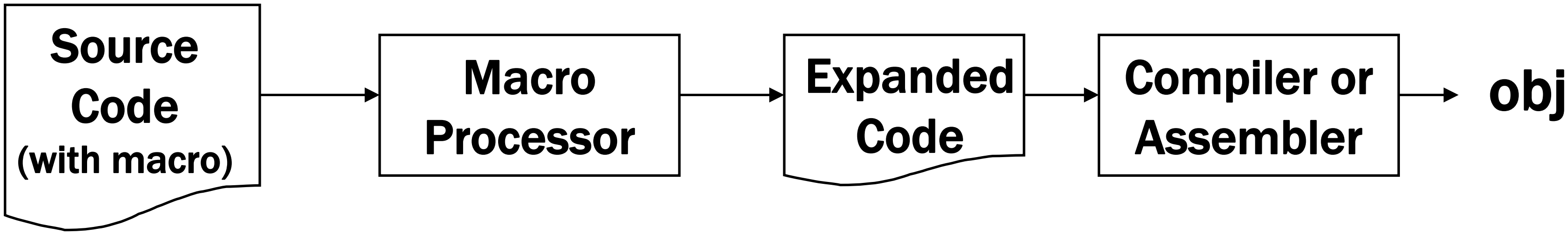
Chapter 5
Compilers



Terminology

- Statement (敘述)
  - Declaration, assignment containing expression (運算式)
- Grammar (文法)
  - A set of rules specify the form of legal statements
- Syntax (語法) vs. Semantics (語意)
  - Example: assuming I, J, K:integer and X,Y:float
  - I:=J+K vs. I:= X+Y
- Compilation (編譯)
  - Matching statements written by the programmer to structures defined by the grammar and generating the appropriate object code.

Basic Compiler

- Lexical analysis (字彙分析) - scanner
  - Scanning the source statement, recognizing and classifying the various tokens
- Syntactic analysis (語法分析) – parser (剖析器)
  - Recognizing the statement as some language construct.
  - Construct a parser tree (syntax tree)
- Code generation – code generator
  - Generate assembly language codes
  - Generate machine codes (Object codes)

Scanner

SUM
:=
0
;
SUMSQ
:=

PROGRAM
STATS
VAR
SUM
,
SUMSQ
,
I

FIGURE 5.1 Example of a Pascal progra

```
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.
```

READ
(
VALUE
)
;

Lexical Analysis

- **Function**
  - Scanning the program to be compiled and recognizing the *tokens* that make up the source statements.

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

- **Tokens**
  - Tokens can be keywords, operators, identifiers, integers, floating-point numbers, character strings, etc.
  - Each token is usually represented by some fixed-length code, such as an integer, rather than as a variable-length character string (see Figure 5.5)
  - Token type, Token specifier (value) (see Figure 5.6)

Scanner Output

- Token specifier
  - Identifier name, integer value, (type)
- Token coding scheme
  - Figure 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

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	

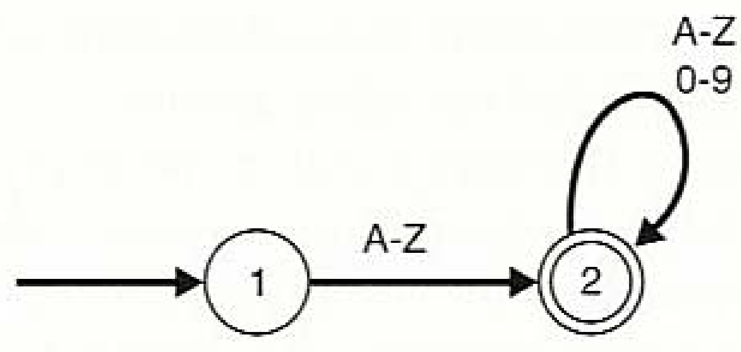
Figure 5.6 Lexical scan of the program from Fig. 5.1.

Token Recognizer

- By grammar

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

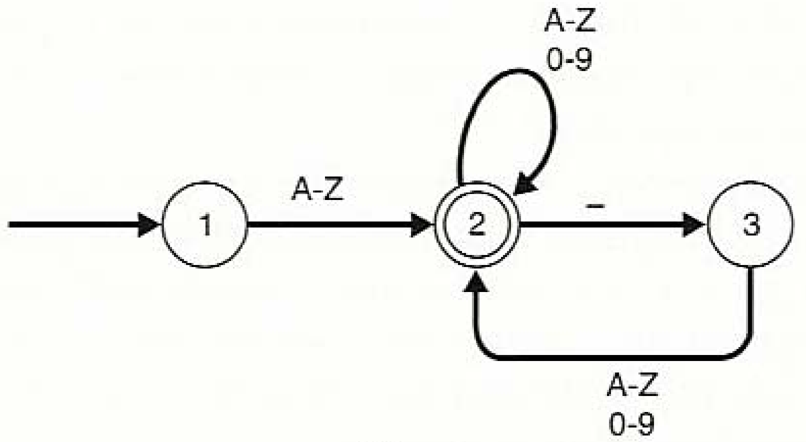
- By scanner - modeling as finite automata (FStateA)
  - Figure 5.8 (a)



(a)

Recognizing Identifier

- Identifiers allowing underscore (_)
  - Figure 5.8 (b)



(b)

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.

Recognizing Identifier

```
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)
```

Recognizing Integer

- **Allowing leading zeroes**
  - Figure 5.8 (c)
- **Disallowing leading zeroes**
  - Figure 5.8 (d)

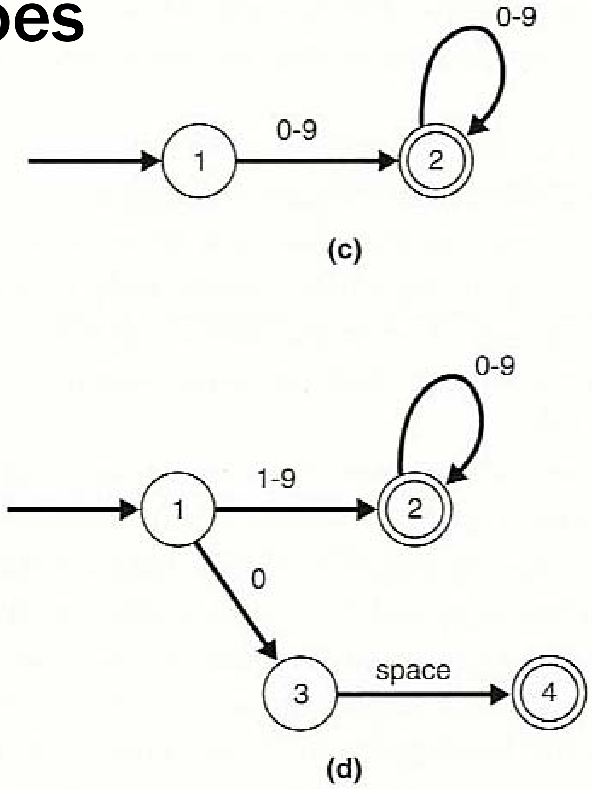


Figure 5.8 Finite automata for typical programming language tokens.

Scanner - Implementation

- Figure 5.10 (a)
  - Algorithmic code for identifier recognition
- Tabular representation of finite automaton for Figure 5.9.

State	A-Z	0-9	;,+-*()	:	=	.
1	2	4	5	6		
2	2	2				3
3						
4		4				
5						
6					7	
7						

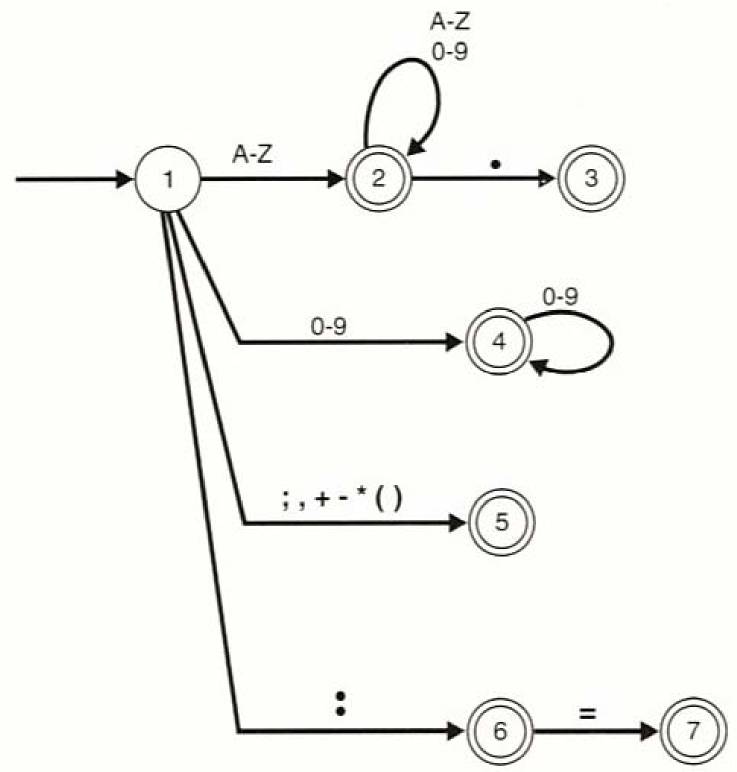


Figure 5.9 Finite automaton to recognize tokens from Fig. 5.5.

