Modern Compiler Design

Dick Grune • Kees van Reeuwijk • Henri E. Bal
Ceriel 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

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

Dick Grune
Kees van Reeuwijk
Henri E. Bal
Ceriel J.H. Jacobs

Delft,
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