Chapter 5

Compilers

In this chapter we discuss the design and operation of compilers for high-level programming languages. Many textbooks and courses are entirely devoted to compiler construction. We obviously cannot hope to cover the subject thoroughly in a single chapter. Instead, our goal is to give the reader an understanding of how a typical compiler works. We introduce the most important concepts and issues related to compilers, and illustrate them with examples. As each subject is discussed, we give references for those readers who want to explore the topic in more detail.

Section 5.1 presents the basic functions of a simple one-pass compiler. We illustrate the operation of such a compiler by following an example program through the entire translation process. This section contains somewhat more detail than the other parts of the chapter because of the fundamental importance of the material.

Section 5.2 discusses machine-dependent extensions to the basic scheme presented in Section 5.1. These extensions are mainly in the area of object code generation and optimization. Section 5.3 describes some machine-independent extensions to the basic scheme.

Section 5.4 describes some compiler design alternatives. These include multi-pass compilers, interpreters, P-code compilers, and compiler-compilers. Finally, Section 5.5 presents five examples of actual compilers and compiler-writing systems, and relates them to the concepts introduced in previous sections.

5.1 BASIC COMPILER FUNCTIONS

This section introduces the fundamental operations that are necessary in compiling a typical high-level language program. We use as an example the Pascal program in Fig. 5.1; however, the concepts and approaches that we discuss can also be applied to the compilation of programs in other languages.

For the purposes of compiler construction, a high-level programming language is usually described in terms of a *grammar*. This grammar specifies the form, or *syntax*, of legal statements in the language. For example, an assignment

```
PROGRAM STATS
VAR
    SUM,SUMSQ,I,VALUE,MEAN,VARIANCE : INTEGER
BEGIN
    SUM  := 0;
    SUMSQ := 0;
    FOR I := 1 TO 100 DO
        BEGIN
            READ(VALUE);
            SUM := SUM + VALUE;
            SUMSQ := SUMSQ + VALUE * VALUE
        END;
    MEAN := SUM DIV 100;
    VARIANCE := SUMSQ DIV 100 - MEAN * MEAN;
    WRITE(MEAN,VARIANCE)
END
```

Figure 5.1 Example of a Pascal program.

statement might be defined by the grammar as a variable name, followed by an assignment operator (:=), followed by an expression. The problem of compilation then becomes one of matching statements written by the programmer to structures defined by the grammar, and generating the appropriate object code for each statement.

It is convenient to regard a source program statement as a sequence of *tokens* rather than simply as a string of characters. Tokens may be thought of as the fundamental building blocks of the language. For example, a token might be a keyword, a variable name, an integer, an arithmetic operator, etc. The task of scanning the source statement, recognizing and classifying the various tokens, is known as *lexical analysis*. The part of the compiler that performs this analytic function is commonly called the *scanner*.

After the token scan, each statement in the program must be recognized as some language construct, such as a declaration or an assignment statement, described by the grammar. This process, which is called *syntactic analysis* or *parsing*, is performed by a part of the compiler that is usually called the *parser*. The last step in the basic translation process is the generation of object code. Most compilers create machine-language programs directly instead of producing a symbolic program for later translation by an assembler.

Although we have mentioned three steps in the compilation process—scanning, parsing, and code generation—it is important to realize that a compiler does not necessarily make three passes over the program being translated. For some languages, it is quite possible to compile a program in a single pass. Our discussions in this section describe how such a one-pass compiler might work. On the other hand, compilers for other languages and compilers that perform sophisticated code optimization or other analysis of the program generally make several passes. In Section 5.4 we discuss the division of a compiler into passes. Section 5.5 gives several examples of the structure of actual compilers.

In the following sections we discuss the basic elements of a simple compilation process, illustrating their application to the example program in Fig. 5.1. Section 5.1.1 introduces some concepts and notation used in specifying grammars for programming languages. Sections 5.1.2 through 5.1.4 discuss, in turn, the functions of lexical analysis, syntactic analysis, and code generation.

5.1.1 Grammars

A grammar for a programming language is a formal description of the *syntax*, or form, of programs and individual statements written in the language. The grammar does not describe the *semantics*, or meaning, of the various statements; such knowledge must be supplied in the code-generation routines. As an illustration of the difference between syntax and semantics, consider the two statements

```
I := J + K
```

and

```
X := Y + I
```

where X and Y are REAL variables and I, J, K are INTEGER variables. These two statements have identical syntax. Each is an assignment statement; the value to be assigned is given by an expression that consists of two variable names separated by the operator +. However, the semantics of the two statements are quite different. The first statement specifies that the variables in the expression are to be added using integer arithmetic operations. The second statement specifies a floating-point addition, with the integer operand I being converted to floating point before adding. Obviously, these two statements would be compiled into very different sequences of machine instructions. However, they would be described in the same way by the grammar. The differences between the statements would be recognized during code generation.

A number of different notations can be used for writing grammars. The one we describe is called BNF (for Backus-Naur Form). BNF is not the most powerful syntax description tool available. In fact, it is not even totally adequate for the description of some real programming languages. It does,

however, have the advantages of being simple and widely used, and it provides capabilities that are sufficient for most purposes. Figure 5.2 gives one possible BNF grammar for a highly restricted subset of the Pascal language. A complete BNF grammar for Pascal can be found in Jensen and Wirth (1974). In the remainder of this section, we discuss this grammar and show how it relates to the example program in Fig. 5.1.

