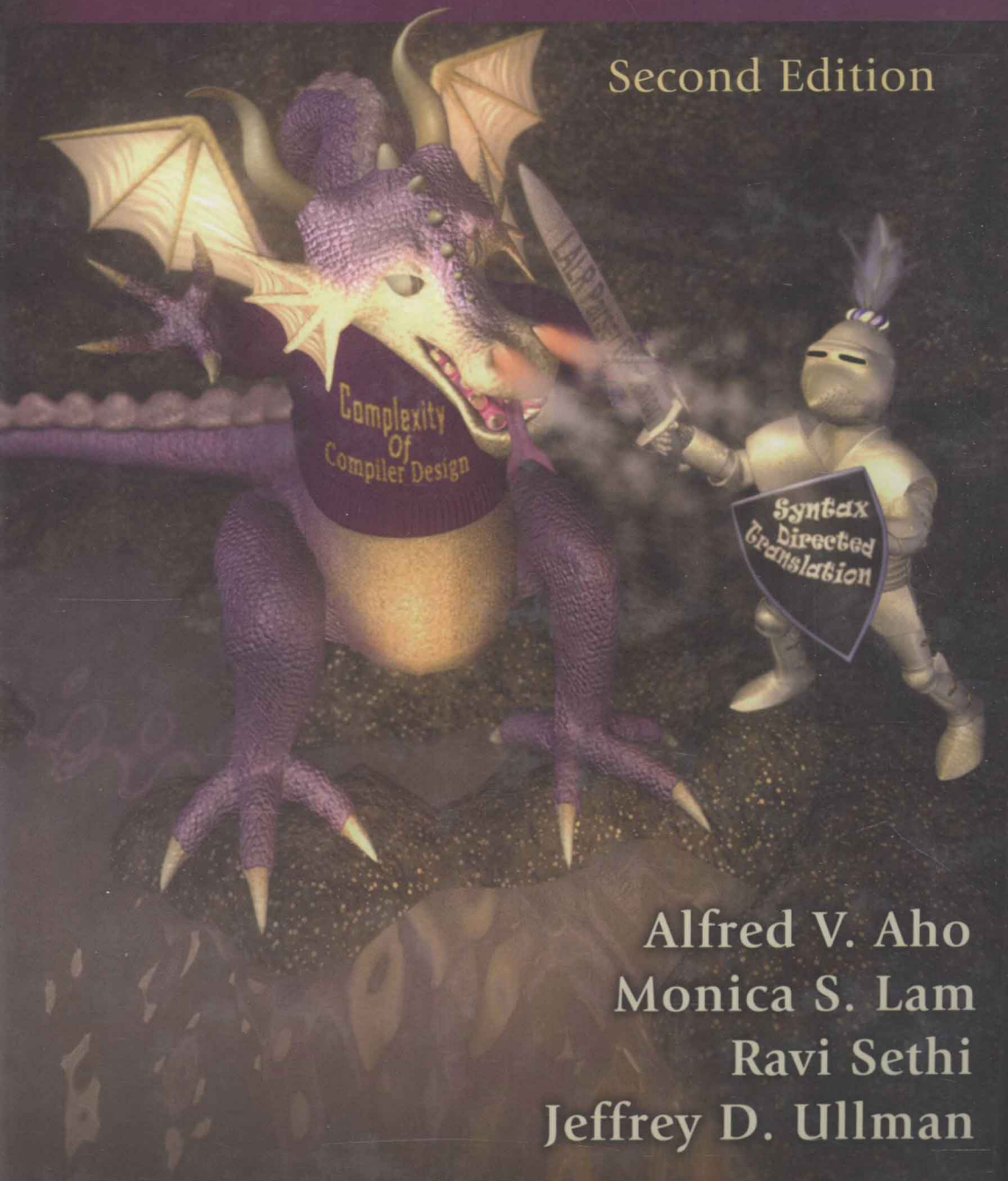
Compilers

Principles, Techniques, & Tools

Second Edition



Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Compilers

Principles, Techniques, & Tools

Second Edition

Alfred V. Aho
Columbia University

Monica S. Lam
Stanford University

Ravi Sethi
Avaya

Jeffrey D. Ullman
Stanford University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher	Greg Tobin
Executive Editor	Michael Hirsch
Acquisitions Editor	Matt Goldstein
Project Editor	Katherine Harutunian
Associate Managing Editor	Jeffrey Holcomb
Cover Designer	Joyce Cosentino Wells
Digital Assets Manager	Marianne Groth
Media Producer	Bethany Tidd
Senior Marketing Manager	Michelle Brown
Marketing Assistant	Sarah Milmore
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Cover Image	Scott Ullman of Strange Tonic Productions (www.strangetonic.com)

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

This interior of this book was composed in LaTeX.

Library of Congress Cataloging-in-Publication Data

Compilers : principles, techniques, and tools / Alfred V. Aho ... [et al.]. -- 2nd ed.
p. cm.
Rev. ed. of: Compilers, principles, techniques, and tools / Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 1986.
ISBN 0-321-48681-1 (alk. paper)
1. Compilers (Computer programs) I. Aho, Alfred V. II. Aho, Alfred V. Compilers, principles, techniques, and tools.
QA76.76.C65A37 2007
005.4'53--dc22

2006024333

2 3 4 5 6 7 8 9 10—CW—10 09 08 07 06

Preface

In the time since the 1986 edition of this book, the world of compiler design has changed significantly. Programming languages have evolved to present new compilation problems. Computer architectures offer a variety of resources of which the compiler designer must take advantage. Perhaps most interestingly, the venerable technology of code optimization has found use outside compilers. It is now used in tools that find bugs in software, and most importantly, find security holes in existing code. And much of the "front-end" technology — grammars, regular expressions, parsers, and syntax-directed translators — are still in wide use.

