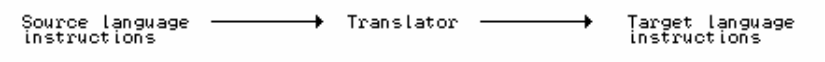
2 TRANSLATOR CLASSIFICATION AND STRUCTURE

In this chapter we provide the reader with an overview of the inner structure of translators, and some idea of how they are classified.

A translator may formally be defined as a function, whose domain is a source language, and whose range is contained in an object or target language.



A little experience with translators will reveal that it is rarely considered part of the translator's function to execute the algorithm expressed by the source, merely to change its representation from one form to another. In fact, at least three languages are involved in the development of translators: the source language to be translated, the object or target language to be generated, and the host language to be used for implementing the translator. If the translation takes place in several stages, there may even be other, intermediate, languages. Most of these - and, indeed, the host language and object languages themselves - usually remain hidden from a user of the source language.

2.1 T-diagrams

A useful notation for describing a computer program, particularly a translator, uses so-called **T-diagrams**, examples of which are shown in Figure 2.1.

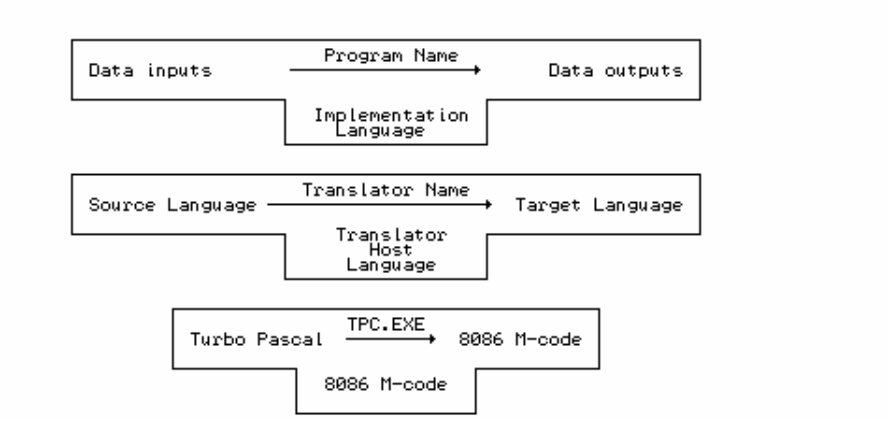


Figure 2.1 T-diagrams. (a) A general program (b) a general translator (c) A Turbo Pascal compiler for an MS-DOS system

We shall use the notation "M-code" to stand for "machine code" in these diagrams. Translation itself is represented by standing the T on a machine, and placing the source program and object program on the left and right arms, as depicted in Figure 2.2.

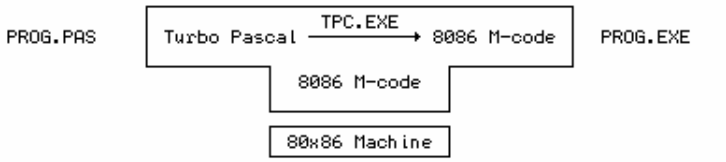


Figure 2.2 A Turbo Pascal compilation on an 80x86 machine

We can also regard this particular combination as depicting an **abstract machine** (sometimes called a **virtual machine**), whose aim in life is to convert Turbo Pascal source programs into their 8086 machine code equivalents.

T-diagrams were first introduced by Bratman (1961). They were further refined by Earley and Sturgis (1970), and are also used in the books by Bennett (1990), Watt (1993), and Aho, Sethi and Ullman (1986).

2.2 Classes of translator

It is common to distinguish between several well-established classes of translator:

- The term **assembler** is usually associated with those translators that map low-level language instructions into machine code which can then be executed directly. Individual source language statements usually map one-for-one to machine-level instructions.

- The term **macro-assembler** is also associated with those translators that map low-level language instructions into machine code, and is a variation on the above. Most source language statements map one- for-one into their target language equivalents, but some *macro* statements map into a sequence of machine- level instructions - effectively providing a text replacement facility, and thereby extending the assembly language to suit the user. (This is not to be confused with the use of procedures or other subprograms to "extend" high-level languages, because the method of implementation is usually very different.)

- The term **compiler** is usually associated with those translators that map high-level language instructions into machine code which can then be executed directly. Individual source language statements usually map into many machine-level instructions.

- The term **pre-processor** is usually associated with those translators that map a superset of a high-level language into the original high-level language, or that perform simple text substitutions before translation takes place. The best-known pre-processor is probably that which forms an integral part of implementations of the language C, and which provides many of the features that contribute to the widely- held perception that C is the only really portable language.

- The term **high-level translator** is often associated with those translators that map one high-level language into another high-level language - usually one for which sophisticated compilers already exist on a range of machines. Such translators are particularly useful as components of a two-stage compiling system, or in assisting with the bootstrapping techniques to be discussed shortly.