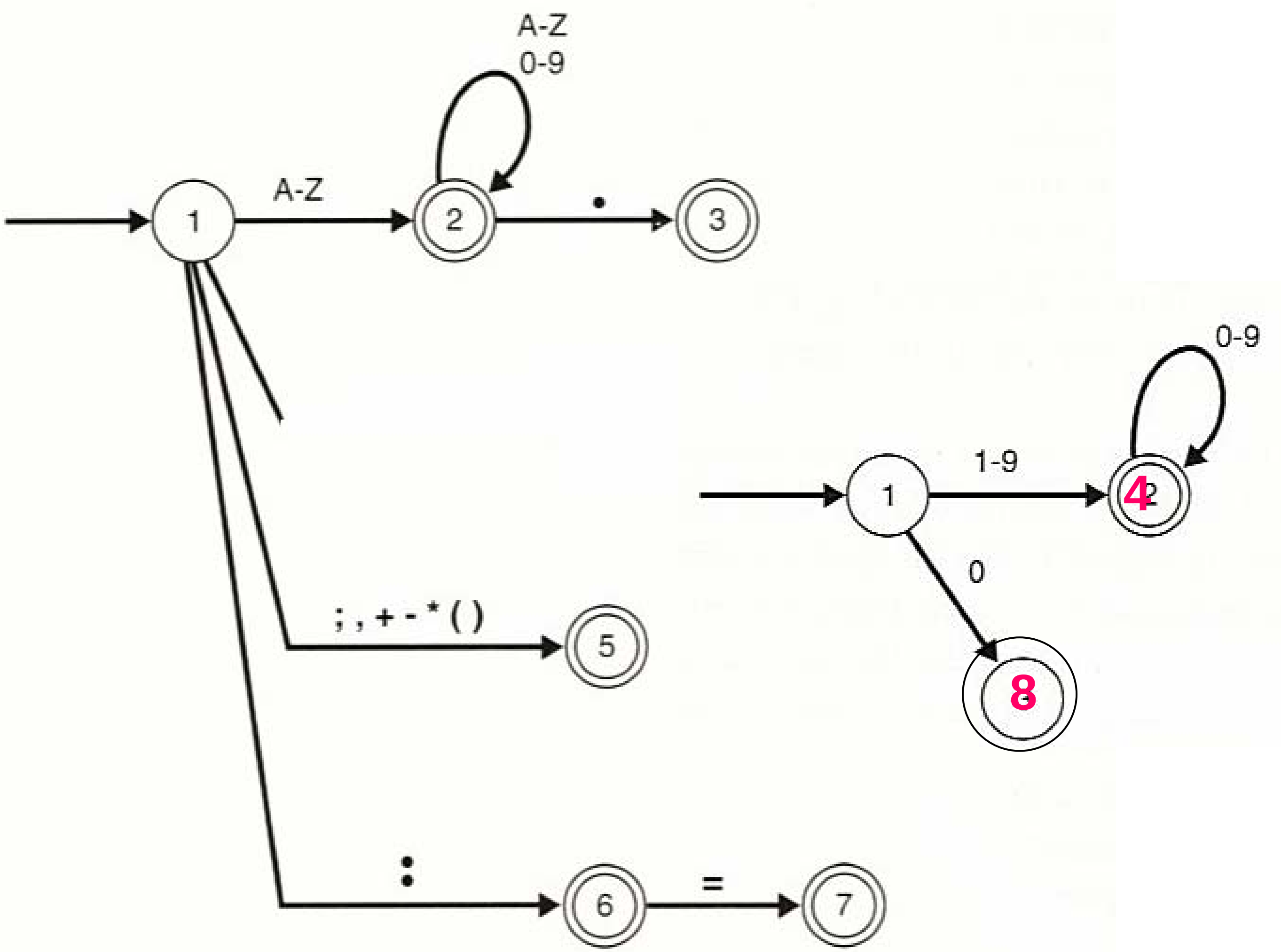


Figure 5.9 Finite automaton to recognize tokens from Fig. 5.5.

Parser

- Grammar: a set of rules
  - Backus-Naur Form (BNF)
  - Ex: Figure 5.2

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

- Terminology
  - Define symbol ::=
  - Nonterminal symbols < >
  - Alternative symbols |
  - Terminal symbols

