

Modern Compiler Design

Dick Grune • Kees van Reeuwijk • Henri E. Bal
Ceriël J.H. Jacobs • Koen Langendoen

Modern Compiler Design

Second Edition

 Springer

Dick Grune
Vrije Universiteit
Amsterdam, The Netherlands

Ceriel J.H. Jacobs
Vrije Universiteit
Amsterdam, The Netherlands

Kees van Reeuwijk
Vrije Universiteit
Amsterdam, The Netherlands

Koen Langendoen
Delft University of Technology
Delft, The Netherlands

Henri E. Bal
Vrije Universiteit
Amsterdam, The Netherlands

Additional material to this book can be downloaded from <http://extras.springer.com>.

ISBN 978-1-4614-4698-9 ISBN 978-1-4614-4699-6 (eBook)
DOI 10.1007/978-1-4614-4699-6
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2012941168

© Springer Science+Business Media New York 2012

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Twelve years have passed since the first edition of *Modern Compiler Design*. For many computer science subjects this would be more than a life time, but since compiler design is probably the most mature computer science subject, it is different. An adult person develops more slowly and differently than a toddler or a teenager, and so does compiler design. The present book reflects that.

Improvements to the book fall into two groups: presentation and content. The ‘look and feel’ of the book has been modernized, but more importantly we have rearranged significant parts of the book to present them in a more structured manner: large chapters have been split and the optimizing code generation techniques have been collected in a separate chapter. Based on reader feedback and experiences in teaching from this book, both by ourselves and others, material has been expanded, clarified, modified, or deleted in a large number of places. We hope that as a result of this the reader feels that the book does a better job of making compiler design and construction accessible.

The book adds new material to cover the developments in compiler design and construction over the last twelve years. Overall the standard compiling techniques and paradigms have stood the test of time, but still new and often surprising optimization techniques have been invented; existing ones have been improved; and old ones have gained prominence. Examples of the first are: procedural abstraction, in which routines are recognized in the code and replaced by routine calls to reduce size; binary rewriting, in which optimizations are applied to the binary code; and just-in-time compilation, in which parts of the compilation are delayed to improve the perceived speed of the program. An example of the second is a technique which extends optimal code generation through exhaustive search, previously available for tiny blocks only, to moderate-size basic blocks. And an example of the third is tail recursion removal, indispensable for the compilation of functional languages. These developments are mainly described in Chapter 9.

Although syntax analysis is the one but oldest branch of compiler construction (lexical analysis being the oldest), even in that area innovation has taken place. Generalized (non-deterministic) LR parsing, developed between 1984 and 1994, is now used in compilers. It is covered in Section 3.5.8.

New hardware requirements have necessitated new compiler developments. The main examples are the need for size reduction of the object code, both to fit the code into small embedded systems and to reduce transmission times; and for lower power

consumption, to extend battery life and to reduce electricity bills. Dynamic memory allocation in embedded systems requires a balance between speed and thrift, and the question is how compiler design can help. These subjects are covered in Sections 9.2, 9.3, and 10.2.8, respectively.

With age comes legacy. There is much legacy code around, code which is so old that it can no longer be modified and recompiled with reasonable effort. If the source code is still available but there is no compiler any more, recompilation must start with a grammar of the source code. For fifty years programmers and compiler designers have used grammars to produce and analyze programs; now large legacy programs are used to produce grammars for them. The recovery of the grammar from legacy source code is discussed in Section 3.6. If just the binary executable program is left, it must be disassembled or even decompiled. For fifty years compiler designers have been called upon to design compilers and assemblers to convert source programs to binary code; now they are called upon to design disassemblers and decompilers, to roll back the assembly and compilation process. The required techniques are treated in Sections 8.4 and 8.5.

The bibliography

The literature list has been updated, but its usefulness is more limited than before, for two reasons. The first is that by the time it appears in print, the Internet can provide more up-to-date and more to-the-point information, in larger quantities, than a printed text can hope to achieve. It is our contention that anybody who has understood a larger part of the ideas explained in this book is able to evaluate Internet information on compiler design.

The second is that many of the papers we refer to are available only to those fortunate enough to have login facilities at an institute with sufficient budget to obtain subscriptions to the larger publishers; they are no longer available to just anyone who walks into a university library. Both phenomena point to paradigm shifts with which readers, authors, publishers and librarians will have to cope.

