Modern Compiler Design
2nd edition

Dick Grune[a], Kees van Reeuwijk[a], Henri E. Bal[a], Ceriel J.H. Jacobs[a], and Koen Langendoen[b]

[a]Vrije Universiteit, Amsterdam
[b]Technische Universiteit, Delft

April 10, 2010

Preface to the Second Edition

Ten 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 ten 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 to 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 Section 8.4.

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.

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.

How to do:	in imperative and object-oriented programs (Chapter 11)	in functional programs (Chapter 12)	in logic programs (Chapter 13)	in parallel/ distributed programs (Chapter 14)
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

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 A 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 John Wiley’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 John Wiley'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 forced 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 Jonathan Shipley, Claire Jardine, Sarah Corney, Nicole Burnett, and Georgia King of John Wiley & Sons Ltd, for their help and encouragement in writing this book.

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.

Dick Grune dick@cs.vu.nl, http://www.cs.vu.nl/~dick
Kees van Reeuwijk reeuwijk@cs.vu.nl, http://www.cs.vu.nl/~reeuwijk
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, March 2010

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? 6
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 11
1.2.3 The language for the demo compiler 12
1.2.4 Lexical analysis for the demo compiler 13
1.2.5 Syntax analysis for the demo compiler 14
1.2.6 Context handling for the demo compiler 19
1.2.7 Code generation for the demo compiler 19
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 24
1.3.3 Short-cuts 25
1.4 Compiler architectures 25
1.4.1 The width of the compiler 25
1.4.2 Who's the boss? 27
1.5 Properties of a good compiler 30
1.6 Portability and retargetability 31
1.7 A short history of compiler construction 31
1.7.1 1945–1960: code generation 31
1.7.2 1960–1975: parsing 32
1.7.3 1975–present: code generation and code optimization; paradigms 32
1.8 Grammars 32

1.8.1 The form of a grammar 33
1.8.2 The grammatical production process 34
1.8.3 Extended forms of grammars 35
1.8.4 Properties of grammars 36
1.8.5 The grammar formalism 37
1.9 Closure algorithms 39
1.9.1 A sample problem 39
1.9.2 The components of a closure algorithm 40
1.9.3 An iterative implementation of the closure algorithm 42
1.10 The code forms used in this book 43
1.11 Conclusion 44

I From Program Text to Abstract Syntax Tree 49

2 Program Text to Tokens — Lexical Analysis 51
2.1 Reading the program text 55
2.1.1 Obtaining and storing the text 55
2.1.2 The troublesome newline 56
2.2 Lexical versus syntactic analysis 57
2.3 Regular expressions and regular descriptions 57
2.3.1 Regular expressions and BNF/EBNF 58
2.3.2 Escape characters in regular expressions 58
2.3.3 Regular descriptions 59
2.4 Lexical analysis 60
2.5 Creating a lexical analyzer by hand 60
2.5.1 Optimization by precomputation 65
2.6 Creating a lexical analyzer automatically 68
2.6.1 Dotted items 70
2.6.2 Concurrent search 74
2.6.3 Precomputing the item sets 78
2.6.4 The final lexical analyzer 81
2.6.5 Complexity of generating a lexical analyzer 82
2.6.6 Transitions to S_ω 83
2.6.7 Complexity of using a lexical analyzer 84
2.7 Transition table compression 84
2.7.1 Table compression by row displacement 85
2.7.2 Table compression by graph coloring 87
2.8 Error handling in lexical analyzers 88
2.9 A traditional lexical analyzer generator—*lex* 91
2.10 Lexical identification of tokens 94
2.11 Symbol tables 96
2.12 Macro processing and file inclusion 97
2.12.1 The input buffer stack 98
2.12.2 Conditional text inclusion 101

