

Preface

The study of programming languages and their implementations is a central theme of computer science. The design of a compiler—a program translating programs written in a high-level language into semantically equivalent programs in another language, typically machine code—is influenced by many aspects of computer science. The compiler allows us to program in high-level languages, and it provides a layer of abstraction so that we do not have to worry when programming about the complex details of the underlying hardware.

The designer of any compiler obviously needs to know the details of both the source language being translated and the language being generated, usually some form of machine code for the target machine. For non-trivial languages, the compiler is itself a non-trivial program and should really be designed by following a standard structure. Describing this standard structure is one of the key aims of this book.

The design of compilers is influenced by the characteristics of formal language specifications, by automata theory, by parsing algorithms, by processor design, by data structure and algorithm design, by operating system services, by the target machine instruction set and other hardware features, by implementation language characteristics and so on, as well as by the needs of the compilers' users. Coding a compiler can be a daunting software task, but the process is greatly simplified by making use of the approaches, experiences, recommendations and algorithms of other compiler writers.

Why Study Compiler Design?

Why should compiler design be studied? Why is this subject considered to be an important component of the education of a computer scientist? After all, only a small proportion of software engineers are employed on large-scale, traditional compiler projects. But knowing something about what happens within a compiler can have many benefits. Understanding the technology and limitations of a

compiler is important knowledge for any user of a compiler. Compilers are complex pieces of code and an awareness of how they work can be very helpful. The algorithms used in a compiler are relevant to many other application areas such as aspects of text decoding and analysis and the development of command-driven interfaces. The need for simple domain-specific languages occurs frequently and the knowledge of compiler design can facilitate their rapid implementation.

Writing a simple compiler is an excellent educational project and enhances skills in programming language understanding and design, data structure and algorithm design and a wide range of programming techniques. Understanding how a high-level language program is translated into a form that can be executed by the hardware gives a good insight into how a program will behave when it runs, where the performance bottlenecks will be, the costs of executing individual high-level language statements and so on. Studying compiler design makes you a better programmer.

Why Another Book?

Why is there now yet another book on compiler design? Many detailed and comprehensive textbooks in this field have already been published. This book is a little different from most of the others. Hopefully, it presents key aspects of the subject in an accessible way, using a practical approach. The algorithms shown are all capable of straightforward implementation in almost any programming language, and the reader is strongly encouraged to read the text and in parallel produce code for implementations of the compiler modules being described. These practical examples are concentrated in areas of compiler design that have general applicability. For example, the algorithms shown for performing lexical and syntax analysis are not restricted for use in compilers alone. They can be applied to the analysis required in a wide range of text-based software.

The field of programming language implementation is huge and this book covers only a small part of it. Just the basic principles, potentially applicable to all compilers, are explained in a practical way.

What's in this Book?

This book introduces the topic of compiler construction using many programmed examples, showing code that could be used in a range of compiler and compiler-related projects. The code examples are nearly all written in C, a mature language and still in widespread use. Translating them into another programming language should not cause any real difficulty. Many existing compiler projects are written in C, many new compiler projects are being written in C and there are many compiler construction tools and utilities designed to support compiler

implementations in C. Character handling and dynamic data structure management are well-handled by C. It is a good language for compiler construction. Therefore, it may have seemed appropriate to choose the construction of a C compiler as a central project for this textbook. However, this would not have been sensible because it is a huge project, and the key algorithms of language analysis and translation would be overwhelmed by the detail necessary to deal with the numerous complexities of a “real” programming language, even one regarded as being simpler than many.

This book is primarily about compiler construction, and it is not specifically about the use of compiler-related algorithms in other application areas. Hopefully, though, there is enough information in the analysis chapters to show how these standard grammar-based techniques can be applied very much more widely.

Although many examples in this book are taken from code that may appear in a complete C compiler, the emphasis is on the development of a compiler for the DL language. This is a very simple language developed for the needs of this book from languages used in a series of practical exercises from various undergraduate and postgraduate compiler construction courses presented by the author. The syntax of DL is loosely based on a subset of C, but with many restrictions. In particular, there is just one data type (the integer), and although functions are supported, their functionality is rather restricted. Nevertheless, DL is sufficiently powerful to be usable for real problems. The syntax of DL is presented in the appendix.

The widely-available *flex* and *bison* tools are introduced, and their use in practical implementations is discussed, especially in the context of generating a compiler for DL. These particular packages provide a good insight into the benefits offered by the many powerful compiler generation tools now available.

The software examples in this book were developed and tested on systems running Fedora Linux on an x86-64 architecture. The C compiler used was GCC. Machine and operating system dependencies are probably inevitable, but any changes needed to move this code to a different computer or operating system should be comparatively minor.

