Introduction to Compiling Techniques

A First Course using ANSI C, LEX and YACC

Introduction to Compiling Techniques

A First Course using ANSI C, LEX and YACC

J.P. Bennett

Lecturer in Computing, School of Mathematical Sciences, Bath University

McGRAW-HILL BOOK COMPANY

London · New York · St Louis · San Francisco · Auckland · Bogotá · Guatemala
Hamburg · Lisbon · Madrid · Mexico · Montreal · New Delhi
Panama · Paris · San Juan · São Paulo · Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
Shoppenhangers Road
Maidenhead, Berkshire, England SL6 2QL
Telephone Maidenhead 0628 23432
Fax 0628 35895

British Library Cataloguing in Publication Data
Bennett, J.P. (Jeremy Peter), *1960–*
Introduction to compiling techniques: a first course using ANSI C, LEX and YACC
1. Computer systems. Compilers. Writing
I. Title
005.4'53

ISBN 0-07-707215-4

Library of Congress Cataloging-in-Publication Data
Bennett, J.P. (Jeremy Peter), 1960.
Introduction to compiling techniques: a first course using ANSI C, LEX and YACC J.P. Bennett.
p. cm.
Includes bibliographical references.
ISBN 0-07-707215-4
1. Compiling (Electronic computers) I. Title.
QA76.76.C65B46 1990
005.4'53--dc20 89-37070

1234 IP 9210

Typeset by Vision Typesetting, Manchester
Printed and bound in Great Britain by Information Press Ltd, Oxford

Preface

There are now several excellent comprehensive texts on compiling on the market. The problem with all these books is that they are very large and detailed. The present work is at a lower level and gives an introduction to compilers and their construction.

This book is aimed at two audiences. The first is the second- or third-year honours undergraduate in computer science taking a first course in compiling techniques. The second is the working programmer, who although not having formal qualifications in computer science needs to use compiler technology in his or her work. To this end a balance is maintained between providing enough theoretical background to enable a clear understanding of the subject and giving a practical presentation that will both illustrate concepts and allow the reader to develop effective compilers.

A certain amount of knowledge is assumed on the part of the reader. Familiarity with C is essential. Throughout we have used ANSI standard C, but this is readily comprehensible to anyone more familiar with K&R C. A general familiarity with machine language is required at the level that would be taught in any introductory computing course.

At its heart compilation is based on a few very elegant algorithms. This book is short enough to allow the reader to see the elegance of the subject without being frustrated by obscure detail. The main parsing, translation, and code generation techniques in use today are presented with many diagrams and examples. Methods that are no longer of importance are either omitted or only presented to the extent that they help the understanding of modern practice.

Examples throughout are presented using the C programming language. The section on compiler generators uses the LEX scanner generator and YACC parser generator, highlighting the modern emphasis on tools. These are widely available under Unix with similar programs being available under other operating systems. This is far and away the commonest teaching and development environment today, making the practical examples directly available to a wide audience. The book culminates with the presentation of a compiler for a

simple programming language, VSL. Compilation is for a simple abstract machine, VAM, which has many of the properties of modern microprocessors.

Exercises are suggested at the end of each chapter. Some of these are practical programming problems to help the student understand the workings of practical compilers. Others are essay subjects and questions suitable for exam revision. Finally there are discussion topics, which lead beyond the level of this book into more advanced areas of computation. A brief reading list at the end of each chapter provides initial assistance in further developing the ideas introduced.

I am indebted to my colleagues in the School of Mathematical Sciences at Bath University for their assistance. In particular Dr Dan Richardson provided much perceptive criticism of the manuscript. Undergraduates on his language theory course used the first draft as their textbook in the spring of 1989 and suggested many improvements which I have adopted. Mr Richard Nuttall of Torch Computers made many helpful suggestions on improving the text from the perspective of an industrial programmer. Finally I must express my gratitude to the anonymous reviewer from McGraw-Hill who gave the manuscript a very thorough analysis, making many perceptive suggestions that have greatly improved the text.