A BNF grammar consists of a set of *rules*, each of which defines the syntax of some construct in the programming language. Consider, for example, Rule 13 in Fig. 5.2:

```
<read> ::= READ ( <id-list> )
```

This is a definition of the syntax of a Pascal READ statement that is denoted in the grammar as <read>. The symbol ::= can be read "is defined to be." On the left of this symbol is the language construct being defined, <read>, and on the right is a description of the syntax being defined for it. Character strings enclosed between the angle brackets < and > are called *nonterminal symbols*. These are the names of constructs defined in the grammar. Entries not enclosed in angle brackets are *terminal symbols* of the grammar (i.e., tokens). In this rule, the nonterminal symbols are <read> and <id-list>, and the terminal symbols are the tokens READ, (, and). Thus this rule specifies that a <read> consists of the token READ, followed by the token (, followed by a language construct <id-list>, followed by the token). The blank spaces in the grammar rules are not significant. They have been included only to improve readability.

```
1  <prog>          ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2  <prog-name>     ::= id
3  <dec-list>      ::= <dec> | <dec-list> ; <dec>
4  <dec>           ::= <id-list> : <type>
5  <type>          ::= INTEGER
6  <id-list>       ::= id | <id-list> , id
7  <stmt-list>     ::= <stmt> | <stmt-list> ; <stmt>
8  <stmt>          ::= <assign> | <read> | <write> | <for>
9  <assign>        ::= id := <exp>
10 <exp>           ::= <term> | <exp> + <term> | <exp> - <term>
11 <term>          ::= <factor> | <term> * <factor> | <term> DIV <factor>
12 <factor>        ::= id | int | ( <exp> )
13 <read>          ::= READ ( <id-list> )
14 <write>         ::= WRITE ( <id-list> )
15 <for>           ::= FOR <index-exp> DO <body>
16 <index-exp>     ::= id := <exp> TO <exp>
17 <body>          ::= <stmt> | BEGIN <stmt-list> END
```

Figure 5.2 Simplified Pascal grammar.

To recognize a <read>, of course, we also need the definition of <id-list>. This is provided by Rule 6 in Fig. 5.2:

```
<id-list>  ::= id | <id-list>, id
```

This rule offers two possibilities, separated by the | symbol, for the syntax of an <id-list>. The first alternative specifies that an <id-list> may consist simply of a token **id** (the notation **id** denotes an identifier that is recognized by the scanner). The second syntax alternative is an <id-list>, followed by the token "," (comma), followed by a token **id**. Note that this rule is recursive, which means the construct <id-list> is defined partially in terms of itself. By trying a few examples you should be able to see that this rule includes in the definition of <id-list> any sequence of one or more **id**'s separated by commas. Thus

```
ALPHA
```

is an <id-list> that consists of a single **id** ALPHA;

```
ALPHA BETA
```

is an <id-list> that consists of another <id-list> ALPHA, followed by a comma, followed by an **id** BETA, and so forth.

It is often convenient to display the analysis of a source statement in terms of a grammar as a tree. This tree is usually called the *parse tree*, or *syntax tree*, for the statement. Figure 5.3(a) shows the parse tree for the statement

```
READ (VALUE)
```

in terms of the two rules just discussed.

Rule 9 of the grammar in Fig. 5.2 provides a definition of the syntax of an assignment statement:

```
<assign>  ::= id := <exp>
```

That is, an <assign> consists of an **id**, followed by the token :=, followed by an expression <exp>. Rule 10 gives a definition of an <exp>:

```
<exp>  ::= <term>  |  <exp> + <term>  |  <exp> - <term>
```

Figure 5.3 Parse trees for two statements from Fig. 5.1.

By reasoning similar to that applied to <id-list>, we can see that this rule defines an expression <exp> to be any sequence of <term>s connected by the operators + and –. Similarly, Rule 11 defines a <term> to be any sequence of <factor>s connected by * and DIV. Rule 12 specifies that a <factor> may consist of an identifier **id** or an integer **int** (which is also recognized by the scanner) or an <exp> enclosed in parentheses.

Figure 5.3(b) shows the parse tree for statement 14 from Fig. 5.1 in terms of the rules just described. You should examine this figure carefully to be sure you understand the analysis of the source statement according to the rules of the grammar. In Section 5.1.3, we discuss methods for performing this sort of syntactic analysis in a compiler.

Note that the parse tree in Fig. 5.3(b) implies that multiplication and division are done before addition and subtraction. The terms SUMSQ DIV 100 and MEAN * MEAN must be calculated first since these intermediate results are the operands (left and right subtrees) for the – operation. Another way of saying this is that multiplication and division have higher *precedence* than addition and subtraction. These rules of precedence are implied by the way Rules 10–12 are constructed (see Exercise 5.1.3). In Section 5.1.3 we see a way to make use of such precedence relationships during the parsing process.

The parse trees shown in Fig. 5.3 represent the only possible ways to analyze these two statements in terms of the grammar of Fig. 5.2. For some grammars, this might not be the case. If there is more than one possible parse tree for a given statement, the grammar is said to be *ambiguous*. We prefer to use unambiguous grammars in compiler construction because, in some cases, an ambiguous grammar would leave doubt about what object code should be generated.

Figure 5.4 shows the parse tree for the entire program in Fig. 5.1. You should examine this figure carefully to see how the form and structure of the program correspond to the rules of the grammar in Fig. 5.2.

5.1.2 Lexical Analysis

Lexical analysis involves scanning the program to be compiled and recognizing the tokens that make up the source statements. Scanners are usually designed to recognize keywords, operators, and identifiers, as well as integers, floating-point numbers, character strings, and other similar items that are written as part of the source program. The exact set of tokens to be recognized, of course, depends upon the programming language being compiled and the grammar being used to describe it.

Items such as identifiers and integers are usually recognized directly as single tokens. As an alternative, these tokens could be defined as a part of the grammar. For example, an identifier might be defined by the rules

```
<ident>    ::=  <letter> | <ident> <letter> | <ident> <digit>
<letter>   ::=  A | B | C | D | ... | Z
<digit>    ::=  0 | 1 | 2 | 3 | ... | 9
```

In such a case the scanner would recognize as tokens the single characters A, B, 0, 1, and so on. The parser would interpret a sequence of such characters as the language construct <ident>. However, this approach would require the parser

Figure 5.4 Parse tree for the program from Fig. 5.1.

to recognize simple identifiers using general parsing techniques such as those discussed in the next section. A special-purpose routine such as the scanner can perform this same function much more efficiently. Since a large part of the source program consists of such multiple-character identifiers, this saving in compilation time can be highly significant. In addition, restrictions such as a limitation on the length of identifiers are easier to include in a scanner than in a general-purpose parsing routine.