Simplified Pascal Grammar

```
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.

Parser

- `READ(VALUE)`
- `SUM := 0`
- `SUM := SUM + VALUE`
- `MEAN := SUM DIV 100`

- `<read> ::= READ (<id-list>)`
- `<id-list>::= id | <id-list>,id`
- `<assign>::= id := <exp>`
- `<exp> ::= <term> | <exp>+<term> | <exp>-<term>`
- `<term>::=<factor> | <term>*<factor> | <term> DIV <factor>`
- `<factor>::= id | int | (<exp>)`

Syntax Tree

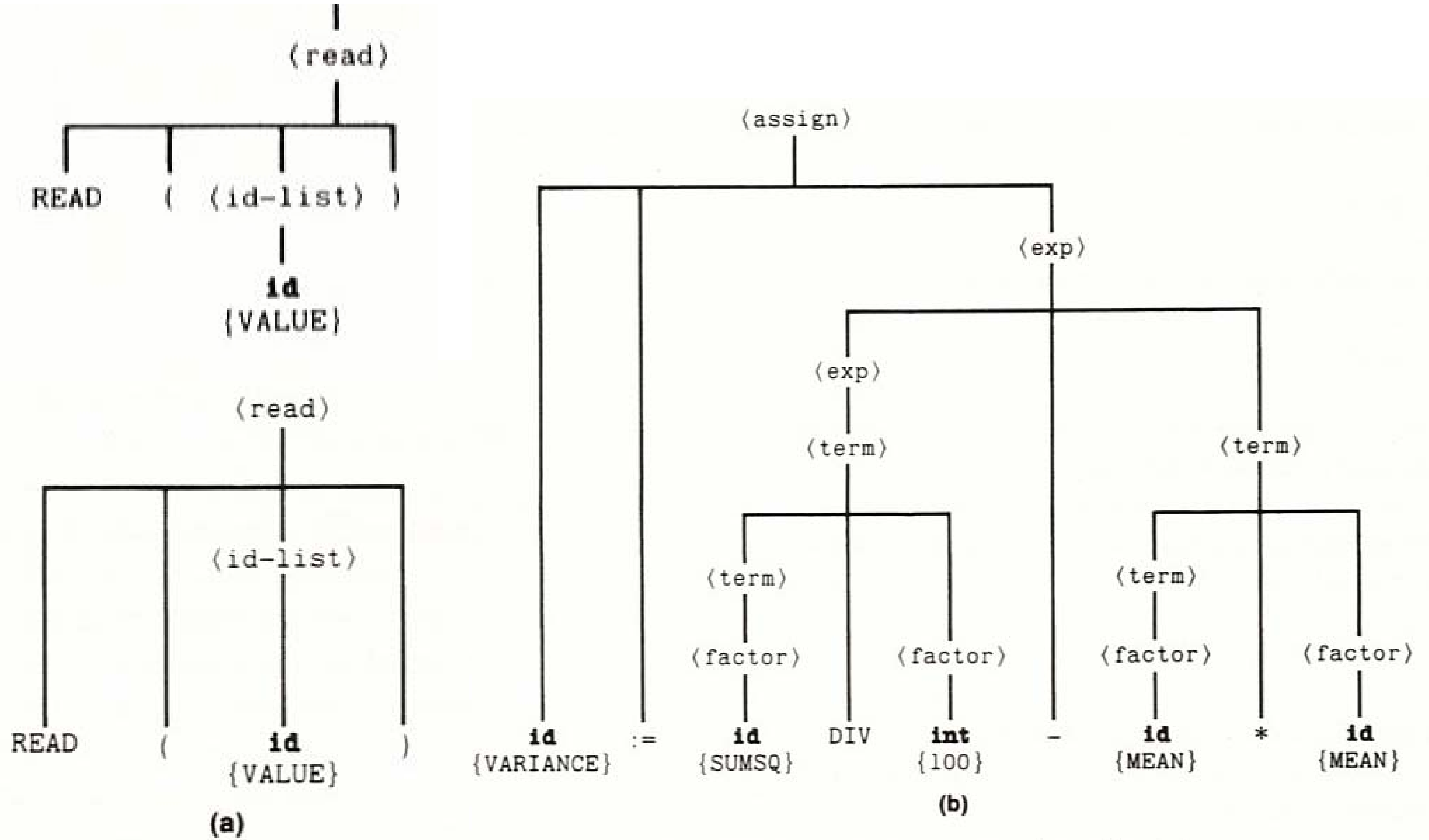
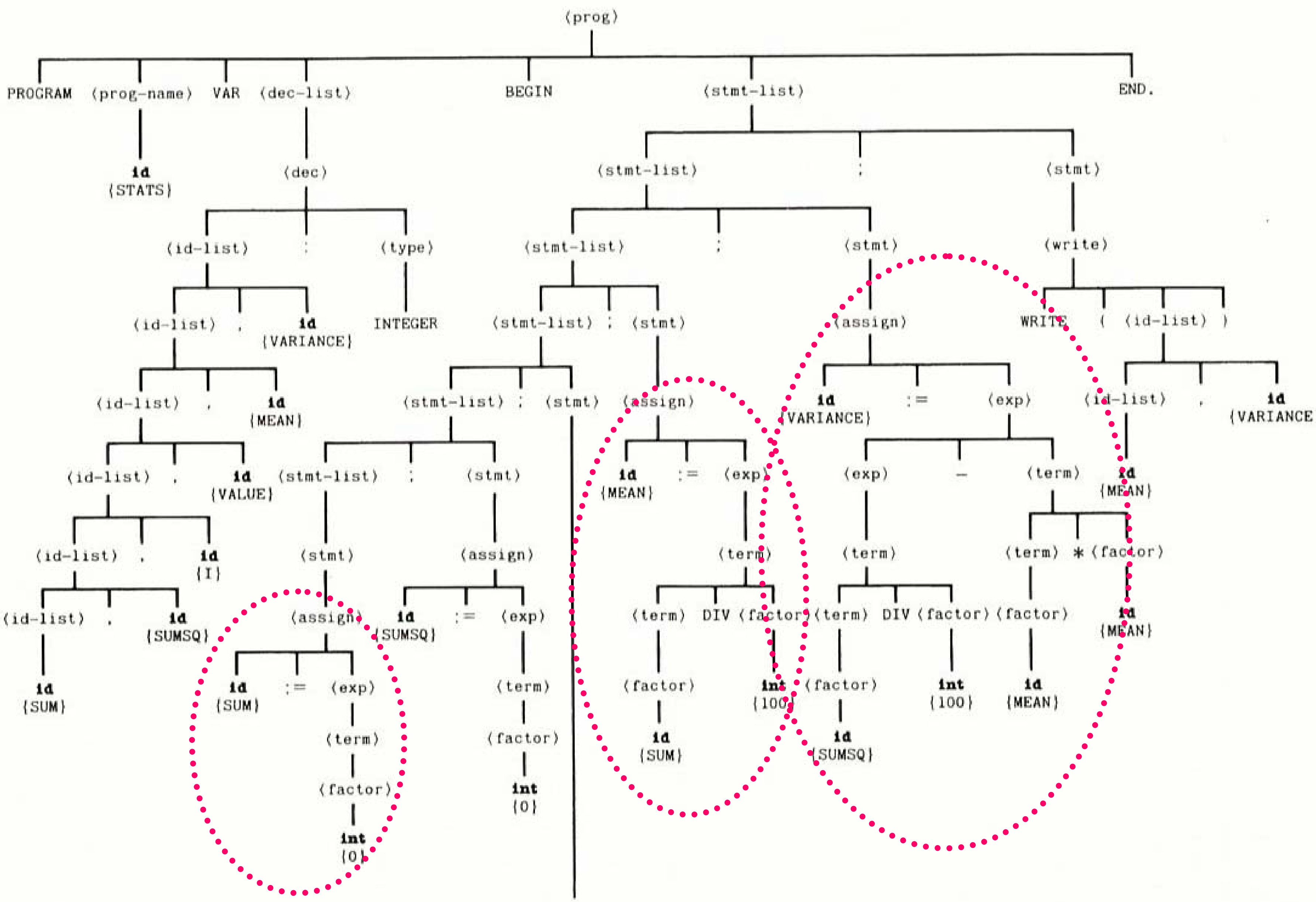


Figure 5.3 Parse trees for two statements from Fig. 5.1.

Syntax Tree for Program 5.1



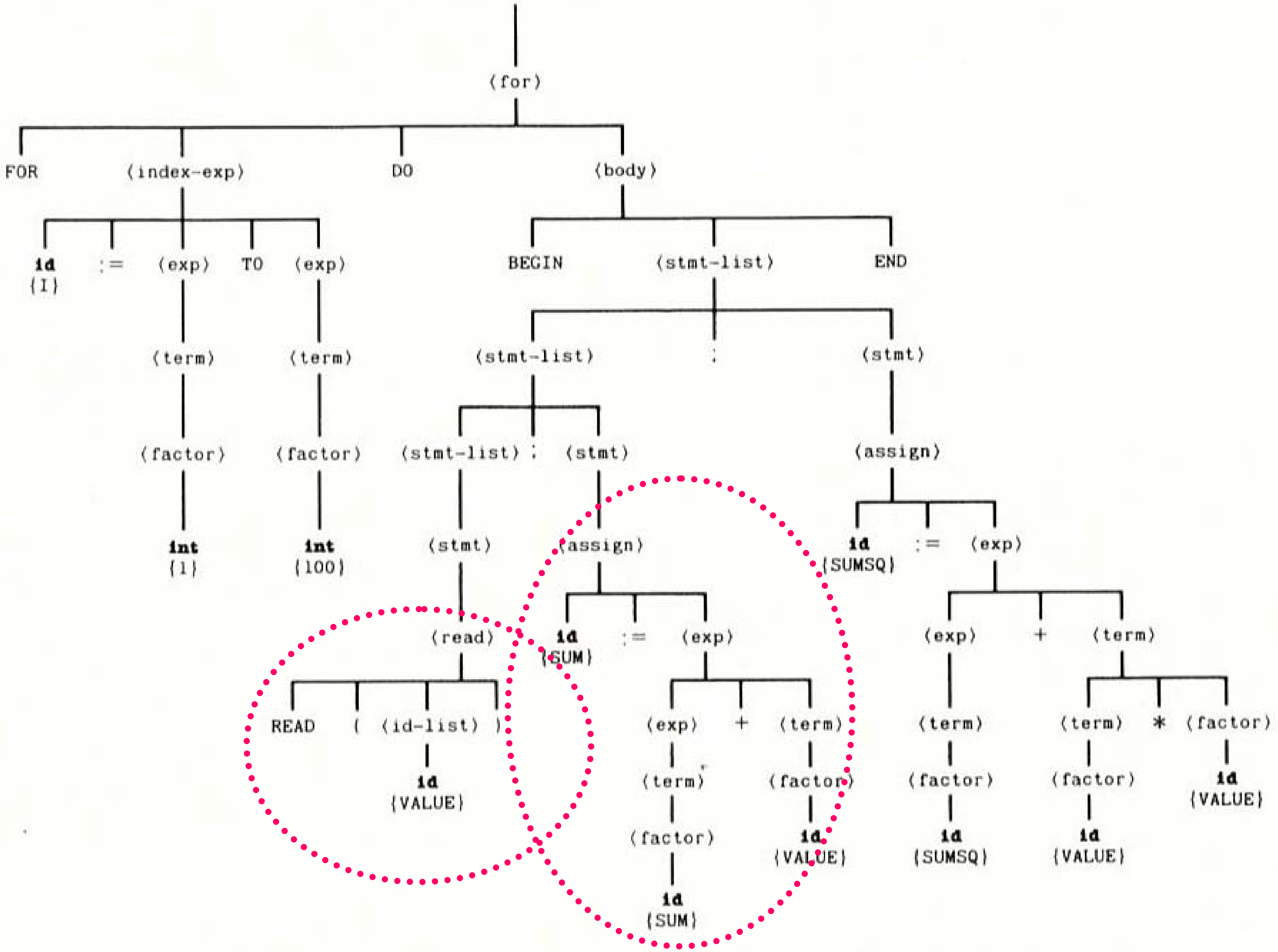


Figure 5.4 Parse tree for the program from Fig. 5.1.

Syntactic Analysis

- Recognize source statements as language constructs or build the parse tree for the statements.
  - Bottom-up
    - Operator-precedence parsing
