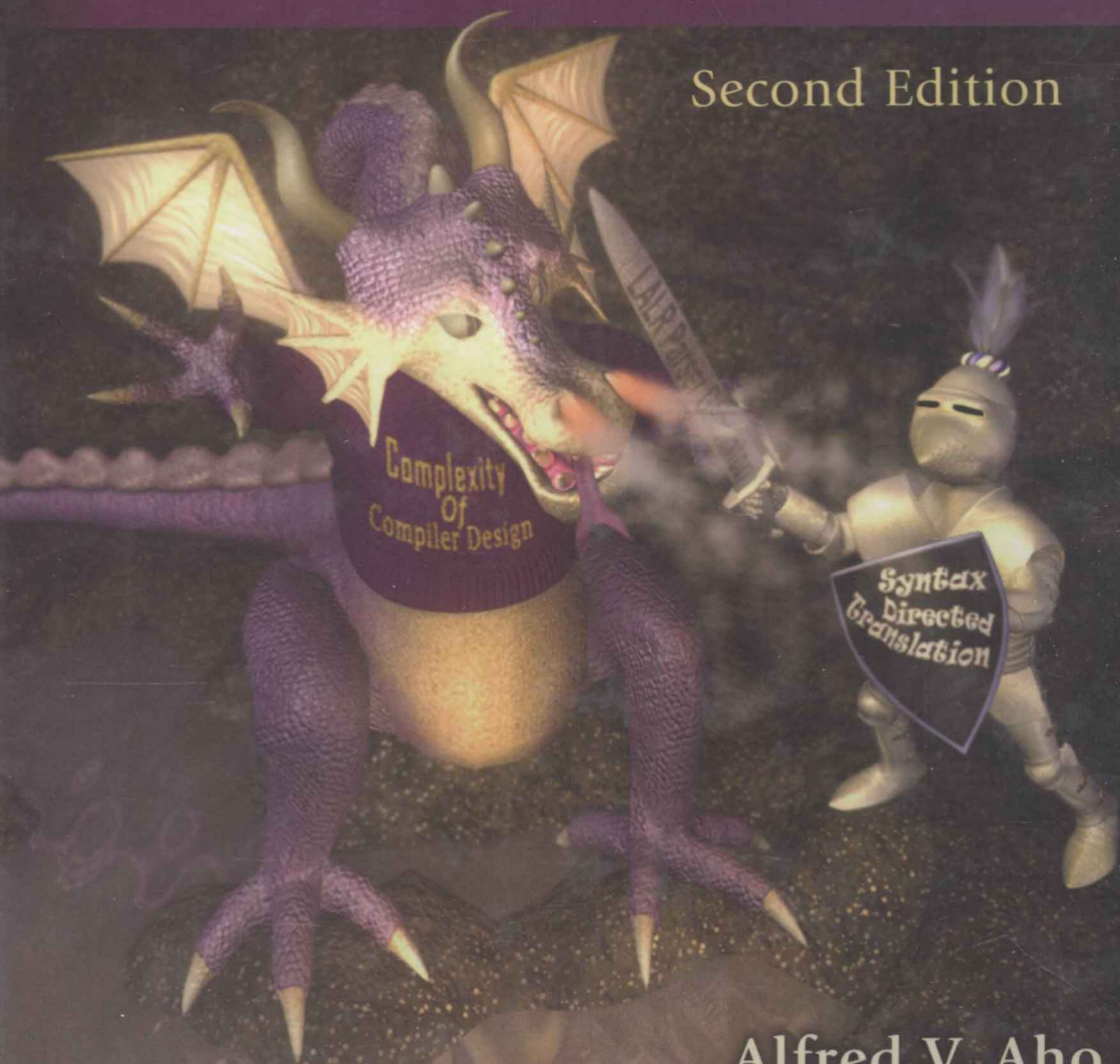


Compilers

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 L^AT_EX.

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)

I. 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

Copyright © 2007 Pearson Education, Inc. 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. For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contracts Department, 75 Arlington Street, Suite 300, Boston, MA 02116, fax your request to 617-848-7047, or e-mail at <http://www.pearsoned.com/legal/permissions.htm>.

