Unit 5: Compiler Design [5 hrs]

5.1: BASIC COMPILER FUNCTIONS

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

Example of a Pascal program (5.1)

```
 1   PROGRAM STATS
 2   VAR
 3       SUM,SUMSQ,I,VALUE,MEAN,VARIANCE : INTEGER
 4   BEGIN
 5       SUM   :=  0;
 6       SUMSQ := 0;
 7       FOR I := 1 TO 100 DO
 8           BEGIN
 9               READ(VALUE);
10               SUM := SUM + VALUE;
11               SUMSQ := SUMSQ + VALUE * VALUE
12           END;
13       MEAN := SUM DIV 100; .
14       VARIANCE := SUMSQ DIV 100 - MEAN * MEAN;
15       WRITE(MEAN,VARIANCE)
16   END.
```

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.
- In the remainder of this section, we discuss this grammar and show how it relates to the example program in Fig. 5.1.

Simplified Pascal grammar (5.2)

```
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
```

- 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.

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

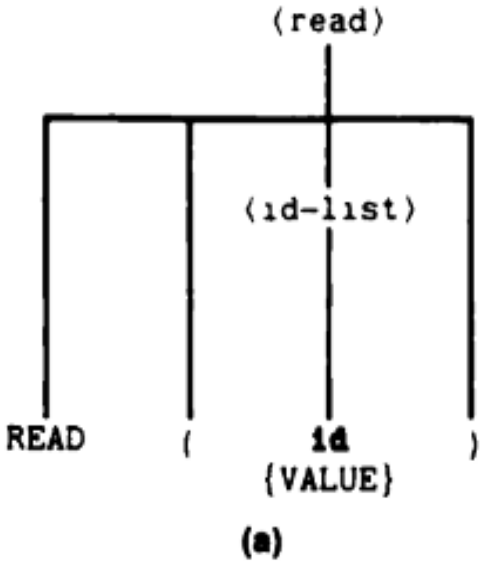
  <id-list> ::= id I <id-list>, id
- This rule offers two possibilities, separated by the **I** symbol, for the syntax of an **<id-list>**.
- The first alternative specifies that an **<id-list>** may consist simply of a token **id** (the notation **id** denotes an identifier that is recognized by the scanner).
- 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.
- 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.**