Similarly, the scanner generally recognizes both single- and multiple-character tokens directly. For example, the character string READ would be interpreted as a single token rather than as a sequence of four tokens R, E, A, D. The string := would be recognized as a single assignment operator, not as : followed by =. It is, of course, possible to handle multiple-character tokens one character at a time, but such an approach creates considerably more work for the parser.

The output of the scanner consists of a sequence of tokens. For efficiency of later use, each token is usually represented by some fixed-length code, such as an integer, rather than as a variable-length character string. In such a coding scheme for the grammar of Fig. 5.2 (shown in Fig. 5.5) the token PROGRAM

Token	Code
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
,	14
:=	15
+	16
-	17
*	18
DIV	19
(	20
)	21
id	22
int	23

Figure 5.5 Token coding scheme for the grammar from Fig. 5.2.

would be represented by the integer value 1, an identifier **id** would be represented by the value 22, and so on.

When the token being scanned is a keyword or an operator, such a coding scheme gives sufficient information. In the case of an identifier, however, it is also necessary to specify the particular identifier name that was scanned. The same is true of integers, floating-point values, character-string constants, etc. This can be accomplished by associating a *token specifier* with the type code for such tokens. This specifier gives the identifier name, integer value, etc., that was found by the scanner. Some scanners are designed to enter identifiers directly into a symbol table, which is similar to the symbol table used by an assembler, when they are first recognized. In that case, the token specifier for an identifier might be a pointer to the symbol-table entry for that identifier. This approach avoids much of the need for table searching during the rest of the compilation process.

Figure 5.6 shows the output from a scanner for the program in Fig. 5.1, using the token coding scheme in Fig. 5.5. For token type 22 (identifier), the token specifier is a pointer to a symbol-table entry (denoted by ^SUM, ^SUMSQ, etc.). For token type 23 (integer), the specifier is the value of the integer (denoted by #0, #100, etc.).

We have shown the output from the scanner as a list of token codings; however, this does not mean that the entire program is scanned at one time, before any other processing. More often, the scanner operates as a procedure that is called by the parser when it needs another token. In this case, each call to the scanner would produce the coding (and specifier, if any) for the next token in the source program. The parser would be responsible for saving any tokens that it might require for later analysis. An example of this can be found in the next section.

In addition to its primary function of recognizing tokens, the scanner usually is responsible for reading the lines of the source program as needed, and possibly for printing the source listing. Comments are ignored by the scanner, except for printing on the output listing, so they are effectively removed from the source statements before parsing begins.

Modeling Scanners as Finite Automata

The tokens of most programming languages can be recognized by a *finite automaton*. Mathematically, a finite automaton consists of a finite set of *states* and a set of *transitions* from one state to another. One of the states is designated as the *starting state*, and one or more states are designated as *final states*.

Finite automata are often represented graphically, as illustrated in Fig. 5.7(a). States are represented by circles, and transitions by arrows from one state to another. Each arrow is labeled with a character or a set of characters that cause the specified transition to occur. The starting state has an arrow

Line	Token type	Token specifier
1	1	
	22	^STATS
2	2	
3	22	^SUM
	14	
	22	^SUMSQ
	14	
	22	^I
	14	
	22	^VALUE
	14	
	22	^MEAN
	14	
	22	^VARIANCE
	13	
	6	
4	3	
5	22	^SUM
	15	
	23	#0
	12	
6	22	^SUMSQ
	15	
	23	#0
	12	
7	7	
	22	^I
	15	
	23	#1
	10	
	23	#100
	11	
8	3	
9	8	
	20	
	22	^VALUE
	21	
	12	

Line	Token type	Token specifier
10	22	^SUM
	15	
	22	^SUM
	16	
	22	^VALUE
	12	
11	22	^SUMSQ
	15	
	22	^SUMSQ
	16	
	22	^VALUE
	18	
	22	^VALUE
12	4	
	12	
13	22	^MEAN
	15	
	22	^SUM
	19	
	23	#100
	12	
14	22	^VARIANCE
	15	
	22	^SUMSQ
	19	
	23	#100
	17	
	22	^MEAN
	18	
	22	^MEAN
	12	
15	9	
	20	
	22	^MEAN
	14	
	22	^VARIANCE
	21	
16	5	

Figure 5.6 Lexical scan of the program from Fig. 5.1.

(a)

a b c	{recognized}
a b c c a b c	{recognized}
a c	{not recognized}

(b)

Figure 5.7 Graphical representation of a finite automaton.

entering it that is not connected to anything else [see State 1 in Fig. 5.7(a)]. Final states are identified by double circles (see State 4).

We can visualize the finite automaton as beginning in the starting state, and moving from one state to another as it examines the characters being scanned. It stops when there is no transition from its current state that matches the next character to be scanned (or when there are no more characters to scan). If the automaton stops in a final state, we say that it *recognizes* (or *accepts*) the string being scanned. If it stops in a non-final state, it fails to recognize (or rejects) the string.

Consider, for example, the finite automaton shown in Fig. 5.7(a) and the first input string in Fig. 5.7(b). The automaton starts in State 1 and examines the first character of the input string. The character *a* causes the automaton to move from State 1 to State 2. The *b* causes a transition from State 2 to State 3, and the *c* causes a transition from State 3 to State 4. At this point, all characters have been scanned, so the finite automaton stops in State 4. Because this is a final state, the automaton recognizes the string *abc*.

Similarly, consider the second input string shown in Fig. 5.7(b). The scanning of the first three characters happens exactly as described above. This time, however, there are still characters left in the input string. The fourth character of the string (the second *c*) causes the automaton to remain in State 4 (note the arrow labeled with *c* that loops back to State 4). The following *a* takes the automaton back to State 2. At the end of the input string, the finite automaton is again in State 4, so it recognizes the string *abccabc*.

On the other hand, consider the third input string in Fig. 5.7(b). The finite automaton begins in State 1, as before, and the *a* causes a transition from

State 1 to State 2. Now the next character to be scanned is *c*. However, there is no transition from State 2 that is labeled with *c*. Therefore, the automaton must stop in State 2. Because this is not a final state, the finite automaton fails to recognize the input string. If you try some other examples, you will discover that the finite automaton in Fig. 5.7(a) recognizes tokens of the form *abc...abc...* where the grouping *abc* is repeated one or more times, and the *c* within each grouping may also be repeated.