    - Shift-reduce parsing
    - LR(0) parsing
    - LR(1) parsing
    - SLR(1) parsing
    - LALR(1) parsing
  - Top-down
    - Recursive-descent parsing
    - LL(1) parsing

Operator-Precedence Parsing

- Operator
  - Any terminal symbol (or any token)
- Precedence
  - * » +
  - \+ « *
- Operator-precedence
  - Precedence relations between operators

A + B * C - D
⋖ ⋗

PROGRAM ≐ VAR

and

BEGIN ⋖ FOR

; ⋗ END

but

END ⋗ ;

Precedence Matrix for the Fig. 5.2

	VAR	BEGIN	END	END.	INTEGER	FOR	READ	WRITE	TO	DO	;	:	,	:=	+	−	*	DIV	(	)	id	int
PROGRAM	≐																				⋖	
VAR		≐									⋖	⋖	⋖								⋖	
BEGIN			≐	≐		⋖	⋖	⋖			⋖										⋖	
END			⋗	⋗							⋗											
INTEGER		⋗									⋗											
FOR										≐											⋖	
READ																			≐			
WRITE																			≐			
TO										⋗					⋖	⋖	⋖	⋖	⋖		⋖	⋖
DO		⋖	⋗	⋗		⋖	⋖	⋖			⋗										⋖	
;		⋗	⋗	⋗		⋖	⋖	⋖			⋗	⋖	⋖								⋖	
