PREFACE

This book has been written to support a practically oriented course in programming language translation for senior undergraduates in Computer Science. More specifically, it is aimed at students who are probably quite competent in the art of imperative programming (for example, in C++, Pascal, or Modula-2), but whose mathematics may be a little weak; students who require only a solid introduction to the subject, so as to provide them with insight into areas of language design and implementation, rather than a deluge of theory which they will probably never use again; students who will enjoy fairly extensive case studies of translators for the sorts of languages with which they are most familiar; students who need to be made aware of compiler writing tools, and to come to appreciate and know how to use them. It will hopefully also appeal to a certain class of hobbyist who wishes to know more about how translators work.

The reader is expected to have a good knowledge of programming in an imperative language and, preferably, a knowledge of data structures. The book is practically oriented, and the reader who cannot read and write code will have difficulty following quite a lot of the discussion. However, it is difficult to imagine that students taking courses in compiler construction will not have that sort of background!

There are several excellent books already extant in this field. What is intended to distinguish this one from the others is that it attempts to mix theory and practice in a disciplined way, introducing the use of attribute grammars and compiler writing tools, at the same time giving a highly practical and pragmatic development of translators of only moderate size, yet large enough to provide considerable challenge in the many exercises that are suggested.

Overview

The book starts with a fairly simple overview of the translation process, of the constituent parts of a compiler, and of the concepts of porting and bootstrapping compilers. This is followed by a chapter on machine architecture and machine emulation, as later case studies make extensive use of code generation for emulated machines, a very common strategy in introductory courses. The next chapter introduces the student to the notions of regular expressions, grammars, BNF and EBNF, and the value of being able to specify languages concisely and accurately.

Two chapters follow that discuss simple features of assembler language, accompanied by the development of an assembler/interpreter system which allows not only for very simple assembly, but also for conditional assembly, macro-assembly, error detection, and so on. Complete code for such an assembler is presented in a highly modularized form, but with deliberate scope left for extensions, ranging from the trivial to the extensive.

Three chapters follow on formal syntax theory, parsing, and the manual construction of scanners and parsers. The usual classifications of grammars and restrictions on practical grammars are discussed in some detail. The material on parsing is kept to a fairly simple level, but with a thorough discussion of the necessary conditions for LL(1) parsing. The parsing method treated in most detail is the method of recursive descent, as is found in many Pascal compilers; LR parsing is only briefly discussed.

The next chapter is on syntax directed translation, and stresses to the reader the importance and usefulness of being able to start from a context-free grammar, adding attributes and actions that allow for the manual or mechanical construction of a program that will handle the system that it defines. Obvious applications come from the field of translators, but applications in other areas such as simple database design are also used and suggested.

The next two chapters give a thorough introduction to the use of Coco/R, a compiler generator based on L- attributed grammars. Besides a discussion of Cocol, the specification language for this tool, several in-depth case studies are presented, and the reader is given some indication of how parser generators are themselves constructed.

The next two chapters discuss the construction of a recursive descent compiler for a simple Pascal-like source language, using both hand-crafted and machine-generated techniques. The compiler produces pseudo-code for a hypothetical stack-based computer (for which an interpreter was developed in an earlier chapter). "On the fly" code generation is discussed, as well as the use of intermediate tree construction.

The last chapters extend the simple language (and its compiler) to allow for procedures and functions, demonstrate the usual stack-frame approach to storage management, and go on to discuss the implementation of simple concurrent programming. At all times the student can see how these are handled by the compiler/interpreter system, which slowly grows in complexity and usefulness until the final product enables the development of quite sophisticated programs.

The text abounds with suggestions for further exploration, and includes references to more advanced texts where these can be followed up. Wherever it seems appropriate the opportunity is taken to make the reader more aware of the strong and weak points in topical imperative languages. Examples are drawn from several languages, such as Pascal, Modula-2, Oberon, C, C++, Edison and Ada.

Support software

An earlier version of this text, published by Addison-Wesley in 1986, used Pascal throughout as a development tool. By that stage Modula-2 had emerged as a language far better suited to serious programming. A number of discerning teachers and programmers adopted it enthusiastically, and the material in the present book was originally and successfully developed in Modula-2. More recently, and especially in the USA, one has witnessed the spectacular rise in popularity of C++, and so as to reflect this trend, this has been adopted as the main language used in the present text. Although offering much of value to skilled practitioners, C++ is a complex language. As the aim of the text is not to focus on intricate C++programming, but compiler construction, the supporting software has been written to be as clear and as simple as possible. Besides the C++ code, complete source for all the case studies has also been provided on an accompanying IBM-PC compatible diskette in Turbo Pascal and Modula-2, so that readers who are proficient programmers in those languages but only have a reading knowledge of C++ should be able to use the material very successfully.