Thus, our philosophy from previous versions of the book has not changed. We recognize that few readers will build, or even maintain, a compiler for a major programming language. Yet the models, theory, and algorithms associated with a compiler can be applied to a wide range of problems in software design and software development. We therefore emphasize problems that are most commonly encountered in designing a language processor, regardless of the source language or target machine.

Use of the Book

It takes at least two quarters or even two semesters to cover all or most of the material in this book. It is common to cover the first half in an undergraduate course and the second half of the book — stressing code optimization — in a second course at the graduate or mezzanine level. Here is an outline of the chapters:

Chapter 1 contains motivational material and also presents some background issues in computer architecture and programming-language principles.

Chapter 2 develops a miniature compiler and introduces many of the important concepts, which are then developed in later chapters. The compiler itself appears in the appendix.

Chapter 3 covers lexical analysis, regular expressions, finite-state machines, and scanner-generator tools. This material is fundamental to text-processing of all sorts.

Chapter 4 covers the major parsing methods, top-down (recursive-descent, LL) and bottom-up (LR and its variants).

Chapter 5 introduces the principal ideas in syntax-directed definitions and syntax-directed translations.

Chapter 6 takes the theory of Chapter 5 and shows how to use it to generate intermediate code for a typical programming language.

Chapter 7 covers run-time environments, especially management of the run-time stack and garbage collection.

Chapter 8 is on object-code generation. It covers construction of basic blocks, generation of code from expressions and basic blocks, and register-allocation techniques.

Chapter 9 introduces the technology of code optimization, including flow graphs, data-flow frameworks, and iterative algorithms for solving these frameworks.

Chapter 10 covers instruction-level optimization. The emphasis is on the extraction of parallelism from small sequences of instructions and scheduling them on single processors that can do more than one thing at once.

Chapter 11 talks about larger-scale parallelism detection and exploitation. Here, the emphasis is on numeric codes that have many tight loops that range over multidimensional arrays.