:		⋗			⋖						⋗											
,																					≐	
:=			⋗	⋗					≐		⋗				⋖	⋖	⋖	⋖	⋖		⋖	⋖
+			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋖	⋖	⋖	⋗	⋖	⋖
−			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋖	⋖	⋖	⋗	⋖	⋖
*			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋗	⋗	⋖	⋗	⋖	⋖
DIV			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋗	⋗	⋖	⋗	⋖	⋖
(													⋖		⋖	⋖	⋖	⋖	⋖	≐	⋖	⋖
)			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋗	⋗		⋗		
id	⋗		⋗	⋗					⋗	⋗	⋗	⋗	⋗	≐	⋗	⋗	⋗	⋗		⋗		
int			⋗	⋗					⋗	⋗	⋗				⋗	⋗	⋗	⋗		⋗		

Figure 5.11 Precedence matrix for the grammar from Fig. 5.2.

Operator-Precedence Parse Example

BEGIN READ (VALUE) ;

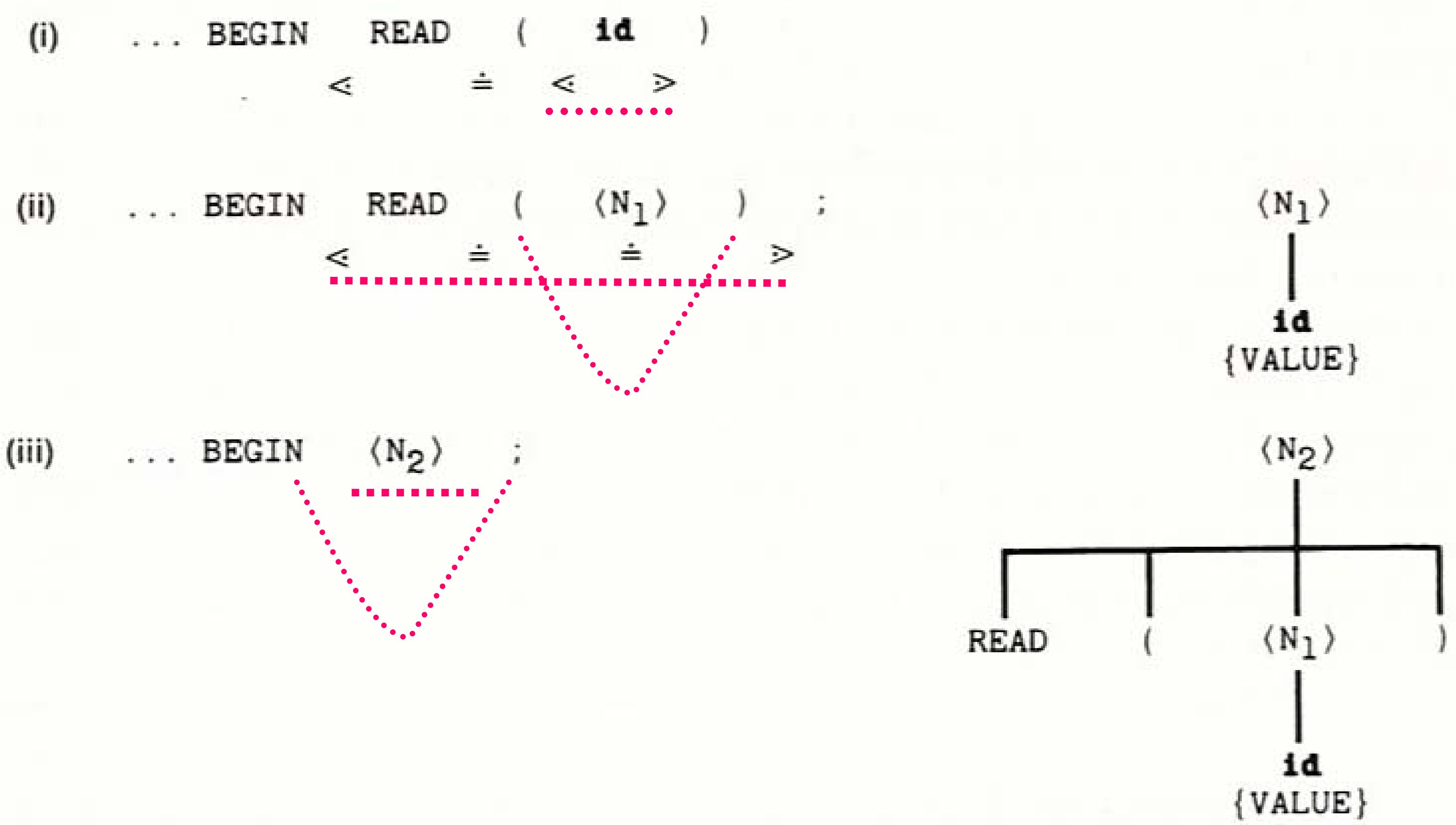
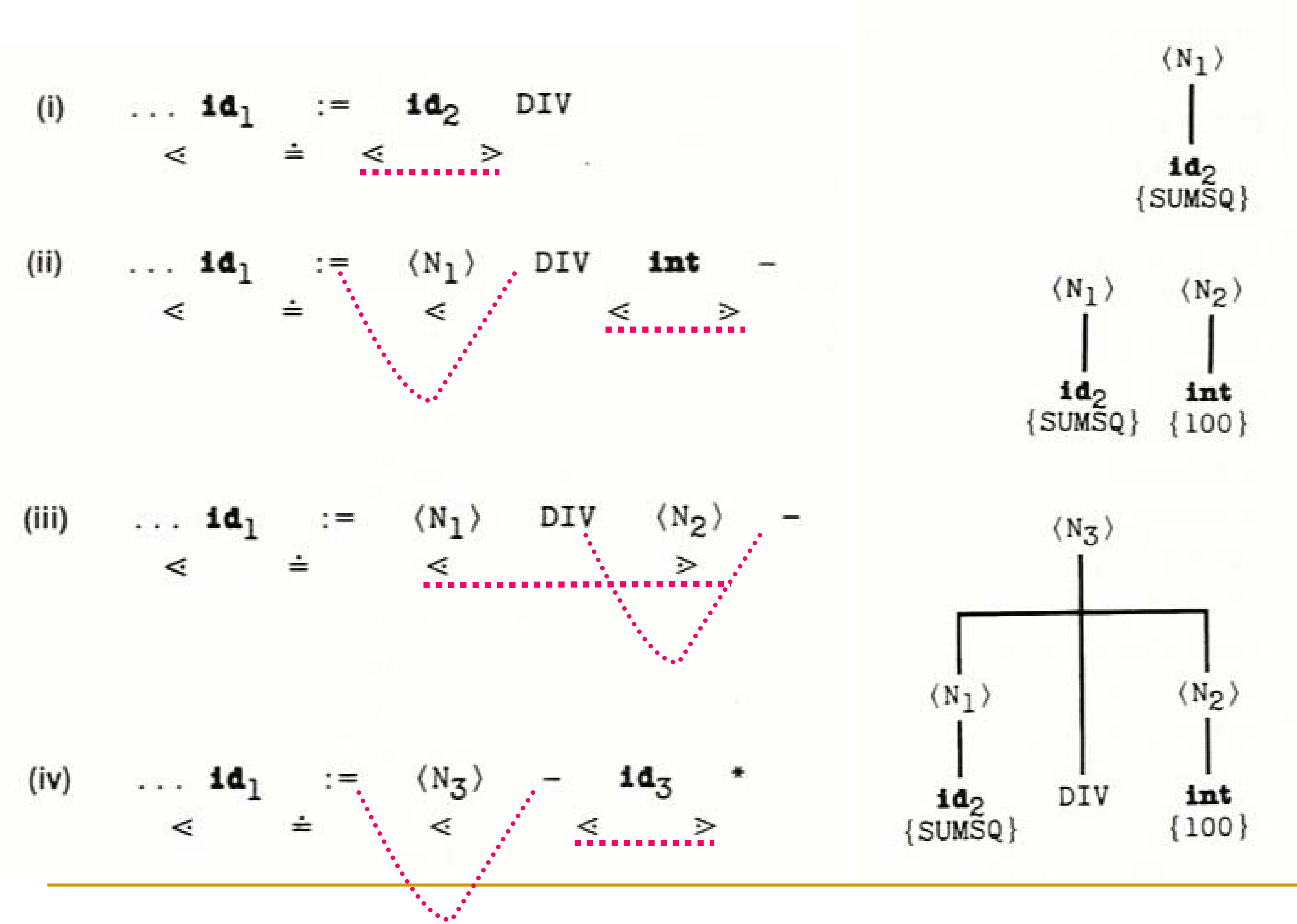


Figure 5.12 Operator-precedence parse of a READ statement.

Operator-Precedence Parse Example



