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

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 is 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