- The terms **decompiler** and **disassembler** refer to translators which attempt to take object code at a low level and regenerate source code at a higher level. While this can be done quite successfully for the production of assembler level code, it is much more difficult when one tries to recreate source code originally written in, say, Pascal.

Many translators generate code for their host machines. These are called **self-resident translators**. Others, known as **cross-translators**, generate code for machines other than the host machine. Cross-translators are often used in connection with microcomputers, especially in embedded systems, which may themselves be too small to allow self-resident translators to operate satisfactorily. Of course, cross-translation introduces additional problems in connection with transferring the object code from the donor machine to the machine that is to execute the translated program, and can lead to delays and frustration in program development.

The output of some translators is absolute machine code, left loaded at fixed locations in a machine ready for immediate execution. Other translators, known as **load-and-go** translators, may even initiate execution of this code. However, a great many translators do not produce fixed-address machine code. Rather, they produce something closely akin to it, known as **semicompiled** or **binary symbolic** or **relocatable** form. A frequent use for this is in the development of composite libraries of special purpose routines, possibly originating from a mixture of source languages. Routines compiled in this way are linked together by programs called **linkage editors** or **linkers**, which may be regarded almost as providing the final stage for a multi-stage translator. Languages that encourage the separate compilation of parts of a program - like Modula-2 and C++ - depend critically on the existence of such linkers, as the reader is doubtless aware. For developing really large software projects such systems are invaluable, although for the sort of "throw away" programs on which most students cut their teeth, they can initially appear to be a nuisance, because of the overheads of managing several files, and of the time taken to link their contents together.

T-diagrams can be combined to show the interdependence of translators, loaders and so on. For example, the FST Modula-2 system makes use of a compiler and linker as shown in Figure 2.3.

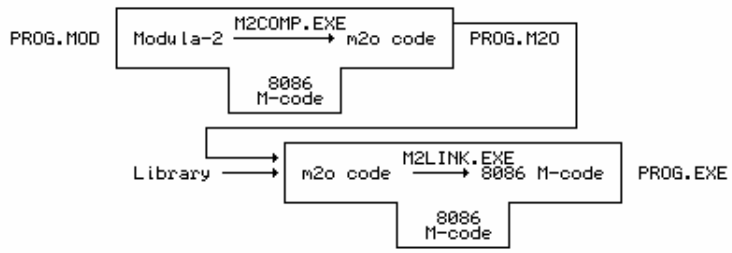


Figure 2.3 Compiling and linking Modula-2 program on the FST system

---

Exercises

2.1 Make a list of as many translators as you can think of that can be found on your system.

2.2 Which of the translators known to you are of the load-and-go type?

2.3 Do you know whether any of the translators you use produce relocatable code? Is this of a standard form? Do you know the names of the linkage editors or loaders used on your system?

2.3 Phases in translation

Translators are highly complex programs, and it is unreasonable to consider the translation process as occurring in a single step. It is usual to regard it as divided into a series of **phases**. The simplest breakdown recognizes that there is an **analytic phase**, in which the source program is analysed to determine whether it meets the syntactic and static semantic constraints imposed by the language. This is followed by a **synthetic phase** in which the corresponding object code is generated in the target language. The components of the translator that handle these two major phases are said to comprise the **front end** and the **back end** of the compiler. The front end is largely independent of the target machine, the back end depends very heavily on the target machine. Within this structure we can recognize smaller components or phases, as shown in Figure 2.4.

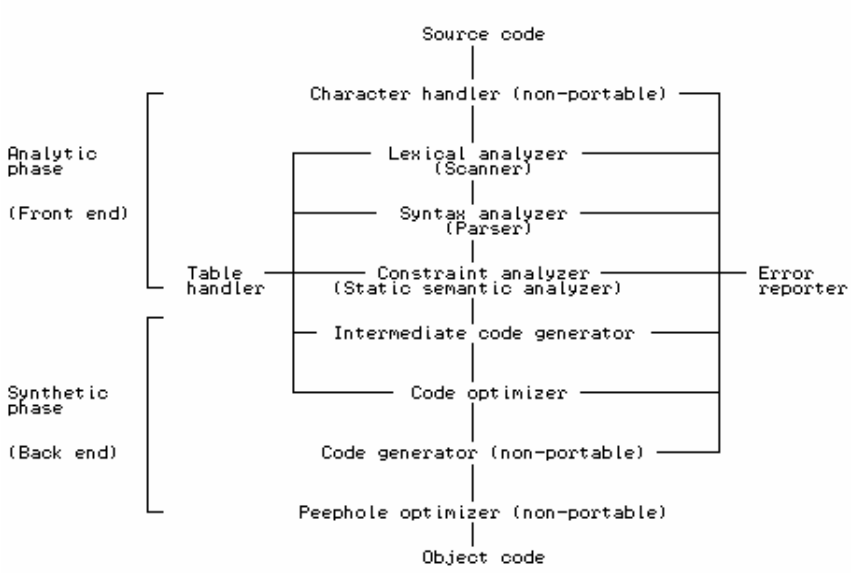


Figure 2.4 Structure and phases of a compiler

The **character handler** is the section that communicates with the outside world, through the operating system, to read in the characters that make up the source text. As character sets and file handling vary from system to system, this phase is often machine or operating system dependent.