2.12.3 Generics by controlled macro processing . . . 102
2.13 Conclusion . . . 103

3 Tokens to Syntax Tree — Syntax Analysis 109
3.1 Two classes of parsing methods . . . 111
3.1.1 Principles of top-down parsing . . . 113
3.1.2 Principles of bottom-up parsing . . . 114
3.2 Error detection and error recovery . . . 115
3.3 Creating a top-down parser manually . . . 116
3.3.1 Recursive descent parsing . . . 116
3.3.2 Disadvantages of recursive descent parsing . . . 118
3.4 Creating a top-down parser automatically . . . 119
3.4.1 LL(1) parsing . . . 120
3.4.2 LL(1) conflicts as an asset . . . 125
3.4.3 LL(1) conflicts as a liability . . . 127
3.4.4 The LL(1) push-down automaton . . . 132
3.4.5 Error handling in LL parsers . . . 135
3.4.6 A traditional top-down parser generator—*LLgen* . . . 142
3.5 Creating a bottom-up parser automatically . . . 150
3.5.1 LR(0) parsing . . . 151
3.5.2 The LR push-down automaton . . . 156
3.5.3 LR(0) conflicts . . . 160
3.5.4 SLR(1) parsing . . . 161
3.5.5 LR(1) parsing . . . 163
3.5.6 LALR(1) parsing . . . 167
3.5.7 Making a grammar (LA)LR(1)—or not . . . 169
3.5.8 Generalized LR parsing . . . 172
3.5.9 Error handling in LR parsers . . . 178
3.5.10 A traditional bottom-up parser generator—*yacc/bison* . . . 181
3.6 Recovering grammars from legacy code . . . 183
3.7 Conclusion . . . 189

II Annotating the Abstract Syntax Tree 197

4 Grammar-based Context Handling 199
4.1 Attribute grammars . . . 201
4.1.1 The attribute evaluator . . . 202
4.1.2 Dependency graphs . . . 205
4.1.3 Attribute evaluation . . . 206
4.1.4 Attribute allocation . . . 221
4.1.5 Multi-visit attribute grammars . . . 221
4.1.6 Summary of the types of attribute grammars . . . 231
4.2 Restricted attribute grammars . . . 233
4.2.1 L-attributed grammars . . . 233
4.2.2 S-attributed grammars . . . 237

4.2.3 Equivalence of L-attributed and S-attributed grammars 238
4.3 Extended grammar notations and attribute grammars 239
4.4 Conclusion 240

5 Manual Context Handling 247
5.1 Threading the AST 248
5.2 Symbolic interpretation 253
5.2.1 Simple symbolic interpretation 255
5.2.2 Full symbolic interpretation 259
5.2.3 Last-def analysis 261
5.3 Data-flow equations 261
5.3.1 Setting up the data-flow equations 263
5.3.2 Solving the data-flow equations 265
5.4 Interprocedural data-flow analysis 268
5.5 Carrying the information upstream—live analysis 269
5.5.1 Live analysis by symbolic interpretation 271
5.5.2 Live analysis by data-flow equations 273
5.6 Symbolic interpretation versus data-flow equations 276
5.7 Conclusion 276

III Processing the Intermediate Code 281

6 Interpretation 283
6.1 Interpreters 285
6.2 Recursive interpreters 285
6.3 Iterative interpreters 289
6.4 Conclusion 293

7 Code Generation 297
7.1 Properties of generated code 297
7.1.1 Correctness 298
7.1.2 Speed 298
7.1.3 Size 299
7.1.4 Power consumption 299
7.1.5 About optimizations 300
7.2 Introduction to code generation 301
7.2.1 The structure of code generation 303
7.2.2 The structure of the code generator 304
7.3 Preprocessing the intermediate code 305
7.3.1 Preprocessing of expressions 305
7.3.2 Preprocessing of if-statements and goto statements 306
7.3.3 Preprocessing of routines 307
7.3.4 Procedural abstraction 309
7.4 Avoiding code generation altogether 311
7.5 Code generation proper 312

7.5.1 Trivial code generation . . . 313
7.5.2 Simple code generation . . . 317
7.6 Postprocessing the generated code . . . 332
7.6.1 Peephole optimization . . . 332
7.6.2 Procedural abstraction of assembly code . . . 335
7.7 Machine code generation . . . 337
7.8 Conclusion . . . 338