J. P. Bennett

Contents

Preface ix

1. What is a compiler? 1

1.1 The need for machine translation 1
1.2 The structure of a compiler 4
1.3 A demonstration compiler 10
Exercises 11
Further reading 12

2. Target languages 13

2.1 Types of target machine 13
2.2 Implementation methods 16
Exercises 26
Further reading 26

3. Formal grammars 28

3.1 Defining the structure of a language 29
3.2 Properties of grammars 35
3.3 Syntax-directed translation 40
Exercises 43
Further reading 44

4. Intermediate representations 45

4.1 Types of intermediate representations 45
4.2 Abstract machines 58
Exercises 58
Further reading 59

5. Lexical analysis 60

5.1 Why have a separate lexical analyser? 60
5.2 *Ad hoc* lexical analysers 66
5.3 Lexical analysis with finite state machines 69
Exercises 76
Further reading 77

6. Syntax analysis methods 78

6.1 Approaches to parsing 78
6.2 Top-down parsing methods 80
6.3 Bottom-up parsing 92
Exercises 110
Further reading 111

7. Error handling 112

7.1 Compile-time error handling 113
7.2 Run-time errors 115
Exercises 116
Further reading 116

8. Parser generators 117

8.1 YACC 117
8.2 YACC and ambiguous grammars 120
8.3 Type consistency in YACC 125
8.4 Error handling 126
Exercises 127
Further reading 128

9. Semantic checking 129

9.1 Type checking 129
9.2 Other semantic checks 136
Exercises 136
Further reading 137

10. Code generation 138

10.1 Declarations and storage allocation 139
10.2 Expressions and assignment 142
10.3 Flow of control 145
Exercises 149
Further reading 150

11. Code optimization **151**

11.1 Basic blocks 151
11.2 Loop optimizations 160
11.3 Register optimization 165
11.4 Other optimizations 166
Exercises 169
Further reading 170

12. A complete compiler for VSL **171**

12.1 The VC compiler for VSL 171
12.2 Description of the compiler 172
12.3 Building the compiler 223
12.4 Running the compiler 224
Exercises 228

Appendix. VSL, VAM, and VAS **230**

A.1 The grammar of VSL 230
A.2 The VSL Abstract Machine 231
A.3 A mnemonic assembler for VAM 233

Index **234**

1

What is a compiler?

A compiler is a translator from a program written in one language, the *source language*, to an equivalent program in a second language, the *target* or *object language* (Fig. 1.1). Typically the source language will be a programming language such as FORTRAN, Pascal, or C, and the target language will be machine code for the computer being used, which is hence known as the *target machine*. The compiler will usually supply error and diagnostic information about the source program being compiled.

1.1 The need for machine translation

The earliest machines were very small and simple. For example, the Manchester Mark 1 produced in 1948 had only seven opcodes and 32 words of main memory. For such a computer, entering programs as sequences of binary digits at a keyboard was not that difficult. However, it was convenient when writing down such programs not to use the sequence of binary digits, but a shorthand notation for the opcodes. Table 1.1 shows some of the early mnemonics for opcodes used with this computer.

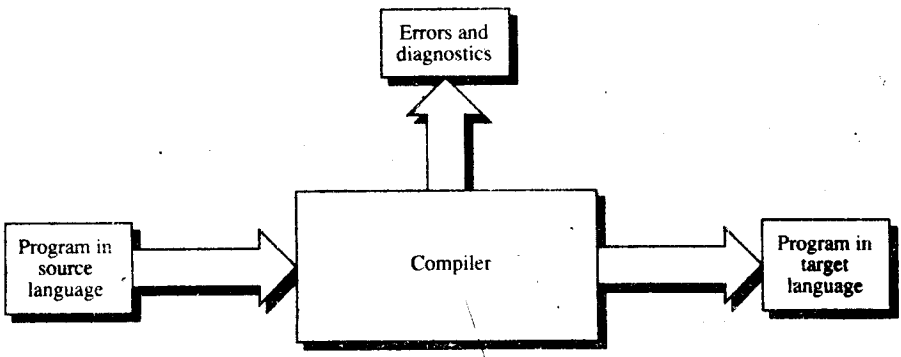

Figure 1.1 Overall structure of a compiler

Table 1.1 Assembler mnemonics for the Manchester Mark 1

Binary opcode	*Mnemonic*	*Meaning*
011	a,S	Store the contents of the accumulator at address S
100	a-s,A	Subtract the value at location S from the accumulator
111	Stop	Stop and await operator action

With more complex machines and longer programs, hand translation of mnemonics into binary for entry into the computer became tedious. In the late 1940s it was pointed out that this translation could perfectly well be done by the computer itself. Programs to do this were known as *assemblers*, and the mnemonic codes as *assembly languages*.

