2. How could a *nonrecursive* macro processor allow for the invocation of macros within macros? What would be the advantages and disadvantages of such an approach?
3. Select two different high-level programming languages with which you are familiar. What differences between these languages might be significant to a macro processor that is intended for use with the language?
4. Select one high-level language and one (real) assembler language with which you are familiar. What differences between these languages might be significant to a macro processor that is intended for use with the language?
5. Outline an algorithm for combining a line-by-line macro processor with an assembler.
6. List utility functions and routines that might be shared by an assembler and an integrated macro processor.
7. Using the methods outlined in Chapter 8, develop a modular design for a two-pass assembler with an integrated macro processor.

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>
```

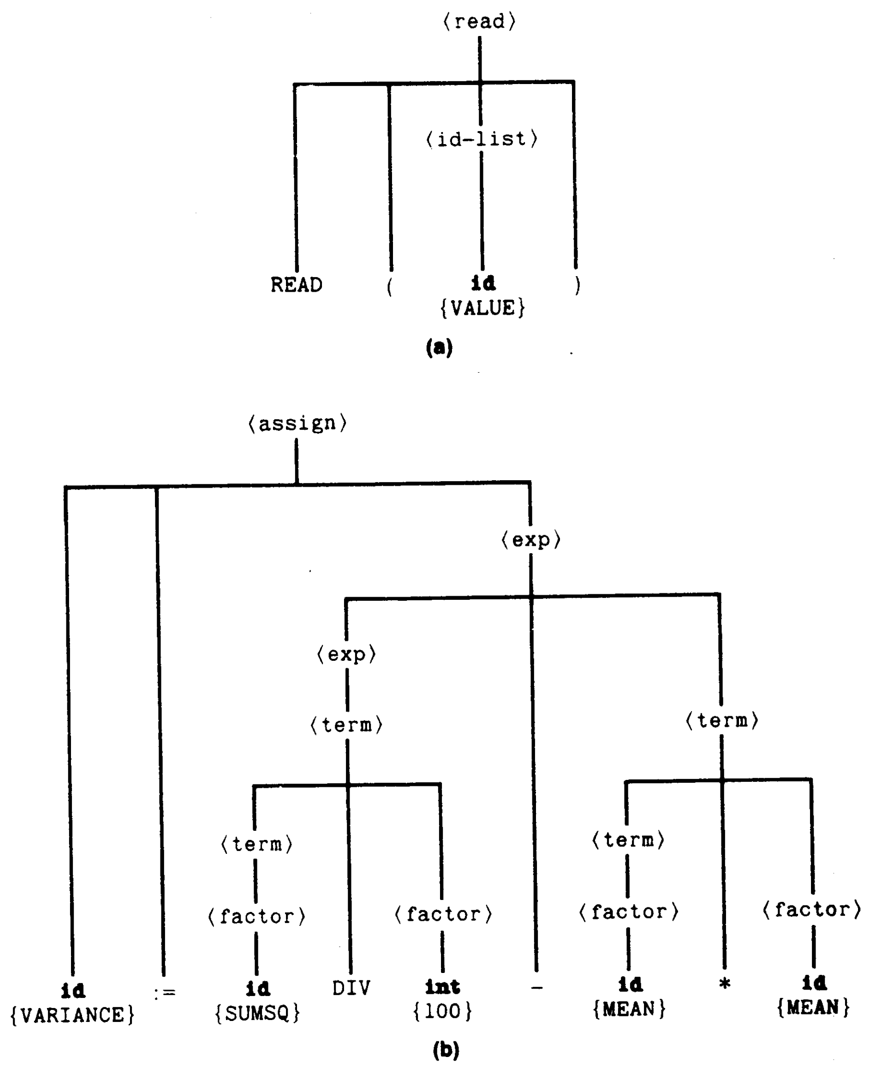


**Figure 5.3** Parse trees for two statements from Fig. 5.1.

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.

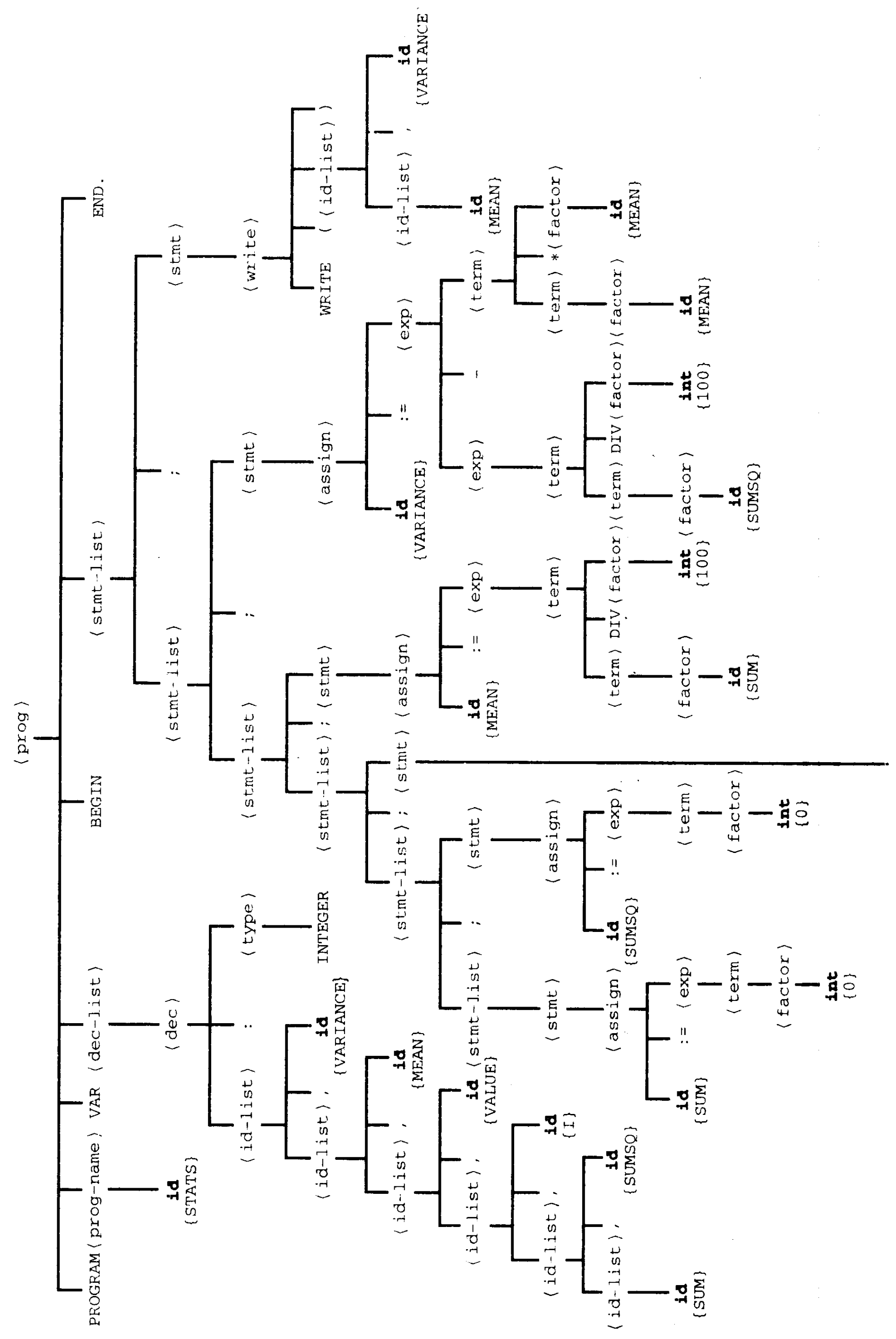
## 5.1.2 Lexical Analysis

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


⟨prog⟩
PROGRAM ⟨prog-name⟩ VAR ⟨dec-list⟩
BEGIN
⟨stmt-list⟩
END.
id {STATS}
⟨dec⟩
⟨id-list⟩ : ⟨type⟩
INTEGER
⟨id-list⟩ , id {VARIANCE}
⟨id-list⟩ , id {MEAN}
⟨id-list⟩ , id {VALUE}
⟨id-list⟩ , id {I}
⟨id-list⟩ , id {SUMSQ}
id {SUM}
⟨stmt-list⟩ ; ⟨stmt⟩
⟨write⟩