The structure of the book

This book is conceptually divided into two parts. The first, comprising Chapters 1 through 10, is concerned with techniques for program processing in general; it includes a chapter on memory management, both in the compiler and in the generated code. The second part, Chapters 11 through 14, covers the specific techniques required by the various programming paradigms. The interactions between the parts of the book are outlined in the adjacent table. The leftmost column shows the four phases of compiler construction: *analysis*, *context handling*, *synthesis*, and *run-time systems*. Chapters in this column cover both the manual and the automatic creation

of the pertinent software but tend to emphasize automatic generation. The other columns show the four paradigms covered in this book; for each paradigm an example of a subject treated by each of the phases is shown. These chapters tend to contain manual techniques only, all automatic techniques having been delegated to Chapters 2 through 9.

	in imperative and object- oriented programs (Chapter 11)	in functional programs (Chapter 12)	in logic programs (Chapter 13)	in parallel/ distributed programs (Chapter 14)
How to do:				
analysis (Chapters 2 & 3)	--	--	--	--
context handling (Chapters 4 & 5)	identifier identification	polymorphic type checking	static rule matching	Linda static analysis
synthesis (Chapters 6–9)	code for while- statement	code for list comprehension	structure unification	marshaling
run-time systems (no chapter)	stack	reduction machine	Warren Abstract Machine	replication

The scientific mind would like the table to be nice and square, with all boxes filled—in short “orthogonal”—but we see that the top right entries are missing and that there is no chapter for “run-time systems” in the leftmost column. The top right entries would cover such things as the special subjects in the program text analysis of logic languages, but present text analysis techniques are powerful and flexible enough and languages similar enough to handle all language paradigms: there is nothing to be said there, for lack of problems. The chapter missing from the leftmost column would discuss manual and automatic techniques for creating run-time systems. Unfortunately there is little or no theory on this subject: run-time systems are still crafted by hand by programmers on an intuitive basis; there is nothing to be said there, for lack of solutions.

Chapter 1 introduces the reader to compiler design by examining a simple traditional modular compiler/interpreter in detail. Several high-level aspects of compiler construction are discussed, followed by a short history of compiler construction and introductions to formal grammars and closure algorithms.

Chapters 2 and 3 treat the program text analysis phase of a compiler: the conversion of the program text to an abstract syntax tree. Techniques for lexical analysis, lexical identification of tokens, and syntax analysis are discussed.

Chapters 4 and 5 cover the second phase of a compiler: context handling. Several methods of context handling are discussed: automated ones using attribute grammars, manual ones using L-attributed and S-attributed grammars, and semi-automated ones using symbolic interpretation and data-flow analysis.

Chapters 6 through 9 cover the synthesis phase of a compiler, covering both interpretation and code generation. The chapters on code generation are mainly concerned with machine code generation; the intermediate code required for paradigm-specific constructs is treated in Chapters 11 through 14.

Chapter 10 concerns memory management techniques, both for use in the compiler and in the generated program.

Chapters 11 through 14 address the special problems in compiling for the various paradigms – imperative, object-oriented, functional, logic, and parallel/distributed. Compilers for imperative and object-oriented programs are similar enough to be treated together in one chapter, Chapter 11.

Appendix B contains hints and answers to a selection of the exercises in the book. Such exercises are marked by a ▷ followed the page number on which the answer appears. A larger set of answers can be found on Springer’s Internet page; the corresponding exercises are marked by ▷www.

Several subjects in this book are treated in a non-traditional way, and some words of justification may be in order.

Lexical analysis is based on the same dotted items that are traditionally reserved for bottom-up syntax analysis, rather than on Thompson’s NFA construction. We see the dotted item as the essential tool in bottom-up pattern matching, unifying lexical analysis, LR syntax analysis, bottom-up code generation and peep-hole optimization. The traditional lexical algorithms are just low-level implementations of item manipulation. We consider the different treatment of lexical and syntax analysis to be a historical artifact. Also, the difference between the lexical and the syntax levels tends to disappear in modern software.

Considerable attention is being paid to attribute grammars, in spite of the fact that their impact on compiler design has been limited. Yet they are the only known way of automating context handling, and we hope that the present treatment will help to lower the threshold of their application.

