

Compiler Design

Reinhard Wilhelm · Helmut Seidl

Compiler Design

Virtual Machines

 Springer

Prof. Dr. Reinhard Wilhelm
Universität des Saarlandes
FB Informatik
Postfach 15 11 50
66041 Saarbrücken Saarland
Germany
wilhelm@cs.uni-sb.de

Prof. Dr. Helmut Seidl
TU München
Fak. Informatik
Boltzmannstr. 3
85748 Garching
Germany
seidl@in.tum.de

ISBN 978-3-642-14908-5 e-ISBN 978-3-642-14909-2
DOI 10.1007/978-3-642-14909-2
Springer Heidelberg Dordrecht London New York

ACM Codes: D.1, D.3, D.2

© Springer-Verlag Berlin Heidelberg 2010

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, 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.

Cover design: KuenkelLopka GmbH

Printed on acid-free paper

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

For Margret, Hannah, Eva, Barbara

R.W.

For Kerstin and Anna

H.S.

Preface

Compilers for high-level programming languages are software systems which are both large and complex. Nonetheless, they have particular characteristics that differentiate them from the majority of other software systems.

Their functionality is (almost) completely well-defined. Ideally, there exist completely formal, or at least rather precise, specifications of the source and target languages. Often additional specifications of the interfaces to the operating system, to programming environments, and to other compilers and libraries are available.

The compilation task can be naturally divided into subtasks. This subdivision results in a modular structure, which, by the way, also leads to a canonical structure of the common compiler design books.

Already in the Fifties it was recognized that the implementation of application systems directly in machine language is both difficult and error-prone, leading to programs that become obsolete as quickly as the computers they were developed for. With the development of higher level machine independent programming languages came the need to offer compilers that are able to translate programs of such programming languages into machine language.

Given this basic challenge, the different subtasks of compilation have been the subject of intensive research since the Fifties. For the subtask of syntactic analysis of programs, concepts from formal language and automata theory, such as regular languages, finite automata, context-free grammars, and pushdown automata were borrowed and were further developed in view of the particular use. The theoretical foundation of the problem was so well-developed that the realization of the components required for syntax analysis could be (almost) completely automated: instead of being implemented *by hand* these components are mainly generated from specifications, in this case context-free grammars. Such automatic generation is also the aim for other components of compilers, although it has not always been achieved yet.

This book is not intended to be a cookbook for compilers. Thus, one will not find recipes like: “To build a compiler of source language X into machine language Y, take ... “. Our presentation instead reflects the special characteristics of compiler design, specially the existence of precise specifications of the subtasks. We invest

some effort to understand these precisely and to provide adequate concepts for their systematic treatment. Ideally, those concepts can build the foundation of a process of automatic generation.

This book is intended for students of Informatics. Knowledge of at least one imperative programming language is assumed. For the chapters on the translation of functional and logic programming languages it is certainly helpful to know a modern functional language and the basic concepts of the logic language PROLOG. On the other hand, these chapters can help to achieve a more profound understanding of such programming languages.

Structure of This Book

For the new edition of the book Wilhelm/Maurer: *Compiler Design*, we decided to divide the contents in multiple volumes. This first volume describes *what* a compiler does: thus, what correspondence it establishes between a source and a target program. To achieve this, for each of an imperative, functional, logic, and object-oriented programming language, a suitable *virtual* machine (called *abstract* machine in previous editions) is specified, and the compilation of programs of each source language into the language of the corresponding virtual machine is presented in detail.

The virtual machines of the previous edition have been fully revised and modernized with the aim of simplifying the translation schemes and, if necessary, to complete them. Compared to before, the various chosen architectures and instruction sets have been made more uniform to clearly highlight the similarities, as well as the differences, of the language concepts. Perhaps the most obvious, if not the most important, feature that readers of earlier editions will easily recognize is that the stack of virtual machines are growing upwards and no longer from *top to bottom*.

Fragments of real programming languages have been used in all example programming languages. As the imperative source language, the programming language PASCAL has been replaced with C – a choice for a more realistic approach. A subset of C++ serves again as object-oriented language. Compared to the presentation of the second edition, however, a detailed discussion of multiple inheritance has been omitted.

In this book, the starting point of the translations of imperative, functional, logic, and object-oriented programs is always a structured internal representation of the source program, for which already simple additional information, such as scope of variables or type information has been added. Later we will call such an analyzed source program an *annotated abstract syntax* of the program.

In the subsequent volumes, the *how* of the compilation process will be described. There, we deal with the question of how to divide the compilation process into a sequence of phases: which tasks each individual phase has to cover, which techniques are used in them, how to describe formally what they do, and how, perhaps, a compiler module can be automatically created out of such a specification.