WRITE ( ⟨id-list⟩ )
⟨id-list⟩ , id {VARIANCE}
id {MEAN}
⟨assign⟩
id {VARIANCE} := ⟨exp⟩
⟨exp⟩ - ⟨term⟩
⟨term⟩ * ⟨factor⟩
⟨factor⟩
id {MEAN}
⟨term⟩ DIV ⟨factor⟩
⟨factor⟩
int {100}
id {SUMSQ}
id {MEAN} := ⟨exp⟩
⟨term⟩
⟨term⟩ DIV ⟨factor⟩
⟨factor⟩
int {100}
id {SUM}
id {SUMSQ} := ⟨exp⟩
⟨term⟩
⟨factor⟩
int {0}
id {SUM} := ⟨exp⟩
⟨term⟩
⟨factor⟩
int {0}

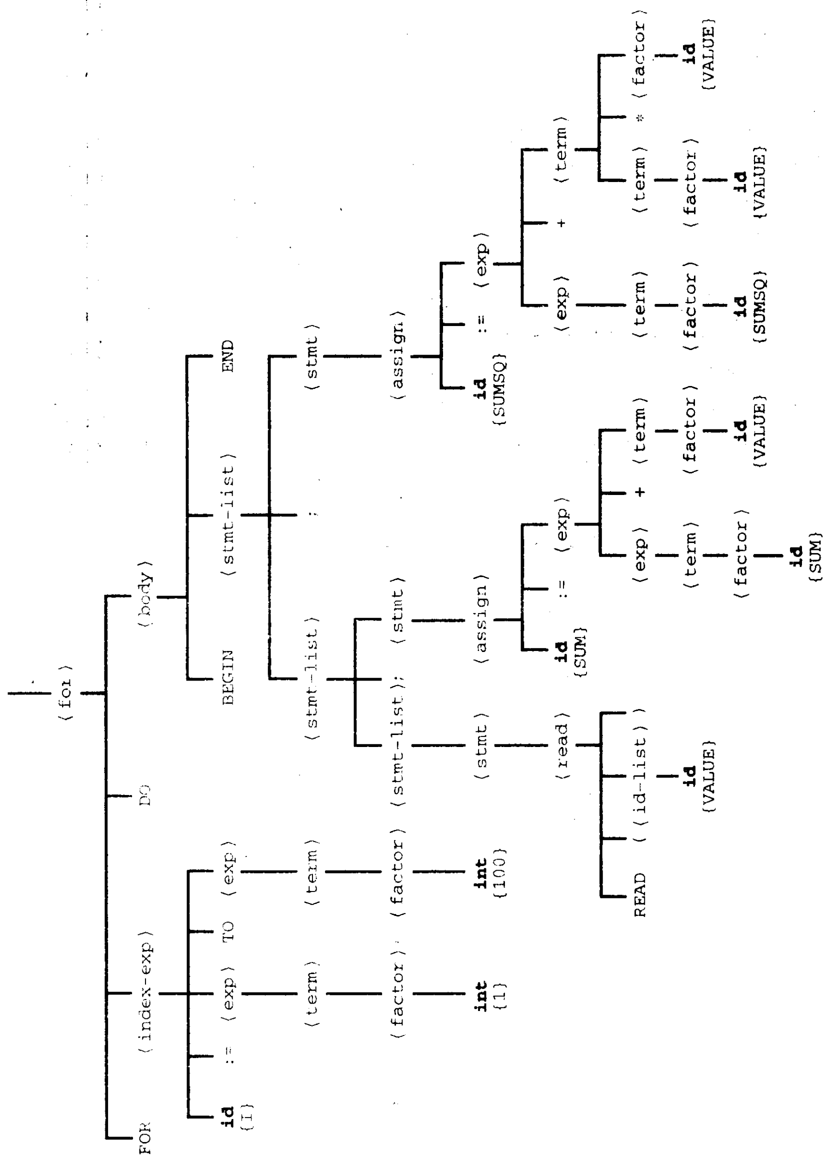


**Figure 5.4** Parse tree for the program from Fig. 5.1.

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.

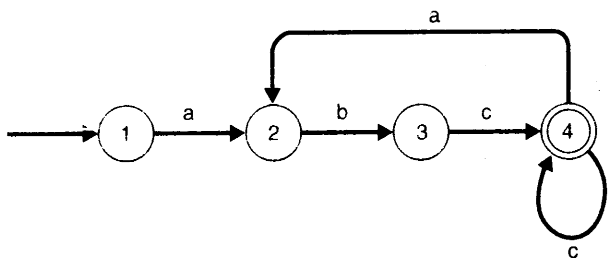
### *Modeling Scanners as Finite Automata*

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 | |
| 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 =.

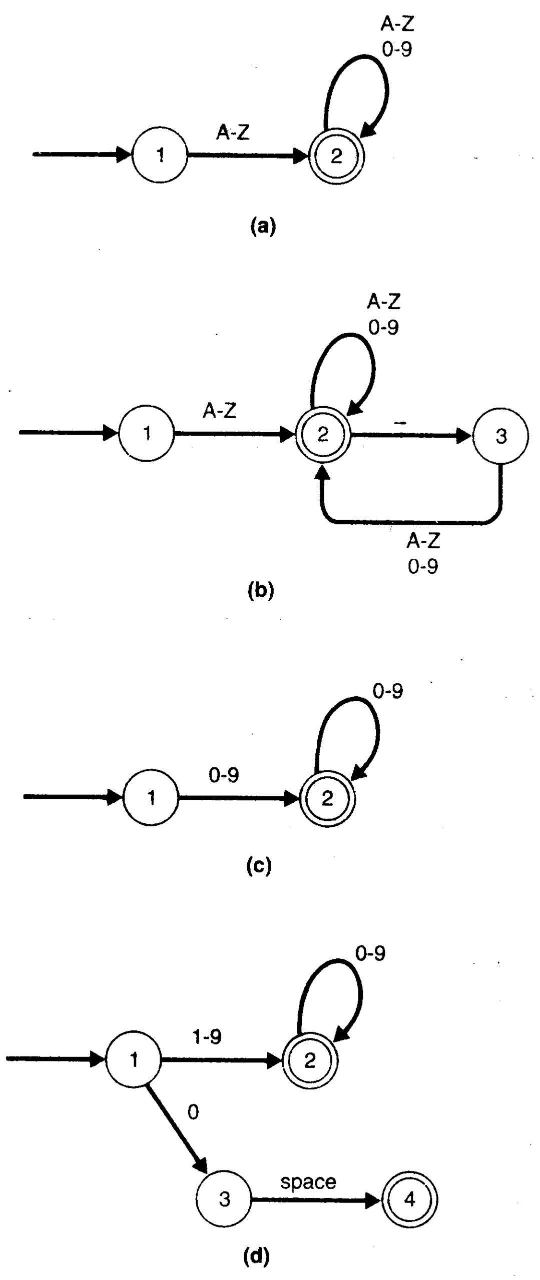


**Figure 5.8** Finite automata for typical programming language tokens.

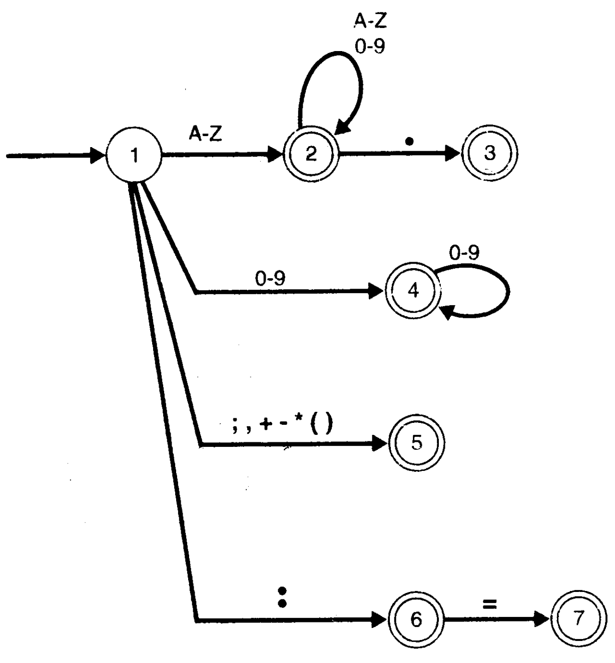


**Figure 5.9** Finite automaton to recognize tokens from Fig. 5.5.

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 the tree so that the terminal nodes match the statements being analyzed. Bottom-up methods begin with the terminal nodes of the tree (the statements being analyzed), and attempt to combine these into successively higher-level nodes until the root is reached.

A large number of different parsing techniques have been devised, most of which are applicable only to grammars that satisfy certain conditions.

