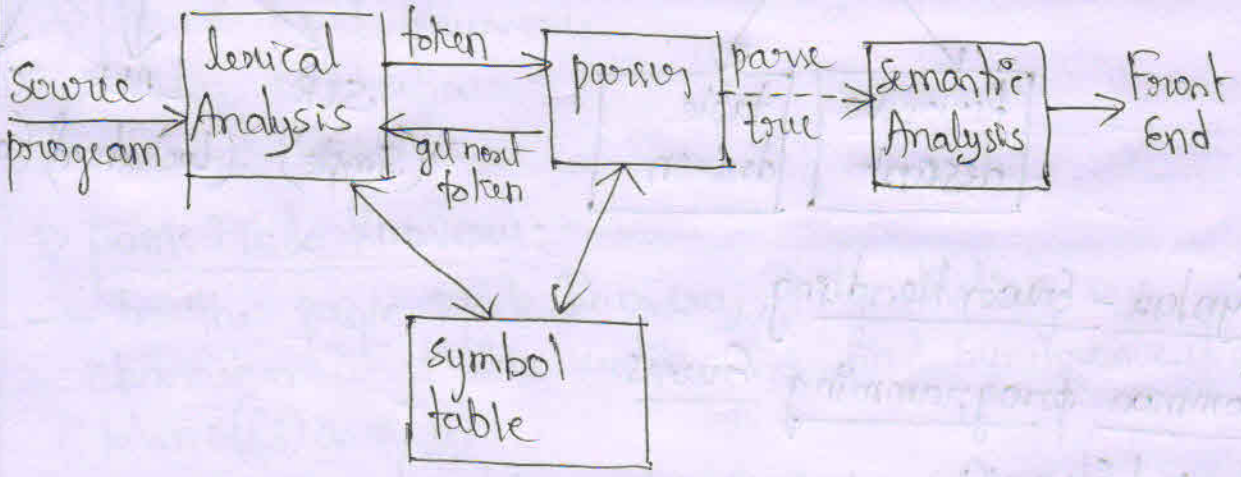
UNIT-2 Syntax Analysis -1

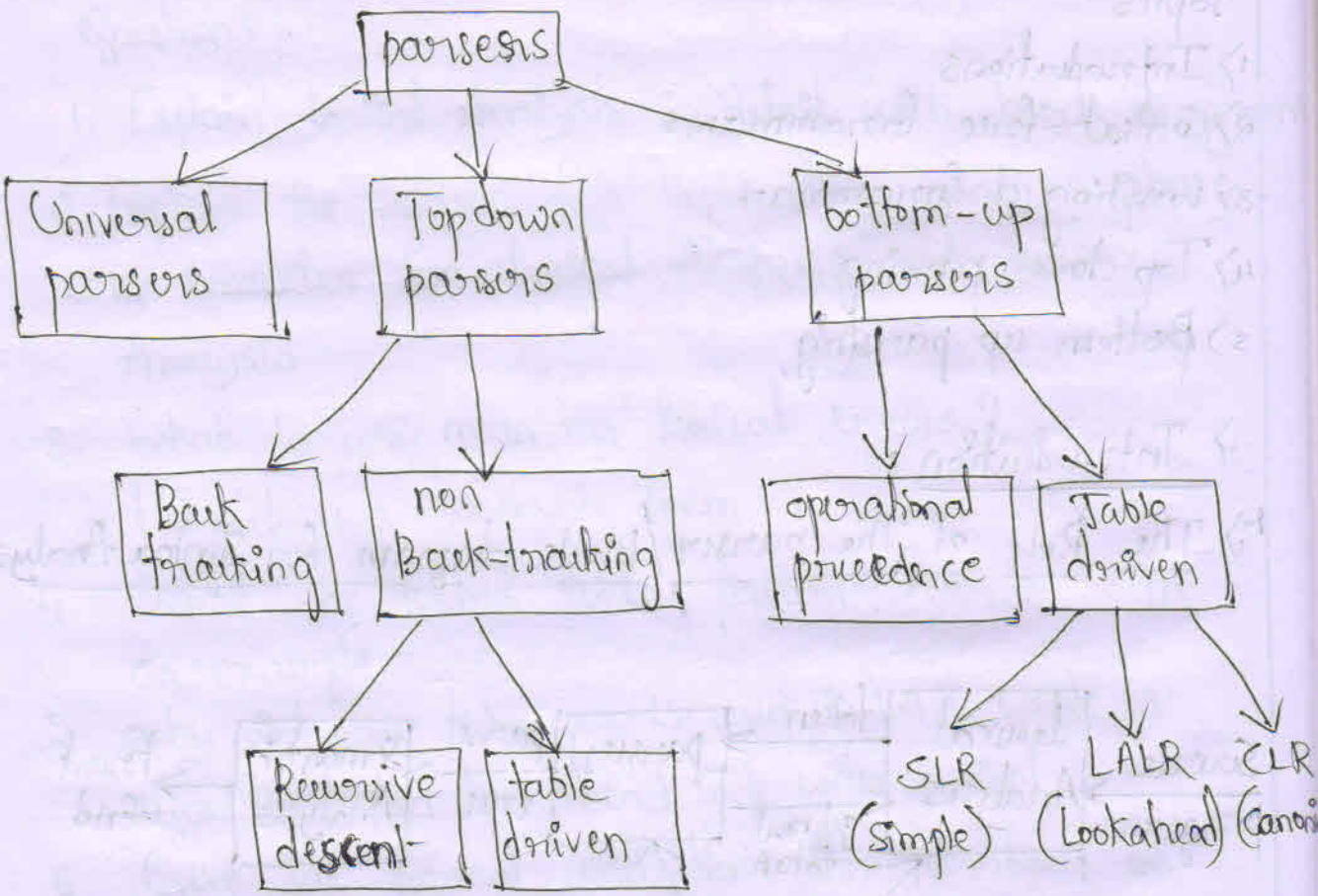
Topics
1) Introductions
2) Context-free Grammars
3) writing a Grammar
4) Top down parsing
5) Bottom up parsing

1) Introduction :

i) The Role of the parser / Block diagram for Syntax Analysis.



The General types of parses for grammers



Syntax - Error Handling

Common programming Errors

i) Lexical Errors:
These include Misspellings of Identifiers, keywords or operators
eg: Use of an identifier elipsesize instead of ellipsesize

ii) Syntactic Errors:
These errors include misplaced Semicolon/Extra or missing braces;

iii) Semantic Errors

These include type mismatches btn operators and operands.

An example: return statement in a Java method with result type Void

iv) Logical Errors:

Can be Anything from incorrect reasoning on the part of the programmer

eg: Using '=' instead of '==' in C programming

Error Recovery Techniques:

i) panic Mode Recovery
ii) phrase level Recovery
iii) Error productions
iv) Global Corrections

i) panic Mode Recovery:

→ In the panic mode Recovery, keep deleting one character at a time until we find Synchronization tokens (';' and '}')

* Synchronization tokens → Semicolon (;)
                         → Epilog (})

eg: int a, ; //Error

ii) phrase level Recovery:

→ It include Insert, delete, update

→ On discovering an error, a parser may perform local correction on the remaining i/p : that is, it may replace a prefix of the remaining i/p by some string that allows the parser to continue

→ It includes → Replacing a comma, by a Semicolon
→ delete an extra Semicolon
→ Inserting a Missing Semicolon

eg: int a, ;
Replace , by ; and delete extra ;

iii) Error productions:

→ By anticipating common errors that might be encountered, we can augment the grammer for the language at hand with productions that generate Incorrect Constructs

→ A parser constructed from a grammer augmented by these error productions detects the anticipated errors when an error productions is used during parsing

iv) Global Corrections:

→ Ideally, we would like a Compiler to make as few changes as possible in processing an incorrect input string. There are algorithm for choosing a minimal sequence of changes to obtain a globally least cost correction.