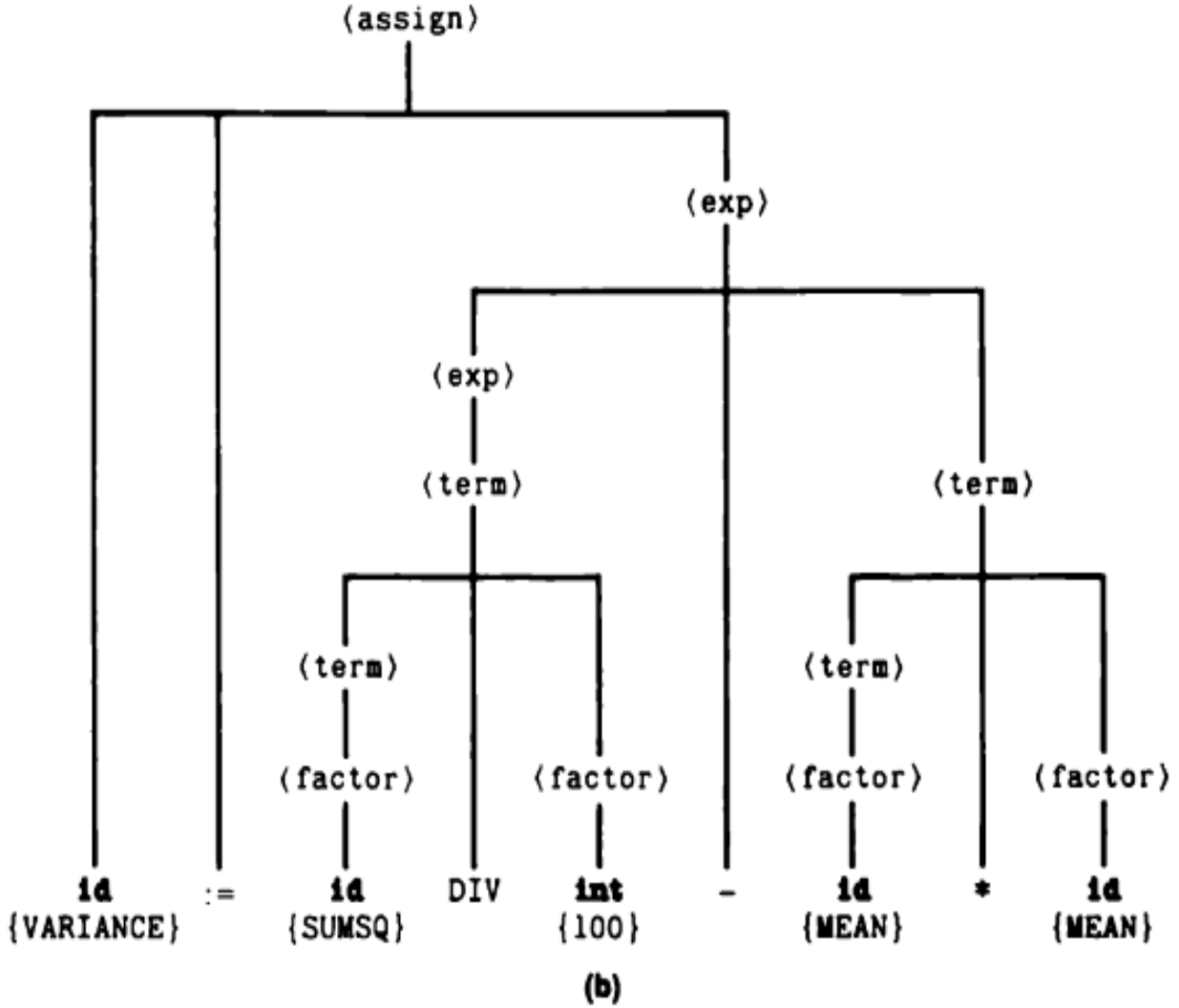
- Rule 9 of the grammar in Fig. 5.2 provides a definition of the syntax of an assignment statement: **<assign> ::= id := <exp>**
- That is, an **<assign>** consists of an **id**, followed by the token **:=**, followed by an expression **<exp>**.
- Rule 10 gives a definition of an **<exp>**:

  <exp> ::= <term> | <exp> + <term> | <exp> - <term>
- 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.
- It is often convenient to display the analysis of a source statement in terms of a grammar as a tree – 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.

Parse trees for two statements from Fig. 5.1 (5.3)



