Compiler construction for digital computers

David Gries
Cornell University

Preface

Compilers and interpreters are a necessary part of any computer system – without them, we would all be programming in assembly language or even machine language –. This has made compiler construction an important, practical area of research in computer science. The object of this book is to present in a coherent fashion the major techniques used in compiler writing, in order to make it easier for novice to enter the field and for the expert to reference the literature.

This book is oriented towards so-called syntax-directed methods of compiling. In fact, over one third of the book is devoted to the subject of formal language theory and automatic syntax recognition. I feel very strongly that anybody involved in compiler writing should have a basic knowledge of the subject. This does not mean that every compiler should be written using automatic syntax methods. There are many programming languages when these methods are not suitable. But a basic knowledge of formal language theory will give the compiler writer more insight into what is happening inside his compiler, and should help him design and program more systematically and efficient.

Syntax analysis, however, is only a small part of compiler construction, and i have included chapters on all the major topics – symbol table organization, error recovery, code generation, code optimization, and so forth. Several topics (e.g. conversion of constants, incremental compilers) have been omitted in order to keep the size of the book reasonable.

The book is meant to serve two needs: it can be used as a self-study and reference book for the professional programmer interested in or involved in compiler construction, and as a text in one-semester course in compiler writing. In fact, the book covers all the topics (and more) listed for the course compiler construction recommended by the ACM curriculum committee in the March 1968 issue of the *communications of the ACM*.

The reader should have at least one year of experience programming in a high-level language (e.g. FORTRAN, ALGOL, PL/I), an ind assembly language, and should be able to read and understand ALGOL programs. In some parts, elementary boolean matrix theory is assumed (with a short introduction). It is assumed the reader knows what a set is, what the union of two sets is, and so on. Beyond this, the reader should have the mathematical experience of, say, a sophomore or junior math major.

The need for experience with a high-level language is obvious: the book is about translating programs written in such languages. Experience with assembly language is similarly necessary. Assembly language experience is more important, however, for the maturity and understanding of how computers work that it provides. Actually, we will have little to do with any specific assembly language. The few IBM 360 assembly language programs scattered throughout

the book can be skipped over without loss of understanding.

A compiler is just a program written in some language. Hence, example of parts of compilers must be given in some programming language, and i have chosen an ALGOL-like language for its readability. The examples are usually very short, so that they can be followed easily. Where too much detail will cause us to lose sigh of the problem at hand, i have taken the liberty to write English instead of ALGOL. I have checked these program segments quite carefully by hand, but the reader is warned that they have not all been debugged on a computer.

A brief description of the bastard ALGOL used appears in the appendix. This description is short and relies heavily on a knowledge of ALGOL. Should the reader not be familiar with ALGOL and syntax description of languages, it is suggested that he wait until after studying chapter 2 to read the appendix.

There is more material that can usually be covered in a one-semester course. The following minimum set is suggested:

Chapter 1. Introduction

Chapter 2. Grammars and languages; omit section 7

Chapter 3. Scanners; omit sections 4, 5, 6

Chapter 4. Top-down parsing; especially section 3

Chapter 5. Simple precedence grammars

Chapter 8. Runtime storage organization; omit sections 6 and 9

Chapter 9. Organizing symbol tables

Chapter 10. The data on the symbol table; omit section 2

Chapter 11. Internal forms of the source programs; omit sections 4, 5

Chapter 12. Introduction to semantic routines

Chapter 13. Semantic routines for ALGOL constructs; omit section 6

Chapter 14. Allocation of storage to runtime variables; omit section 3

Chapter 16. Interpreters

Chapter 22. Hints to the compiler writer

You will notice that i emphasize the simple precedence technique for syntax analysis. This is not because it is the best (it is possibly the worst), but it is the easiest to teach. Should more time be available, the instructor is encouraged to add his favorite bottom-up syntax method: operator, precedence, higher order

precedence, transition matrices, production language, or any other not covered.

The order of presentation may also be changed. In fact, when teaching a course, it is best to break up the study of syntax theory with some practical material. Chapter 8 on runtime storage administration is independent, while chapters 9, 10, 11, and 16 on symbol tables, internal source program forms, and interpreters can be studied in that order at any time.

Chapters 21 deserves special mention. It is a collection of assorted facts and options that a compiler writer should be familiar with. They don't belong anywhere else, or are two important to be buried in some other chapter. The reader should browse through this chapter from time to time and read the sections of current interests.