Functions as first-class data are covered in much greater depth in this book than is usual in compiler design books. After a good start in Algol 60, functions lost much status as manipulatable data in languages like C, Pascal, and Ada, although Ada 95 rehabilitated them somewhat. The implementation of some modern concepts, for example functional and logic languages, iterators, and continuations, however, requires functions to be manipulated as normal data. The fundamental aspects of the implementation are covered in the chapter on imperative and object-oriented languages; specifics are given in the chapters on the various other paradigms.

Additional material, including more answers to exercises, and all diagrams and all code from the book, are available through Springer’s Internet page.

Use as a course book

The book contains far too much material for a compiler design course of 13 lectures of two hours each, as given at our university, so a selection has to be made. An

introductory, more traditional course can be obtained by including, for example,

Chapter 1;
Chapter 2 up to 2.7; 2.10; 2.11; Chapter 3 up to 3.4.5; 3.5 up to 3.5.7;
Chapter 4 up to 4.1.3; 4.2.1 up to 4.3; Chapter 5 up to 5.2.2; 5.3;
Chapter 6; Chapter 7 up to 9.1.1; 9.1.4 up to 9.1.4.4; 7.3;
Chapter 10 up to 10.1.2; 10.2 up to 10.2.4;
Chapter 11 up to 11.2.3.2; 11.2.4 up to 11.2.10; 11.4 up to 11.4.2.3.

A more advanced course would include all of Chapters 1 to 11, possibly excluding Chapter 4. This could be augmented by one of Chapters 12 to 14.

An advanced course would skip much of the introductory material and concentrate on the parts omitted in the introductory course, Chapter 4 and all of Chapters 10 to 14.

Acknowledgments

We owe many thanks to the following people, who supplied us with help, remarks, wishes, and food for thought for this Second Edition: Ingmar Alting, José Fortes, Bert Huijben, Jonathan Joubert, Sara Kalvala, Frank Lippes, Paul S. Moulson, Prasant K. Patra, Carlo Perassi, Marco Rossi, Mooly Sagiv, Gert Jan Schoneveld, Ajay Singh, Evert Wattel, and Freek Zindel. Their input ranged from simple corrections to detailed suggestions to massive criticism. Special thanks go to Stefanie Scherzinger, whose thorough and thoughtful criticism of our outline code format induced us to improve it considerably; any remaining imperfections should be attributed to stubbornness on the part of the authors. The presentation of the program code snippets in the book profited greatly from Carsten Heinz's `listings` package; we thank him for making the package available to the public.

We are grateful to Ann Kostant, Melissa Fearon, and Courtney Clark of Springer US, who, through fast and competent work, have cleared many obstacles that stood in the way of publishing this book. We thank them for their effort and pleasant cooperation.

We mourn the death of Irina Athanasiu, who did not live long enough to lend her expertise in embedded systems to this book.

We thank the Faculteit der Exacte Wetenschappen of the Vrije Universiteit for their support and the use of their equipment.

Amsterdam,
March 2012

Delft,

*Dick Grune
Kees van Reeuwijk
Henri E. Bal
Ceriël J.H. Jacobs
Koen G. Langendoen*

Abridged Preface to the First Edition (2000)

In the 1980s and 1990s, while the world was witnessing the rise of the PC and the Internet on the front pages of the daily newspapers, compiler design methods developed with less fanfare, developments seen mainly in the technical journals, and –more importantly– in the compilers that are used to process today’s software. These developments were driven partly by the advent of new programming paradigms, partly by a better understanding of code generation techniques, and partly by the introduction of faster machines with large amounts of memory.

The field of programming languages has grown to include, besides the traditional imperative paradigm, the object-oriented, functional, logical, and parallel/distributed paradigms, which inspire novel compilation techniques and which often require more extensive run-time systems than do imperative languages. BURS techniques (Bottom-Up Rewriting Systems) have evolved into very powerful code generation techniques which cope superbly with the complex machine instruction sets of present-day machines. And the speed and memory size of modern machines allow compilation techniques and programming language features that were unthinkable before. Modern compiler design methods meet these challenges head-on.

The audience

Our audience are students with enough experience to have at least used a compiler occasionally and to have given some thought to the concept of compilation. When these students leave the university, they will have to be familiar with language processors for each of the modern paradigms, using modern techniques. Although curriculum requirements in many universities may have been lagging behind in this respect, graduates entering the job market cannot afford to ignore these developments.

