Module 10

Compiler Basics

Module Ten

- Compiler Basics - Part Three
- In this presentation, we are going to talk about :
- Syntactical Analysis

Overview

- Previously we talked about:
- Compiler Basic Functions
- Language Definition – Grammar
- Lexical Analysis

Next: Syntactical Analysis

Syntactical Analysis

- Build the Structures.
- Recognize the constructs.
- Build the parse tree.
- BOTTOM_UP
  - Begin with the tokens and attempt to build a structure.
- TOP_DOWN
  - Begin with a goal and attempt to reach it.

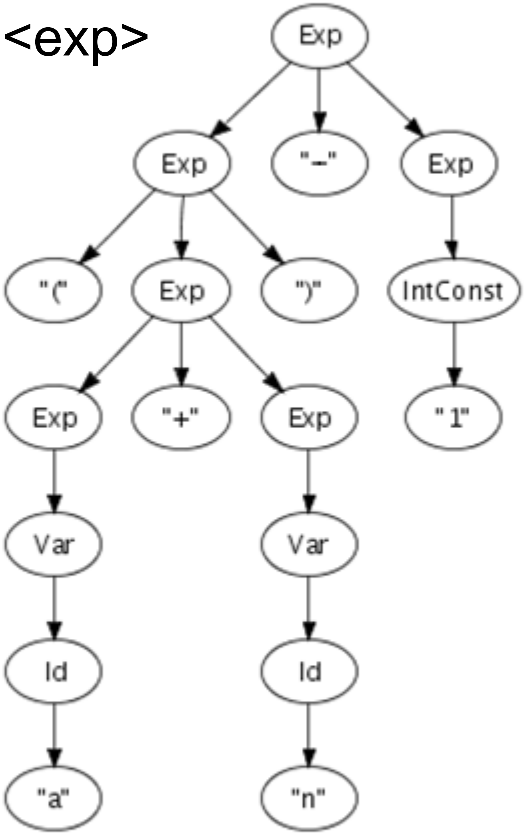
Parse tree

- Used to graphically display the analysis of the source statements.
- Shows the structure - Syntax Tree
- Given: Statements + grammar ==> Parse Tree
- Given: Parse Tree + grammar ==> Statements
- Diagram a Program.
- More than one tree, then ambiguous grammar.

Parse tree

- Given this grammar:
- <exp>::= <exp> | <exp> + <exp> | <exp> - <exp>
- <exp> ::= **id** | **int** | (<exp>)

- (a + n) - 1



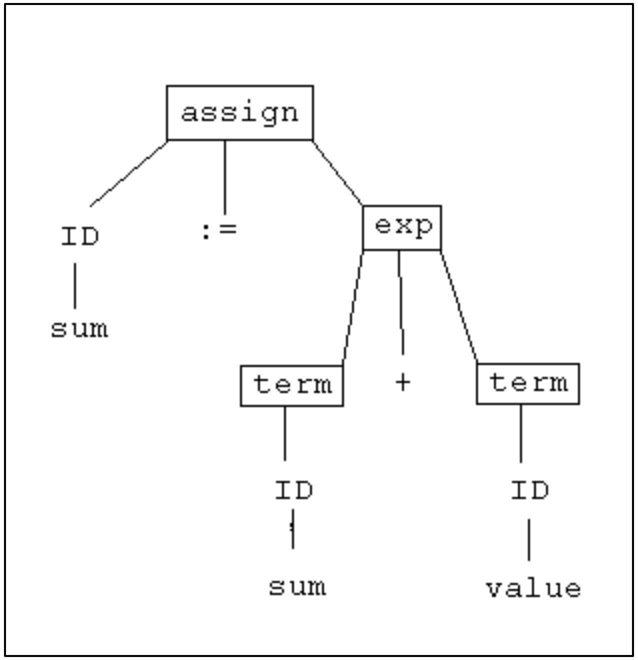
Parse tree

- Given this grammar:

- <assign> ::= **id** := <exp>
- <exp> ::= <term> { + <term> | - <term> }

- and this statement

- **sum := sum + value;**



Syntactical Analysis   BOTTOM_UP

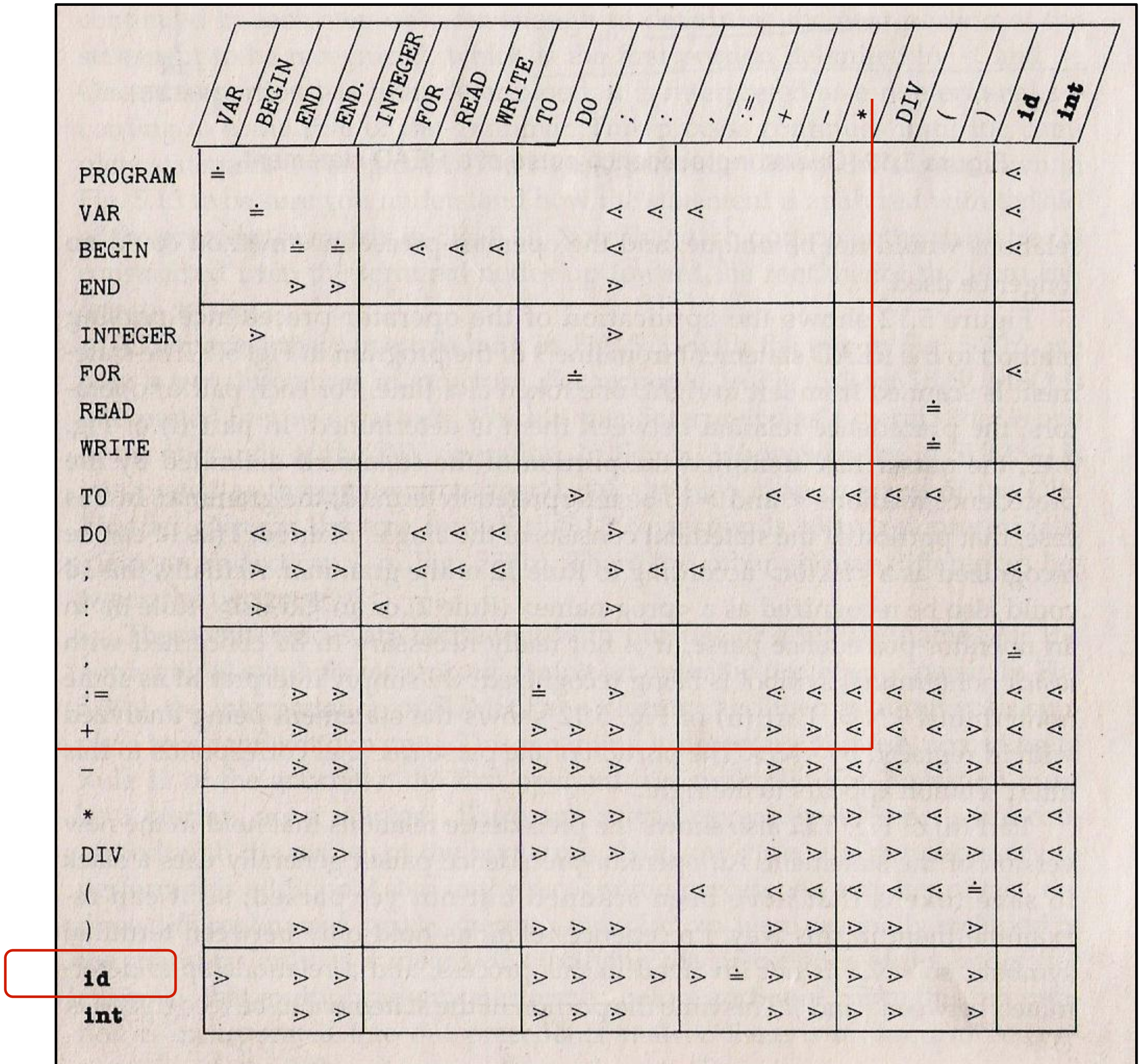
- **Operator Precedence Parsing**

- Operators are the Terminal Symbols of the language.
- Precedence Matrix
- Build (or obtain) the precedence matrix.
- Automatic generation.