The **lexical analyser** or **scanner** is the section that fuses characters of the source text into groups that logically make up the **tokens** of the language - symbols like identifiers, strings, numeric constants, keywords like `while` and `if`, operators like `<=`, and so on. Some of these symbols are very simply represented on the output from the scanner, some need to be associated with various properties such as their names or values.

Lexical analysis is sometimes easy, and at other times not. For example, the Modula-2 statement

```
WHILE A > 3 * B DO A := A - 1 END
```

easily decodes into tokens

```
WHILE       keyword
A           identifier              name A
```

```
>           operator                comparison
3           constant literal        value 3
*           operator                multiplication
B           identifier              name B
DO          keyword
A           identifier              name A
:=          operator                assignment
A           identifier              name A
-           operator                subtraction
1           constant literal        value 1
END         keyword
```

as we read it from left to right, but the Fortran statement

```
        10      DO 20 I = 1 . 30
```

is more deceptive. Readers familiar with Fortran might see it as decoding into

```
10          label
DO          keyword
20          statement label
I           INTEGER identifier
=           assignment operator
1           INTEGER constant literal
,           separator
30          INTEGER constant literal
```

while those who enjoy perversity might like to see it as it really is:

```
10          label
DO20I       REAL identifier
=           assignment operator
1.30        REAL constant literal
```

One has to look quite hard to distinguish the period from the "expected" comma. (Spaces are irrelevant in Fortran; one would, of course *be* perverse to use identifiers with unnecessary and highly suggestive spaces in them.) While languages like Pascal, Modula-2 and C++ have been cleverly designed so that lexical analysis can be clearly separated from the rest of the analysis, the same is obviously not true of Fortran and other languages that do not have reserved keywords.

The **syntax analyser** or **parser** groups the tokens produced by the scanner into syntactic structures - which it does by parsing expressions and statements. (This is analogous to a human analysing a sentence to find components like "subject", "object" and "dependent clauses"). Often the parser is combined with the **contextual constraint analyser**, whose job it is to determine that the components of the syntactic structures satisfy such things as scope rules and type rules within the context of the structure being analysed. For example, in Modula-2 the syntax of a *while* statement is sometimes described as

```
WHILE  Expression  DO  StatementSequence  END
```

It is reasonable to think of a statement in the above form with any type of *Expression* as being syntactically correct, but as being devoid of real meaning unless the value of the *Expression* is constrained (in this context) to be of the Boolean type. No program really has any meaning until it is executed dynamically. However, it is possible with strongly typed languages to predict at compile-time that some source programs can have no sensible meaning (that is, statically, before an attempt is made to execute the program dynamically). Semantics is a term used to describe "meaning", and so the constraint analyser is often called the **static semantic analyser**, or simply the semantic analyser.

The output of the syntax analyser and semantic analyser phases is sometimes expressed in the form of a decorated **abstract syntax tree** (AST). This is a very useful representation, as it can be used in clever ways to optimize code generation at a later stage.

Whereas the **concrete syntax** of many programming languages incorporates many keywords and tokens, the **abstract syntax** is rather simpler, retaining only those components of the language needed to capture the real content and (ultimately) meaning of the program. For example, whereas the concrete syntax of a *while* statement requires the presence of `WHILE`, `DO` and `END` as shown above, the essential components of the *while* statement are simply the (Boolean) *Expression* and the statements comprising the *StatementSequence*.

Thus the Modula-2 statement

```
WHILE  (1 < P)  AND  (P < 9)  DO  P := P + Q  END
```

or its C++ equivalent

```
while  (1 < P && P < 9)  P = P + Q;
```

are both depicted by the common AST shown in Figure 2.5.

Figure 2.5 AST for the statement while (1 < P && P < 9) P = P + Q;

An abstract syntax tree on its own is devoid of some semantic detail; the semantic analyser has the task of adding "type" and other contextual information to the various nodes (hence the term "decorated" tree).

Sometimes, as for example in the case of most Pascal compilers, the construction of such a tree is not explicit, but remains implicit in the recursive calls to procedures that perform the syntax and semantic analysis.

Of course, it is also possible to construct concrete syntax trees. The Modula-2 form of the statement

```
WHILE  (1 < P)  AND  (P < 9)  DO  P := P + Q  END
```

could be depicted in full and tedious detail by the tree shown in Figure 2.6. The reader may have to make reference to Modula-2 syntax diagrams and the knowledge of Modula-2 precedence rules to understand why the tree looks so complicated.