→ Given an Incorrect input-string x and Grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible

drawback of Global Corrections:

→ These generally too costly to Implement in terms of time and space, so these are Currently only a theoretical Interest.

CONTEXT FREE GRAMMARS

defn: Context free grammer is a 4-tuple defined as (V, T, P, S), where

V: Set of Variable

T: set of Terminals

P: Set of production

S is the start Symbols

Differentiate b/n CFGs and RE

CFGs	RE
1. It is the part of the Syntax Analysis	1. It is the part of the lexical Analysis
2. Usefull for describing nested gramatical structure such as balanced paranthesis and so, on.	2. Usefull for describing the structure of, construct / lexical construct such as Identifiers, keywords etc
3. CFGs are Combined using pushdown automata	3. Regular Expressions are Combined using Finite Automata
4. CFGs Can keep track of no. of Symbols Seen so far	4. RE Cannot keep track of no. of symbols Seen so far
5. Every CFG need not be RE	5. Every RE is a CFG
6. CFGs are more powerfull	6. RE are less powerfull as Compared to CFGs
7. Eg: letter → [A-Z a-z _] digit → [0-9] id → letter (letter / digit)	7. Eg: [a-z A-Z 0-9][0-9]*

Q) For the following CFG

a. Give the LMD for the string
b. Give the RMD for the string
c. Give the parse tree for the string
d. Is the grammar ambigous / Unambigous? Justify

1. $S \rightarrow SS+ \mid SS* \mid a \Rightarrow aa+a*$
2. $S \rightarrow 0S1 \mid 01 \Rightarrow 000111$
3. $S \rightarrow +SS \mid *SS \mid a \Rightarrow +*aaa$
4. $S \rightarrow S(S)S \mid \epsilon \Rightarrow (()())$
5. $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a \Rightarrow (a+a)*a$
6. $S \rightarrow (L) \mid a \quad L \rightarrow L,S \mid S \Rightarrow (a,a)$
7. $S \rightarrow asbs \mid bsas \mid \varepsilon \Rightarrow aabbab$
8. $bexpr \rightarrow bexpr$ or $bterm \mid bterm$

   $bterm \rightarrow bterm$ and $bfaulon \mid bfaulon$

   $bfaulon \rightarrow$ not $bfaulon \mid (bexpr) \mid true \mid false$

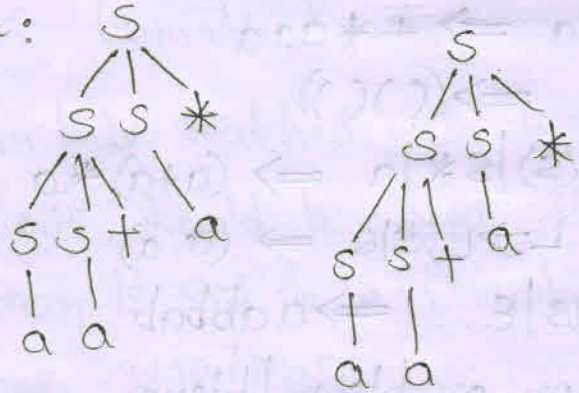
   $\Rightarrow$ not $(true \mid false)$
9. $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id \Rightarrow id+id+id$
10. $S \rightarrow iEtS \mid iEtSeS \mid a \Rightarrow$ If E_1 then If E_2 then S_1 else S_2
11. $R \rightarrow R'|'R \mid RR \mid R* \mid (R) \mid a \mid b \mid c \Rightarrow a|b*c$

1) $S \to SS+ \mid SS* \mid a \Rightarrow aa+a*$

$S \xrightarrow{lm} SS*$
$\Rightarrow SS+S*$
$\Rightarrow as+S*$
$\Rightarrow aa+S*$
$\Rightarrow aa+a*$