Precedence Matrix

A + B * C - D

id1 + id2 *



	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.

Syntactical Analysis BOTTOM_UP

- **Operator Precedence Parsing**
- Scan for subexpressions having operators with higher precedence than the surrounding operators.
- $\lessdot$ $\doteq$ $\gtrdot$
- Scan left to right, only as many tokens as needed.

$$\begin{array}{ccccccc} & A & + & B & * & C & - \quad D \\ \lessdot & & & \lessdot & & \gtrdot & \end{array}$$

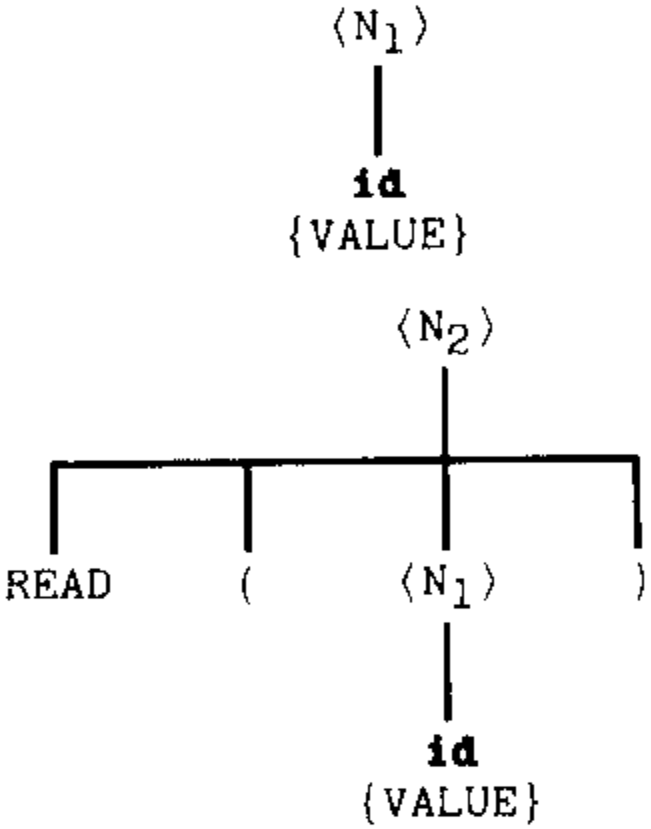
Bottom - Up

- Operator-precedence parse of a READ statement
- … BEGIN READ (VALUE) ;

(i) $\ldots\ \text{BEGIN} \lessdot \text{READ} \doteq (\lessdot \textbf{id} \gtrdot)$

(ii) $\ldots\ \text{BEGIN} \lessdot \text{READ} \doteq (\doteq \langle N_1 \rangle \doteq) \gtrdot ;$

(iii) $\ldots\ \text{BEGIN} \lessdot \langle N_2 \rangle\ ;$



Syntactical Analysis BOTTOM_UP

- **Shift - Reduce Parsing**

- Shift unrecognized tokens onto stack.
- When recognized, Reduce the stack and place non-terminal onto stack.
- Can be applied to a class of grammars known as LR.
  (Left-right scan, Reverse derivation)
- Symbols to be recognized always at top of stack.
- LR(k) Grammars where k is the token lookahead count.

Syntactical Analysis TOP_DOWN

- **Recursive Descent Parsing**
- Procedure for each non-terminal symbol in the Grammar.
- Procedures attempt to find the substring of the input that satisfies the non-terminal symbol.
- May call other procedures, even itself.
- If procedure finds the non-terminal, it advances input token pointer and returns success.
- Otherwise, it returns failure, and / or calls the Error Routine.

ASSIGN procedure

<assign> ::= **id :=** <exp>

```
Procedure ASSIGN
     begin
         FOUND := FALSE
         if TOKEN = id then
              begin
                  get next TOKEN
                  if TOKEN = := then
                       begin
                           get next TOKEN
                           if EXP returns SUCCESS then
                                FOUND := TRUE
                       end if :=
              end if id
         if FOUND = TRUE
              return SUCCESS
         else
              return FAILURE
     end ASSIGN
```

Syntactical Analysis TOP_DOWN

- **Recursive Descent Parsing**
- Grammar needs to be defined to eliminate ‘Left Recursion’

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

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

- Read the next token to determine path among alternatives.
- Some compilers use both TOP_DOWN and BOTTOM_UP parsing.