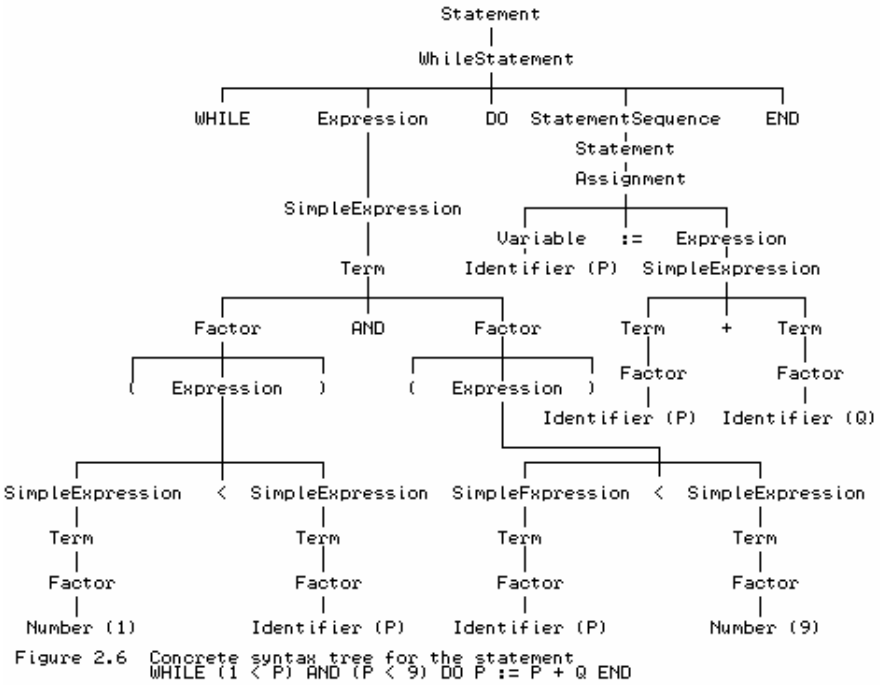


Figure 2.6 Concrete syntax tree for the statement
WHILE (1 < P) AND (P < 9) DO P := P + Q END

The phases just discussed are all analytic in nature. The ones that follow are more synthetic. The first of these might be an **intermediate code generator**, which, in practice, may also be integrated with earlier phases, or omitted altogether in the case of some very simple translators. It uses the data structures produced by the earlier phases to generate a form of code, perhaps in the form of simple code skeletons or macros, or ASSEMBLER or even high-level code for processing by an external assembler or separate compiler. The major difference between intermediate code and actual machine code is that intermediate code need not specify in detail such things as the exact machine registers to be used, the exact addresses to be referred to, and so on.

Our example statement

```
WHILE  (1 < P)  AND  (P < 9)  DO   P := P + Q  END
```

might produce intermediate code equivalent to

```
L0      if 1 < P goto L1
        goto L3
L1      if P < 9 goto L2
        goto L3
L2      P := P + Q
        goto L0
L3      continue
```

Then again, it might produce something like

```
L0      T1 := 1 < P
        T2 := P < 9
        if T1 and T2 goto L1
        goto L2
L1      P := P + Q
        goto L0
L2      continue
```

depending on whether the implementors of the translator use the so-called *sequential conjunction* or *short-circuit* approach to handling compound Boolean expressions (as in the first case) or the so-called *Boolean operator* approach. The reader will recall that Modula-2 and C++ require the short-circuit approach. However, the very similar language Pascal did not specify that one approach

be preferred above the other.

A **code optimizer** may optionally be provided, in an attempt to improve the intermediate code in the interests of speed or space or both. To use the same example as before, obvious optimization would lead to code equivalent to

```
L0      if 1 >= P goto L1
        if P >= 9 goto L1
        P := P + Q
        goto L0
L1      continue
```

The most important phase in the back end is the responsibility of the **code generator**. In a real compiler this phase takes the output from the previous phase and produces the object code, by deciding on the memory locations for data, generating code to access such locations, selecting registers for intermediate calculations and indexing, and so on. Clearly this is a phase which calls for much skill and attention to detail, if the finished product is to be at all efficient. Some translators go on to a further phase by incorporating a so-called **peephole optimizer** in which attempts are made to reduce unnecessary operations still further by examining short sequences of generated code in closer detail.