Appendix A gives instructions for unpacking the software provided on the diskette and installing it on a reader’s computer. In the same appendix will be found the addresses of various sites on the Internet where this software (and other freely available compiler construction software) can be found in various formats. The software provided on the diskette includes

- Emulators for the two virtual machines described in Chapter 4 (one of these is a simple accumulator based machine, the other is a simple stack based machine).
- The one- and two-pass assemblers for the accumulator based machine, discussed in Chapter 6.
- A macro assembler for the accumulator-based machine, discussed in Chapter 7.
- Three executable versions of the Coco/R compiler generator used in the text and described in detail in Chapter 12, along with the frame files that it needs. (The three versions produce Turbo Pascal, Modula-2 or C/C++ compilers)
- Complete source code for hand-crafted versions of each of the versions of the Clang compiler that is developed in a layered way in Chapters 14 through 18. This highly modularized code comes with an "on the fly" code generator, and also with an alternative code generator that builds and then walks a tree representation of the intermediate code.
- Cocol grammars and support modules for the numerous case studies throughout the book that use Coco/R. These include grammars for each of the versions of the Clang compiler.
- A program for investigating the construction of minimal perfect hash functions (as discussed in Chapter 14).
- A simple demonstration of an LR parser (as discussed in Chapter 10).

Use as a course text

The book can be used for courses of various lengths. By choosing a selection of topics it could be used on courses as short as 5-6 weeks (say 15-20 hours of lectures and 6 lab sessions). It could also be used to support longer and more intensive courses. In our university, selected parts of the material have been successfully used for several years in a course of about 35 - 40 hours of lectures with strictly controlled and structured, related laboratory work, given to students in a pre-Honours year. During that time the course has evolved significantly, from one in which theory and formal specification played a very low key, to the present stage where students have come to appreciate the use of specification and syntax-directed compiler-writing systems as very powerful and useful tools in their armoury.

It is hoped that instructors can select material from the text so as to suit courses tailored to their own interests, and to their students' capabilities. The core of the theoretical material is to be found in Chapters 1, 2, 5, 8, 9, 10 and 11, and it is suggested that this material should form part of any course based on the book. Restricting the selection of material to those chapters would deny the student the very important opportunity to see the material in practice, and at least a partial selection of the material in the practically oriented chapters should be studied. However, that part of the material in Chapter 4 on the accumulator-based machine, and Chapters 6 and 7 on writing assemblers for this machine could be omitted without any loss of continuity. The development of the small Clang compiler in Chapters 14 through 18 is handled in a way that allows for the later sections of Chapter 15, and for Chapters 16 through 18 to be omitted if time is short. A very wide variety of laboratory exercises can be selected from those suggested as exercises, providing the students with both a challenge, and a feeling of satisfaction when they rise to meet that challenge. Several of these exercises are based on the idea of developing a small compiler for a language

similar to the one discussed in detail in the text. Development of such a compiler could rely entirely on traditional hand-crafted techniques, or could rely entirely on a tool-based approach (both approaches have been successfully used at our university). If a hand-crafted approach were used, Chapters 12 and 13 could be omitted; Chapter 12 is largely a reference manual in any event, and could be left to the students to study for themselves as the need arose. Similarly, Chapter 3 falls into the category of background reading.

At our university we have also used an extended version of the Clang compiler as developed in the text (one incorporating several of the extensions suggested as exercises) as a system for students to study concurrent programming *per se*, and although it is a little limited, it is more than adequate for the purpose. We have also used a slightly extended version of the assembler program very successfully as our primary tool for introducing students to the craft of programming at the assembler level.

Limitations

It is, perhaps, worth a slight digression to point out some things which the book does not claim to be, and to justify some of the decisions made in the selection of material.

In the first place, while it is hoped that it will serve as a useful foundation for students who are already considerably more advanced, a primary aim has been to make the material as accessible as possible to students with a fairly limited background, to enhance the background, and to make them somewhat more critical of it. In many cases this background is still Pascal based; increasingly it is tending to become C++ based. Both of these languages have become rather large and complex, and I have found that many students have a very superficial idea of how they really fit together. After a course such as this one, many of the pieces of the language jigsaw fit together rather better.