$S \xrightarrow{rm} SS*$
$\Rightarrow Sa*$
$\Rightarrow SS+a*$
$\Rightarrow Sa+a*$
$\Rightarrow aa+a*$

parse tree:



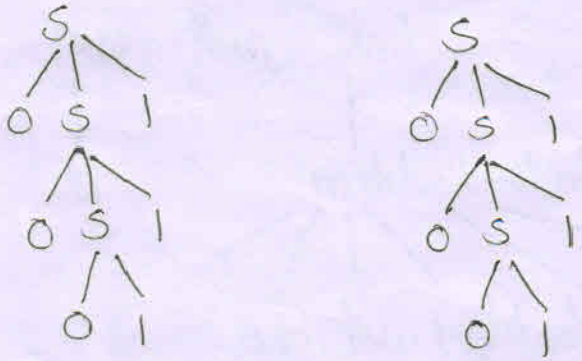
The Grammar Is Unambigous because It has only one LMD and one RMD

2) $S \to 0S1 \mid 01 \Rightarrow 000111$

$S \xrightarrow{lm} 0S1$
$\Rightarrow 00S11$
$\Rightarrow 000111$

$S \xrightarrow{rm} 0S1$
$\Rightarrow 00S11$
$\Rightarrow 000111$

parse tree:

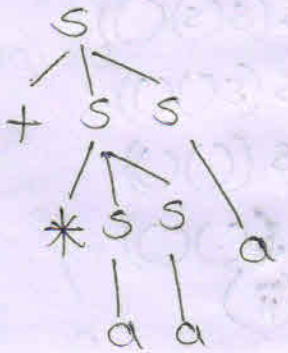


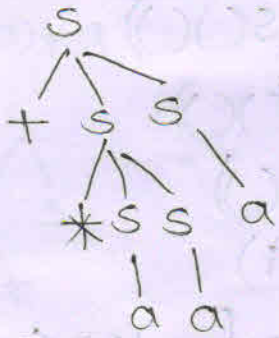
The Grammar Is Unambigous because It has only one LMD and Only one RMD

3) 3) $S \to +SS \mid *SS \mid a \Rightarrow +*aaa$

$S \xrightarrow{lm} +SS$
$\xrightarrow{lm} +*SSS$
$\Rightarrow +*aSS$
$\Rightarrow +*aaS$
$\Rightarrow +*aaa$

$S \xrightarrow{rm} +SS$
$\xrightarrow{rm} +*SSS$
$\Rightarrow +*SSa$
$\Rightarrow +*Saa$
$\Rightarrow +*aaa$





The Grammar is Unambigous because it has only one LMD and one RMD

4) $S \to S(S)S \mid \varepsilon \Rightarrow (()())$

$S \xrightarrow{lm} S(S)S$
$\Rightarrow (S)S$
$\Rightarrow (S(S)S)S$
$\Rightarrow ((S)S)S$
$\Rightarrow (()S)S$
$\Rightarrow (()(S)S)$
$\Rightarrow (()()S)$
$\Rightarrow (()())$
(i)

$S \xrightarrow{rm} S(S)S$
$\xrightarrow{lm} (S)S$
$\xrightarrow{lm} (S(S)S)S$
$\xrightarrow{lm} (S(S)S(S)S)S$
$\Rightarrow ((S)S(S)S)S$
$\Rightarrow (()S(S)S)S$
$\Rightarrow (()(S)S)S$
$\Rightarrow (()()S)S$
$\Rightarrow (()())S$
$\Rightarrow (()())$
(ii)

RMD:

$S \xrightarrow{rm} S(S)S$

$\Rightarrow S(S)$

$\Rightarrow S(S(S)S)$

$\Rightarrow S(S(S)S(S))$

$\Rightarrow S(S(S)S())$

$\Rightarrow S(S(S)())$

$\Rightarrow S(S()())$

$\Rightarrow S(()())$

$\Rightarrow (()())$

(i)