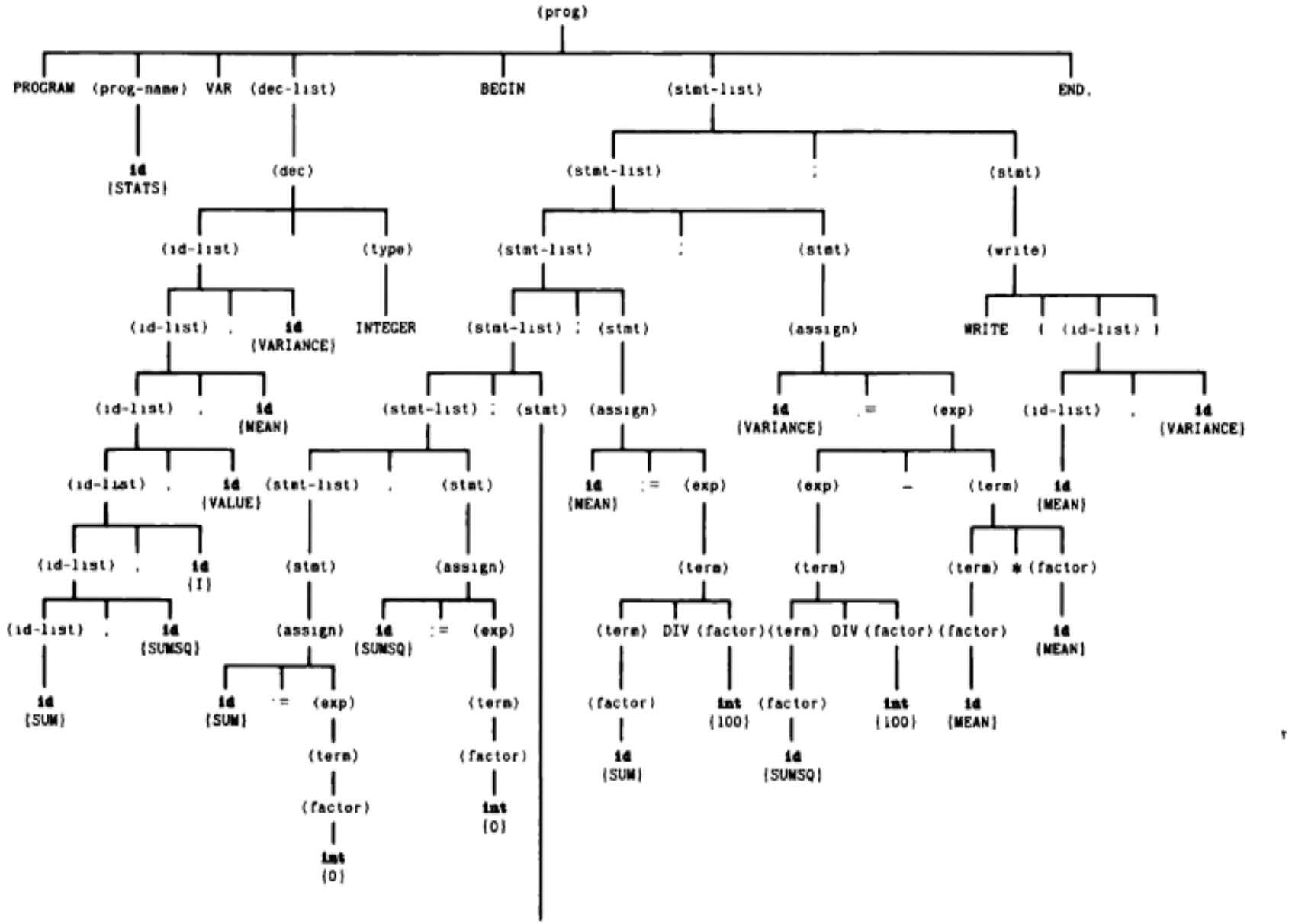
(a)



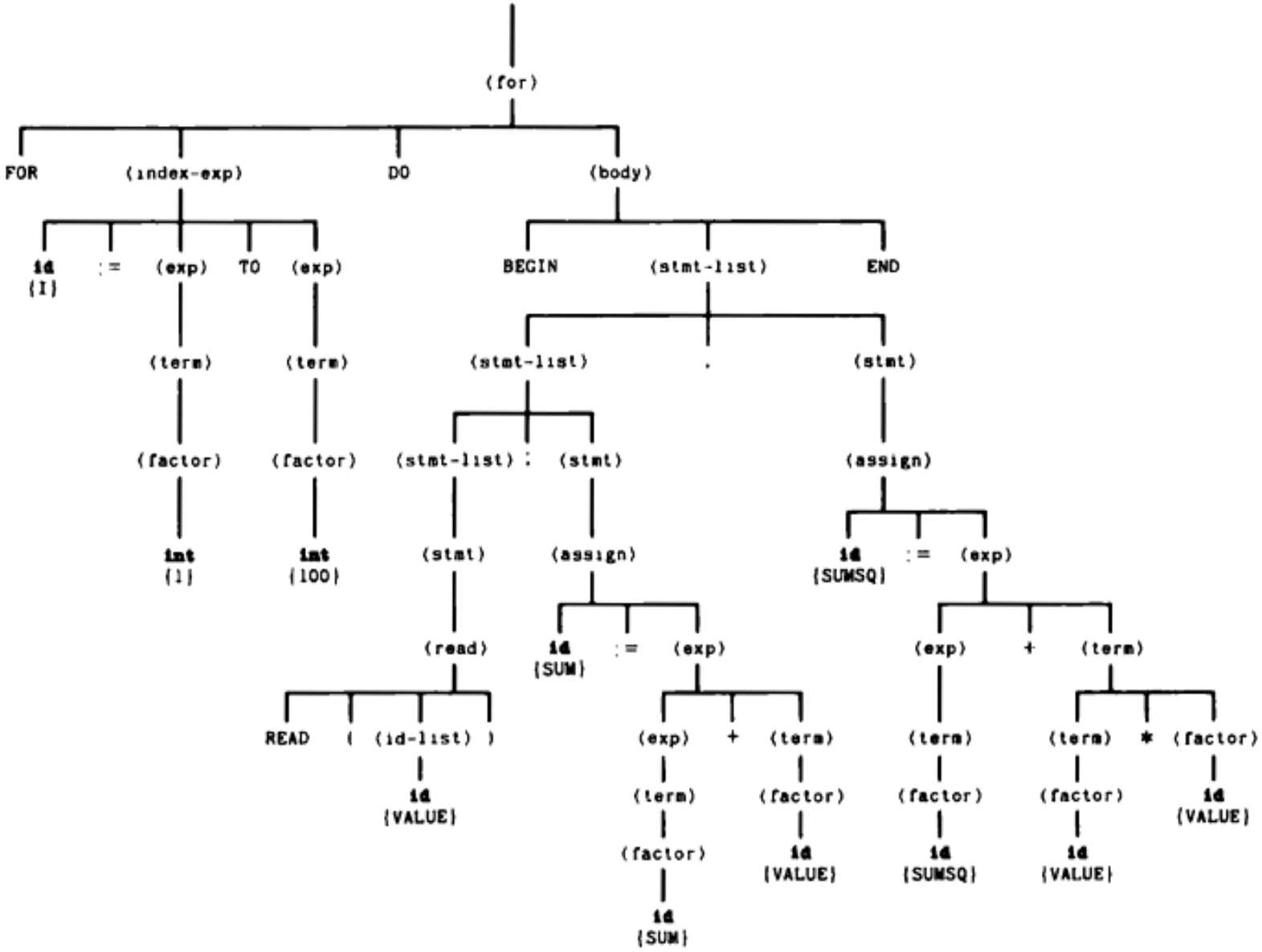
(b)