When introducing the use of compiler writing tools, one might follow the many authors who espouse the classic lex/yacc approach. However, there are now a number of excellent LL(1) based tools, and these have the advantage that the code which is produced is close to that which might be hand-crafted; at the same time, recursive descent parsing, besides being fairly intuitive, is powerful enough to handle very usable languages.

That the languages used in case studies and their translators are relative toys cannot be denied. The Clang language of later chapters, for example, supports only integer variables and simple one-dimensional arrays of these, and has concurrent features allowing little beyond the simulation of some simple textbook examples. The text is not intended to be a comprehensive treatise on systems programming in general, just on certain selected topics in that area, and so very little is said about native machine code generation and optimization, linkers and loaders, the interaction and relationship with an operating system, and so on. These decisions were all taken deliberately, to keep the material readily understandable and as machine-independent as possible. The systems may be toys, but they are very usable toys! Of course the book is then open to the criticism that many of the more difficult topics in translation (such as code generation and optimization) are effectively not covered at all, and that the student may be deluded into thinking that these areas do not exist. This is not entirely true; the careful reader will find most of these topics mentioned somewhere.

Good teachers will always want to put something of their own into a course, regardless of the quality of the prescribed textbook. I have found that a useful (though at times highly dangerous) technique is deliberately not to give the best solutions to a problem in a class discussion, with the

optimistic aim that students can be persuaded to "discover" them for themselves, and even gain a sense of achievement in so doing. When applied to a book the technique is particularly dangerous, but I have tried to exploit it on several occasions, even though it may give the impression that the author is ignorant.

Another dangerous strategy is to give too much away, especially in a book like this aimed at courses where, so far as I am aware, the traditional approach requires that students make far more of the design decisions for themselves than my approach seems to allow them. Many of the books in the field do not show enough of how something is actually done: the bridge between what they give and what the student is required to produce is in excess of what is reasonable for a course which is only part of a general curriculum. I have tried to compensate by suggesting what I hope is a very wide range of searching exercises. The solutions to some of these are well known, and available in the literature. Again, the decision to omit explicit references was deliberate (perhaps dangerously so). Teachers often have to find some way of persuading the students to search the literature for themselves, and this is not done by simply opening the journal at the right page for them.

Acknowledgements

I am conscious of my gratitude to many people for their help and inspiration while this book has been developed.

Like many others, I am grateful to Niklaus Wirth, whose programming languages and whose writings on the subject of compiler construction and language design refute the modern trend towards ever-increasing complexity in these areas, and serve as outstanding models of the way in which progress should be made.

This project could not have been completed without the help of Hanspeter Mössenböck (author of the original Coco/R compiler generator) and Francisco Arzu (who ported it to C++), who not only commented on parts of the text, but also willingly gave permission for their software to be distributed with the book. My thanks are similarly due to Richard Cichelli for granting permission to distribute (with the software for Chapter 14) a program based on one he wrote for computing minimal perfect hash functions, and to Christopher Cockburn for permission to include his description of tonic sol-fa (used in Chapter 13).

I am grateful to Volker Pohlers for help with the port of Coco/R to Turbo Pascal, and to Dave Gillespie for developing `p2c`, a most useful program for converting Modula-2 and Pascal code to C/C++.

I am deeply indebted to my colleagues Peter Clayton, George Wells and Peter Wentworth for many hours of discussion and fruitful suggestions. John Washbrook carefully reviewed the manuscript, and made many useful suggestions for its improvement. Shaun Bangay patiently provided incomparable technical support in the installation and maintenance of my hardware and software, and rescued me from more than one disaster when things went wrong. To Rhodes University I am indebted for the use of computer facilities, and for granting me leave to complete the writing of the book. And, of course, several generations of students have contributed in intangible ways by their reaction to my courses.

The development of the software in this book relied heavily on the use of electronic mail, and I am grateful to Randy Bush, compiler writer and network guru extraordinaire, for his friendship, and for his help in making the Internet a reality in developing countries in Africa and elsewhere.

But, as always, the greatest debt is owed to my wife Sally and my children David and Helen, for their love and support through the many hours when they must have wondered where my priorities lay.

Pat Terry
Rhodes University
Grahamstown

Trademarks