$S \xrightarrow{rm} S(S)S$

$\Rightarrow S(S)$

$\Rightarrow S(S(S)S)$

$\Rightarrow S(S(S))$

$\Rightarrow S(S())$

$\Rightarrow S(S(S)S())$

$\Rightarrow S(S(S)())$

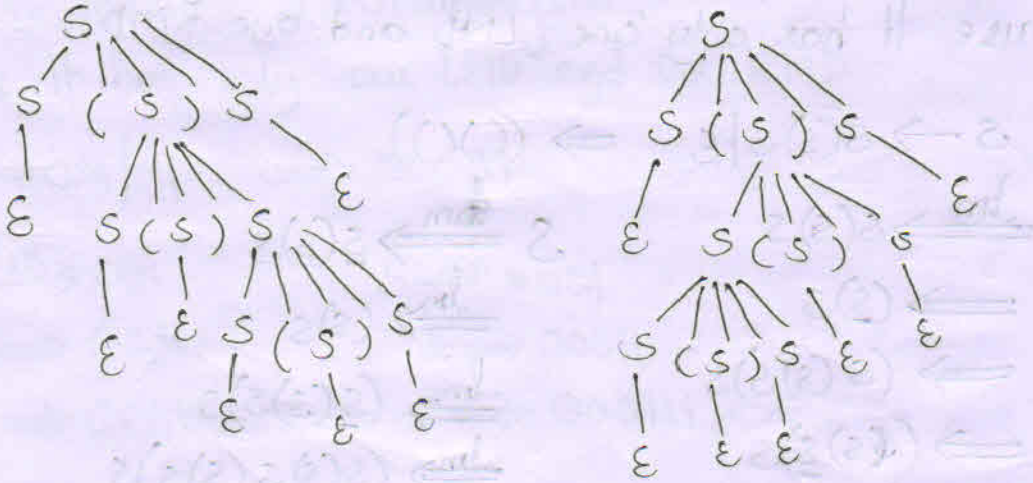
$\Rightarrow S(S()())$

$\Rightarrow S(()())$

$\Rightarrow (()())$

(ii)

parse tree for LMD (i) and (ii)

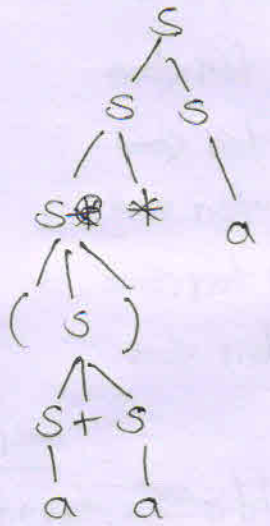


The Grammar is ambigous
Since it has 2 LMD and 2 RMD

5) $S \rightarrow S+S|SS|(S)|S*|a \Rightarrow (a+a)*a$

$S \xrightarrow{lm} SS$
$\Rightarrow S*S$
$\Rightarrow (S)*S$
$\Rightarrow (S+S)*S$
$\Rightarrow (a+S)*S$
$\Rightarrow (a+a)*S$
$\Rightarrow (a+a)*a$

$S \xrightarrow{rm} SS$
$\Rightarrow S*S$
$\Rightarrow S*a$
$\Rightarrow (S)*a$
$\Rightarrow (S+S)*a$
$\Rightarrow (S+a)*a$
$\Rightarrow (a+a)*a$

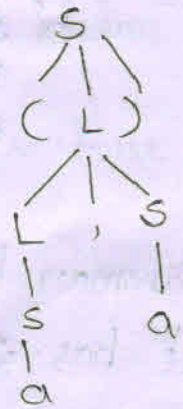


The Grammar is Unambigous because it has only one RMD and LMD

6) $S \rightarrow (L)|a$
$L \rightarrow L,S|S \Rightarrow (a,a)$

$S \xrightarrow{lm} (L)$
$\Rightarrow (L,S)$
$\Rightarrow (S,S)$
$\Rightarrow (a,S)$
$\Rightarrow (a,a)$