Figure 5.8 shows several finite automata that are designed to recognize typical programming language tokens. Figure 5.8(a) recognizes identifiers and keywords that begin with a letter and may continue with any sequence of letters and digits. Notice the notation A–Z, which specifies that any character from A to Z may cause the indicated transition. For simplicity, we have considered only uppercase letters in this example.

Some languages allow identifiers such as NEXT_LINE, which contains the underscore character (_). Figure 5.8(b) shows a finite automaton that recognizes identifiers of this type. Notice that this automaton does not allow identifiers that begin or end with an underscore, or that contain two consecutive underscores.

The finite automaton in Fig. 5.8(c) recognizes integers that consist of a string of digits, including those that contain leading zeroes, such as 000025. Figure 5.8(d) shows an automaton that does not allow leading zeroes, except in the case of the integer 0. An integer that consists only of the digit 0 must be followed by a space to separate it from the following token. You are encouraged to try several example strings with the finite automata in Fig. 5.8, to be sure you see how they work.

Each of the finite automata we have seen so far was designed to recognize one particular type of token. Figure 5.9 shows a finite automaton that can recognize all of the tokens listed in Fig. 5.5. Notice that for simplicity we have chosen to recognize all identifiers and keywords with one final state (State 2). A separate table look-up operation could then be used to distinguish keywords. Likewise, a separate check could be made to ensure that identifiers are of a length permitted by the language definition. (Finite automata cannot easily represent limitations on the length of strings being recognized.)

A similar kind of special case occurs in State 3. This state is included to recognize the keyword "END." (token type 5 in Fig. 5.5). Suppose, however, that the scanner encounters an erroneous token such as "VAR.". When the automaton stops in State 3, the scanner should perform a check to see whether the string being recognized is "END.". If it is not, the scanner could, in effect, back up to State 2 (recognizing the "VAR"). The period would then be re-scanned as part of the following token the next time the scanner is called. Notice that this kind of backup is not required with State 7. The sequence := is always recognized as an assignment operator, not as : followed by =.

Figure 5.8 Finite automata for typical programming language tokens.

Figure 5.9 Finite automaton to recognize tokens from Fig. 5.5.

Finite automata provide an easy way to visualize the operation of a scanner. However, the real advantage of this kind of representation is in ease of implementation. Consider again the problem of recognizing identifiers that may contain underscores. Figure 5.10(a) shows a typical algorithm to recognize such a token.

Figure 5.10(b) shows the finite automaton from Fig. 5.8(b) represented in a tabular form. Each row of the table corresponds to one of the states of the automaton. Each entry in the row specifies the transition that occurs when a character listed in the heading of that column is scanned. If there is no entry in a column, there is no transition corresponding to that character, and the automaton halts. For example, the first row (corresponding to State 1) specifies a transition to State 2 if one of the characters A–Z is scanned; the automaton halts if 0–9 or underscore is scanned.

It is easy to imagine a program that simulates the operation of a finite automaton by traversing the tabular representation. With such a program, the implementation of the automaton would simply require writing down the entries in the table. The tabular representation is usually much clearer and less error-prone than an algorithmic representation such as Fig. 5.10(a). It is also much easier to change table entries than to modify nested loops or procedure calls.

```
get first Input_Character
if Input_Character in ['A'..'Z'] then
   begin
      while Input_Character in ['A'..'Z', '0'..'9'] do
         begin
            get next Input_Character
            if Input_Character = '_' then
               begin
                  get next Input_Character
                  Last_Char_Is_Underscore := true
               end  {if '_'}
            else
               Last_Char_Is_Underscore := false
         end  {while}
      if Last_Char_Is_Underscore then
         return (Token_Error)
      else
         return (Valid_Token)
   end  {if first in ['A'..'Z']}
else
   return (Token_Error)
```

(a)

State	A-Z	0-9	_	
1	2			{starting state}
2	2	2	3	{final state}
3	2	2		

(b)

Figure 5.10 Token recognition using (a) algorithmic code and (b) tabular representation of finite automaton.

Further discussions of finite automata and the implementation of scanners can be found in Aho et al. (1988).

5.1.3 Syntactic Analysis

During syntactic analysis, the source statements written by the programmer are recognized as language constructs described by the grammar being used. We may think of this process as building the parse tree for the statements being translated. Parsing techniques are divided into two general classes—*bottom-up* and *top-down*—according to the way in which the parse tree is constructed. Top-down methods begin with the rule of the grammar that specifies the goal of the analysis (i.e., the root of the tree), and attempt to construct

Suppose now that procedure C calls D. (This is allowed because the identifier D is defined in procedure A, which contains C. For simplicity, we have assumed that no special "forward call" declarations are required.) The resulting stack and display are shown in Fig. 5.28(c). An activation record for D has been created in the usual way and added to the stack. Note, however, that the display now contains only two pointers: one each to the activation records for D and A. This is because procedure D cannot refer to variables in B or C, except through parameters that are passed to it, even though it was called from C. According to the rules for the scope of names in a block-structured language, procedure D can refer only to variables that are declared by D or by some block that contains D in the source program (in this case, procedure A).

A similar situation, illustrated in Fig. 5.28(d), occurs if procedure D now calls B. Procedure B is allowed to refer only to variables declared by either B or A, which is reflected in the contents of the display. After procedure B returns to D, the contents of the stack and display will again appear as they were in Fig. 5.28(c).

It is important to be aware of the difference between the run-time allocation of variables, as represented by the stack of activation records, and the rules for referring to variables in the block-structured program, as represented by the display. You should carefully examine Figs. 5.27 and 5.28 to be sure you understand why the stack and display appear as they do in each situation.

The compiler for a block-structured language must include code at the beginning of a block to initialize the display for that block. At the end of the block, it must include code to restore the previous display contents. For the details of how this is accomplished, see Aho et al. (1988).