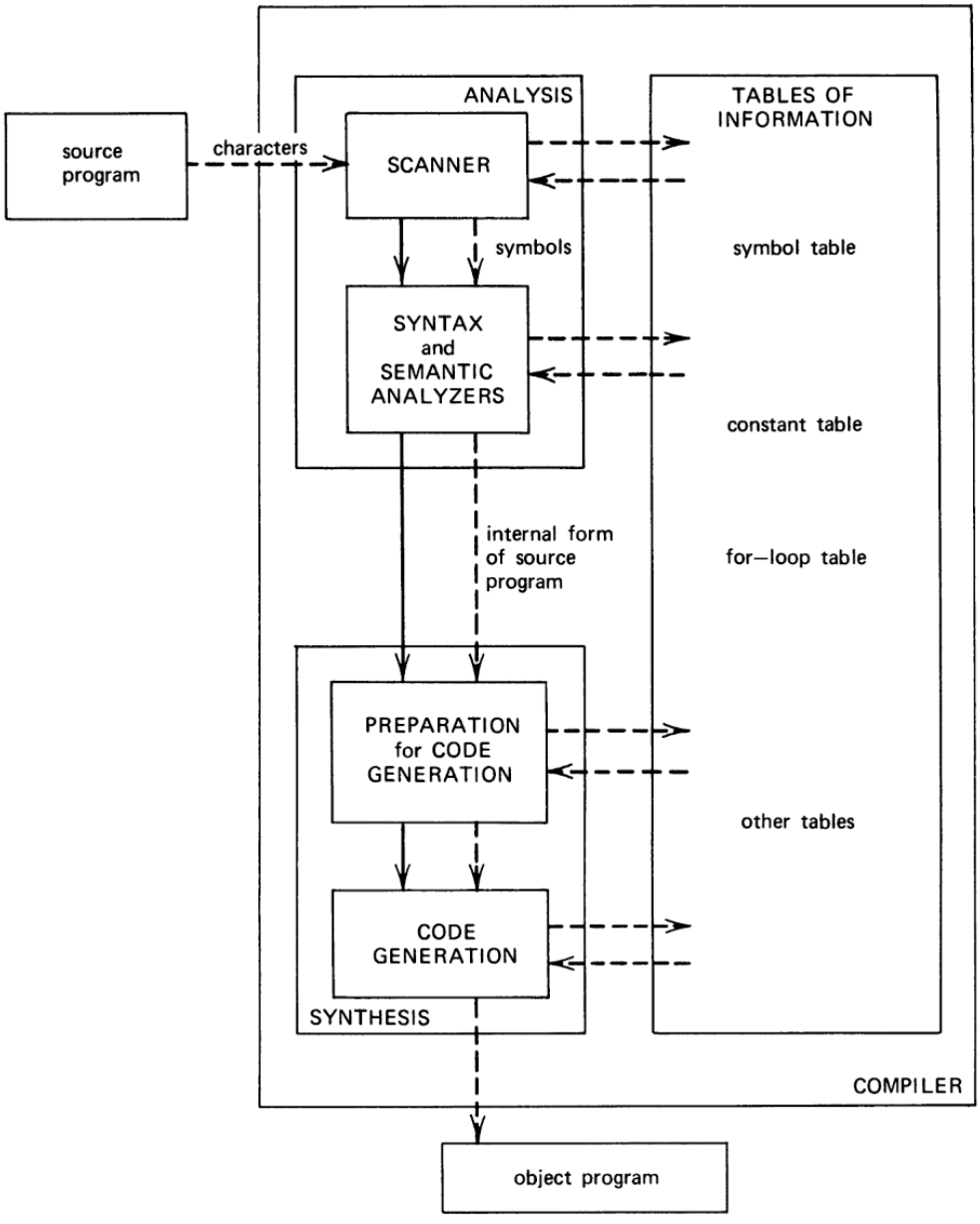
A compiler-writing course should be a laboratory course. Students should write and debug a compiler or interpreter for some simple language, in groups pf one to three people. Only then will they really understand what goes into a compiler. An interpreter is best, since students don't have to worry about messy machine language details; the ideas are important, not the details. Following this line, the whole project should be programmed in a high-level language. My experience is that PL/I or an ALGOL-like language is better than FORTRAN. Compilers in FORTRAN tend to be larger and much more difficult to read. A translator writing system should be used if available.

To produce some variability and creativity, start with a basic, simple language containing integer variables, assignment statements, expressions, labels and branches, conditional statements, and finally simple read and write statements. Then let each group extend it by adding one or two features. Examples are arrays, records (structures), different data types, block structure, procedures, macros, and iterative statements.

The compiler can be written and checked out in stages as the course progresses. First the scanner, then, the syntax analyzer, then the symbol table routines, and finally the semantic routines. The interpreter itself can be designed and implemented as soon as the chapters dealing with it have been covered. In this way, the work is spread out evenly throughout the semester, and is not bunched up at the end.

Table of Contents

Introduction


source program
characters
ANALYSIS
SCANNER
symbols
SYNTAX and SEMANTIC ANALYZERS
internal form of source program
PREPARATION for CODE GENERATION
CODE GENERATION
SYNTHESIS
TABLES OF INFORMATION
symbol table
constant table
for–loop table
other tables
COMPILER
object program