Experience has shown us that a considerable number of techniques traditionally taught in compiler construction are special cases of more fundamental techniques. Often these special techniques work for imperative languages only; the fundamental techniques have a much wider application. An example is the stack as an optimized representation for activation records in strictly last-in-first-out languages. Therefore, this book

- focuses on principles and techniques of wide application, carefully distinguishing between the essential (= material that has a high chance of being useful to the student) and the incidental (= material that will benefit the student only in exceptional cases);
- provides a first level of implementation details and optimizations;
- augments the explanations by pointers for further study.

The student, after having finished the book, can expect to:

- have obtained a thorough understanding of the concepts of modern compiler design and construction, and some familiarity with their practical application;
- be able to start participating in the construction of a language processor for each of the modern paradigms with a minimal training period;
- be able to read the literature.

The first two provide a firm basis; the third provides potential for growth.

Acknowledgments

We owe many thanks to the following people, who were willing to spend time and effort on reading drafts of our book and to supply us with many useful and sometimes very detailed comments: Mirjam Bakker, Raoul Bhoedjang, Wilfred Dittmer, Thomer M. Gil, Ben N. Hasnai, Bert Huijben, Jaco A. Imthorn, John Romein, Tim Rühl, and the anonymous reviewers. We thank Ronald Veldema for the Pentium code segments.

We are grateful to Simon Plumtree, Gaynor Redvers-Mutton, Dawn Booth, and Jane Kerr of John Wiley & Sons Ltd, for their help and encouragement in writing this book. Lambert Meertens kindly provided information on an older ABC compiler, and Ralph Griswold on an Icon compiler.

We thank the Faculteit Wiskunde en Informatica (now part of the Faculteit der Exacte Wetenschappen) of the Vrije Universiteit for their support and the use of their equipment.

Dick Grune

dick@cs.vu.nl, <http://www.cs.vu.nl/~dick>

Henri E. Bal

bal@cs.vu.nl, <http://www.cs.vu.nl/~bal>

Ceriel J.H. Jacobs

ceriel@cs.vu.nl, <http://www.cs.vu.nl/~ceriel>

Koen G. Langendoen

koen@pds.twi.tudelft.nl, <http://pds.twi.tudelft.nl/~koen>

Amsterdam, May 2000

Contents

1	Introduction	1
1.1	Why study compiler construction?	5
1.1.1	Compiler construction is very successful	6
1.1.2	Compiler construction has a wide applicability	8
1.1.3	Compilers contain generally useful algorithms	9
1.2	A simple traditional modular compiler/interpreter	9
1.2.1	The abstract syntax tree	9
1.2.2	Structure of the demo compiler	12
1.2.3	The language for the demo compiler	13
1.2.4	Lexical analysis for the demo compiler	14
1.2.5	Syntax analysis for the demo compiler	15
1.2.6	Context handling for the demo compiler	20
1.2.7	Code generation for the demo compiler	20
1.2.8	Interpretation for the demo compiler	21
1.3	The structure of a more realistic compiler	22
1.3.1	The structure	23
1.3.2	Run-time systems	25
1.3.3	Short-cuts	26
1.4	Compiler architectures	26
1.4.1	The width of the compiler	26
1.4.2	Who's the boss?	28
1.5	Properties of a good compiler	31
1.6	Portability and retargetability	32
1.7	A short history of compiler construction	33
1.7.1	1945–1960: code generation	33
1.7.2	1960–1975: parsing	33
1.7.3	1975–present: code generation and code optimization; paradigms	34
1.8	Grammars	34
1.8.1	The form of a grammar	35
1.8.2	The grammatical production process	36

1.8.3	Extended forms of grammars	37
1.8.4	Properties of grammars	38
1.8.5	The grammar formalism	39
1.9	Closure algorithms	41
1.9.1	A sample problem	41
1.9.2	The components of a closure algorithm	43
1.9.3	An iterative implementation of the closure algorithm	44
1.10	The code forms used in this book	46
1.11	Conclusion	47