$S \xrightarrow{rm} (L)$
$\Rightarrow (L,S)$
$\Rightarrow (L,a)$
$\Rightarrow (S,a)$
$\Rightarrow (a,a)$



The Grammar is Unambigous because it has only one LMD and RMD

5) $S \rightarrow S+S|SS|(S)|S*|a \Rightarrow (a+a)*a$

$S \xrightarrow{lm} SS$

$\Rightarrow S*S$

$\Rightarrow (S)*S$

$\Rightarrow (S+S)*S$

$\Rightarrow (a+S)*S$

$\Rightarrow (a+a)*S$

$\Rightarrow (a+a)*a$

$S \xrightarrow{rm} SS$

$\Rightarrow S*S$

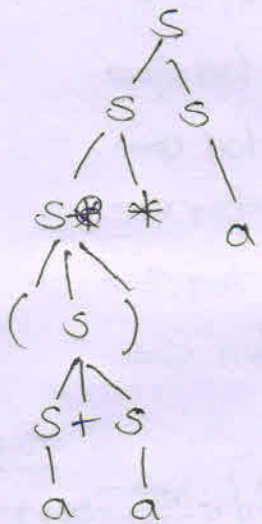
$\Rightarrow S*a$

$\Rightarrow (S)*a$

$\Rightarrow (S+S)*a$

$\Rightarrow (S+a)*a$

$\Rightarrow (a+a)*a$



The Grammar is Unambigous because it has only one RMD and LMD

6) $S \rightarrow (L)|a$

$L \rightarrow L,S|S \Rightarrow (a,a)$

$S \xrightarrow{lm} (L)$

$\Rightarrow (L,S)$

$\Rightarrow (S,S)$

$\Rightarrow (a,S)$

$\Rightarrow (a,a)$

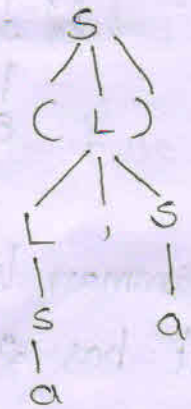
$S \xrightarrow{rm} (L)$

$\Rightarrow (L,S)$

$\Rightarrow (L,a)$

$\Rightarrow (S,a)$

$\Rightarrow (a,a)$



The Grammar is Unambigous because it has only one LMD and RMD

7) $S \rightarrow asbs \mid bsas \mid \varepsilon \Rightarrow aabbab$

LMD:

$S \xrightarrow{lm} asbs$
$\Rightarrow aasbsbs$
$\Rightarrow aabsbs$
$\Rightarrow aabbsasbs$
$\Rightarrow aabbasbs$
$\Rightarrow aabbabs$
$\Rightarrow aabbab$

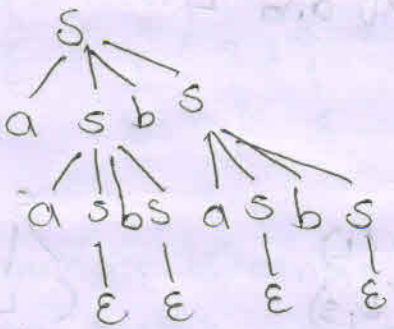
$S \xrightarrow{lm} asbs$
$\Rightarrow aasbsbs$
$\Rightarrow aabsbs$
$\Rightarrow aabbasbs$
$\Rightarrow aabbabs$
$\Rightarrow aabbab$

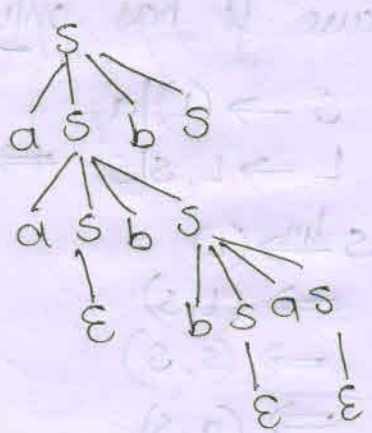
RMD:

