To Margret, Hannah, Eva, and Barbara
R. W.

To Kerstin and Anna
H. S.

To Kerstin, Charlotte, and Marlene
S. H.

Preface

Compilers for high-level programming languages are large and complex software systems. They have, however, several distinct properties by which they favorably differ from most other software systems. Their semantics is (almost) well defined. Ideally, completely formal or at least precise descriptions exist both of the source and the target languages. Often, additional descriptions are provided of the interfaces to the operating system, to programming environments, to other compilers, and to program libraries.

The task of compilation can be naturally decomposed into subtasks. This decomposition results in a modular structure which, by the way, is also reflected in the structure of most books about compilers.

As early as the 1950s, it was observed that implementing application systems directly as machine code is both difficult and error-prone, and results in programs which become outdated as quickly as the computers for which they have been developed. High-level machine-independent programming languages, on the other hand, immediately made it mandatory to provide compilers, which are able to automatically translate high-level programs into low-level machine code.

Accordingly, the various subtasks of compilation have been subject to intensive research. For the subtasks of lexical and syntactic analysis of programs, concepts like regular expressions, finite automata, context-free grammars and pushdown automata have been borrowed from the theory of automata and formal languages and adapted to the particular needs of compilers. These developments have been extremely successful. In the case of syntactic analysis, they led to fully automatic techniques for generating the required components solely from corresponding specifications, i.e., context-free grammars. Analogous automatic generation techniques would be desirable for further components of compilers as well, and have, to a certain extent, also been developed.

The current book does not attempt to be a cookbook for compiler writers. Accordingly, there are no recipes such as "in order to construct a compiler from source language X into target language Y, take ...". Instead, our presentation elaborates on the fundamental aspects such as the technical concepts, the specification

formalisms for compiler components and methods how to systematically derive implementations. Ideally, this approach may result in fully automatic generator tools.

The book is written for students of computer science. Some knowledge about an object-oriented programming language such as JAVA and very basic principles of a functional language such as OCAML or SML are required. Knowledge about formal languages or automata is useful, but is not mandatory as the corresponding background is provided within the presentation.

Organization of the Book

For the new edition of the book *Wilhelm/Maurer: Compiler Construction*, we decided to distribute the contents over several volumes. The first volume *Wilhelm/Seidl: Compiler Design – Virtual Machines* speaks about *what* a compiler does, i.e., which correspondence it realizes between source program and target program. This is exemplified for an imperative, a functional, a logic-based and an object-oriented language. For each language, a corresponding *virtual* machine (in earlier editions also called *abstract* machine) is designed, and implementations are provided for each construct of the programming language by means of the instructions of the virtual machine.

The subsequent three volumes then describe *how* the compilation process is structured and organized. Each corresponds to one phase of compilation, namely to *syntactic and semantic* analysis, to *program transformation*, and to *code generation*. The present volume is dedicated to the analysis phase, realized by the *front-end* of the compiler. We also briefly discuss how the compilation process as a whole can be organized into phases, which tasks should be assigned to different phases, and which techniques can be used for their implementation.

Further material related to this book is provided on the webpage: http://www2.informatik.tu-_muenchen.de/~seidl/compilers/

Acknowledgement

Besides the helpers from former editions, we would like to thank Michael Jacobs and Jörg Herter for carefully proofreading the chapter on syntactic analysis. When revising the description of the recursive descent parser, we also were supported by Christoph Mallon and his invaluable experience with practical parser implementations.

We hope that our readers will enjoy the present volume and that our book may encourage them to realize compilers of their own for their favorite programming languages.

Saarbrücken and München, August 2012 Reinhard Wilhelm, Helmut Seidl and Sebastian Hack

http://www.springer.com/978-3-642-17539-8

Compiler Design
Syntactic and Semantic Analysis
Wilhelm, R.; Seidl, H.; Hack, S.
2013, X, 225 p., Hardcover
ISBN: 978-3-642-17539-8