Chapter 12 is on interprocedural analysis. It covers pointer analysis, aliasing, and data-flow analysis that takes into account the sequence of procedure calls that reach a given point in the code.

Courses from material in this book have been taught at Columbia, Harvard, and Stanford. At Columbia, a senior/first-year graduate course on programming languages and translators has been regularly offered using material from the first eight chapters. A highlight of this course is a semester-long project in which students work in small teams to create and implement a little language of their own design. The student-created languages have covered diverse application domains including quantum computation, music synthesis, computer graphics, gaming, matrix operations and many other areas. Students use compiler-component generators such as ANTLR, Lex, and Yacc and the syntax-directed translation techniques discussed in chapters two and five to build their compilers. A follow-on graduate course has focused on material in Chapters 9 through 12, emphasizing code generation and optimization for contemporary machines including network processors and multiprocessor architectures.

At Stanford, a one-quarter introductory course covers roughly the material in Chapters 1 through 8, although there is an introduction to global code optimization from Chapter 9. The second compiler course covers Chapters 9 through 12, plus the more advanced material on garbage collection from Chapter 7. Students use a locally developed, Java-based system called `Joeq` for implementing data-flow analysis algorithms.

Prerequisites

The reader should possess some "computer-science sophistication," including at least a second course on programming, and courses in data structures and discrete mathematics. Knowledge of several different programming languages is useful.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Gradiance On-Line Homeworks

A feature of the new edition is that there is an accompanying set of on-line homeworks using a technology developed by Gradiance Corp. Instructors may assign these homeworks to their class, or students not enrolled in a class may enroll in an "omnibus class" that allows them to do the homeworks as a tutorial (without an instructor-created class). Gradiance questions look like ordinary questions, but your solutions are sampled. If you make an incorrect choice you are given specific advice or feedback to help you correct your solution. If your instructor permits, you are allowed to try again, until you get a perfect score.

A subscription to the Gradiance service is offered with all new copies of this text sold in North America. For more information, visit the Addison-Wesley web site `www.aw.com/gradiance` or send email to `computing@aw.com`.

Support on the World Wide Web

The book's home page is

```
dragonbook.stanford.edu
```

Here, you will find errata as we learn of them, and backup materials. We hope to make available the notes for each offering of compiler-related courses as we teach them, including homeworks, solutions, and exams. We also plan to post descriptions of important compilers written by their implementers.

Acknowledgements

Cover art is by S. D. Ullman of Strange Tonic Productions.

Jon Bentley gave us extensive comments on a number of chapters of an earlier draft of this book. Helpful comments and errata were received from:

Domenico Bianculli, Peter Bosch, Marcio Buss, Marc Eaddy, Stephen Edwards, Vibhav Garg, Kim Hazelwood, Gaurav Kc, Wei Li, Mike Smith, Art Stamness, Krysta Svore, Olivier Tardieu, and Jia Zeng. The help of all these people is gratefully acknowledged. Remaining errors are ours, of course.

In addition, Monica would like to thank her colleagues on the SUIF compiler team for an 18-year lesson on compiling: Gerald Aigner, Dzintars Avots, Saman Amarasinghe, Jennifer Anderson, Michael Carbin, Gerald Cheong, Amer Diwan, Robert French, Anwar Ghuloum, Mary Hall, John Hennessy, David Heine, Shih-Wei Liao, Amy Lim, Benjamin Livshits, Michael Martin, Dror Maydan, Todd Mowry, Brian Murphy, Jeffrey Oplinger, Karen Pieper, Martin Rinard, Olatunji Ruwase, Constantine Sapuntzakis, Patrick Sathyanathan, Michael Smith, Steven Tjiang, Chau-Wen Tseng, Christopher Unkel, John Whaley, Robert Wilson, Christopher Wilson, and Michael Wolf.

A. V. A., Chatham NJ
M. S. L., Menlo Park CA
R. S., Far Hills NJ
J. D. U., Stanford CA
June, 2006

Table of Contents