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

1	Introduction	1
1.1	Language Processors	1
1.1.1	Exercises for Section 1.1	3
1.2	The Structure of a Compiler	4
1.2.1	Lexical Analysis	5
1.2.2	Syntax Analysis	8
1.2.3	Semantic Analysis	8
1.2.4	Intermediate Code Generation	9
1.2.5	Code Optimization	10
1.2.6	Code Generation	10
1.2.7	Symbol-Table Management	11
1.2.8	The Grouping of Phases into Passes	11
1.2.9	Compiler-Construction Tools	12
1.3	The Evolution of Programming Languages	12
1.3.1	The Move to Higher-level Languages	13
1.3.2	Impacts on Compilers	14
1.3.3	Exercises for Section 1.3	14
1.4	The Science of Building a Compiler	15
1.4.1	Modeling in Compiler Design and Implementation	15
1.4.2	The Science of Code Optimization	15
1.5	Applications of Compiler Technology	17
1.5.1	Implementation of High-Level Programming Languages	17
1.5.2	Optimizations for Computer Architectures	19
1.5.3	Design of New Computer Architectures	21
1.5.4	Program Translations	22
1.5.5	Software Productivity Tools	23
1.6	Programming Language Basics	25
1.6.1	The Static/Dynamic Distinction	25
1.6.2	Environments and States	26
1.6.3	Static Scope and Block Structure	28
1.6.4	Explicit Access Control	31
1.6.5	Dynamic Scope	31
1.6.6	Parameter Passing Mechanisms	33

1.6.7	Aliasing	35
1.6.8	Exercises for Section 1.6	35
1.7	Summary of Chapter 1	36
1.8	References for Chapter 1	38
2	A Simple Syntax-Directed Translator	39
2.1	Introduction	40
2.2	Syntax Definition	42
2.2.1	Definition of Grammars	42
2.2.2	Derivations	44
2.2.3	Parse Trees	45
2.2.4	Ambiguity	47
2.2.5	Associativity of Operators	48
2.2.6	Precedence of Operators	48
2.2.7	Exercises for Section 2.2	51
2.3	Syntax-Directed Translation	52
2.3.1	Postfix Notation	53
2.3.2	Synthesized Attributes	54
2.3.3	Simple Syntax-Directed Definitions	56
2.3.4	Tree Traversals	56
2.3.5	Translation Schemes	57
2.3.6	Exercises for Section 2.3	60
2.4	Parsing	60
2.4.1	Top-Down Parsing	61
2.4.2	Predictive Parsing	64
2.4.3	When to Use ϵ -Productions	65
2.4.4	Designing a Predictive Parser	66
2.4.5	Left Recursion	67
2.4.6	Exercises for Section 2.4	68
2.5	A Translator for Simple Expressions	68
2.5.1	Abstract and Concrete Syntax	69
2.5.2	Adapting the Translation Scheme	70
2.5.3	Procedures for the Nonterminals	72
2.5.4	Simplifying the Translator	73
2.5.5	The Complete Program	74
2.6	Lexical Analysis	76
2.6.1	Removal of White Space and Comments	77
2.6.2	Reading Ahead	78
2.6.3	Constants	78
2.6.4	Recognizing Keywords and Identifiers	79
2.6.5	A Lexical Analyzer	81
2.6.6	Exercises for Section 2.6	84
2.7	Symbol Tables	85
2.7.1	Symbol Table Per Scope	86
2.7.2	The Use of Symbol Tables	89