Part I From Program Text to Abstract Syntax Tree

2	Program Text to Tokens — Lexical Analysis	55
2.1	Reading the program text	59
2.1.1	Obtaining and storing the text	59
2.1.2	The troublesome newline	60
2.2	Lexical versus syntactic analysis	61
2.3	Regular expressions and regular descriptions	61
2.3.1	Regular expressions and BNF/EBNF	63
2.3.2	Escape characters in regular expressions	63
2.3.3	Regular descriptions	63
2.4	Lexical analysis	64
2.5	Creating a lexical analyzer by hand	65
2.5.1	Optimization by precomputation	70
2.6	Creating a lexical analyzer automatically	73
2.6.1	Dotted items	74
2.6.2	Concurrent search	79
2.6.3	Precomputing the item sets	83
2.6.4	The final lexical analyzer	86
2.6.5	Complexity of generating a lexical analyzer	87
2.6.6	Transitions to S_ω	87
2.6.7	Complexity of using a lexical analyzer	88
2.7	Transition table compression	89
2.7.1	Table compression by row displacement	90
2.7.2	Table compression by graph coloring	93
2.8	Error handling in lexical analyzers	95
2.9	A traditional lexical analyzer generator— <i>lex</i>	96
2.10	Lexical identification of tokens	99
2.11	Symbol tables	101
2.12	Macro processing and file inclusion	102
2.12.1	The input buffer stack	104
2.12.2	Conditional text inclusion	107
2.12.3	Generics by controlled macro processing	108
2.13	Conclusion	109

3	Tokens to Syntax Tree — Syntax Analysis	115
3.1	Two classes of parsing methods	117
3.1.1	Principles of top-down parsing	117
3.1.2	Principles of bottom-up parsing	119
3.2	Error detection and error recovery	120
3.3	Creating a top-down parser manually	122
3.3.1	Recursive descent parsing	122
3.3.2	Disadvantages of recursive descent parsing	124
3.4	Creating a top-down parser automatically	126
3.4.1	LL(1) parsing	126
3.4.2	LL(1) conflicts as an asset	132
3.4.3	LL(1) conflicts as a liability	133
3.4.4	The LL(1) push-down automaton	139
3.4.5	Error handling in LL parsers	143
3.4.6	A traditional top-down parser generator— <i>LLgen</i>	148
3.5	Creating a bottom-up parser automatically	156
3.5.1	LR(0) parsing	159
3.5.2	The LR push-down automaton	166
3.5.3	LR(0) conflicts	167
3.5.4	SLR(1) parsing	169
3.5.5	LR(1) parsing	171
3.5.6	LALR(1) parsing	176
3.5.7	Making a grammar (LA)LR(1)—or not	178
3.5.8	Generalized LR parsing	181
3.5.9	Making a grammar unambiguous	185
3.5.10	Error handling in LR parsers	188
3.5.11	A traditional bottom-up parser generator— <i>yacc/bison</i>	191
3.6	Recovering grammars from legacy code	193
3.7	Conclusion	199

Part II Annotating the Abstract Syntax Tree

4	Grammar-based Context Handling	209
4.1	Attribute grammars	210
4.1.1	The attribute evaluator	212
4.1.2	Dependency graphs	215
4.1.3	Attribute evaluation	217
4.1.4	Attribute allocation	232
4.1.5	Multi-visit attribute grammars	232
4.1.6	Summary of the types of attribute grammars	244
4.2	Restricted attribute grammars	245
4.2.1	L-attributed grammars	245
4.2.2	S-attributed grammars	250
4.2.3	Equivalence of L-attributed and S-attributed grammars	250
4.3	Extended grammar notations and attribute grammars	252

4.4	Conclusion	253
5	Manual Context Handling	261
5.1	Threading the AST	262
5.2	Symbolic interpretation	267
5.2.1	Simple symbolic interpretation	270
5.2.2	Full symbolic interpretation	273
5.2.3	Last-def analysis	275
5.3	Data-flow equations	276
5.3.1	Setting up the data-flow equations	277
5.3.2	Solving the data-flow equations	280
5.4	Interprocedural data-flow analysis	283
5.5	Carrying the information upstream—live analysis	285
5.5.1	Live analysis by symbolic interpretation	286
5.5.2	Live analysis by data-flow equations	288
5.6	Symbolic interpretation versus data-flow equations	291
5.7	Conclusion	292