8 (Dis-)Assemblers, Linkers, and Loaders 345
8.1 The tasks of an assembler . . . 345
8.1.1 The running program . . . 346
8.1.2 The executable code file . . . 346
8.1.3 Object files and linkage . . . 346
8.1.4 Alignment requirements and endianness . . . 348
8.2 Assembler design issues . . . 349
8.2.1 Handling internal addresses . . . 350
8.2.2 Handling external addresses . . . 352
8.3 Linker design issues . . . 352
8.4 Disassembly and decompilation . . . 353
8.4.1 Distinguishing between instructions and data . . . 354
8.4.2 Disassembly with indirection . . . 355
8.4.3 Disassembly with relocation information . . . 358
8.5 Decompilation . . . 358
8.6 Conclusion . . . 363

9 Optimization Techniques 367
9.1 General optimization . . . 368
9.1.1 Compilation by symbolic interpretation . . . 368
9.1.2 Code generation for basic blocks . . . 370
9.1.3 Almost optimal code generation . . . 387
9.1.4 BURS code generation and dynamic programming . . . 387
9.1.5 Register allocation by graph coloring . . . 406
9.1.6 Supercompilation . . . 411
9.1.7 Evaluation of code generation techniques . . . 412
9.1.8 Debugging of code optimizers . . . 413
9.2 Code size reduction . . . 415
9.2.1 General code size reduction techniques . . . 415
9.2.2 Code compression . . . 416
9.2.3 Discussion . . . 420
9.3 Power reduction and energy saving . . . 421
9.3.1 Just compiling for speed . . . 423
9.3.2 Trading speed for power . . . 423
9.3.3 Instruction scheduling and bit switching . . . 424
9.3.4 Register relabeling . . . 426
9.3.5 Avoiding the dynamic scheduler . . . 427
9.3.6 Domain-specific optimizations . . . 427

9.3.7 Discussion . . . 427
9.4 Just-In-Time compilation . . . 428
9.5 Compilers versus computer architectures . . . 428
9.6 Conclusion . . . 430

IV Memory Management 437

10 Explicit and Implicit Memory Management 439
10.1 Data allocation with explicit deallocation . . . 442
10.1.1 Basic memory allocation . . . 443
10.1.2 Optimizations for basic memory allocation . . . 445
10.1.3 Compiler applications of basic memory allocation . . . 447
10.1.4 Embedded-systems considerations . . . 450
10.2 Data allocation with implicit deallocation . . . 451
10.2.1 Basic garbage collection algorithms . . . 452
10.2.2 Preparing the ground . . . 453
10.2.3 Reference counting . . . 460
10.2.4 Mark and scan . . . 463
10.2.5 Two-space copying . . . 468
10.2.6 Compaction . . . 470
10.2.7 Generational garbage collection . . . 472
10.2.8 Implicit deallocation in embedded systems . . . 473
10.3 Conclusion . . . 474

V From Abstract Syntax Tree to Intermediate Code 481

11 Imperative and Object-Oriented Programs 483
11.1 Context handling . . . 485
11.1.1 Identification . . . 486
11.1.2 Type checking . . . 493
11.1.3 Discussion . . . 502
11.2 Source language data representation and handling . . . 503
11.2.1 Basic types . . . 503
11.2.2 Enumeration types . . . 504
11.2.3 Pointer types . . . 504
11.2.4 Record types . . . 508
11.2.5 Union types . . . 510
11.2.6 Array types . . . 510
11.2.7 Set types . . . 513
11.2.8 Routine types . . . 514
11.2.9 Object types . . . 514
11.2.10 Interface types . . . 524
11.3 Routines and their activation . . . 525
11.3.1 Activation records . . . 525

11.3.2 The contents of an activation record . . . 526
11.3.3 Routines . . . 528
11.3.4 Operations on routines . . . 529
11.3.5 Non-nested routines . . . 533
11.3.6 Nested routines . . . 534
11.3.7 Lambda lifting . . . 542
11.3.8 Iterators and coroutines . . . 543
11.4 Code generation for control flow statements . . . 544
11.4.1 Local flow of control . . . 545
11.4.2 Routine invocation . . . 555
11.4.3 Run-time error handling . . . 564
11.5 Code generation for modules . . . 568
11.5.1 Name generation . . . 569
11.5.2 Module initialization . . . 569
11.5.3 Code generation for generics . . . 570
11.6 Conclusion . . . 572