Operator-Precedence Parse Example

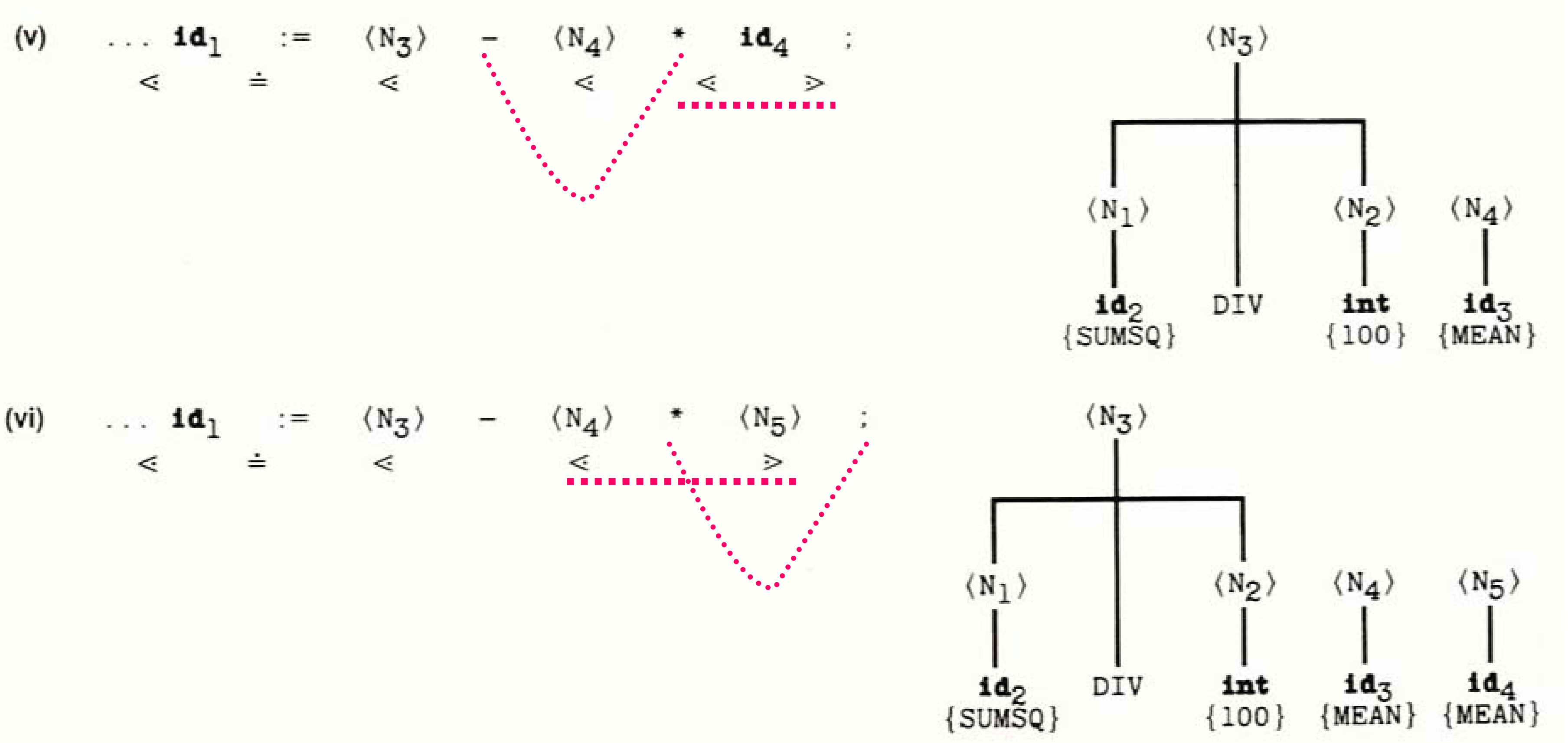
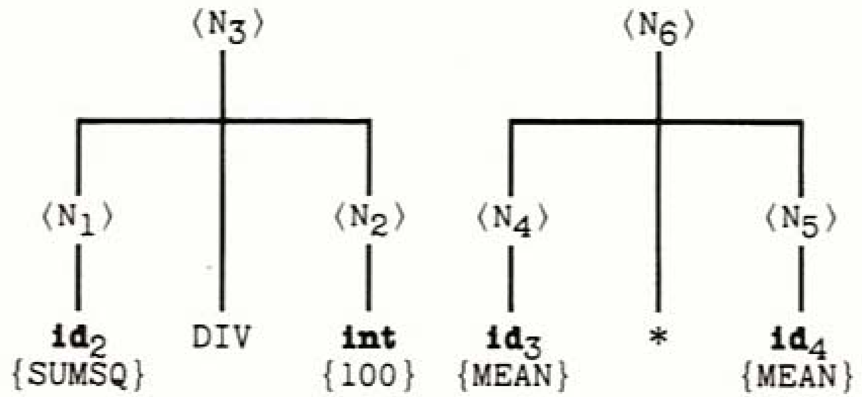


Figure 5.13 Operator-precedence parse of an assignment statement.

Operator-Precedence Parse Example

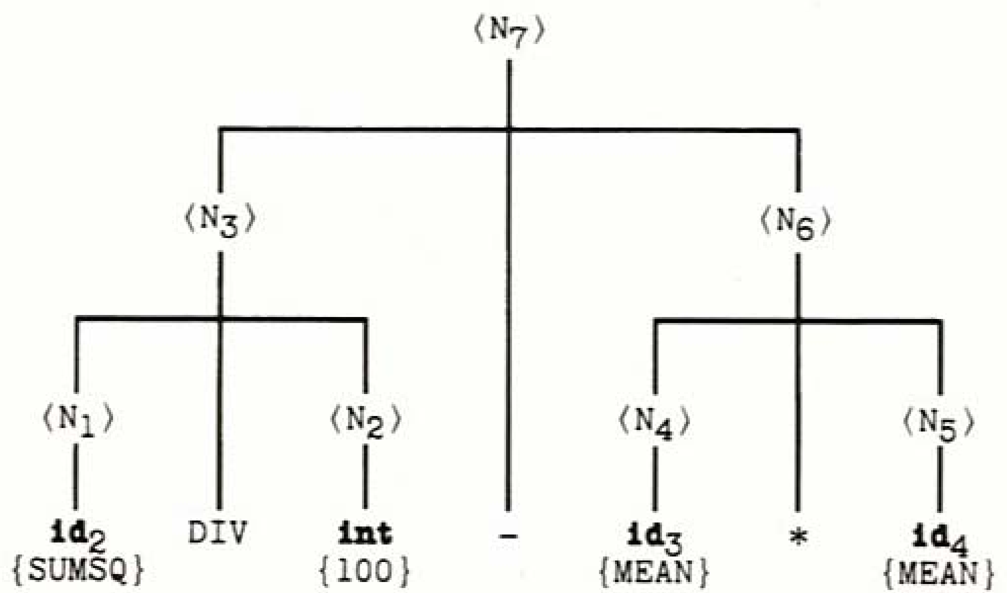
(vii) $\ldots\ \mathbf{id}_1\ :=\ \langle N_3 \rangle\ -\ \langle N_6 \rangle\ ;$

$\lessdot\ \doteq\ \lessdot\ \gtrdot$



(viii) $\ldots\ \mathbf{id}_1\ :=\ \langle N_7 \rangle\ ;$

$\lessdot\ \doteq\ \gtrdot$



Operator-Precedence Parsing

- Bottom-up parsing
- Generating precedence matrix
  - Aho et al. (1988)

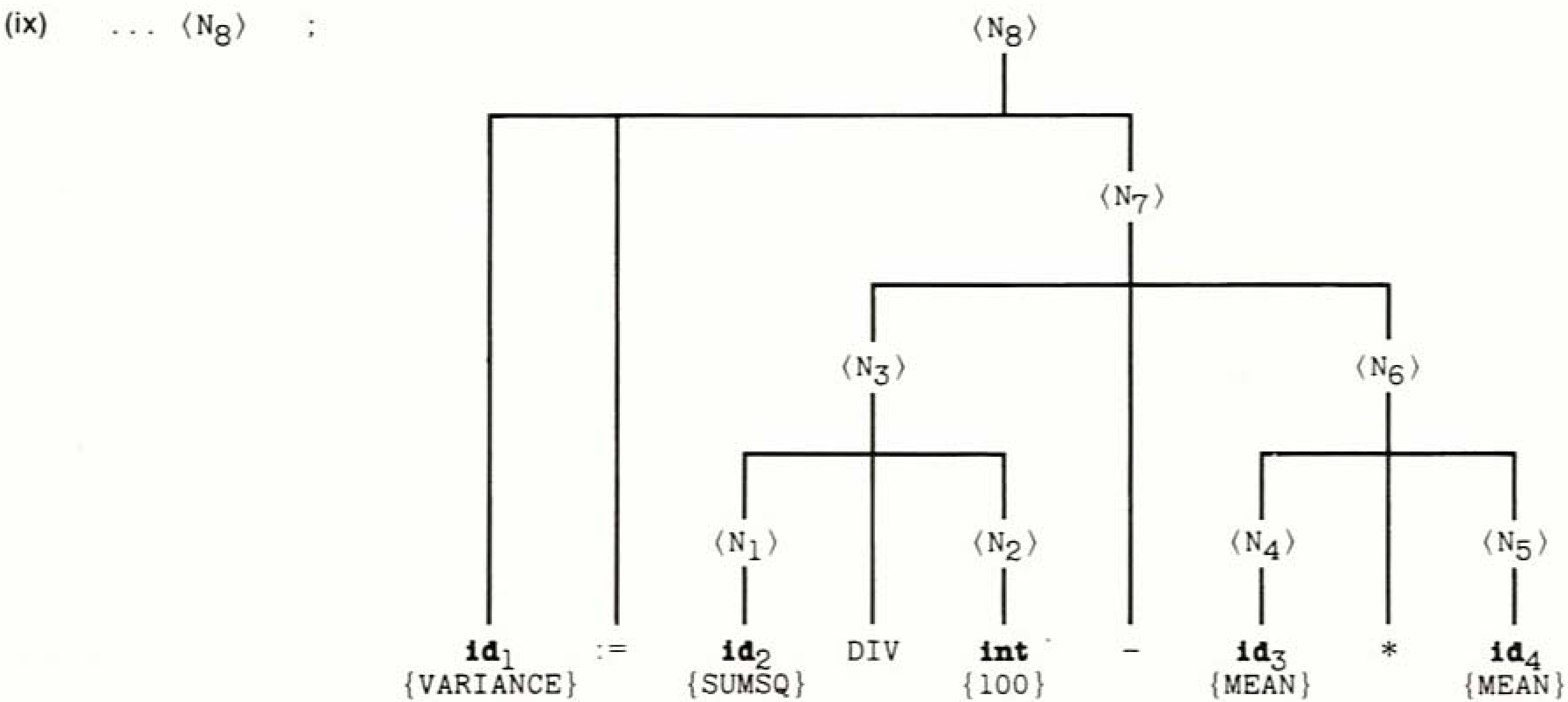


Figure 5.13 (*cont'd*)

Shift-reduce Parsing with Stack

- Figure 5.14

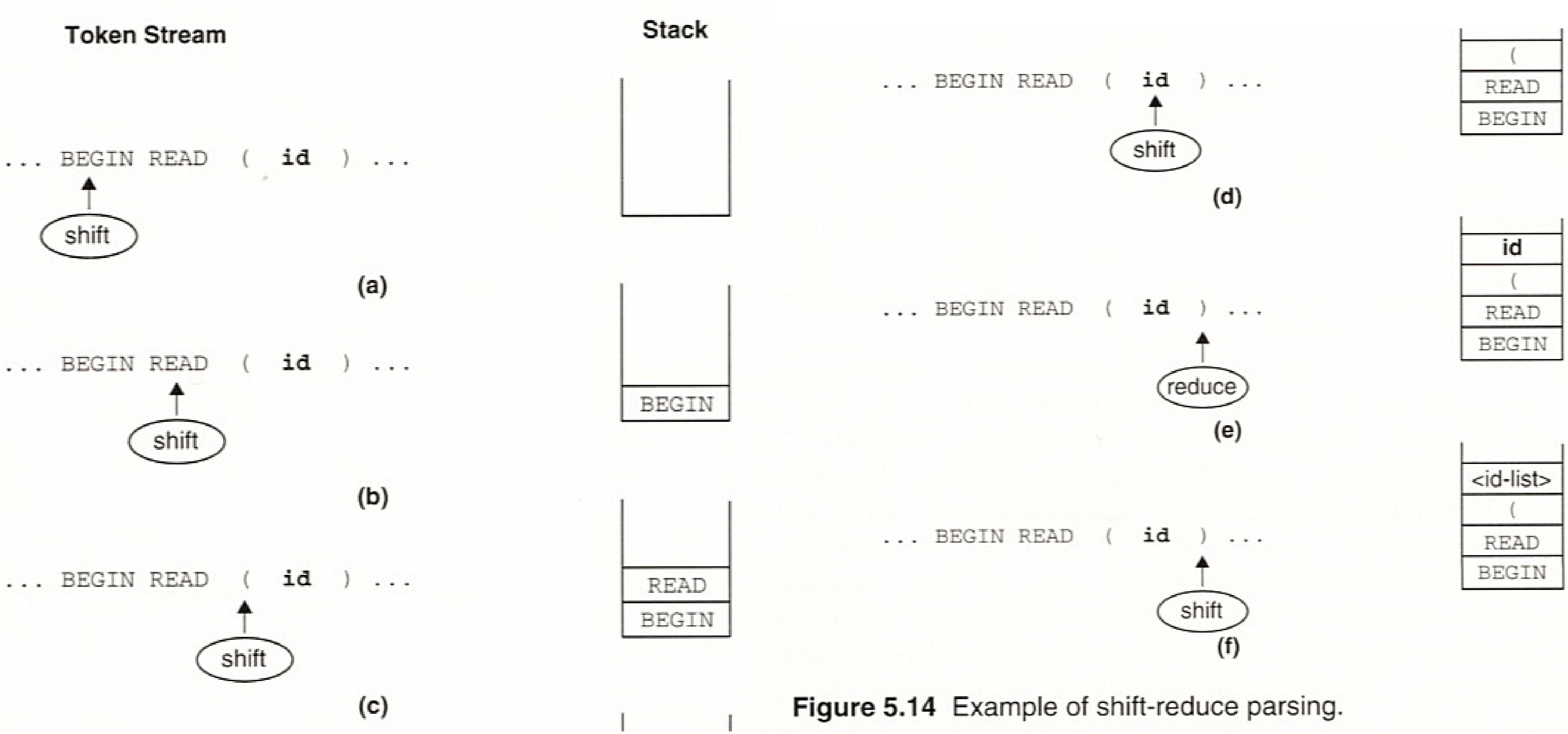


Figure 5.14 Example of shift-reduce parsing.

Recursive-Descent Parsing

- Each nonterminal symbol in the grammar is associated with a procedure.