## Recursive-Descent Parsing

A top-down method which is known as *recursive descent,* is made up of a procedure for each nonterminal symbol in the grammar. When a procedure is called, it attempts to find a substring of the input, beginning with the current token, that can be interpreted as the nonterminal with which the procedure is associated. In the process of doing this, it may call other procedures, or even call itself recursively, to search for other nonterminals. If a procedure finds the nonterminal that is its goal, it returns an indication of success to its caller. It also advances the current-token pointer past the substring it has just recognized. If the procedure is unable to find a substring that can be interpreted as the desired nonterminal, it returns an indication of failure, or invokes an error diagnosis and recovery routine.

As an example of this, consider Rule 13 of the grammar in Fig. 5.2. The procedure for <read> in a recursive-descent parser first examines the next two input tokens, looking for READ and (. If these are found, the procedure for <read> then calls the procedure for <id-list>. If that procedure succeeds, the <read> procedure examines the next input token, looking for ). If all these tests are successful, the <read> procedure returns an indication of success to its caller and advances to the next token following ). Otherwise, the procedure returns an indication of failure.

The procedure is only slightly more complicated when there are several alternatives defined by the grammar for a nonterminal. In that case, the procedure must decide which of the alternatives to try. For the recursive-descent technique, it must be possible to decide which alternative to use by examining the next input token. There are other top-down methods that remove this requirement; however, they are not as efficient as recursive descent. Thus the procedure for <stmt> looks at the next token to decide which of its four alternatives to try. If the token is READ, it calls the procedure for <read>; if the token is **id**, it calls the procedure for <assign> because this is the only alternative that can begin with the token **id**, and so on.

If we attempted to write a complete set of procedures for the grammar of Fig. 5.2, we would discover a problem. The procedure for <id-list>, corresponding to Rule 6, would be unable to decide between its two alternatives since both **id** and <id-list> can begin with **id**. There is, however, a more fundamental difficulty. If the procedure somehow decided to try the second alternative (<id-list>, **id**), it would immediately call itself recursively to find an <id-list>. This could result in another immediate recursive call, which leads to an unending chain. The reason for this is that one of the alternatives for <id-list> begins with <id-list>. Top-down parsers cannot be directly used with a grammar that contains this kind of immediate *left recursion*. The same problems also occur with respect to Rules 3, 7, 10, and 11. Methods for eliminating left recursion from a grammar are described in Aho et al. (1988).

Figure 5.11 shows the grammar from Fig. 5.2 with left recursion eliminated. Consider, for example, Rule 6a in Fig. 5.11:

```
<id-list> ::= id { , id }
```

This notation, which is a common extension to BNF, specifies that the terms between { and } may be omitted, or repeated one or more times. Thus Rule 6a defines <id-list> as being composed of an **id** followed by zero or more occurrences of ", **id**". This is clearly equivalent to Rule 6 of Fig. 5.2. With the revised definition, the procedure for <id-list> simply looks first for an **id**, and then keeps scanning the input as long as the next two tokens are a comma (,) and **id**. This eliminates the problem of left recursion and also the difficulty of deciding which alternative for <id-list> to try.

```
1   <prog>        ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2   <prog-name>   ::= id
3a  <dec-list>    ::= <dec> { ; <dec> }
4   <dec>         ::= <id-list> : <type>
5   <type>        ::= INTEGER
6a  <id-list>     ::= id { , id }
7a  <stmt-list>   ::= <stmt> { ; <stmt> }
8   <stmt>        ::= <assign> | <read> | <write> | <for>
9   <assign>      ::= id := <exp>
10a <exp>         ::= <term> { + <term> | - <term> }
11a <term>        ::= <factor> { * <factor> | 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 5.11** Simplified Pascal grammar modified for recursive-descent parse.

Similar changes have been made in Rules 3a, 7a, 10a, and 11a in Fig. 5.11. You should compare these rules to the corresponding definitions in Fig. 5.2 to be sure you understand the changes made. Note that the grammar itself is still recursive: <exp> is defined in terms of <term>, which is defined in terms of <factor>, and one of the alternatives for <factor> involves <exp>. This means that recursive calls among the procedures of the parser are still possible. However, direct left recursion has been eliminated. A chain of calls from <exp> to <term> to <factor> and back to <exp> must always consume at least one token from the input statement.

Figure 5.12 illustrates a recursive-descent parse of the READ statement on line 9 of Fig. 5.1, using the grammar in Fig. 5.11. Figure 5.12(a) shows the procedures for the nonterminals <read> and <id-list>, which follow the verbal descriptions just given. It is assumed that TOKEN contains the type of the next input token, using the coding scheme shown in Fig. 5.5. You should examine these procedures carefully to be sure you understand how they were derived from the grammar.

In the procedure IDLIST, note that a comma (,) that is not followed by an **id** is considered to be an error, and the procedure returns an indication of failure to its caller. If a sequence of tokens such as "**id,id,**" could be a legal construct according to the grammar, this recursive-descent technique would not work properly. For such a grammar, it would be necessary to use a more complex parsing method that would allow the top-down parser to backtrack after recognizing that the last comma was not followed by an **id**.

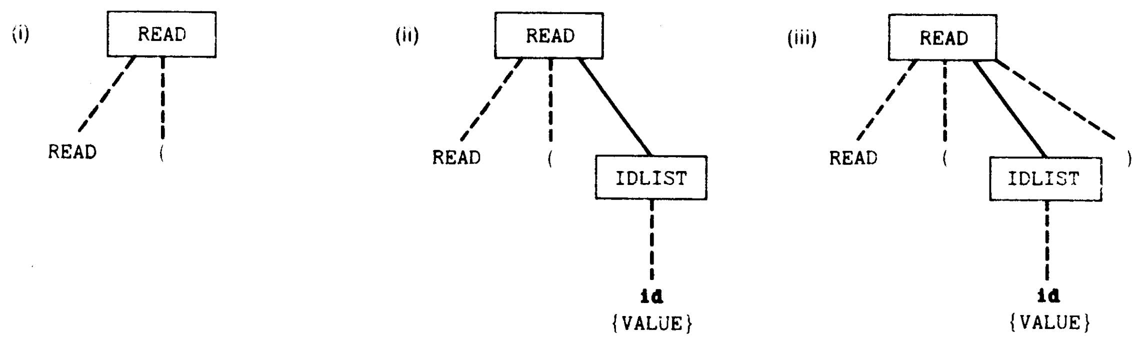
Figure 5.12(b) gives a graphic representation of the recursive-descent parsing process for the statement being analyzed. In part (i), the READ procedure has been invoked and has examined the tokens READ and ( from the input stream (indicated by the dashed lines). In part (ii), READ has called IDLIST (indicated by the solid line), which has examined the token **id**. In part (iii), IDLIST has returned to READ, indicating success; READ has then examined the input token ). This completes the analysis of the source statement. The procedure READ will now return to its caller, indicating that a <read> was successfully found. Note that the sequence of procedure calls and token examinations has completely defined the structure of the READ statement. The representation in part (iii) is the same as the parse tree in Fig. 5.3(a). Note also that the parse tree was constructed beginning at the root, hence the term *top-down parsing*.

Figure 5.13 illustrates a recursive-descent parse of the assignment statement on line 14 of Fig. 5.1. Figure 5.13(a) shows the procedures for the nonterminal symbols that are involved in parsing this statement. You should carefully compare these procedures to the corresponding rules of the grammar. Figure 5.13(b) is a step-by-step representation of the procedure calls and token examinations similar to that shown in Fig. 5.12(b). You are urged to follow through each step of the analysis of this statement, using the procedures in Fig. 5.13(a). Compare

```
procedure READ
      begin
         FOUND := FALSE
         if TOKEN = 8 {READ} then
            begin
               advance to next token
               if TOKEN = 20 { ( } then
                  begin
                     advance to next token
                     if IDLIST returns success then
                        if TOKEN = 21 { ) } then
                           begin
                              FOUND := TRUE
                              advance to next token
                           end {if ) }
                  end {if ( }
            end {if READ}
         if FOUND = TRUE then
            return success
         else
            return failure
      end {READ}

procedure IDLIST
      begin
         FOUND := FALSE
         if TOKEN = 22 {id} then
            begin
               FOUND := TRUE
               advance to next token
               while (TOKEN = 14 {,}) and (FOUND = TRUE) do
                  begin
                     advance to next token
                     if TOKEN = 22 {id} then
                        advance to next token
                     else
                        FOUND := FALSE
                  end {while}
            end {if id}
         if FOUND = TRUE then
            return success
         else
            return failure
      end {IDLIST}
```

**(a)**



**(b)**

**Figure 5.12** Recursive-descent parse of a READ statement.

the parse tree built in Fig. 5.13(b) to the one in Fig. 5.3(b). Note that the differences between these two trees correspond exactly to the differences between the grammars of Figs. 5.11 and 5.2.