Below we list the actual code generated by various MS-DOS compilers for this statement. It is readily apparent that the code generation phases in these compilers are markedly different. Such differences can have a profound effect on program size and execution speed.

```
Borland C++ 3.1 (47 bytes)

CS:A0 BBB702     MOV  BX,02B7
CS:A3 C746FE5100 MOV  WORD PTR[BP-2],0051
CS:A8 EB07       JMP  B1
CS:AA 8BC3       MOV  AX,BX
CS:AC 0346FE     ADD  AX,[BP-2]
CS:AF 8BD8       MOV  BX,AX
CS:B1 83FB01     CMP  BX,1
CS:B4 7E05       JLE  BB
CS:B6 B80100     MOV  AX,1
CS:B9 EB02       JMP  BD
CS:BB 33C0       XOR  AX,AX
CS:BD 50         PUSH AX
CS:BE 83FB09     CMP  BX,9
CS:C1 7D05       JGE  C8
CS:C3 B80100     MOV  AX,1
CS:C6 EB02       JMP  CA
CS:C8 33C0       XOR  AX,AX
CS:CA 5A         POP  DX
CS:CB 85D0       TEST DX,AX
CS:CD 75DB       JNZ  AA
```

```
Turbo Pascal (46 bytes)
(with no short circuit evaluation)

CS:09 833E3E0009 CMP  WORD PTR[003E],9
CS:0E 7C04       JL   14
CS:10 B000       MOV  AL,0
CS:12 EB02       JMP  16
CS:14 B001       MOV  AL,1
CS:16 8AD0       MOV  DL,AL
CS:18 833E3E0001 CMP  WORD PTR[003E],1
CS:1D 7F04       JG   23
CS:1F B000       MOV  AL,0
CS:21 EB02       JMP  25
CS:23 B001       MOV  AL,01
CS:25 22C2       AND  AL,DL
CS:27 08C0       OR   AL,AL
CS:29 740C       JZ   37
CS:2B A13E00     MOV  AX,[003E]
CS:2E 03064000   ADD  AX,[0040]
CS:32 A33E00     MOV  [003E],AX
CS:35 EBD2       JMP  9
```

```
JPI TopSpeed Modula-2 (29 bytes)

CS:19 2E         CS:
CS:1A 8E1E2700   MOV  DS,[0027]
CS:1E 833E000001 CMP  WORD PTR[0000],1
CS:23 7E11       JLE  36
CS:25 833E000009 CMP  WORD PTR[0000],9
CS:2A 7D0A       JGE  36
CS:2C 8B0E0200   MOV  CX,[0002]
CS:30 010E0000   ADD  [0000],CX
CS:34 EBE3       JMP  19
```

```
Stony Brook QuickMod (24 bytes)

CS:69 BB2D00     MOV  BX,2D
CS:6C B90200     MOV  CX,2
CS:6F E90200     JMP  74
CS:72 01D9       ADD  CX,BX
CS:74 83F901     CMP  CX,1
CS:77 7F03       JG   7C
CS:79 E90500     JMP  81
CS:7C 83F909     CMP  CX,9
CS:7F 7CF1       JL   72
```

A translator inevitably makes use of a complex data structure, known as the **symbol table**, in which it keeps track of the names used by the program, and associated properties for these, such as their type, and their storage requirements (in the case of variables), or their values (in the case of constants).

As is well known, users of high-level languages are apt to make many errors in the development of even quite simple programs. Thus the various phases of a compiler, especially the earlier ones, also communicate with an **error handler** and **error reporter** which are invoked when errors are detected. It is desirable that compilation of erroneous programs be continued, if possible, so that the user can clean several errors out of the source before recompiling. This raises very interesting issues regarding the design of **error recovery** and **error correction** techniques. (We speak of error recovery when the translation process attempts to carry on after detecting an error, and of error correction or error repair when it attempts to correct the error from context - usually a contentious subject, as the correction may be nothing like what the programmer originally had in mind.)

Error detection at compile-time in the source code must not be confused with error detection at run-time when executing the object code. Many code generators are responsible for adding error-checking code to the object program (to check that subscripts for arrays stay in bounds, for example). This may be quite rudimentary, or it may involve adding considerable code and data structures for use with sophisticated debugging systems. Such ancillary code can drastically reduce the efficiency of a program, and some compilers allow it to be suppressed.

Sometimes mistakes in a program that are detected at compile-time are known as *errors*, and errors that show up at run-time are known as *exceptions*, but there is no universally agreed terminology for this.

Figure 2.4 seems to imply that compilers work serially, and that each phase communicates with the next by means of a suitable intermediate language, but in practice the distinction between the various phases often becomes a little blurred. Moreover, many compilers are actually constructed around a central parser as the dominant component, with a structure rather more like the one in Figure 2.7.