2.8	Intermediate Code Generation	91
2.8.1	Two Kinds of Intermediate Representations	91
2.8.2	Construction of Syntax Trees	92
2.8.3	Static Checking	97
2.8.4	Three-Address Code	99
2.8.5	Exercises for Section 2.8	105
2.9	Summary of Chapter 2	105
3	Lexical Analysis	109
3.1	The Role of the Lexical Analyzer	109
3.1.1	Lexical Analysis Versus Parsing	110
3.1.2	Tokens, Patterns, and Lexemes	111
3.1.3	Attributes for Tokens	112
3.1.4	Lexical Errors	113
3.1.5	Exercises for Section 3.1	114
3.2	Input Buffering	115
3.2.1	Buffer Pairs	115
3.2.2	Sentinels	116
3.3	Specification of Tokens	116
3.3.1	Strings and Languages	117
3.3.2	Operations on Languages	119
3.3.3	Regular Expressions	120
3.3.4	Regular Definitions	123
3.3.5	Extensions of Regular Expressions	124
3.3.6	Exercises for Section 3.3	125
3.4	Recognition of Tokens	128
3.4.1	Transition Diagrams	130
3.4.2	Recognition of Reserved Words and Identifiers	132
3.4.3	Completion of the Running Example	133
3.4.4	Architecture of a Transition-Diagram-Based Lexical Analyzer	134
3.4.5	Exercises for Section 3.4	136
3.5	The Lexical-Analyzer Generator Lex	140
3.5.1	Use of Lex	140
3.5.2	Structure of Lex Programs	141
3.5.3	Conflict Resolution in Lex	144
3.5.4	The Lookahead Operator	144
3.5.5	Exercises for Section 3.5	146
3.6	Finite Automata	147
3.6.1	Nondeterministic Finite Automata	147
3.6.2	Transition Tables	148
3.6.3	Acceptance of Input Strings by Automata	149
3.6.4	Deterministic Finite Automata	149
3.6.5	Exercises for Section 3.6	151
3.7	From Regular Expressions to Automata	152

3.7.1	Conversion of an NFA to a DFA	152
3.7.2	Simulation of an NFA	156
3.7.3	Efficiency of NFA Simulation	157
3.7.4	Construction of an NFA from a Regular Expression	159
3.7.5	Efficiency of String-Processing Algorithms	163
3.7.6	Exercises for Section 3.7	166
3.8	Design of a Lexical-Analyzer Generator	166
3.8.1	The Structure of the Generated Analyzer	167
3.8.2	Pattern Matching Based on NFA's	168
3.8.3	DFA's for Lexical Analyzers	170
3.8.4	Implementing the Lookahead Operator	171
3.8.5	Exercises for Section 3.8	172
3.9	Optimization of DFA-Based Pattern Matchers	173
3.9.1	Important States of an NFA	173
3.9.2	Functions Computed From the Syntax Tree	175
3.9.3	Computing <i>nullable</i> , <i>firstpos</i> , and <i>lastpos</i>	176
3.9.4	Computing <i>followpos</i>	177
3.9.5	Converting a Regular Expression Directly to a DFA	179
3.9.6	Minimizing the Number of States of a DFA	180
3.9.7	State Minimization in Lexical Analyzers	184
3.9.8	Trading Time for Space in DFA Simulation	185
3.9.9	Exercises for Section 3.9	186
3.10	Summary of Chapter 3	187
3.11	References for Chapter 3	189
4	Syntax Analysis	191
4.1	Introduction	192
4.1.1	The Role of the Parser	192
4.1.2	Representative Grammars	193
4.1.3	Syntax Error Handling	194
4.1.4	Error-Recovery Strategies	195
4.2	Context-Free Grammars	197
4.2.1	The Formal Definition of a Context-Free Grammar	197
4.2.2	Notational Conventions	198
4.2.3	Derivations	199
4.2.4	Parse Trees and Derivations	201
4.2.5	Ambiguity	203
4.2.6	Verifying the Language Generated by a Grammar	204
4.2.7	Context-Free Grammars Versus Regular Expressions	205
4.2.8	Exercises for Section 4.2	206
4.3	Writing a Grammar	209
4.3.1	Lexical Versus Syntactic Analysis	209
4.3.2	Eliminating Ambiguity	210
4.3.3	Elimination of Left Recursion	212
4.3.4	Left Factoring	214