Our examples of recursive-descent parsing have involved only single statements; however, the same technique can be applied to an entire program. In that case, the syntactic analysis would consist simply of a call to the procedure for <prog>. The calls from this procedure would create the parse tree for the program. You may want to write the procedures for the other nonterminals, following the grammar of Fig. 5.11, and apply this method to the program in

```
procedure ASSIGN
      begin
         FOUND := FALSE
         if TOKEN = 22 {id} then
            begin
               advance to next token
               if TOKEN = 15 { := } then
                  begin
                     advance to next token
                     if EXP returns success then
                        FOUND := TRUE
                  end {if := }
            end {if id}
         if FOUND = TRUE then
            return success
         else
            return failure
      end {ASSIGN}


procedure EXP
      begin
         FOUND := FALSE
         if TERM returns success then
            begin
               FOUND := TRUE
               while ((TOKEN = 16 {+}) or (TOKEN = 17 {-}))
                     and (FOUND = TRUE) do
                  begin
                     advance to next token
                     if TERM returns failure then
                        FOUND := FALSE
                  end {while}
            end {if TERM}
         if FOUND = TRUE then
            return success
         else
            return failure
      end {EXP}
```

**Figure 5.13** Recursive-descent parse of an assignment statement.

```
procedure TERM
    begin
        FOUND := FALSE
        if FACTOR returns success then
            begin
                FOUND := TRUE
                while ({TOKEN = 18 {*}) or (TOKEN = 19 {DIV})
                    and (FOUND = TRUE) do
                        begin
                            advance to next token
                            if FACTOR returns failure then
                                FOUND := FALSE
                        end {while}
            end {if FACTOR}
        if FOUND = TRUE then
            return success
        else
            return failure
    end {TERM}


procedure FACTOR
    begin
        FOUND := FALSE
        if (TOKEN = 22 {id}) or (TOKEN = 23 {int}) then
            begin
                FOUND := TRUE
                advance to next token
            end {if id or int}
        else
            if TOKEN = 20 { ( } then
                begin
                    advance to next token
                    if EXP returns success then
                        if TOKEN = 21 { ) } then
                            begin
                                FOUND := TRUE
                                advance to next token
                            end {if )}
                end {if ( }
        if FOUND = TRUE then
            return success
        else
            return failure
    end {FACTOR}
```