5.4 COMPILER DESIGN OPTIONS

In this section we consider some of the possible alternatives for the design and construction of a compiler. The discussions in this section are necessarily very brief. Our purpose is to introduce terms and concepts rather than to give a comprehensive discussion of any of these topics.

The compilation scheme presented in Section 5.1 was a simple one-pass design. Sections 5.2 and 5.3 described many features that usually require more than one pass to implement. In Section 5.4.1 we briefly discuss the general question of dividing a compiler into passes, and consider the advantages of one-pass and multi-pass designs.

Section 5.4.2 discusses interpreters, which execute an intermediate form of the program instead of translating it into machine code. Section 5.4.3 introduces the related topic of P-code systems, which compile high-level language programs into object code for a hypothetical machine.

Finally, Section 5.4.4 describes compiler-writing systems, which use software tools to automate much of the process of compiler construction.

5.4.1 Division into Passes

In Section 5.1 we presented a simple one-pass compilation scheme for a subset of the Pascal language. In this design, the compiler was driven by the parsing process. The lexical scanner was called when the parser needed another input token, and a code-generation routine was invoked as each language construct was recognized by the parser. The object code produced was not highly efficient. Most of the code-optimization techniques discussed in Sections 5.2 and 5.3 could not be applied in such a one-pass compiler. However, the compilation process itself, which required only one pass over the program and no intermediate code-generation step, was quite efficient.

Not all languages can be translated by such a one-pass compiler. In Pascal, declarations of variables must appear in the program before the statements that use these variables. In FORTRAN, declarations may appear at the beginning of the program; any variable that is not declared is assigned characteristics by default. However, in some languages the declaration of an identifier may appear after it has been used in the program. One-pass compilers must have the ability to fix up forward references in jump instructions, using techniques like those discussed for one-pass assemblers. Forward references to data items, however, present a much more serious problem.

Consider, for example, the assignment statement

```
X := Y * Z
```

If all of the variables X, Y, and Z are of type INTEGER, the object code for this statement might consist of a simple integer multiplication followed by storage of the result. If the variables are a mixture of REAL and INTEGER types, one or more conversion operations will need to be included in the object code, and floating-point arithmetic instructions may be used. Obviously the compiler cannot decide what machine instructions to generate for this statement unless information about the operands is available. The statement may even be illegal for certain combinations of operand types. Thus a language that allows forward references to data items cannot be compiled in one pass.

Some programming languages, because of other characteristics, require more than two passes to compile. For example Hunter (1981) showed that ALGOL 68 required at least three passes.

There are a number of factors that should be considered in deciding between one-pass and multi-pass compiler designs (assuming that the language in question can be compiled in one pass). If speed of compilation is important,

a one-pass design might be preferred. For example, computers running student jobs tend to spend a large amount of time performing compilations. The resulting object code is usually executed only once or twice for each compilation; these test runs are normally very short. In such an environment, improvements in the speed of compilation can lead to significant benefits in system performance and job turnaround time.

If programs are executed many times for each compilation, or if they process large amounts of data, then speed of execution becomes more important than speed of compilation. In such a case, we might prefer a multi-pass compiler design that could incorporate sophisticated code-optimization techniques. Multi-pass compilers are also used when the amount of memory, or other system resources, is severely limited. The requirements of each pass can be kept smaller if the work of compilation is divided into several passes.

Other factors may also influence the design of the compiler. If a compiler is divided into several passes, each pass becomes simpler and, therefore, easier to understand, write, and test. Different passes can be assigned to different programmers and can be written and tested in parallel, which shortens the overall time required for compiler construction.

For further discussion of the problem of dividing a compiler into passes, see Hunter (1981) and Aho et al. (1988).

5.4.2 Interpreters

An *interpreter* processes a source program written in a high-level language, just as a compiler does. The main difference is that interpreters execute a version of the source program directly, instead of translating it into machine code.

An interpreter usually performs lexical and syntactic analysis functions like those we have described for a compiler, and then translates the source program into an internal form. Many different internal forms can be used. One possibility is a sequence of quadruples like those discussed in Section 5.2. It is even possible to use the original source program itself as the internal form; however, it is generally much more efficient to perform some preprocessing of the program before execution.

After translating the source program into an internal form, the interpreter executes the operations specified by the program. During this phase, an interpreter can be viewed as a set of subroutines. The execution of these subroutines is driven by the internal form of the program.

The process of translating a source program into some internal form is simpler and faster than compiling it into machine code. However, execution of the translated program by an interpreter is much slower than execution of the machine code produced by a compiler. Thus an interpreter would not normally be used if speed of execution is important. If speed of translation is of primary concern, and execution of the translated program will be short, then an interpreter may be a good choice.

The real advantage of an interpreter over a compiler, however, is in the debugging facilities that can easily be provided. The symbol table, source line numbers, and other information from the source program are usually retained by the interpreter. During execution, these can be used to produce symbolic dumps of data values, traces of program execution related to the source statements, etc. Thus interpreters are especially attractive in an educational environment where the emphasis is on learning and program testing. Discussions of the implementation of debugging tools in an interpreter can be found in Watt (1993).

Most programming languages can be either compiled or interpreted successfully. However, some languages are particularly well suited to the use of an interpreter. As we have seen, compilers usually generate calls to library routines to perform functions such as I/O and complex conversion operations. For some languages, such as SNOBOL and APL, a large part of the compiled program would consist of calls to such routines. In such cases, an interpreter might be preferred because of its speed of translation. Most of the execution time for the translated program would be consumed by the standard library routines. These routines would be the same, regardless of whether a compiler or an interpreter were used.

Certain languages also have features that lend themselves naturally to interpretation. For example, in some languages the type of a variable can change during the execution of a program. Other languages use *dynamic scoping* instead of the more usual *static scoping* we discussed in Section 5.3.4. With dynamic scoping, the variables that can be referred to by a function or a subroutine are determined by the sequence of calls made during execution, not by the nesting of blocks in the source program. It would be very difficult to compile such languages efficiently and allow for dynamic changes in the types of variables and the scope of names. These features can be more easily handled by an interpreter, which provides delayed binding of symbolic variable names to data types and locations.

Further discussions of the construction and use of interpreters can be found in Watt (1993).