Part III Processing the Intermediate Code

6	Interpretation	299
6.1	Interpreters	301
6.2	Recursive interpreters	301
6.3	Iterative interpreters	305
6.4	Conclusion	310
7	Code Generation	313
7.1	Properties of generated code	313
7.1.1	Correctness	314
7.1.2	Speed	314
7.1.3	Size	315
7.1.4	Power consumption	315
7.1.5	About optimizations	316
7.2	Introduction to code generation	317
7.2.1	The structure of code generation	319
7.2.2	The structure of the code generator	320
7.3	Preprocessing the intermediate code	321
7.3.1	Preprocessing of expressions	322
7.3.2	Preprocessing of if-statements and goto statements	323
7.3.3	Preprocessing of routines	323
7.3.4	Procedural abstraction	326
7.4	Avoiding code generation altogether	328
7.5	Code generation proper	329
7.5.1	Trivial code generation	330
7.5.2	Simple code generation	335
7.6	Postprocessing the generated code	349

7.6.1	Peephole optimization	349
7.6.2	Procedural abstraction of assembly code	353
7.7	Machine code generation	355
7.8	Conclusion	356
8	Assemblers, Disassemblers, Linkers, and Loaders	363
8.1	The tasks of an assembler	363
8.1.1	The running program	363
8.1.2	The executable code file	364
8.1.3	Object files and linkage	364
8.1.4	Alignment requirements and endianness	366
8.2	Assembler design issues	367
8.2.1	Handling internal addresses	368
8.2.2	Handling external addresses	370
8.3	Linker design issues	371
8.4	Disassembly	372
8.4.1	Distinguishing between instructions and data	372
8.4.2	Disassembly with indirection	374
8.4.3	Disassembly with relocation information	377
8.5	Decompilation	377
8.6	Conclusion	382
9	Optimization Techniques	385
9.1	General optimization	386
9.1.1	Compilation by symbolic interpretation	386
9.1.2	Code generation for basic blocks	388
9.1.3	Almost optimal code generation	405
9.1.4	BURS code generation and dynamic programming	406
9.1.5	Register allocation by graph coloring	427
9.1.6	Supercompilation	432
9.1.7	Evaluation of code generation techniques	433
9.1.8	Debugging of code optimizers	434
9.2	Code size reduction	436
9.2.1	General code size reduction techniques	436
9.2.2	Code compression	437
9.2.3	Discussion	442
9.3	Power reduction and energy saving	443
9.3.1	Just compiling for speed	445
9.3.2	Trading speed for power	445
9.3.3	Instruction scheduling and bit switching	446
9.3.4	Register relabeling	448
9.3.5	Avoiding the dynamic scheduler	449
9.3.6	Domain-specific optimizations	449
9.3.7	Discussion	450
9.4	Just-In-Time compilation	450

9.5	Compilers versus computer architectures	451
9.6	Conclusion	452

Part IV Memory Management

10	Explicit and Implicit Memory Management	463
10.1	Data allocation with explicit deallocation	465
10.1.1	Basic memory allocation	466
10.1.2	Optimizations for basic memory allocation	469
10.1.3	Compiler applications of basic memory allocation	471
10.1.4	Embedded-systems considerations	475
10.2	Data allocation with implicit deallocation	476
10.2.1	Basic garbage collection algorithms	476
10.2.2	Preparing the ground	478
10.2.3	Reference counting	485
10.2.4	Mark and scan	489
10.2.5	Two-space copying	494
10.2.6	Compaction	496
10.2.7	Generational garbage collection	498
10.2.8	Implicit deallocation in embedded systems	500
10.3	Conclusion	501

Part V From Abstract Syntax Tree to Intermediate Code

11	Imperative and Object-Oriented Programs	511
11.1	Context handling	513
11.1.1	Identification	514
11.1.2	Type checking	521
11.1.3	Discussion	532
11.2	Source language data representation and handling	532
11.2.1	Basic types	532
11.2.2	Enumeration types	533
11.2.3	Pointer types	533
11.2.4	Record types	538
11.2.5	Union types	539
11.2.6	Array types	540
11.2.7	Set types	543
11.2.8	Routine types	544
11.2.9	Object types	544
11.2.10	Interface types	554
11.3	Routines and their activation	555
11.3.1	Activation records	556
11.3.2	The contents of an activation record	557
11.3.3	Routines	559
11.3.4	Operations on routines	562