**(a)**

**Figure 5.13** (*cont'd*)

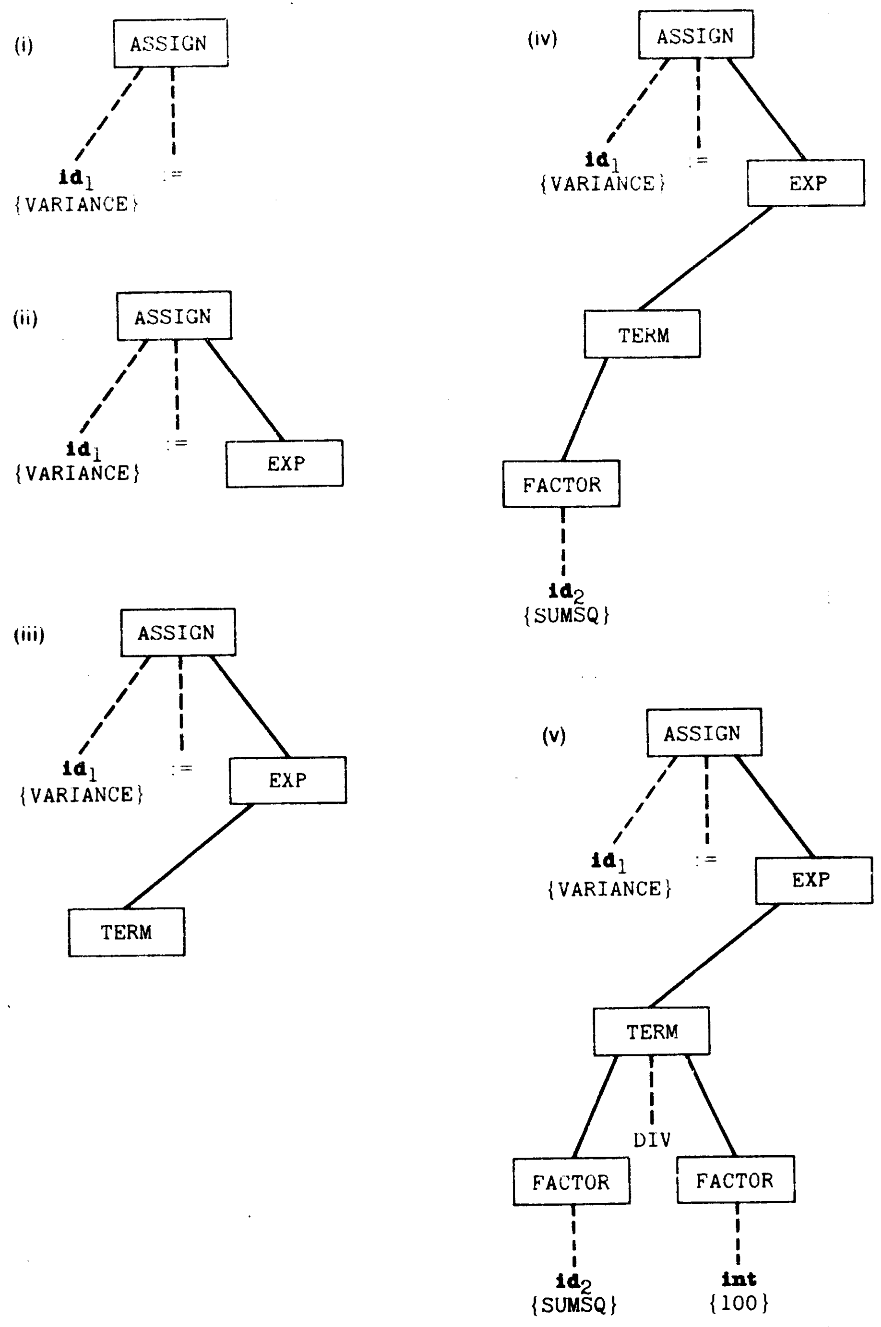
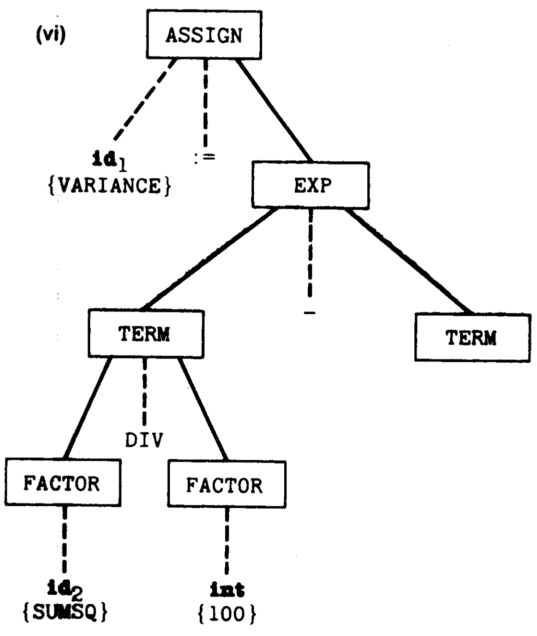
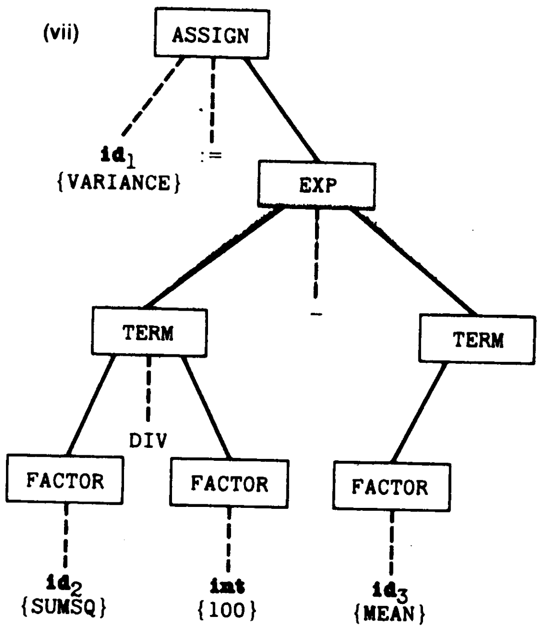
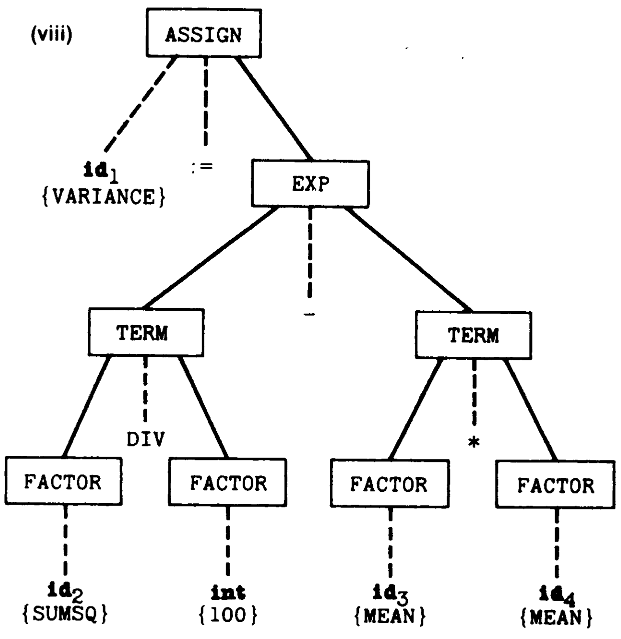


**Figure 5.13** *(cont'd)*







(b)

**Figure 5.13** (*cont'd*)

Fig. 5.1. The result should be similar to the parse tree in Fig. 5.4. The only differences should be ones created by the modifications made to the grammar in Fig. 5.11.