5.4.3 P-Code Compilers

P-code compilers (also called *bytecode* compilers) are very similar in concept to interpreters. In both cases, the source program is analyzed and converted into an intermediate form, which is then executed interpretively. With a P-code compiler, however, this intermediate form is the machine language for a hypothetical

such a P-code compiler is illustrated in Fig. 5.29. The source program is compiled, with the resulting object program being in P-code. This P-code program is then read and executed under the control of a P-code interpreter.

The main advantage of this approach is portability of software. It is not necessary for the compiler to generate different code for different computers, because the P-code object programs can be executed on any machine that has a P-code interpreter. Even the compiler itself can be transported if it is written in the language that it compiles. To accomplish this, the source version of the compiler is compiled into P-code; this P-code can then be interpreted on another computer. In this way, a P-code compiler can be used without modification on a wide variety of systems if a P-code interpreter is written for each different machine. Although writing such an interpreter is not a trivial task, it is certainly easier than writing a new compiler for each different machine. The same approach can also be used to transport other types of system software without rewriting.

The design of a P-machine and the associated P-code is often related to the requirements of the language being compiled. For example, the P-code for a Pascal compiler might include single P-instructions that perform array-subscript calculations, handle the details of procedure entry and exit, and perform elementary operations on sets. This simplifies the code-generation process, leading to a smaller and more efficient compiler. In addition, the P-code object program is often much smaller than a corresponding machine-code program would be. This is particularly useful on machines with severely limited memory size.

Figure 5.29 Translation and execution using a P-code compiler.

Obviously the interpretive execution of a P-code program may be much slower than the execution of the equivalent machine code. Depending upon the environment, however, this may not be a problem. Many P-code compilers are designed for a single user running on a dedicated microcomputer system. In that case, speed of execution may be relatively insignificant because the limiting factor in system performance may be the response time and "think time" of the user. If execution speed is important, some P-code compilers support the use of machine-language subroutines. By rewriting a small number of commonly used routines in machine language, rather than P-code, it is often possible to achieve substantial improvements in performance. Of course, this approach sacrifices some of the portability associated with the use of P-code compilers.

Section 5.5.2 of this text describes a recently developed P-code compiler for the Java language.

5.4.4 Compiler-Compilers

The process of writing a compiler usually involves a great deal of time and effort. In some areas, particularly the construction of scanners and parsers, it is possible to perform much of this work automatically. A *compiler-compiler* is a software tool that can be used to help in the task of compiler construction. Such tools are also often called *compiler generators* or *translator-writing systems*.

The process of using a typical compiler-compiler is illustrated in Fig. 5.30. The user (i.e., the compiler writer) provides a description of the language to be translated. This description may consist of a set of lexical rules for defining tokens and a grammar for the source language. Some compiler-compilers use this information to generate a scanner and a parser directly. Others create tables for use by standard table-driven scanning and parsing routines that are supplied by the compiler-compiler.

Figure 5.30 Automated compiler construction using a compiler-compiler.

In addition to the description of the source language, the user provides a set of semantic or code-generation routines. Often there is one such routine for each rule of the grammar, as we discussed in Section 5.1. This routine is called by the parser each time it recognizes the language construct described by the associated rule. However, some compiler-compilers can parse a larger section of the program before calling a semantic routine. In that case, an internal form of the statements that have been analyzed, such as a portion of the parse tree, may be passed to the semantic routine. This latter approach is often used when code optimization is to be performed. Compiler-compilers frequently provide special languages, notations, data structures, and other similar facilities that can be used in the writing of semantic routines.

The main advantage of using a compiler-compiler is, of course, ease of compiler construction and testing. The amount of work required from the user varies considerably from one compiler-compiler to another depending upon the degree of flexibility provided. Compilers that are generated in this way tend to require more memory and compile programs more slowly than hand-written compilers. However, the object code generated by the compiler may actually be better when a compiler-compiler is used. Because of the automatic construction of scanners and parsers, and the special tools provided for writing semantic routines, the compiler writer is freed from many of the mechanical details of compiler construction. The writer can therefore focus more attention on good code generation and optimization.

A brief description of one compiler-compiler (YACC) is given in Section 5.5.3. Further discussions and examples of compiler-writing tools can be found in Fischer and LeBlanc (1988).

5.5 IMPLEMENTATION EXAMPLES

In this section we briefly discuss the design of several real compilers. Section 5.5.1 describes the SunOS C compiler, which runs on a variety of hardware platforms.

Section 5.5.2 discusses the Java programming language and run-time environment recently developed by Sun Microsystems.

Section 5.5.3 presents a description of the YACC compiler-compiler, originally developed at Bell Laboratories for use with the UNIX operating system. We also briefly describe LEX, a scanner generator that is commonly used with YACC.

As in our previous discussions of real systems, we do not attempt to give a complete description of any of these compilers. References are provided for those readers who want more information.

5.5.1 SunOS C Compiler

The SunOS C compiler runs on a variety of hardware platforms, including SPARC, x86, and PowerPC. It conforms to the ANSI C standard, which is described in Schildt (1990). It also supports traditional (pre-ANSI) C features, as described in Kernighan and Ritchie (1978). The user of the compiler can specify which sets of language features are to be accepted by the compiler and which are to generate warning messages during compilation.

The translation process begins with the execution of the C preprocessor, which performs such functions as file inclusion and macro processing. (See Section 4.4.3 for a brief discussion of some of these features.) The output from the preprocessor goes to the C compiler itself. Several different levels of code optimization can be specified by the user. The compiler generates assembler language, which is then translated by an assembler. The preprocessor and compiler also accept source files that contain assembler language subprograms, and pass these on to the assembly phase.

The preprocessing phase consists of the following conceptual steps. The logical ordering of these steps is specified by the ANSI standard to eliminate possible ambiguities. The implementation of a particular preprocessor may in fact combine several of these steps. However, the effect must be the same as if they were executed separately in the sequence given.