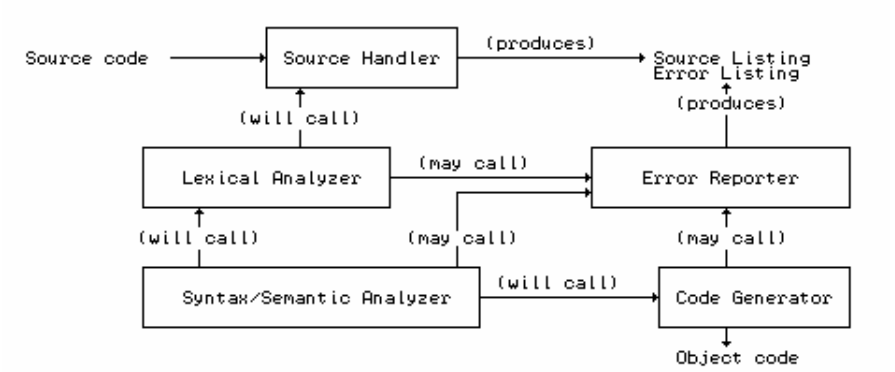


Figure 2.7 Structure of a parser-directed compiler

Exercises

2.5 What sort of problems can you foresee a Fortran compiler having in analysing statements beginning

```
      IF ( I(J) - I(K) ) ........
      CALL IF (4 ,    ...........
      IF (3 .EQ. MAX) GOTO ......
100   FORMAT(X3H)=(I5)
```

2.6 What sort of code would *you* have produced had you been coding a statement like "`WHILE (1 <`

`P) AND (P < 9) DO P := P + Q END`" into your favourite ASSEMBLER language?

2.7 Draw the concrete syntax tree for the C++ version of the *while* statement used for illustration in this section.

2.8 Are there any reasons why short-circuit evaluation should be preferred over the Boolean operator approach? Can you think of any algorithms that would depend critically on which approach was adopted?

2.9 Write down a few other high-level constructs and try to imagine what sort of ASSEMBLER-like machine code a compiler would produce for them.

2.10 What do you suppose makes it relatively easy to compile Pascal? Can you think of any aspects of Pascal which could prove really difficult?

2.11 We have used two undefined terms which at first seem interchangeable, namely "separate" and "independent" compilation. See if you can discover what the differences are.

2.12 Many development systems - in particular debuggers - allow a user to examine the object code produced by a compiler. If you have access to one of these, try writing a few very simple (single statement) programs, and look at the sort of object code that is generated for them.

2.4 Multi-stage translators

Besides being conceptually divided into phases, translators are often divided into **passes**, in each of which several phases may be combined or interleaved. Traditionally, a pass reads the source program, or output from a previous pass, makes some transformations, and then writes output to an intermediate file, whence it may be rescanned on a subsequent pass.

These passes may be handled by different integrated parts of a single compiler, or they may be handled by running two or more separate programs. They may communicate by using their own specialized forms of intermediate language, they may communicate by making use of internal data structures (rather than files), or they may make several passes over the same original source code.

The number of passes used depends on a variety of factors. Certain languages require at least two passes to be made if code is to be generated easily - for example, those where declaration of identifiers may occur after the first reference to the identifier, or where properties associated with an identifier cannot be readily deduced from the context in which it first appears. A multi-pass compiler can often save space. Although modern computers are usually blessed with far more memory than their predecessors of only a few years back, multiple passes may be an important consideration if one wishes to translate complicated languages within the confines of small systems. Multi-pass compilers may also allow for better provision of code optimization, error reporting and error handling. Lastly, they lend themselves to team development, with different members of the team assuming responsibility for different passes. However, multi-pass compilers are usually slower than single-pass ones, and their probable need to keep track of several files makes them slightly awkward to write and to use. Compromises at the design stage often result in languages that are well suited to single-pass compilation.

In practice, considerable use is made of two-stage translators in which the first stage is a high-level

translator that converts the source program into ASSEMBLER, or even into some other relatively high-level language for which an efficient translator already exists. The compilation process would then be depicted as in Figure 2.8 - our example shows a Modula-3 program being prepared for execution on a machine that has a Modula-3 to C converter:

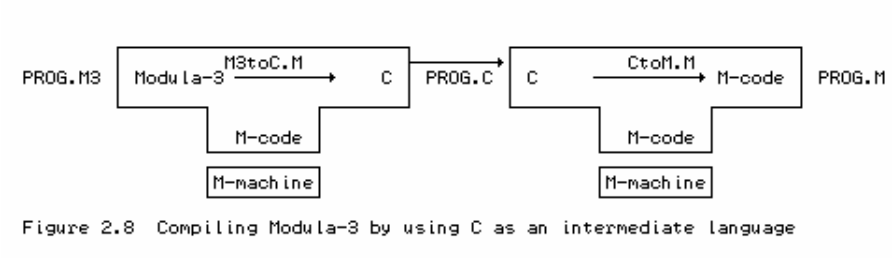


Figure 2.8 Compiling Modula-3 by using C as an intermediate language

It is increasingly common to find compilers for high-level languages that have been implemented using C, and which themselves produce C code as output. The success of these is based on the premises that "all modern computers come equipped with a C compiler" and "source code written in C is truly portable". Neither premise is, unfortunately, completely true. However, compilers written in this way are as close to achieving the dream of themselves being portable as any that exist at the present time. The way in which such compilers may be used is discussed further in Chapter 3.