4.3.5	Non-Context-Free Language Constructs	215
4.3.6	Exercises for Section 4.3	216
4.4	Top-Down Parsing	217
4.4.1	Recursive-Descent Parsing	219
4.4.2	FIRST and FOLLOW	220
4.4.3	LL(1) Grammars	222
4.4.4	Nonrecursive Predictive Parsing	226
4.4.5	Error Recovery in Predictive Parsing	228
4.4.6	Exercises for Section 4.4	231
4.5	Bottom-Up Parsing	233
4.5.1	Reductions	234
4.5.2	Handle Pruning	235
4.5.3	Shift-Reduce Parsing	236
4.5.4	Conflicts During Shift-Reduce Parsing	238
4.5.5	Exercises for Section 4.5	240
4.6	Introduction to LR Parsing: Simple LR	241
4.6.1	Why LR Parsers?	241
4.6.2	Items and the LR(0) Automaton	242
4.6.3	The LR-Parsing Algorithm	248
4.6.4	Constructing SLR-Parsing Tables	252
4.6.5	Viable Prefixes	256
4.6.6	Exercises for Section 4.6	257
4.7	More Powerful LR Parsers	259
4.7.1	Canonical LR(1) Items	260
4.7.2	Constructing LR(1) Sets of Items	261
4.7.3	Canonical LR(1) Parsing Tables	265
4.7.4	Constructing LALR Parsing Tables	266
4.7.5	Efficient Construction of LALR Parsing Tables	270
4.7.6	Compaction of LR Parsing Tables	275
4.7.7	Exercises for Section 4.7	277
4.8	Using Ambiguous Grammars	278
4.8.1	Precedence and Associativity to Resolve Conflicts	279
4.8.2	The “Dangling-Else” Ambiguity	281
4.8.3	Error Recovery in LR Parsing	283
4.8.4	Exercises for Section 4.8	285
4.9	Parser Generators	287
4.9.1	The Parser Generator <i>Yacc</i>	287
4.9.2	Using <i>Yacc</i> with Ambiguous Grammars	291
4.9.3	Creating <i>Yacc</i> Lexical Analyzers with <i>Lex</i>	294
4.9.4	Error Recovery in <i>Yacc</i>	295
4.9.5	Exercises for Section 4.9	297
4.10	Summary of Chapter 4	297
4.11	References for Chapter 4	300

5	Syntax-Directed Translation	303
5.1	Syntax-Directed Definitions	304
5.1.1	Inherited and Synthesized Attributes	304
5.1.2	Evaluating an SDD at the Nodes of a Parse Tree	306
5.1.3	Exercises for Section 5.1	309
5.2	Evaluation Orders for SDD's	310
5.2.1	Dependency Graphs	310
5.2.2	Ordering the Evaluation of Attributes	312
5.2.3	S-Attributed Definitions	312
5.2.4	L-Attributed Definitions	313
5.2.5	Semantic Rules with Controlled Side Effects	314
5.2.6	Exercises for Section 5.2	317
5.3	Applications of Syntax-Directed Translation	318
5.3.1	Construction of Syntax Trees	318
5.3.2	The Structure of a Type	321
5.3.3	Exercises for Section 5.3	323
5.4	Syntax-Directed Translation Schemes	324
5.4.1	Postfix Translation Schemes	324
5.4.2	Parser-Stack Implementation of Postfix SDT's	325
5.4.3	SDT's With Actions Inside Productions	327
5.4.4	Eliminating Left Recursion From SDT's	328
5.4.5	SDT's for L-Attributed Definitions	331
5.4.6	Exercises for Section 5.4	336
5.5	Implementing L-Attributed SDD's	337
5.5.1	Translation During Recursive-Descent Parsing	338
5.5.2	On-The-Fly Code Generation	340
5.5.3	L-Attributed SDD's and LL Parsing	343
5.5.4	Bottom-Up Parsing of L-Attributed SDD's	348
5.5.5	Exercises for Section 5.5	352
5.6	Summary of Chapter 5	353
5.7	References for Chapter 5	354
6	Intermediate-Code Generation	357
6.1	Variants of Syntax Trees	358
6.1.1	Directed Acyclic Graphs for Expressions	359
6.1.2	The Value-Number Method for Constructing DAG's	360
6.1.3	Exercises for Section 6.1	362
6.2	Three-Address Code	363
6.2.1	Addresses and Instructions	364
6.2.2	Quadruples	366
6.2.3	Triples	367
6.2.4	Static Single-Assignment Form	369
6.2.5	Exercises for Section 6.2	370
6.3	Types and Declarations	370
6.3.1	Type Expressions	371