1. Trigraph sequences are replaced by their single-character equivalents. The trigraph sequences are provided as a way to specify C language characters that may not available on some terminals. For example, the sequence ??< is replaced by {.
2. Any source line that ends with a backslash (\) and a newline is spliced together with the following line by deleting the backslash and newline.
3. The source file is partitioned into preprocessing tokens and sequences of white-space characters. Each comment is, in effect, replaced by one space. Preprocessing tokens include the regular tokens of the C language, plus tokens such as header file names and special numeric forms that are used only in the preprocessing stage.
4. Preprocessing directives are executed, and macros are expanded. Any source files that are included in response to an #include directive are processed from step 1 through 4.
5. Escape sequences in character constants and string literals are converted to their character equivalents. For example, \n is converted to a newline, and \0 is converted to the character with ASCII code 0.

6. Adjacent string literals are concatenated. For example, "hello," "world" is converted to "hello, world".

After preprocessing is complete, the actual process of program translation begins. The lexical analysis of the program is performed during preprocessing (step 3 above). Thus the compiler itself begins with syntactic analysis, followed by semantic analysis and code generation.

Like most implementations of UNIX, the SunOS operating system itself is largely written in C. Many compilers, editors, and other pieces of UNIX system software are also written in C. Because of this, it is important that a C compiler for such a system be able to generate efficient object code. It is also desirable that the compiler include tools to assist programmers in analyzing the performance of their programs. We will focus on these aspects of the SunOS C compiler.

Four different levels of code optimization can be specified by the user when a program is compiled. These levels are designated by O1 through O4. The O1 level does only a minimal amount of local (peephole) optimization. This type of optimization is performed at the assembler-language level, after the compilation itself is complete.

The O2 level provides basic local and global optimization. This includes register allocation and merging of basic blocks (see Section 5.2.2) as well as elimination of common subexpressions and removal of loop invariants (see Section 5.3.2). It also includes a number of other optimizations such as algebraic simplification and tail recursion elimination. In general, the O2 level of optimization results in the minimum object code size. This is the default that is provided unless otherwise requested by the user.

The O3 and O4 levels include optimizations that can improve execution speed, but usually produce a larger object program. For example, O3 optimization performs loop unrolling, which partially converts loops into straight-line code. The O4 level automatically converts calls to user-written functions into in-line code. This eliminates the overhead of calling and returning from the functions. Optimizations such as these can be guided by the user via compile-time options. For example, the user can specify the names of functions that should (or should not) be converted to in-line code.

When requested, the SunOS C compiler can insert special code into the object program to gather information about its execution. For example, one option accumulates a count of how many times each basic block is executed. Another option invokes a run-time recording mechanism. The resulting data can be analyzed by other SunOS software tools to produce profiles of program execution. For example, one such profile shows the percentage of execution time spent in different parts of the program.

Other program analysis tools provide support for reordering object code at the function level. To use these tools, the user instructs the C compiler to place each function in a separate section. After the execution profile is analyzed, the object program is relinked to rearrange the functions. This can produce an executable program with improved locality of reference that runs more efficiently under a virtual memory management system. (Issues of virtual memory and locality of reference are discussed in Section 6.2.5.)

Like many other compilers, SunOS C can also generate information that supports the operation of debugging tools. For example, one option requests the compiler to include source code information in the object program. Using this information, the symbolic debugger can allow the user to examine variable values, control execution, browse the source file, and so on.

Further information about the SunOS C compiler can be found in Sun Microsystems (1994c).

5.5.2 Java Compiler and Environment

Java is a new programming language and operating environment developed by Sun Microsystems. It was designed to support applications in a diverse environment such as the Internet. Using Java, programmers can create high-performance applications that are portable without modification to multiple operating systems and hardware platforms. Java also provides features that are intended to make distributed applications more reliable and more secure.

The Java language itself is derived from C and C++. However, many of the features found in these languages have been removed to make Java programming as simple and error-free as possible. For example, Java has no "go to" statement and no pointers. Memory management is handled automatically, thus freeing the programmer from a complex and error-prone task. Data conversions that might cause loss of precision must be done explicitly by using built-in language features. This makes it possible for the compiler to perform strong type-checking, which leads to early detection of many programming errors.

Java is an object-oriented language. (If you are unfamiliar with the principles of object-oriented programming, you may want to review the discussion in Section 8.4.1.) The object-orientation in Java is stronger than in many other languages, such as C++. Except for a few primitive data types, everything in Java is an object. Arrays and strings are treated as objects. Even the primitive data types can be encapsulated inside objects if necessary. There are no procedures or functions in Java; classes and methods are used instead. Thus programmers are constrained to use a "pure" object-oriented style, rather than mixing the procedural and object-oriented approaches.

Java provides built-in support for multiple *threads* of execution. This feature allows different parts of an application's code to be executed concurrently.

For example, an interactive application might use one thread to run an animation, another to control sound effects, and a third to scroll a window. In Java, threads are implemented as objects. The Java library provides methods that can be invoked to start or stop a thread, check on the status of a thread, and synchronize the operation of multiple threads.

The Java compiler follows the P-code approach we discussed in Section 5.4.3. It does not generate machine code or assembly language for a particular target machine. Instead, the compiler generates *bytecodes*—a high-level, machine-independent code for a hypothetical machine (the Java Virtual Machine). This hypothetical machine is implemented on each target computer by an interpreter and run-time system. Thus a Java application can be run, without modification and without recompiling, on any computer for which a Java interpreter exists. The Java compiler itself is written in Java. Therefore the compiler can also run on any machine with a Java interpreter.

The bytecode approach also allows easy integration of Java applications into the World Wide Web. A segment of Java bytecode (often called an *applet*) can be included in an HTML page, in much the same way an image can be included. When a Java-compatible Web browser is used to view the page, the code for the applet is downloaded and executed by the browser. The applet can then perform animation, play sound, and generally interact with the user in real time.

The Java Virtual Machine supports a standard set of primitive data types: 1-, 2-, 4-, and 8-byte integers, single- and double-precision floating-point numbers, and 16-bit character codes. These data representations are independent of the architecture of the target machine. The interpreter is responsible for emulating these data types using the underlying hardware. Thus, for example, the floating-point formats and the big- or little-endian storage of integers on the target machine have no effect on an application program.