The code examples are concentrated on the compiler’s front-end. Code for intermediate code optimisation, target machine code generation and optimisation tends to be long, complex and often overwhelmed by target machine detail. Hence, code examples from the back-end are largely avoided in this book so that no introduction to or detailed discussion of assembly language programming is included. Instead, the text presents the basic principles of back-end design from which code generators for diverse target architectures can be developed. References are given to sources providing further algorithm examples.

The source code of a complete DL compiler is *not* presented in this book. The real reason for this is that there is an underlying assumption that one of the most important practical exercises of the book is to produce a complete compiler for DL. A large number of code examples taken from a compiler are included in the text to illustrate the principles being described so that the reader will not be coding from scratch.

How Should this Book be Used?

This book can be used to accompany taught courses in programming language implementation and compiler design, and it can also be used for self-study. There is an assumption that students using this book will have some programming skills but not necessarily significant experience of writing large software systems. A working understanding of basic data structures such as trees is essential. The examples in the book are coded in C, but a comprehensive knowledge of C is really not required to understand these examples and the accompanying text. A basic knowledge of computer hardware is also assumed, including just the rudiments of the principles of assembly-level programming.

Each chapter ends with a few exercises. They vary a great deal in complexity. Some involve just a few minutes of thought, whereas others are major programming projects. Many of the exercises are appropriate for group discussion and some may form the basis of group projects involving code implementation as well as research.

It is especially important to make the most of the practical aspects of this subject by coding parts of a compiler as the book is being read. This will help greatly to alleviate boredom and will hugely help with the process of understanding. For example, for the newcomer to recursive descent parsing, the power and elegance of the technique can only be fully appreciated when a working implementation has been written.

The obvious project work associated with this book is to write a complete compiler for the DL language. Assuming that a simple target machine is chosen, the project is of a reasonable size and can fit well into an average size university or college course. Extensions to such a compiler by including optimisation and register allocation can follow in a more advanced course. The project can be taken even further by developing the DL compiler into a complete C compiler, but the time required to do this should not be underestimated. Writing a simple compiler following the steps described in this book is not a huge task. But it is important not to abandon the standard techniques. I have seen some students getting into major difficulties with the implementation of their compilers, coded using a “much better algorithm” of their own devising! The correct approach is reliable and really does involve a careful and systematic implementation with extensive testing of each module before it is combined with others.

Although the DL language is used as an example in most of the chapters, this book is not intended to be a tutorial guide for writing DL compilers. Its aims are much broader than this—it tries to present the principles of compiler design and the implementation of certain types of programming language, and where appropriate, DL-targeted examples are presented. Should the reader want to accept the challenge of writing a complete DL compiler (and I would certainly recommend this), then the key practical information about lexical and syntax analysis is easy to find in Chaps. 3 and 5 and semantic analysis in Chap. 6. There is then some information about DL-specific issues of code generation in Chap. 8.

Turning the compiler construction project into a group project worked well. Programming teams can be made responsible for the construction of a complete compiler. The development can be done entirely by members of the team or it may be possible for teams to trade with other teams. This is a good test of well-documented interfaces. Producing a set of good test programs to help verify that a compiler works is an important part of the set of software modules produced by each team.

Generating standard-format object code files for real machines in an introductory compilers course may be trying to go a little too far. Generating assembly code for a simple processor or for a simple subset of a processor's features is probably a better idea. Coding an emulator for a simple target machine is not difficult—just use the techniques described in this book, of course. Alternatively, there are many virtual target architecture descriptions with corresponding emulator software freely available. The MIPS architecture, with the associated SPIM software [1], despite its age, is still very relevant today and is a good target for code generation. The pleasure of writing a compiler that produces code that actually runs is considerable!

Acknowledgement This book is loosely based on material presented in several undergraduate and postgraduate lecture courses at the University of Sussex. I should like to thank all the students who took these courses and who shared my enthusiasm for the subject. Over the years, I watched thousands of compilers being developed and discovered which parts of the process they usually found difficult. I hope that I have addressed those issues properly in this book.

Thanks also go to my colleagues at the University of Sussex—in particular to all the staff and students in the Foundations of Software Systems research group who provided such a supportive and stimulating work environment. Particular thanks go to Bernhard Reus for all his suggestions and corrections.

I'm really grateful to Ian Mackie, the UTICS series editor, and to Helen Desmond at Springer for their constant enthusiasm for the book. They always provided advice and support just when it was needed.

Finally, and most important, I should like to thank Wendy, Helen and Jonathan for tolerating my disappearing to write and providing unfailing encouragement.

Sussex, UK

Des Watson

Reference

1. Larus JR (1990) SPIM S20: a MIPS R2000 simulator. Technical Report 966. University of Wisconsin-Madison, Madison, WI, Sept 1990



<http://www.springer.com/978-3-319-52787-1>

A Practical Approach to Compiler Construction

Watson, D.

2017, XV, 254 p. 26 illus., Softcover

ISBN: 978-3-319-52787-1