Exercises

2.13 Try to find out which of the compilers you have used are single-pass, and which are multi-pass, and for the latter, find out how many passes are involved. Which produce relocatable code needing further processing by linkers or linkage editors?

2.14 Do any of the compilers in use on your system produce ASSEMBLER, C or other such code during the compilation process? Can you foresee any particular problems that users might experience in using such compilers?

2.15 One of several compilers that translates from Modula-2 to C is called `mtc`, and is freely available from several ftp sites. If you are a Modula-2 programmer, obtain a copy, and experiment with it.

2.16 An excellent compiler that translates Pascal to C is called `p2c`, and is widely available for Unix systems from several ftp sites. If you are a Pascal programmer, obtain a copy, and experiment with it.

2.17 Can you foresee any practical difficulties in using C as an intermediate language?

2.5 Interpreters, interpretive compilers, and emulators

Compilers of the sort that we have been discussing have a few properties that may not immediately be apparent. Firstly, they usually aim to produce object code that can run at the full speed of the target machine. Secondly, they are usually arranged to compile an entire section of code before any of it can be executed.

In some interactive environments the need arises for systems that can execute part of an application without preparing all of it, or ones that allow the user to vary his or her course of action on the fly. Typical scenarios involve the use of spreadsheets, on-line databases, or batch files or shell scripts for operating systems. With such systems it may be feasible (or even desirable) to exchange some of the advantages of speed of execution for the advantage of procuring results on demand.

Systems like these are often constructed so as to make use of an **interpreter**. An interpreter is a translator that effectively accepts a source program and executes it directly, without, seemingly, producing any object code first. It does this by fetching the source program instructions one by one, analysing them one by one, and then "executing" them one by one. Clearly, a scheme like this, if it is to be successful, places some quite severe constraints on the nature of the source program. Complex program structures such as nested procedures or compound statements do not lend themselves easily to such treatment. On the other hand, one-line queries made of a data base, or simple manipulations of a row or column of a spreadsheet, can be handled very effectively.

This idea is taken quite a lot further in the development of some translators for high-level languages, known as **interpretive compilers**. Such translators produce (as output) intermediate code which is intrinsically simple enough to satisfy the constraints imposed by a practical interpreter, even though it may still be quite a long way from the machine code of the system on which it is desired to execute the original program. Rather than continue translation to the level of machine code, an alternative approach that may perform acceptably well is to use the intermediate code as part of the input to a specially written interpreter. This in turn "executes" the original algorithm, by simulating a virtual machine for which the intermediate code effectively *is* the machine code. The distinction between the machine code and pseudo-code approaches to execution is summarized in Figure 2.9.

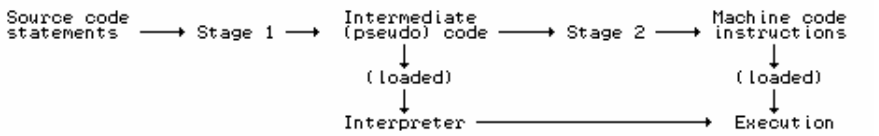


Figure 2.9 Differences between native-code and pseudo-code compilers

We may depict the process used in an interpretive compiler running under MS-DOS for a toy language like Clang, the one illustrated in later chapters, in T-diagram form (see Figure 2.10).

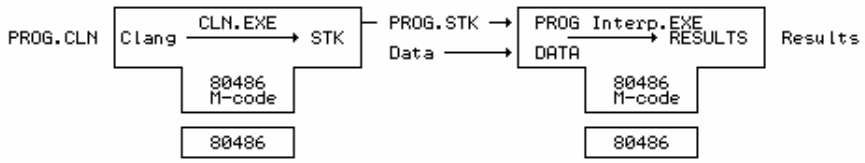


Figure 2.10 An interpretive compiler/interpreter for Clang

It is not necessary to confine interpreters merely to work with intermediate output from a translator. More generally, of course, even a real machine can be viewed as a highly specialized interpreter - one that executes the machine level instructions by fetching, analysing, and then interpreting them one by one. In a real machine this all happens "in hardware", and hence very quickly. By carrying on this train of thought, the reader should be able to see that a program could be written to allow one real machine to emulate any other real machine, albeit perhaps slowly, simply by writing an interpreter - or, as it is more usually called, an **emulator** - for the second machine.

For example, we might develop an emulator that runs on a Sun SPARC machine and makes it appear to be an IBM PC (or the other way around). Once we have done this, we are (in principle) in a position to execute any software developed for an IBM PC on the Sun SPARC machine - effectively the PC software becomes portable!

The T-diagram notation is easily extended to handle the concept of such virtual machines. For example, running Turbo Pascal on our Sun SPARC machine could be depicted by Figure 2.11.