6.3.2	Type Equivalence	372
6.3.3	Declarations	373
6.3.4	Storage Layout for Local Names	373
6.3.5	Sequences of Declarations	376
6.3.6	Fields in Records and Classes	376
6.3.7	Exercises for Section 6.3	378
6.4	Translation of Expressions	378
6.4.1	Operations Within Expressions	378
6.4.2	Incremental Translation	380
6.4.3	Addressing Array Elements	381
6.4.4	Translation of Array References	383
6.4.5	Exercises for Section 6.4	384
6.5	Type Checking	386
6.5.1	Rules for Type Checking	387
6.5.2	Type Conversions	388
6.5.3	Overloading of Functions and Operators	390
6.5.4	Type Inference and Polymorphic Functions	391
6.5.5	An Algorithm for Unification	395
6.5.6	Exercises for Section 6.5	398
6.6	Control Flow	399
6.6.1	Boolean Expressions	399
6.6.2	Short-Circuit Code	400
6.6.3	Flow-of-Control Statements	401
6.6.4	Control-Flow Translation of Boolean Expressions	403
6.6.5	Avoiding Redundant Gotos	405
6.6.6	Boolean Values and Jumping Code	408
6.6.7	Exercises for Section 6.6	408
6.7	Backpatching	410
6.7.1	One-Pass Code Generation Using Backpatching	410
6.7.2	Backpatching for Boolean Expressions	411
6.7.3	Flow-of-Control Statements	413
6.7.4	Break-, Continue-, and Goto-Statements	416
6.7.5	Exercises for Section 6.7	417
6.8	Switch-Statements	418
6.8.1	Translation of Switch-Statements	419
6.8.2	Syntax-Directed Translation of Switch-Statements	420
6.8.3	Exercises for Section 6.8	421
6.9	Intermediate Code for Procedures	422
6.10	Summary of Chapter 6	424
6.11	References for Chapter 6	425

7	Run-Time Environments	427
7.1	Storage Organization	427
7.1.1	Static Versus Dynamic Storage Allocation	429
7.2	Stack Allocation of Space	430
7.2.1	Activation Trees	430
7.2.2	Activation Records	433
7.2.3	Calling Sequences	436
7.2.4	Variable-Length Data on the Stack	438
7.2.5	Exercises for Section 7.2	440
7.3	Access to Nonlocal Data on the Stack	441
7.3.1	Data Access Without Nested Procedures	442
7.3.2	Issues With Nested Procedures	442
7.3.3	A Language With Nested Procedure Declarations	443
7.3.4	Nesting Depth	443
7.3.5	Access Links	445
7.3.6	Manipulating Access Links	447
7.3.7	Access Links for Procedure Parameters	448
7.3.8	Displays	449
7.3.9	Exercises for Section 7.3	451
7.4	Heap Management	452
7.4.1	The Memory Manager	453
7.4.2	The Memory Hierarchy of a Computer	454
7.4.3	Locality in Programs	455
7.4.4	Reducing Fragmentation	457
7.4.5	Manual Deallocation Requests	460
7.4.6	Exercises for Section 7.4	463
7.5	Introduction to Garbage Collection	463
7.5.1	Design Goals for Garbage Collectors	464
7.5.2	Reachability	466
7.5.3	Reference Counting Garbage Collectors	468
7.5.4	Exercises for Section 7.5	470
7.6	Introduction to Trace-Based Collection	470
7.6.1	A Basic Mark-and-Sweep Collector	471
7.6.2	Basic Abstraction	473
7.6.3	Optimizing Mark-and-Sweep	475
7.6.4	Mark-and-Compact Garbage Collectors	476
7.6.5	Copying collectors	478
7.6.6	Comparing Costs	482
7.6.7	Exercises for Section 7.6	482
7.7	Short-Pause Garbage Collection	483
7.7.1	Incremental Garbage Collection	483
7.7.2	Incremental Reachability Analysis	485
7.7.3	Partial-Collection Basics	487
7.7.4	Generational Garbage Collection	488
7.7.5	The Train Algorithm	490