Revised Grammar

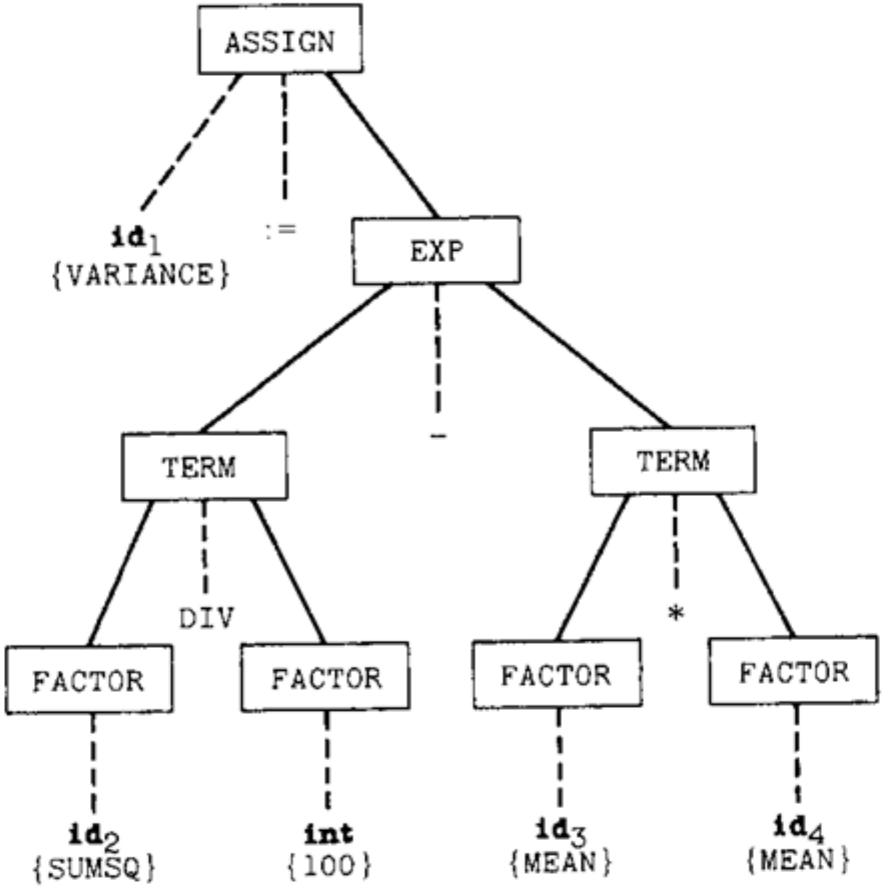
Recursive Descent

- <prog>::= Program <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
- <prog-name> ::= **id**
- **<dec-list>** ::= <dec> { ; <dec> }
- <dec> ::= <id-list> : <type>
- <type> ::= INTEGER
- **<id-list>** ::= **id** { , **id** }
- **<stmt-list>** ::= <stmt> { ; <stmt> }
- <stmt> ::= <assign> | <read> | <write> | <for>
- <assign> ::= i**id** := <exp>
- **<exp>** ::= <term> { + <term> | - <term> }
- **<term>** ::= <factor> { * <factor> | DIV <factor> }
- <factor> ::= **id** | **int** | (<exp>)
- <read> ::= READ (<id-list>)
- <write> ::= WRITE (<id-list>)
- <for> ::= FOR <index-exp> DO <body>
- <index-exp> ::= **id** := <exp> TO <exp>
- <body> ::= <stmt> | begin <stmt-list> END

Top-Down

- Recursive-descent parsing of an ASSIGN statement

```
VARIANCE := SUMSQ DIV 100 - MEAN * MEAN
```



Summary

- Syntax Analysis
  - Parsing
  - Bottom Up
  - Top Down

Next: Semantic Analysis