Because of their simplicity, being essentially a one-to-one mapping from mnemonic to machine opcode, assembly language programs are very verbose. More complex languages, known as *autocodes*, were developed to describe programs more concisely. Each autocode instruction could represent several machine code operations. Programs to translate these *high-level languages* into machine code were more complex than assemblers and became known as *compilers*.

During the 1950s high-level computer languages evolved to describe problems independently of the machine code of any particular computer. Early languages such as FORTRAN and the autocodes from which it was derived were strongly influenced by the available operations in the underlying machine code. For example, FORTRAN IV had a restriction of 3 on the number of dimensions of an array partly because its original target machine, the IBM 709, had only three registers for indexing arrays. Even C, designed in the mid-1970s, has some constructs (e.g. the increment operator ++) because of the availability of an equivalent opcode on the original target machine, a PDP-11.

Algol 60, which was actually proposed in 1958, heralded a new approach to high-level languages. It was designed with problem solving in mind, and questions of how it might ever be translated to be run on real machines ignored. For example, it allowed local variables and recursive routine calls. How were these to be translated to run on machines with a single address space and just a jump to subroutine opcode? Most modern computer languages since, such as Pascal, Modula 2, and Ada, have also been designed to be independent of any particular target architecture.

Compiled high-level languages are now well established. Advantages are their conciseness which improves programmer productivity, semantic restrictions (such as type checking) to reduce logical errors, and ease of debugging. Disadvantages are their speed (typically 2–10 times slower than hand-written assembler) and size, both of the compilers and the compiled code.

Compiler theory has developed to enable languages such as these to be translated without difficulty. Many tools have been developed to automate much

of the process. While the first FORTRAN compiler took 18 man years of effort to construct, it is now perfectly feasible for an undergraduate to write a simple compiler for Pascal in a term.

1.1.1 APPROACHES TO MACHINE TRANSLATION

There are two ways of running a program written in a high-level language on a computer. The first is to translate the program into an equivalent program in the machine code of the computer. This is the process of compilation described in the previous section.

The second approach is to write a program that can interpret the statements of the high-level language program as they are encountered and carry out their actions. Such programs are called *interpreters*.

Compilation has the advantage that we have to analyse and translate our high-level language program only once, although this may be a time-consuming process. Thereafter we just run the equivalent machine code program produced by the compiler. A disadvantage is that if our program goes wrong we will get an error in the machine code program and must try and work back from this to find the corresponding error in the high-level language program.

Interpretation is much slower than compilation, since we must analyse each high-level language statement to determine its meaning each time we encounter it. However, if there is an error we are still dealing with the original high-level language program and can immediately pinpoint its source. This is often a great help when developing and debugging programs.

These two approaches are the extremes, and many machine translators are a bit of both. A common approach is to compile the high-level language not into the machine code of the target machine, but into a lower-level *intermediate code* which is then interpreted. This intermediate code is chosen to be easy to compile into and efficient to execute, so we end up with a system where compilation is not too time consuming, programs run reasonably fast, and if there is an error we see it in an intermediate code that is easier to relate to the source language than machine code. Compiler/interpreters like this have been written for many languages. A good example is the UCSD Pascal compiler, which generates an intermediate code, PCODE, for interpretation.