- Figure 5.3(b) shows the parse tree for statement 14 from Fig. 5.1 in terms of the rules just described.
- 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.

Parse tree for the program from Fig. 5.1 (5.4)



(5.4 Continued)



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 I B I C I D I … I 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 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** would be represented by the integer value **1**, an identifier **id** would be represented by the value **22**, and so on.

Token coding scheme for the grammar from Fig. 5.2 (5.5)

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

- 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.

Lexical scan of the program from Fig. 5.1 (5.6)

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	

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.

Operator-Precedence Parsing

- The **bottom-up parsing** technique we consider is called the ***operator precedence* method**.
- This method is based on examining pairs of consecutive operators in the source program, and making decisions about which operation should be performed first.
- Consider, for example, the arithmetic expression: **A + B * C - D**
- According to the usual rules of arithmetic, multiplication and division are performed before addition and subtraction, that is, multiplication and division have higher ***precedence*** *than addition and subtraction.*
- *If we examine the first* two operators (**+** and *****), we find that **+** has lower precedence than *****. This is often written as: **+ < ***
- Similarly, for the next pair of operators **(*** and **-**), we would find that ***** has higher precedence than **-**. We may write this as: *** > -**
- The **operator-precedence** method uses such observations to guide the parsing process.
- Our previous discussion has led to the following precedence relations for the expression being considered.

A + B * C - D

< >

- This implies that the subexpression **B * C** is to be computed before either of the other operations in the expression is performed.
- In terms of the **parse tree**, this means that the * operation appears at a lower level than does either **+** or **-**.
- Thus a **bottom-up parser** should recognize **B * C**, by interpreting it in terms of the grammar, before considering the surrounding terms.
- The preceding discussion illustrates the fundamental idea behind **operator-precedence parsing**.
- During this process, the statement being analyzed is scanned for a subexpression whose operators have higher precedence than the surrounding operators.
- Then this subexpression is interpreted in terms of the rules of the grammar.
- This process continues until the root of the tree is reached, at which time the analysis is complete.

Recursive-Descent Parsing

- The other **parsing** technique we discuss in this section is a **top-down** method known as ***recursive descent.***
- *A **recursive-descent parser** 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 **nontenninals**.
- 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.

- 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 taken is **id**, it calls the procedure for **<assign>** because this is the only alternative that can begin with the token **id**, and so on.

Code Generation

- See yourself!!!