```
<read>::=READ (<id-list>)
<stmt>::=<assign> | <read> | <write> | <for>
```

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

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

- Left recursion
  - `<dec-list> ::= <dec> | <dec-list>;<dec>`
- Modification
  - `<dec-list> ::= <dec> {;<dec>}`

Recursive-Descent Parsing (cont'd.)

```
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 5.15 Simplified Pascal grammar modified for recursive-descent parse.

Recursive-Descent Parsing of READ

```
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}
```

Recursive-Descent Parsing of IDLIST

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

```
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)

Figure 5.16 Recursive-descent parse of a READ statement.

Recursive-Descent Parsing (cont'd.)

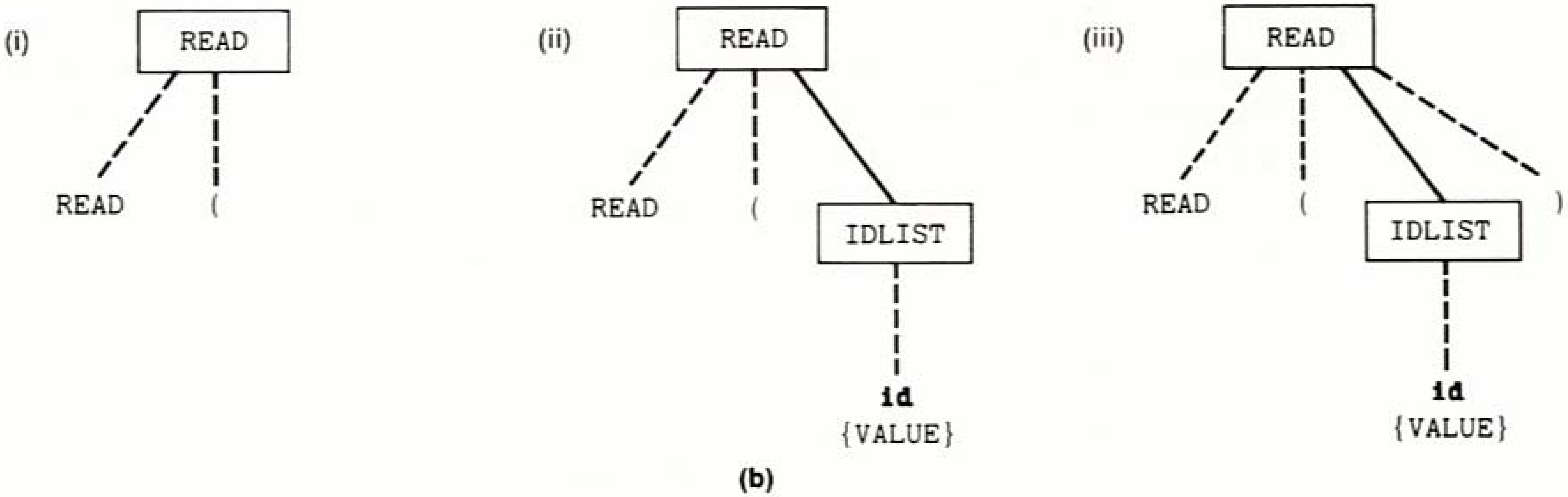


Figure 5.16 (*cont'd*)

Recursive-Descent Parsing of ASSIGN

```
9     <assign>     ::= id := <exp>
10a   <exp>        ::= <term> { + <term> | - <term> }
11a   <term>       ::= <factor> { * <factor> | DIV < factor> }
```

```
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}
```

Recursive-Descent Parsing of EXP

```
                  9    <assign>        ::= id := <exp>
procedure EXP     10a  <exp>           ::= <term> { + <term> | - <term> }
      begin       11a  <term>          ::= <factor> { * <factor> | DIV < factor> }
         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 5.17 Recursive-descent parse of an assignment statement.

Recursive-Descent Parsing of TERM

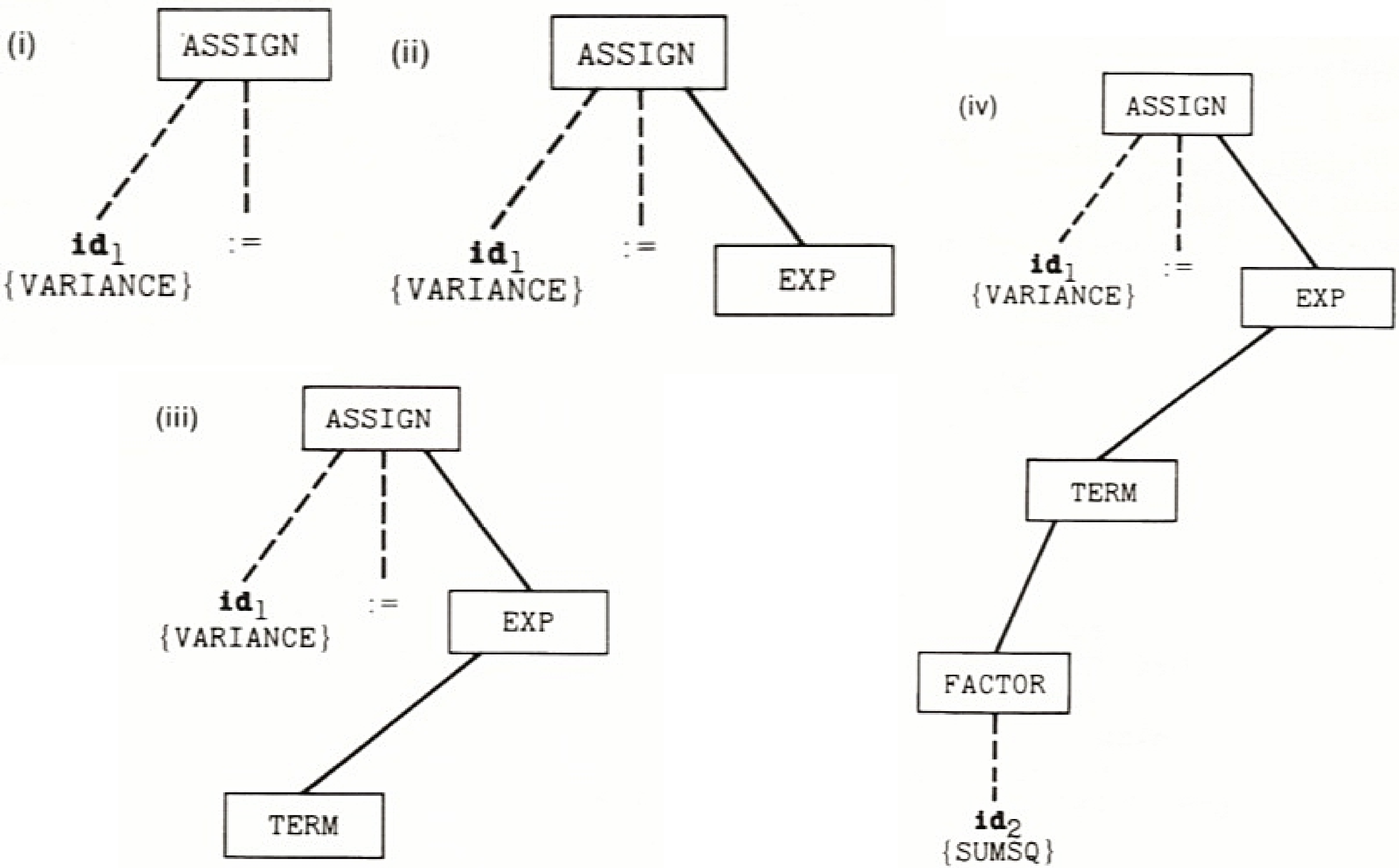
```
9     <assign>     ::= id := <exp>
10a   <exp>        ::= <term> { + <term> | - <term> }
11a   <term>       ::= <factor> { * <factor> | DIV < factor> }
```

```
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}
```