The choice of whether to compile or interpret is to a large extent influenced by the nature of the high-level language and the environment in which it is used. FORTRAN is relatively simple, designed for translation to machine code, and often used for solving big numerical problems on mainframe computers, where speed of execution is essential. It is thus invariably compiled. BASIC, on the other hand, is mainly used on personal microcomputers where clear error handling is important, and where lack of processing power and memory could make compilation very difficult. It is invariably interpreted, although modern interpreters often do an element of compilation, analysing keywords as the program is typed in. LISP is a language that often uses both interpretation and compilation. Programs are interpreted during program development to avoid

time-consuming compilations each time the program is changed and to give clear error handling, and then compiled when development is complete.

Although this book is essentially concerned with the techniques involved in compilation, much of the information is of use in writing interpreters. Analysis of the source code is much the same in both cases and finding the most efficient way of interpreting a particular construct is not dissimilar to finding the best code for a compiler to generate.

1.1.2 THE WIDER USE OF COMPILERS

By far the commonest use of compilation techniques is in the translation of high-level programming languages into machine code for execution on a target machine. However, these techniques have relevance throughout software engineering. Source languages need not be programming languages, but may be word-processing languages, natural languages such as English, or special languages to describe the layout of silicon integrated circuits. Target languages may be driver codes for laser printers, other natural languages, or integrated circuit masks. In all these the comprehension of computer languages and translation into other languages is important.

This book concentrates on compilation from conventional procedural languages, such as Pascal or C, into conventional machine codes as typically found on a modern microprocessor. Ideas are illustrated throughout by examples using VSL, a very simple block-structured procedural language and its compilation for VAM, a byte stream machine with a reduced instruction set.

1.2 The structure of a compiler

The translation of a programming language naturally breaks down into a number of logical phases. These phases may run simultaneously, or they may run consecutively. At its simplest level we may break down a compiler into a *front end*, responsible for the analysis of the structure and meaning of the source text; and a *back end*, responsible for generating the target language.

Each of these may be further subdivided into logical blocks. The front end can be divided into *lexical analyser*, *syntax analyser*, and *semantic analyser*. The lexical analyser, sometimes also called the *scanner*, carries out the simplest level of structural analysis. It will group the individual symbols of the source program text into their logical entities. Thus the sequence of characters 'W', 'H', 'I', 'L', and 'E' would be identified as the word 'WHILE' and the sequence of characters '1', '.', and '0' would be identified as the floating point number, 1.0.

The syntax analyser, often also called the *parser*, analyses the overall structure of the whole program, grouping the simple entities identified by the scanner into the larger constructs, such as statements, loops, and routines, that make up a complete program. Just as the structure of English prose is determined by the rules of English grammar, so we have *formal grammars* to describe the structure of computer programs.

Once the structure of the program has been determined we can then analyse its meaning (or *semantics*). We can determine which variables are to hold integers, and which to hold floating point numbers, we can check that the size of all arrays is defined and so on.

At this stage the program has been partially translated into some intermediate representation. The back end of the compiler takes this and with the information provided on the structure and meaning of the source program, generates an equivalent program in the target language. Often this involves more than one phase, to ensure an efficient translation of the high-level language program.

First of all an *intermediate code optimizer* may transform the intermediate representation into a more efficient equivalent. After this comes the *code generator*, generating an equivalent program for the target machine. Finally there may be a *target code optimizer* to improve further the generated code.

Good optimizers can be very time consuming. It is not at all uncommon for the optimizer not to be used during program development and only to be brought in for the last compilation of the complete program.

This is not quite all that is involved in a compiler. It is usual to have to provide a *run-time system* to support the compiled language. Some high-level language constructs, such as input and output and interrupt handling, are inherently complex. Compiled code would be immensely large if such constructs were translated into target code each time they were encountered and so instead we provide them as subroutines and compile calls to the subroutines when necessary. These subroutines form the *run-time library*. In addition the run-time system will include some start-up code to initialize the machine before the compiled program is run, and some termination code to put the system back in a standard state at the end of a run.

The operation of a compiler is summarized in Fig. 1.2.

1.2.1 LANGUAGES FOR WRITING COMPILERS

Compilers are relatively large programs involving a lot of programming effort and we wish them to be as portable as possible. The front end can remain the same for a wide range of target machines. The back end differs for each target machine, but even here there is scope for reuse of code.