Note that there is nothing inherent in a programming language that requires the use of any particular parsing technique. We have used one bottom-up parsing method and one top-down method to parse the same program, using essentially the same grammar. It is even possible to use a combination of techniques. Some compilers use recursive descent for high-level constructs (for example, down to the statement level), and then switch to a technique such as operator precedence to analyze constructs such as arithmetic expressions. Further discussions of parsing methods can be found in Aho et al. (1988).

### 5.1.4 Code Generation

After the syntax of a program has been analyzed, the last task of compilation is the generation of object code. In this section we discuss a simple code-generation technique that creates the object code for each part of the program as soon as its syntax has been recognized.

The code-generation technique we describe involves a set of routines, one for each rule or alternative rule in the grammar. When the parser recognizes a portion of the source program according to some rule of the grammar, the corresponding routine is executed. Such routines are often called *semantic routines* because the processing performed is related to the meaning we associate with the corresponding construct in the language. In our simple scheme, these semantic routines generate object code directly, so we refer to them as *code-generation routines*. In more complex compilers, the semantic routines might generate an intermediate form of the program that would be analyzed further in an attempt to generate more efficient object code. We discuss this possibility in more detail in Sections 5.2 and 5.3.

The code-generation routines we discuss in this section are designed for use with the grammar in Fig. 5.2. As we have seen, neither of the parsing techniques discussed in Section 5.1.3 recognizes exactly the constructs specified by this grammar. The operator-precedence method ignores certain nonterminals, and the recursive-descent method must use a slightly modified grammar. However, there are parsing techniques not much more complicated than those we have discussed that can parse according to the grammar in Fig. 5.2. We choose to use this grammar in our discussion of code generation to emphasize the point that code-generation techniques need not be associated with any particular parsing method.

The specific code to be generated clearly depends upon the computer for which the program is being compiled. In this section we use as an example the generation of object code for a SIC/XE machine.

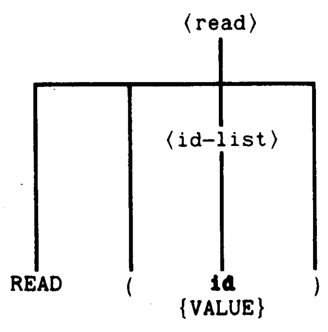
Our code-generation routines make use of two data structures for working storage: a list and a stack. Items inserted into the list are removed in the order of their insertion, first in–first out. Items pushed onto the stack are removed (popped from the stack) in the opposite order, last in–first out. The variable LISTCOUNT is used to keep a count of the number of items currently in the list. The code-generation routines also make use of the token specifiers described in Section 5.1.2; these specifiers are denoted by S(token). For a token **id**, S(**id**) is the name of the identifier, or a pointer to the symbol-table entry for it. For a token **int**, S(**int**) is the value of the integer, such as #100.

Many of our code-generation routines, of course, create segments of object code for the compiled program. We give a symbolic representation of this code, using SIC assembler language. You should remember, however, that the actual code generated is usually machine language, not assembler language. As each piece of object code is generated, we assume that a location counter LOCCTR is updated to reflect the next available address in the compiled program (exactly as it is in an assembler).

Figure 5.14 illustrates the application of this process to the READ statement on line 9 of the program in Fig. 5.1. The parse tree for this statement is repeated for convenience in Fig. 5.14(a). This tree can be generated with many different parsing methods. Regardless of the technique used, however, the parser always recognizes at each step the leftmost substring of the input that can be interpreted according to a rule of the grammar. In an operator-precedence parse, this recognition occurs when a substring of the input is reduced to some nonterminal <$N_i$>. In a recursive-descent parse, the recognition occurs when a procedure returns to its caller, indicating success. Thus the parser first recognizes the **id** VALUE as an <id-list>, and then recognizes the complete statement as a <read>.

Figure 5.14(c) shows a symbolic representation of the object code to be generated for the READ statement. This code consists of a call to a subroutine XREAD, which would be part of a standard library associated with the compiler. The subroutine XREAD can be called by any program that wants to perform a READ operation. XREAD is linked together with the generated object program by a linking loader or a linkage editor. (The compiler includes enough information in the object program to specify this linking operation, perhaps using Modification records such as those discussed in Chapter 2.) This technique is commonly used for the compilation of statements that perform relatively complex functions. The use of a subroutine avoids the repetitive generation of large amounts of in-line code, which makes the object program smaller.

Since XREAD may be used to perform any READ operation, it must be passed parameters that specify the details of the READ. In this case, the parameter list for XREAD is defined immediately after the JSUB that calls it. The first word in this parameter list contains a value that specifies the number of variables that will be assigned values by the READ. The following words give



(a)

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

              add S(id) to list
              add 1 to LISTCOUNT


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

              add S(id) to list
              add 1 to LISTCOUNT


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

              generate [+JSUB  XREAD]
              record external reference to XREAD
              generate [WORD   LISTCOUNT]
              for each item on list do
                  begin
                      remove S(ITEM) from list
                      generate [WORD S(ITEM)]
                  end
              LISTCOUNT := 0
```

(b)

```
+JSUB  XREAD
 WORD  1
 WORD  VALUE
```

(c)

**Figure 5.14** Code generation for a READ statement.