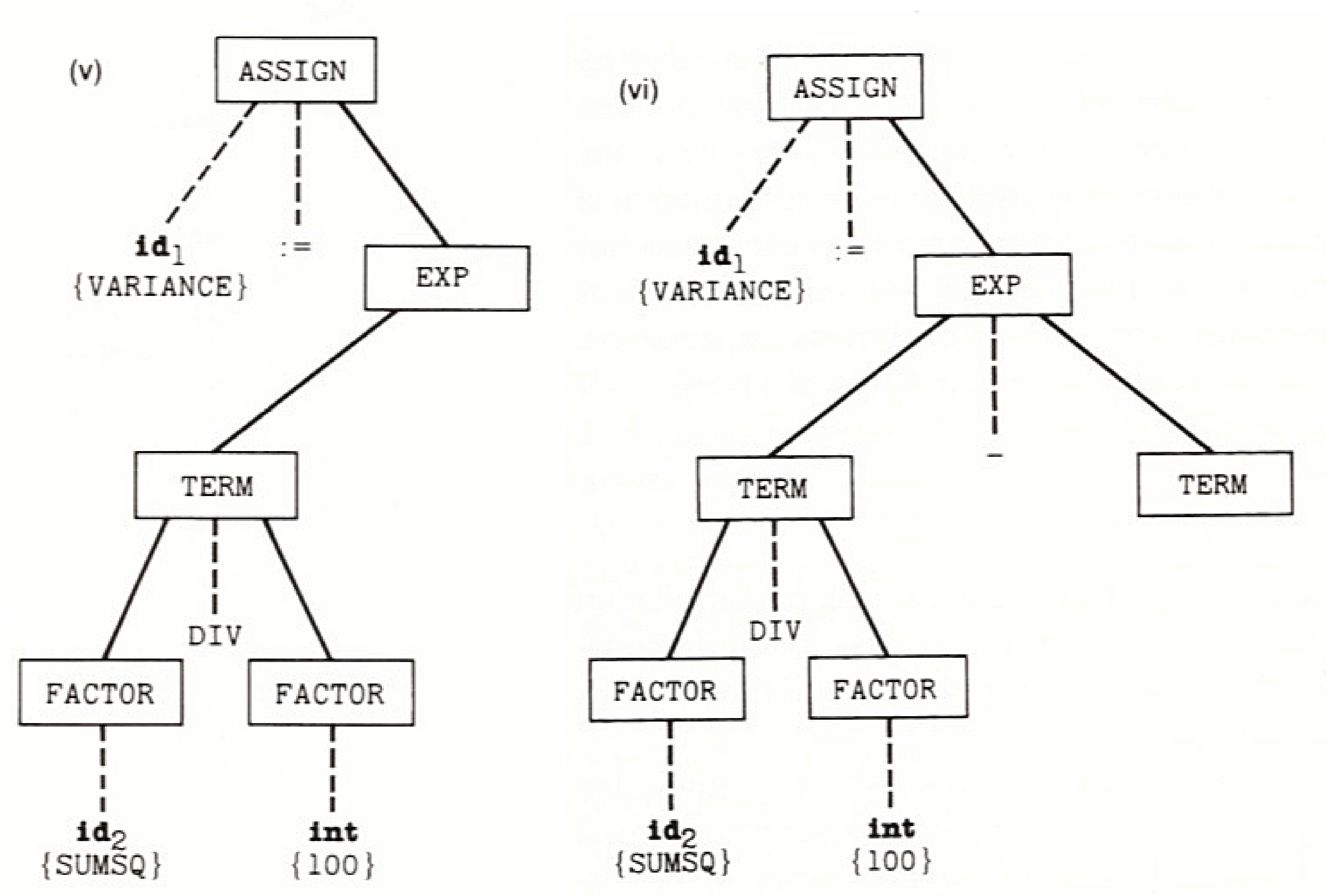
Recursive-Descent Parsing of FACTOR

```
procedure FACTOR 12   <factor>        ::= id | int | ( <exp> )
        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}
```

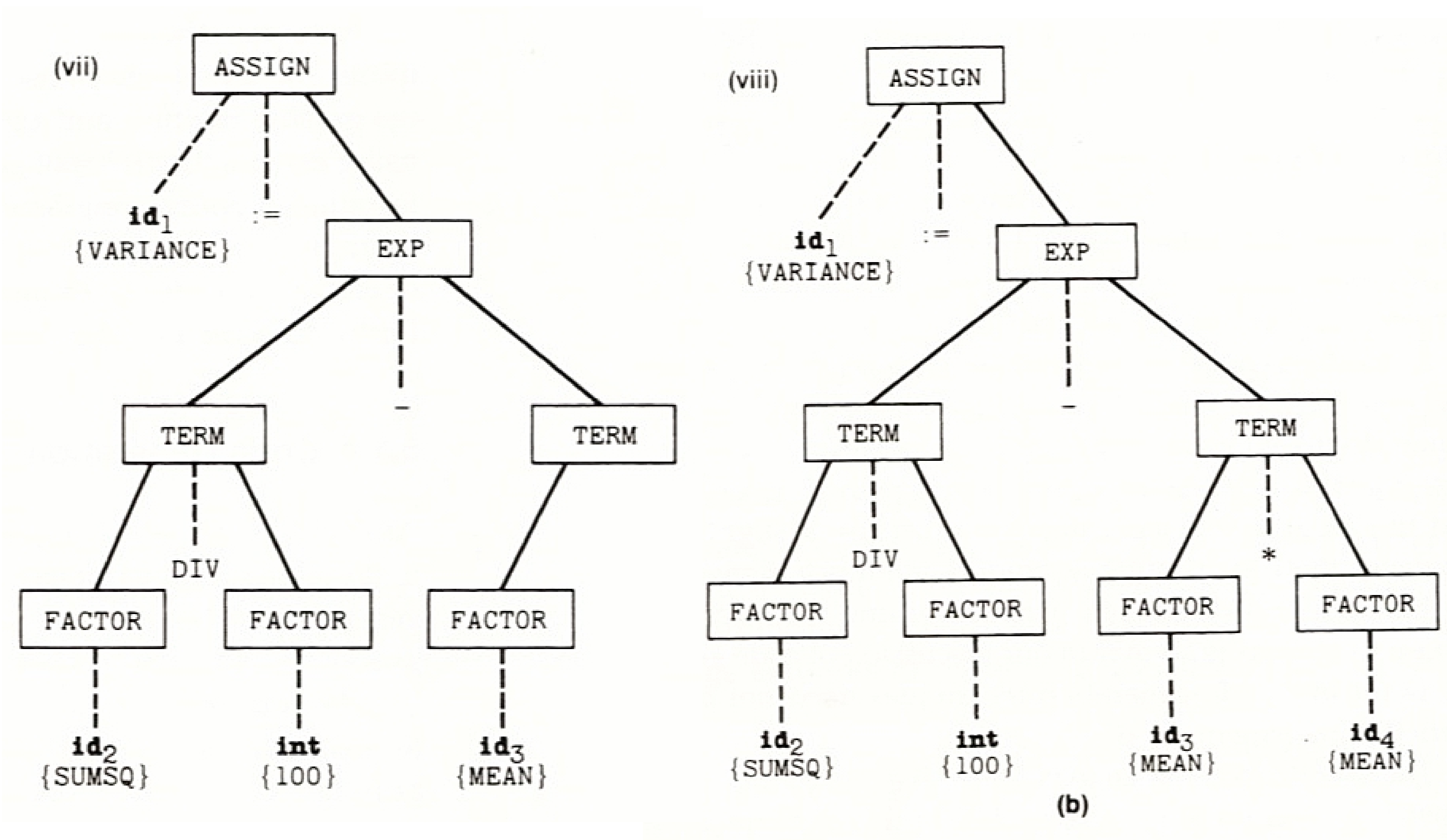
Recursive-Descent Parsing (cont'd.)



Recursive-Descent Parsing (cont'd.)

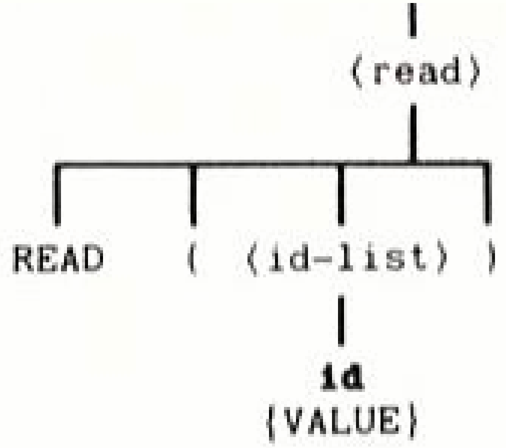


Recursive-Descent Parsing (cont'd.)



Code Generation

Add S(id) to LIST and LISTCOUNT++



(a)

```
<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)

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

        add S(id) to list
        add 1 to LISTCOUNT
```

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

(c)

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

        add S(id) to list
        add 1 to LISTCOUNT
```

Figure 5.18 Code generation for a READ statement.