A bytecode instruction on the Java Virtual Machine consists of a 1-byte opcode followed by zero or more operands. Many opcodes require no explicit operands in the instruction; instead, they take their operand values from a stack. A stack organization was chosen so that it would be easy to emulate the machine on a computer with few general-purpose registers (such as the x86 architecture).

For example, the "iadd" instruction adds two integers together. It expects that the integers to be added are the top two words on the operand stack, pushed there by previous instructions. Both integers are popped from the stack, and their sum is pushed back onto the stack. Each primitive data type has specialized instructions that must be used to operate on items of that type.

There are also single bytecode instructions that perform higher-level operations. For example, one instruction allocates a new array of a particular type. Other instructions can be used to transfer elements of an array to or from the operand stack.

Similarly, the bytecode instructions provide direct support for the object-oriented nature of Java. One instruction creates a new object of a specified type. Another instruction tests whether an object is an instance of a particular class. There are four instructions that can be used (depending upon the situation) to invoke a method on an object. Another group of instructions is used to manipulate fields within an object.

Performance is always a consideration, especially with interpreted execution. The Java interpreter is designed to run as fast as possible, without needing to check the run-time environment. The automatic *garbage collection* system used to manage memory runs as a low-priority background thread. Experiments conducted on modern (1995) systems such as workstations and high-end personal computers show generally good response running interactive graphical applications. For example, creating a new object typically requires about 8 msec and invoking a method on an object requires about 2 msec.

However, there are times when higher performance may be needed. In such cases, the Java bytecodes can be translated at execution time into machine code for the computer on which the application is running. Measurements on execution of bytecodes converted to machine code show performance roughly the same as the equivalent application coded directly in C or C++.

A description of the Java language can be found in Arnold and Gosling (1996). Further information about the implementation of Java can be found in Lindholm and Yellin (1996) and Sun Microsystems (1995b).

5.5.3 The YACC Compiler-Compiler*

YACC (Yet Another Compiler-Compiler) is a parser generator that is available on UNIX systems. YACC has been used in the production of compilers for Pascal, RATFOR, APL, C, and many other programming languages. It has also been used for several less conventional applications, including a typesetting language and a document retrieval system. In this section we give brief descriptions of YACC and LEX, the scanner generator that is related to YACC. Further information about these software tools can be found in Levine et al. (1992).

A lexical scanner must be supplied for use with YACC. This scanner is called by the parser whenever a new input token is needed. It returns an integer that identifies the type of token found, as described in Section 5.1. The scanner may also make entries in a symbol table for the identifiers that are processed.

LEX is a scanner generator that can be used to create scanners of the type required by YACC. A portion of an input specification for LEX is shown in

*Adapted from "Language Development Tools on the Unix System" by S.C. Johnson, from the IEEE publication *Computer*, Vol. 13, No. 8, pp. 16–21, August 1980. © 1980 IEEE.

Fig. 5.31(a). Each entry in the left-hand column is a pattern to be matched against the input stream. When a pattern is matched, the corresponding action routine in the right-hand column is invoked. These action routines are written in the programming language C. The routines usually return an indication of the token that was recognized. They may also make entries in tables and perform other similar tasks.

In the example shown in Fig. 5.31(a), the first pattern has no associated action; the effect of this is to delete blanks as the input is scanned. The actions for the next three patterns simply return a token type: the token LET for the keyword *let*, MUL for the operator *, and ASSIGN for the operator =. As discussed above, the internal representations of LET, MUL, and the other tokens are integers. The fifth pattern specifies the form of identifiers to be recognized. The first character must be in the range a–z or A–Z. This may be followed by any number of characters in the ranges a–z, A–Z, or 0–9. The * in this pattern indicates that an arbitrary number of repetitions of the preceding

```
.
.
.
" "                          ; /* ignore blanks */
let                          return(LET);
"*"                          return(MUL);
"="                          return(ASSIGN);
[a-zA-Z] [a-zA-Z0-9]*        {make entries in tables; return(ID)};
.
.
.
```

(a)

```
%token ASSIGN ID LET MUL ...
.
.
.
statement   :   LET  ID  ASSIGN expr
                     { ... }
expr        :   expr  MUL  expr
                     { $$ = build(MUL,$1,$3);}
expr        :   ID
                     { ... }
.
.
.
```

(b)

Figure 5.31 Example of input specifications for LEX and YACC.

item are allowed. The action routine for this pattern makes entries in the appropriate tables to describe the identifier found and then returns the token type ID.

According to the specifications given in Fig. 5.31(a), the input

```
let x = y * z
```

would be scanned as the sequence of tokens

```
LET ID ASSIGN ID MUL ID
```

Note that the first pattern that matches the input stream is selected, so the keyword *let* is recognized as the token LET, not as ID.

LEX can be used to produce quite complicated scanners. Some languages, such as FORTRAN, however, have lexical analyzers that must still be generated or modified by hand.

The YACC parser generator accepts as input a grammar for the language being compiled and a set of actions corresponding to rules of the grammar. A portion of such an input specification appears in Fig. 5.31(b). The first line shown is a declaration of the token types used. The other entries are rules of the grammar. The YACC parser calls the semantic routine associated with each rule as the corresponding language construct is recognized. Each routine may return a value by assigning it to the variable $$. Values returned by previous routines, or by the scanner, may be referred to as $1, $2, etc. These variables designate the values returned for the components on the right-hand side of the corresponding rule, reading from left to right.

An example of the use of such values is shown in Fig. 5.31(b). The semantic routine associated with the rule

```
expr : expr MUL expr
```

constructs a portion of the parse tree for the statement, using a tree-building function *build*. This subtree is returned from the semantic routine by assigning the subtree to $$. The arguments passed to the *build* function are the operator MUL and the values (i.e., the subtrees) returned when the operands were recognized. These values are denoted by $1 and $3.

It is sometimes useful to perform semantic processing as each part of a rule is recognized. YACC permits this by allowing semantic routines to be written in the middle of a rule as well as at the end. The value returned by such a routine is available to any of the routines that appear later in the rule. It is also possible for the user to define global variables that can be used by all of the semantic routines and by the lexical scanner.