Which language should we choose for maximum portability? Machine code has the advantage that it is always available. This was invariably the approach used with early compilers, a typical example being the FORTRAN compiler for the IBM 709. However, it is the least portable, since our compiler will only run on one machine. Furthermore, writing a program as large as a compiler in machine code is extremely demanding.

An alternative is to implement in an existing high-level language that is already widely available. We then have the programming power of a high-level language and our compiler can be ported to any machine that supports the implementation language. This does not give us the widest portability, since we are restricted to those machines supporting the implementation language. It does not help

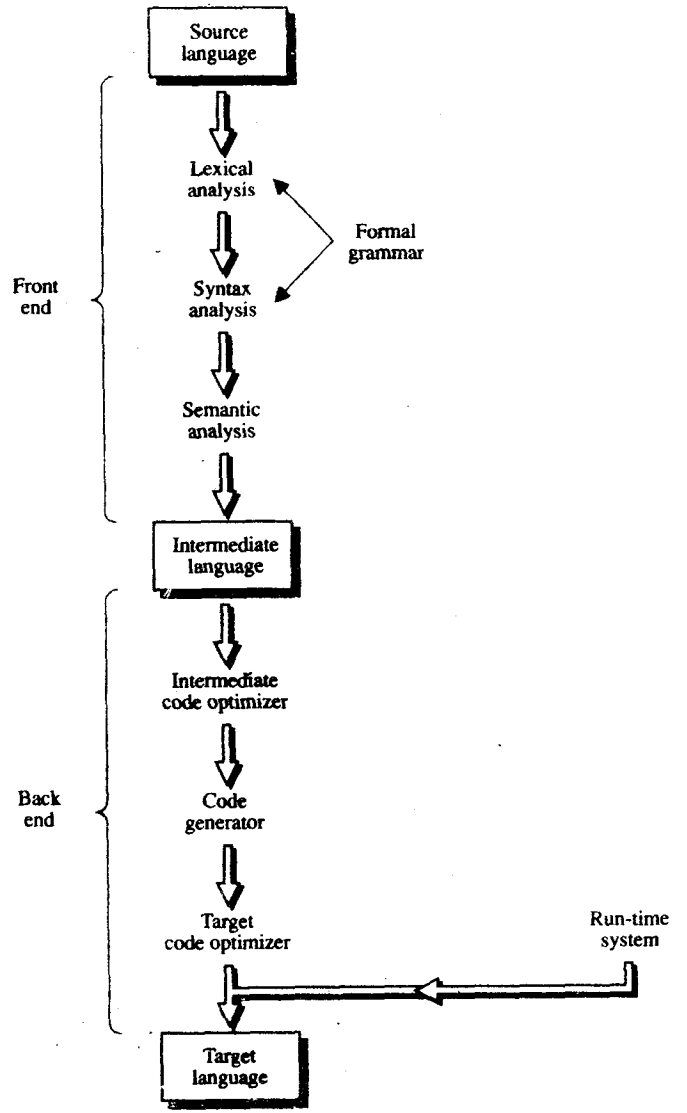

Figure 1.2 Operation of a compiler

matters that the two most widely available computer languages, FORTRAN and COBOL, are far from ideal for compiler writing. Despite these drawbacks this approach has been widely used, particularly for compilers under the Unix operating system. Under Unix the C programming language is always available and often used for implementing compilers. The approach is also common with specialist research languages, usually because the compiler writer needs the