11.3.5	Non-nested routines	564
11.3.6	Nested routines	566
11.3.7	Lambda lifting	573
11.3.8	Iterators and coroutines	576
11.4	Code generation for control flow statements	576
11.4.1	Local flow of control	577
11.4.2	Routine invocation	587
11.4.3	Run-time error handling	597
11.5	Code generation for modules	601
11.5.1	Name generation	602
11.5.2	Module initialization	602
11.5.3	Code generation for generics	604
11.6	Conclusion	606
12	Functional Programs	617
12.1	A short tour of Haskell	619
12.1.1	Offside rule	619
12.1.2	Lists	620
12.1.3	List comprehension	621
12.1.4	Pattern matching	622
12.1.5	Polymorphic typing	623
12.1.6	Referential transparency	624
12.1.7	Higher-order functions	625
12.1.8	Lazy evaluation	627
12.2	Compiling functional languages	628
12.2.1	The compiler structure	628
12.2.2	The functional core	630
12.3	Polymorphic type checking	631
12.4	Desugaring	633
12.4.1	The translation of lists	634
12.4.2	The translation of pattern matching	634
12.4.3	The translation of list comprehension	637
12.4.4	The translation of nested functions	639
12.5	Graph reduction	641
12.5.1	Reduction order	645
12.5.2	The reduction engine	647
12.6	Code generation for functional core programs	651
12.6.1	Avoiding the construction of some application spines	653
12.7	Optimizing the functional core	655
12.7.1	Strictness analysis	656
12.7.2	Boxing analysis	662
12.7.3	Tail calls	663
12.7.4	Accumulator transformation	664
12.7.5	Limitations	666
12.8	Advanced graph manipulation	667

12.8.1	Variable-length nodes	667
12.8.2	Pointer tagging	667
12.8.3	Aggregate node allocation	668
12.8.4	Vector apply nodes	668
12.9	Conclusion	669
13	Logic Programs	677
13.1	The logic programming model	679
13.1.1	The building blocks	679
13.1.2	The inference mechanism	681
13.2	The general implementation model, interpreted	682
13.2.1	The interpreter instructions	684
13.2.2	Avoiding redundant goal lists	687
13.2.3	Avoiding copying goal list tails	687
13.3	Unification	688
13.3.1	Unification of structures, lists, and sets	688
13.3.2	The implementation of unification	691
13.3.3	Unification of two unbound variables	694
13.4	The general implementation model, compiled	696
13.4.1	List procedures	697
13.4.2	Compiled clause search and unification	699
13.4.3	Optimized clause selection in the WAM	704
13.4.4	Implementing the “cut” mechanism	708
13.4.5	Implementing the predicates assert and retract	709
13.5	Compiled code for unification	715
13.5.1	Unification instructions in the WAM	716
13.5.2	Deriving a unification instruction by manual partial evaluation	718
13.5.3	Unification of structures in the WAM	721
13.5.4	An optimization: read/write mode	725
13.5.5	Further unification optimizations in the WAM	728
13.6	Conclusion	730
14	Parallel and Distributed Programs	737
14.1	Parallel programming models	740
14.1.1	Shared variables and monitors	741
14.1.2	Message passing models	742
14.1.3	Object-oriented languages	744
14.1.4	The Linda Tuple space	745
14.1.5	Data-parallel languages	747
14.2	Processes and threads	749
14.3	Shared variables	751
14.3.1	Locks	751
14.3.2	Monitors	752
14.4	Message passing	753

14.4.1	Locating the receiver	754
14.4.2	Marshaling	754
14.4.3	Type checking of messages	756
14.4.4	Message selection	756
14.5	Parallel object-oriented languages	757
14.5.1	Object location	757
14.5.2	Object migration	759
14.5.3	Object replication	760
14.6	Tuple space	761
14.6.1	Avoiding the overhead of associative addressing	762
14.6.2	Distributed implementations of the tuple space	765
14.7	Automatic parallelization	767
14.7.1	Exploiting parallelism automatically	768
14.7.2	Data dependencies	770
14.7.3	Loop transformations	772
14.7.4	Automatic parallelization for distributed-memory machines	773
14.8	Conclusion	776
A	Machine Instructions	783
B	Hints and Solutions to Selected Exercises	785
	References	799
	Index	813