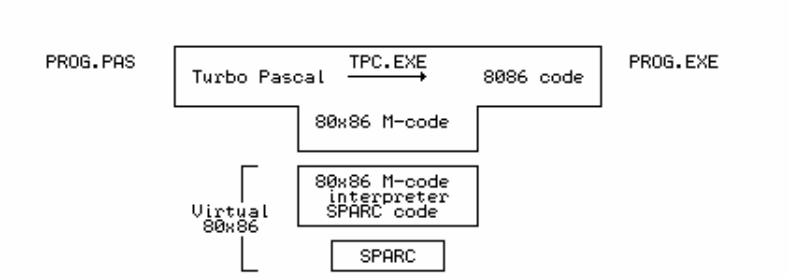


Figure 2.11 Executing the Turbo Pascal compiler on a Sun SPARC

The interpreter/emulator approach is widely used in the design and development both of new machines themselves, and the software that is to run on those machines.

An interpretive approach may have several points in its favour:

- It is far easier to generate hypothetical machine code (which can be tailored towards the quirks of the original source language) than real machine code (which has to deal with the uncompromising quirks of real machines).

- A compiler written to produce (as output) well-defined pseudo-machine code capable of easy interpretation on a range of machines can be made highly portable, especially if it is written in a host language that is widely available (such as ANSI C), or even if it is made available already implemented in its own pseudo- code.

- It can more easily be made "user friendly" than can the native code approach. Since the interpreter works closer to the source code than does a fully translated program, error messages and other debugging aids may readily be related to this source.

- A whole range of languages may quickly be implemented in a useful form on a wide range of different machines relatively easily. This is done by producing intermediate code to a well-defined standard, for which a relatively efficient interpreter should be easy to implement on any particular real machine.

- It proves to be useful in connection with cross-translators such as were mentioned earlier. The code produced by such translators can sometimes be tested more effectively by simulated execution on the donor machine, rather than after transfer to the target machine - the delays inherent in the transfer from one machine to the other may be balanced by the degradation of execution time in an interpretive simulation.

- Lastly, intermediate languages are often very compact, allowing large programs to be handled, even on relatively small machines. The success of the once very widely used UCSD Pascal and UCSD p-System stands as an example of what can be done in this respect.

For all these advantages, interpretive systems carry fairly obvious overheads in execution speed, because execution of intermediate code effectively carries with it the cost of virtual translation into machine code each time a hypothetical machine instruction is obeyed.

One of the best known of the early **portable interpretive compilers** was the one developed at Zürich and known as the "Pascal-P" compiler (Nori *et al.*, 1981). This was supplied in a kit of three components:

- The first component was the source form of a Pascal compiler, written in a very complete subset of the language, known as Pascal-P. The aim of this compiler was to translate Pascal-P source programs into a well-defined and well-documented intermediate language, known as P-code, which was the "machine code" for a hypothetical stack-based computer, known as the P-machine.

- The second component was a compiled version of the first - the P-codes that would be produced by the Pascal-P compiler, were it to compile itself.

- Lastly, the kit contained an interpreter for the P-code language, supplied as a Pascal algorithm.

The interpreter served primarily as a model for writing a similar program for the target machine, to allow it to emulate the hypothetical P-machine. As we shall see in a later chapter, emulators are relatively easy to develop - even, if necessary, in ASSEMBLER - so that this stage was usually fairly painlessly achieved. Once one had loaded the interpreter - that is to say, the version of it tailored to a local real machine - into a real machine, one was in a position to "execute" P-code, and in particular the P-code of the P-compiler. The compilation and execution of a user program could then be achieved in a manner depicted in Figure 2.12.

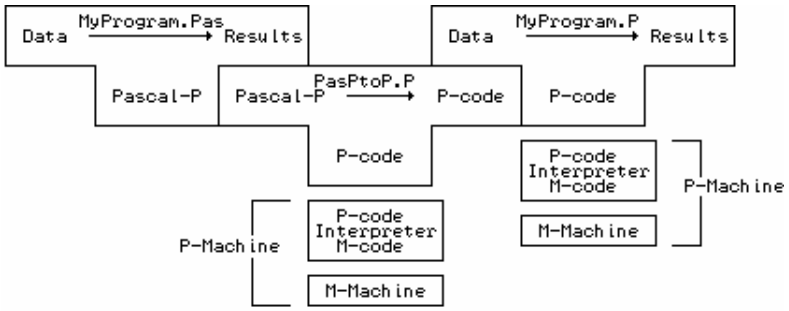


Figure 2.12 Compiling and executing a program with the P-compiler

---

Exercises

2.18 Try to find out which of the translators you have used are interpreters, rather than full compilers.

2.19 If you have access to both a native-code compiler and an interpreter for a programming language known to you, attempt to measure the loss in efficiency when the interpreter is used to run a large program (perhaps one that does substantial number-crunching).