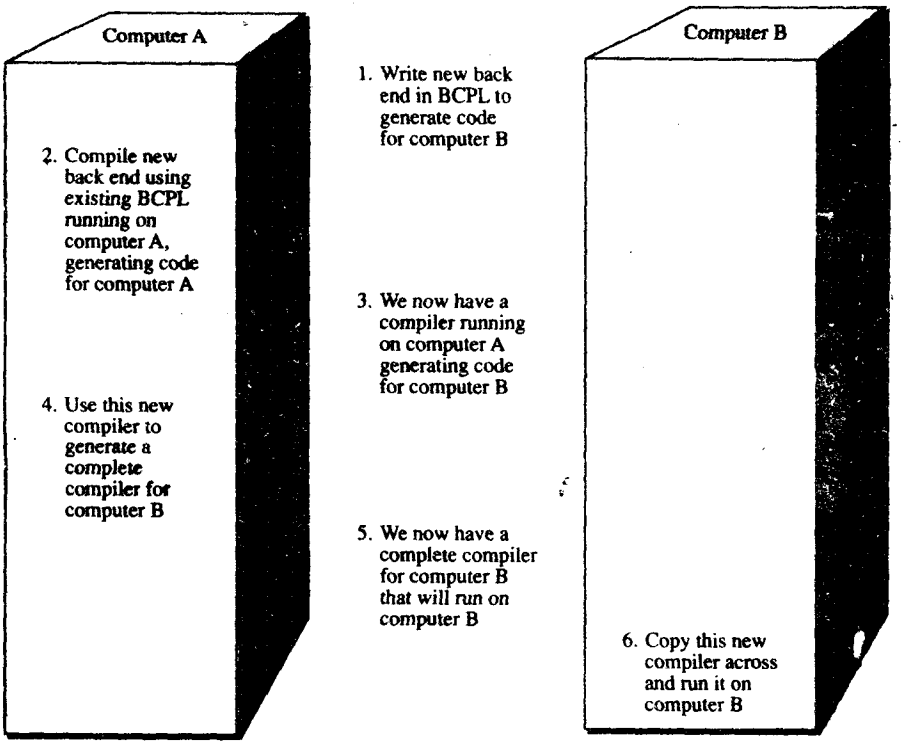

Figure 1.3 Porting the BCPL compiler

flexibility of a high-level language, rather than for reasons of portability. A typical example of this is the compiler for the Ponder higher-order functional programming language, which is written in Algol 68.

A compromise approach is to write our compiler in a 'generalized' assembly language. To port the compiler to a new machine we have to write an assembler for this language to generate our target machine code. Greater portability is achieved if the target language of our compiler is the same generalized assembly language, since we can use the assembler to translate this into the target language. Using this approach we have simplified the problem of porting a compiler to one of porting an assembler. An example of this approach is the Macro-SPITBOL compiler for the SNOBOL4 string-processing language, which is implemented in the generalized assembly language, MINIMAL. The target language for this compiler is in fact an intermediate code for interpretation, the interpreter for which is also written in MINIMAL, aiding portability.

There is one other approach to compiler writing, which gives the greatest flexibility of all, and that is to write the compiler in its own source language. For example the standard BCPL compiler is written in BCPL, and C compilers under

Unix are written in C. Portability is achieved by the technique of *cross-compilation*. Let us consider porting a compiler for BCPL, written in BCPL, from an existing machine, A, to a new machine, B. We take our existing compiler, running on machine A and modify its back end to generate machine code for machine B. We then compile this on machine A using the existing compiler. This gives us a compiler for BCPL that runs on machine A and generates code for machine B. We then run this compiler on machine A, using it to compile our modified compiler for machine B. We now have a compiled version of the modified compiler in the machine code of machine B. We can then copy this across to machine B, giving us a BCPL compiler on machine B, generating code for machine B. From now on we can work on machine B alone, since we have a working BCPL compiler. This sequence of operations is shown in Fig. 1.3. We still have the problem of writing the first ever BCPL compiler, which must be done using one of the techniques described earlier. However once we have this initial compiler running on one machine, then cross-compilation gives us a very powerful way of writing a portable compiler.

T-diagrams

When considering how compilers are implemented it is often helpful to show the programs required using *T-diagrams*. For each program in the system we draw a T, with the name of the program across the top. The left and right arms of the T show the source and target language of the program, respectively, and the bottom leg of the T shows the language in which the program itself is implemented. If the implementation language itself must be translated we can slot together a number of T-diagrams, showing the translations that must be carried out for the system to work. Figure 1.4 shows T-diagrams for the examples described in the previous section.