12 Functional Programs 585
12.1 A short tour of Haskell . . . 587
12.1.1 Offside rule . . . 587
12.1.2 Lists . . . 588
12.1.3 List comprehension . . . 589
12.1.4 Pattern matching . . . 590
12.1.5 Polymorphic typing . . . 590
12.1.6 Referential transparency . . . 591
12.1.7 Higher-order functions . . . 592
12.1.8 Lazy evaluation . . . 594
12.2 Compiling functional languages . . . 595
12.2.1 The compiler structure . . . 595
12.2.2 The functional core . . . 597
12.3 Polymorphic type checking . . . 598
12.4 Desugaring . . . 600
12.4.1 The translation of lists . . . 600
12.4.2 The translation of pattern matching . . . 601
12.4.3 The translation of list comprehension . . . 603
12.4.4 The translation of nested functions . . . 605
12.5 Graph reduction . . . 607
12.5.1 Reduction order . . . 611
12.5.2 The reduction engine . . . 613
12.6 Code generation for functional core programs . . . 616
12.6.1 Avoiding the construction of some application spines . . . 619
12.7 Optimizing the functional core . . . 621
12.7.1 Strictness analysis . . . 621
12.7.2 Boxing analysis . . . 627
12.7.3 Tail calls . . . 628

12.7.4 Accumulator transformation 629
12.7.5 Limitations 631
12.8 Advanced graph manipulation 631
12.8.1 Variable-length nodes 631
12.8.2 Pointer tagging 632
12.8.3 Aggregate node allocation 632
12.8.4 Vector apply nodes 633
12.9 Conclusion 633

13 Logic Programs **641**
13.1 The logic programming model 643
13.1.1 The building blocks 643
13.1.2 The inference mechanism 645
13.2 The general implementation model, interpreted 646
13.2.1 The interpreter instructions 647
13.2.2 Avoiding redundant goal lists 650
13.2.3 Avoiding copying goal list tails 651
13.3 Unification 651
13.3.1 Unification of structures, lists, and sets 652
13.3.2 The implementation of unification 654
13.3.3 Unification of two unbound variables 657
13.4 The general implementation model, compiled 659
13.4.1 List procedures 660
13.4.2 Compiled clause search and unification 662
13.4.3 Optimized clause selection in the WAM 667
13.4.4 Implementing the "cut" mechanism 671
13.4.5 Implementing the predicates assert and retract 672
13.5 Compiled code for unification 677
13.5.1 Unification instructions in the WAM 679
13.5.2 Deriving a unification instruction by manual partial evaluation 680
13.5.3 Unification of structures in the WAM 682
13.5.4 An optimization: read/write mode 688
13.5.5 Further unification optimizations in the WAM 690
13.6 Conclusion 693

14 Parallel and Distributed Programs **699**
14.1 Parallel programming models 702
14.1.1 Shared variables and monitors 703
14.1.2 Message passing models 704
14.1.3 Object-oriented languages 706
14.1.4 The Linda Tuple space 707
14.1.5 Data-parallel languages 709
14.2 Processes and threads 709
14.3 Shared variables 711
14.3.1 Locks 712
14.3.2 Monitors 712

14.4 Message passing . . . 714
14.4.1 Locating the receiver . . . 714
14.4.2 Marshaling . . . 715
14.4.3 Type checking of messages . . . 716
14.4.4 Message selection . . . 716
14.5 Parallel object-oriented languages . . . 717
14.5.1 Object location . . . 717
14.5.2 Object migration . . . 719
14.5.3 Object replication . . . 719
14.6 Tuple space . . . 721
14.6.1 Avoiding the overhead of associative addressing . . . 721
14.6.2 Distributed implementations of the tuple space . . . 724
14.7 Automatic parallelization . . . 726
14.7.1 Exploiting parallelism automatically . . . 727
14.7.2 Data dependencies . . . 729
14.7.3 Loop transformations . . . 731
14.7.4 Automatic parallelization for distributed-memory machines . . . 732
14.8 Conclusion . . . 735

A Hints and Solutions to Selected Exercises **741**

B Machine Instructions **753**

Bibliography **755**

Index **771**