Acknowledgments

Besides the coworkers of previous editions, we would like to thank all the students who participated in courses again and again using different versions of virtual machines and who gave us invaluable feedback. The visualization of the virtual machines by Peter Ziewer added a lot to the comprehension. For subtle insight into the semantics of C++ and Java we thank Thomas Gawlitza and Michael Petter. Special thanks go to Jörg Herter, who inspected multiple versions of the book carefully for inconsistencies and who drew our attention to multiple mistakes and oddities.

In the meantime we wish the eager reader lots of fun with this volume and hope that the book will whet her appetite to quickly create her own compiler for the favorite programming language.

Saarbrücken and München, May 2010

Reinhard Wilhelm, Helmut Seidl

Further material for this book can be found on the following Web page:

<http://www2.informatik.tu-muenchen.de/~seidl/compilers/>

Contents

1	Introduction	1
1.1	High-Level Programming Languages	1
1.2	Implementation of Programming Languages	2
1.2.1	Interpreters	2
1.2.2	Compilers	3
1.2.3	Real and Virtual Machines	4
1.2.4	Combined Compilation and Interpretation	4
1.3	General References	5
2	Imperative Programming Languages	7
2.1	Language Concepts and Their Compilation	7
2.2	The Architecture of the C-Machine	8
2.3	Simple Expressions and Assignments	9
2.4	Statements and Statement Sequences	15
2.5	Conditional and Iterative Statements	16
2.6	Memory Allocation for Variables of Basic Types	22
2.7	Memory Allocation for Arrays and Structures	23
2.8	Pointers and Dynamic Memory Allocation	27
2.9	Functions	32
2.9.1	Memory Organization of the C-Machine	35
2.9.2	Dealing with Local Variables	37
2.9.3	Function Call and Return	40
2.10	Translation of Programs	45
2.11	Exercises	48
2.12	List of CMA Registers	55
2.13	List of Code Functions of the CMA	55
2.14	List of CMA Instructions	55
2.15	References	55

3	Functional Programming Languages	57
3.1	Basic Concepts and Introductory Examples	57
3.2	A Simple Functional Programming Language	59
3.3	The Architecture of the MAMA	63
3.4	Translation of Simple Expressions	66
3.5	Access to Variables	68
3.6	<i>let</i> Expressions	73
3.7	Function Definitions	74
3.8	Function Application	76
3.9	Under- and Oversupply with Arguments	80
3.10	Recursive Variable Definitions	83
3.11	Closures and Their Evaluation	86
3.12	Optimization I: Global Variables	89
3.13	Optimization II: Closures	90
3.14	Translating Program Expressions	91
3.15	Structured Data	92
3.15.1	Tuples	92
3.15.2	Lists	93
3.15.3	Closures for Tuples and Lists	96
3.16	Optimization III: Last Calls	97
3.17	Exercises	99
3.18	List of MAMA Registers	103
3.19	List of Code Functions of the MAMA	103
3.20	List of MAMA Instructions	104
3.21	References	104
4	Logic Programming Languages	105
4.1	The Language PROL	105
4.2	The Architecture of the W1M	108
4.3	Allocation of Terms in the Heap	109
4.4	The Translation of Literals	114
4.5	Unification	115
4.6	Clauses	126
4.7	The Translation of Predicates	127
4.7.1	Backtracking	127
4.7.2	Putting It All Together	130
4.8	The Finalization of Clauses	131
4.9	Queries and Programs	134
4.10	Optimization I: Last Goals	135
4.11	Optimization II: Trimming of Stack Frames	138
4.12	Optimization III: Clause Indexing	140
4.13	Extension: The Cut Operator	142
4.14	Digression: Garbage Collection	144
4.15	Exercises	149
4.16	List of W1M Registers	152

4.17	List of Code Functions of the WiM	152
4.18	List of WiM Instructions	152
4.19	References	153
5	Object-Oriented Programming Languages	155
5.1	Concepts of Object-Oriented Languages	155
5.1.1	Objects	155
5.1.2	Object Classes	156
5.1.3	Inheritance	157
5.1.4	Genericity	158
5.1.5	Information Encapsulation	158
5.1.6	Summary	159
5.2	An Object-Oriented Extension of C	159
5.3	The Memory Organization for Objects	160
5.4	Method Calls	163
5.5	The Definition of Methods	165
5.6	The Use of Constructors	166
5.7	The Definition of Constructors	168
5.8	Perspective: Multiple Inheritance	169
5.9	Exercises	171
5.10	List of Additional Registers	177
5.11	CMa Instructions for Objects	177
5.12	References	177
	References	179
	Index	183