1.2.2 THE IMPORTANCE OF VARIOUS PARTS OF A COMPILER

Different languages make different demands on the compiler writer. Conventional block structured languages, such as Algol, Pascal, or C, have relatively complex formal grammars. The syntax analysers for such languages are a major part of the compiler. Once the structure is understood, then simple code generation is relatively straightforward, because modern computer architectures are orientated towards such languages. The large numbers of cases that have to be considered mean such code generators are not small, but they are not too difficult to write and execute quickly. Should a code optimizer be added then this will increase the size, and reduce the performance of the whole back end. In addition, such languages need a run-time system, but again this need not be too large. In general with this type of language we see a fairly even balance between front end and back end of the compiler.

Other types of language place different emphasis on the parts of the compiler. Functional languages and their relatives, such as LISP or ML, often have simple

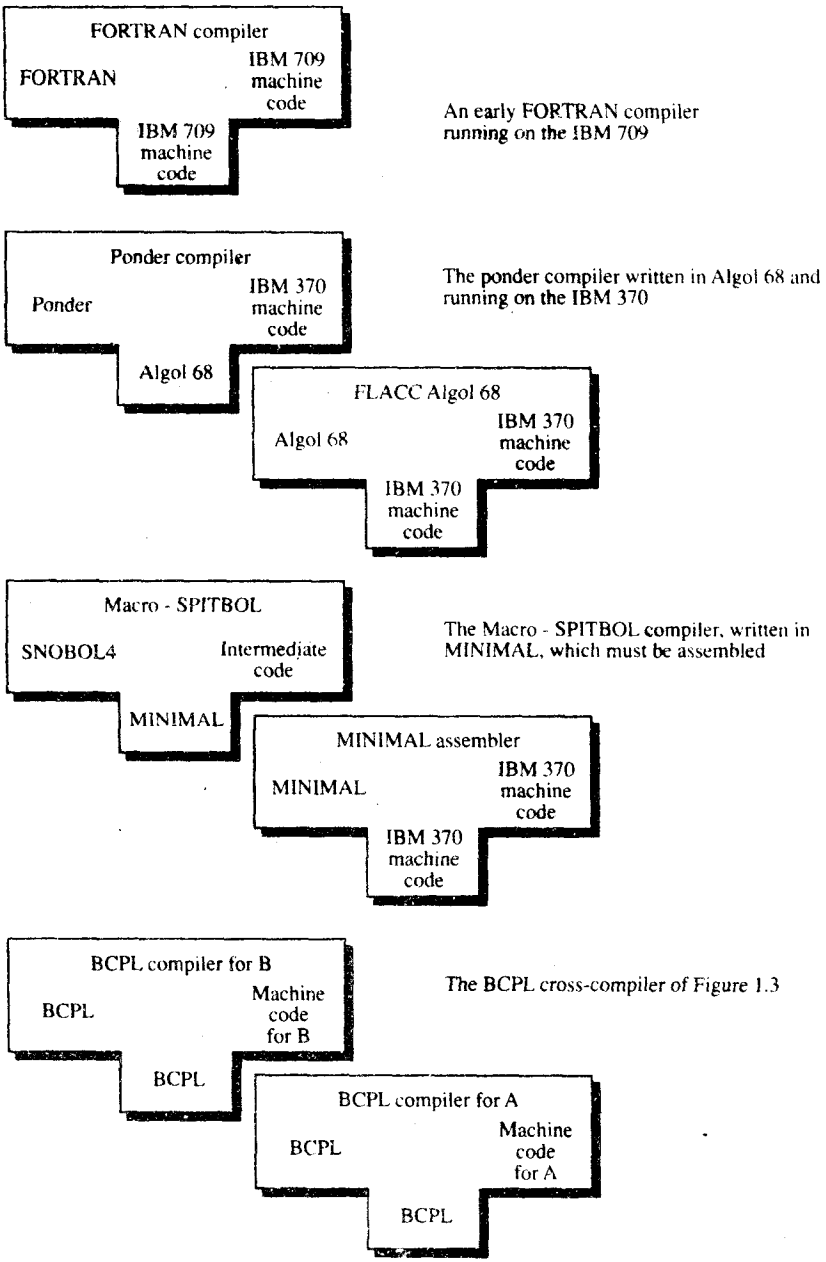

Figure 1.4 Some typical T-diagrams