7.7.6	Exercises for Section 7.7	493
7.8	Advanced Topics in Garbage Collection	494
7.8.1	Parallel and Concurrent Garbage Collection	495
7.8.2	Partial Object Relocation	497
7.8.3	Conservative Collection for Unsafe Languages	498
7.8.4	Weak References	498
7.8.5	Exercises for Section 7.8	499
7.9	Summary of Chapter 7	500
7.10	References for Chapter 7	502
8	Code Generation	505
8.1	Issues in the Design of a Code Generator	506
8.1.1	Input to the Code Generator	507
8.1.2	The Target Program	507
8.1.3	Instruction Selection	508
8.1.4	Register Allocation	510
8.1.5	Evaluation Order	511
8.2	The Target Language	512
8.2.1	A Simple Target Machine Model	512
8.2.2	Program and Instruction Costs	515
8.2.3	Exercises for Section 8.2	516
8.3	Addresses in the Target Code	518
8.3.1	Static Allocation	518
8.3.2	Stack Allocation	520
8.3.3	Run-Time Addresses for Names	522
8.3.4	Exercises for Section 8.3	524
8.4	Basic Blocks and Flow Graphs	525
8.4.1	Basic Blocks	526
8.4.2	Next-Use Information	528
8.4.3	Flow Graphs	529
8.4.4	Representation of Flow Graphs	530
8.4.5	Loops	531
8.4.6	Exercises for Section 8.4	531
8.5	Optimization of Basic Blocks	533
8.5.1	The DAG Representation of Basic Blocks	533
8.5.2	Finding Local Common Subexpressions	534
8.5.3	Dead Code Elimination	535
8.5.4	The Use of Algebraic Identities	536
8.5.5	Representation of Array References	537
8.5.6	Pointer Assignments and Procedure Calls	539
8.5.7	Reassembling Basic Blocks From DAG's	539
8.5.8	Exercises for Section 8.5	541
8.6	A Simple Code Generator	542
8.6.1	Register and Address Descriptors	543
8.6.2	The Code-Generation Algorithm	544

8.6.3	Design of the Function <i>getReg</i>	547
8.6.4	Exercises for Section 8.6	548
8.7	Peephole Optimization	549
8.7.1	Eliminating Redundant Loads and Stores	550
8.7.2	Eliminating Unreachable Code	550
8.7.3	Flow-of-Control Optimizations	551
8.7.4	Algebraic Simplification and Reduction in Strength	552
8.7.5	Use of Machine Idioms	552
8.7.6	Exercises for Section 8.7	553
8.8	Register Allocation and Assignment	553
8.8.1	Global Register Allocation	553
8.8.2	Usage Counts	554
8.8.3	Register Assignment for Outer Loops	556
8.8.4	Register Allocation by Graph Coloring	556
8.8.5	Exercises for Section 8.8	557
8.9	Instruction Selection by Tree Rewriting	558
8.9.1	Tree-Translation Schemes	558
8.9.2	Code Generation by Tiling an Input Tree	560
8.9.3	Pattern Matching by Parsing	563
8.9.4	Routines for Semantic Checking	565
8.9.5	General Tree Matching	565
8.9.6	Exercises for Section 8.9	567
8.10	Optimal Code Generation for Expressions	567
8.10.1	Ershov Numbers	567
8.10.2	Generating Code From Labeled Expression Trees	568
8.10.3	Evaluating Expressions with an Insufficient Supply of Reg- isters	570
8.10.4	Exercises for Section 8.10	572
8.11	Dynamic Programming Code-Generation	573
8.11.1	Contiguous Evaluation	574
8.11.2	The Dynamic Programming Algorithm	575
8.11.3	Exercises for Section 8.11	577
8.12	Summary of Chapter 8	578
8.13	References for Chapter 8	579
9	Machine-Independent Optimizations	583
9.1	The Principal Sources of Optimization	584
9.1.1	Causes of Redundancy	584
9.1.2	A Running Example: Quicksort	585
9.1.3	Semantics-Preserving Transformations	586
9.1.4	Global Common Subexpressions	588
9.1.5	Copy Propagation	590
9.1.6	Dead-Code Elimination	591
9.1.7	Code Motion	592
9.1.8	Induction Variables and Reduction in Strength	592