Preface

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 mature students in one of their final years, who have at least used a compiler occasionally and 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.

The structure of the book

This book is conceptually divided into two parts. The first, comprising Chapters 1 through 5, 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 6 through 9, covers the specific techniques required by the various programming paradigms. The interactions between the parts of the book are outlined in the table on the next page.

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 4.

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 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

How to do:	in imperative and object-oriented programs (Chapter 6)	in functional programs (Chapter 7)	in logic programs (Chapter 8)	in parallel/ distributed programs (Chapter 9)
analysis (Chapter 2)	–	–	–	–
context handling (Chapter 3)	identifier identification	polymorphic type checking	static rule matching	Linda static analysis
synthesis (Chapter 4)	code for while-statement	code for list comprehension	structure unification	marshaling
run-time systems (no chapter)	stack	reduction machine	Warren Abstract Machine	replication

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 an introduction to formal grammars.

Chapter 2 treats the 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.

Chapter 3 covers 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.

Chapter 4 covers the synthesis phase of a compiler, covering both interpretation and code generation. The section on code generation is mainly concerned with machine code generation; the intermediate code required for paradigm-specific constructs is treated in Chapters 6 through 9.

Chapter 5 concerns memory management techniques, both for use in the compiler and in the generated program.

Chapters 6 through 9 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 6.

Appendix A discusses a possible but experimental method of object-oriented compiler construction, in which an attempt is made to exploit object-oriented concepts to simplify compiler design.

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, and bottom-up code generation. 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. Still, 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.

An attempt at justifying the outline code used in this book to specify algorithms can be found in Section 1.11.

Additional material, including more answers to exercises, and all diagrams and all code from the book, are available through John Wiley's Web 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. Depending on the maturity of the audience, an introductory, more traditional course can be obtained by including, for example,

Chapter 1;
Chapter 2 up to 2.1.7; 2.1.10; 2.1.11; 2.2 up to 2.2.4.5; 2.2.5 up to 2.2.5.7;
Chapter 3 up to 3.1.2; 3.1.7 up to 3.1.10; 3.2 up to 3.2.2.2; 3.2.3;
Chapter 4 up to 4.1; 4.2 up to 4.2.4.3; 4.2.6 up to 4.2.6.4; 4.2.11;
Chapter 5 up to 5.1.1.1; 5.2 up to 5.2.4;
Chapter 6 up to 6.2.3.2; 6.2.4 up to 6.2.10; 6.4 up to 6.4.2.3.

A more advanced course would include all of Chapters 1 to 6, excluding Section 3.1. This could be augmented by one of Chapters 7 to 9 and perhaps Appendix A.

An advanced course would skip much of the introductory material and concentrate on the parts omitted in the introductory course: Section 3.1 and all of Chapters 5 to 9, plus Appendix A.

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