$S \xrightarrow{rm} asbs$
$\Rightarrow asb$
$\Rightarrow aasbsb$
$\Rightarrow aasbbsasb$
$\Rightarrow aasbbsab$
$\Rightarrow aasbbab$
$\Rightarrow aabbab$

$S \xrightarrow{rm} asbs$
$\Rightarrow asbasbs$
$\Rightarrow asbasb$
$\Rightarrow asbab$
$\Rightarrow aasbsbab$
$\Rightarrow aasbbab$
$\Rightarrow aabbab$

parse tree:



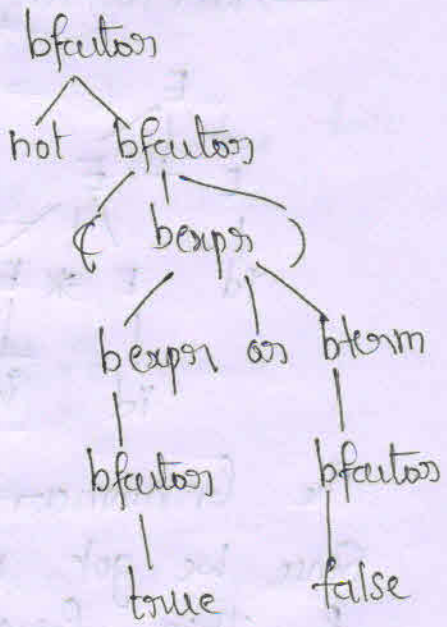


The grammar is ambigous
Since it has 2 LMD and 2RMD

8) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false
not (true or false)

LMD:
bexpr $\xrightarrow{lm}$ bterm
$\Rightarrow$ bfactor
$\Rightarrow$ not bfactor
$\Rightarrow$ not (bexpr)
$\Rightarrow$ not (bexpr or bterm)
$\Rightarrow$ not (bterm or bterm)
$\Rightarrow$ not (bfactor or bterm)
$\Rightarrow$ not (true or bterm)
$\Rightarrow$ not (true or bfactor)
$\Rightarrow$ not (true or false)

RMD:
bexpr $\xrightarrow{rm}$ bterm
$\Rightarrow$ bfactor
$\Rightarrow$ not bfactor
$\Rightarrow$ not (bexpr)
$\Rightarrow$ not (bexpr or bterm)
$\Rightarrow$ not (bexpr or bfactor)
$\Rightarrow$ not (bexpr or false)
$\Rightarrow$ not (bterm or false)
$\Rightarrow$ not (bfactor or false)
$\Rightarrow$ not (true or false)



9) $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$

$\Rightarrow id + id * id$

LMD

$E \xrightarrow{lm} E+E \quad (E \rightarrow E+E)$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

$E \xrightarrow{lm} E*E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

RMD

$E \xrightarrow{rm} E+E$

$\Rightarrow E+E*E$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

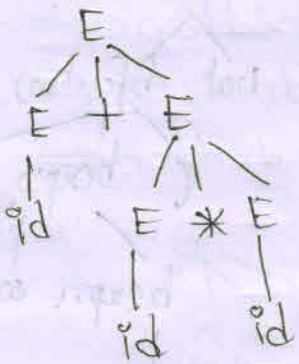
$E \xrightarrow{rm} E*E$

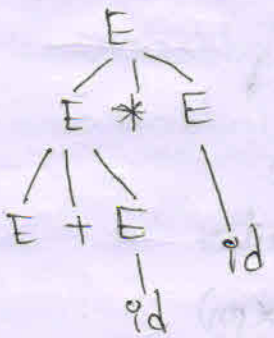
$\Rightarrow E*id$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$





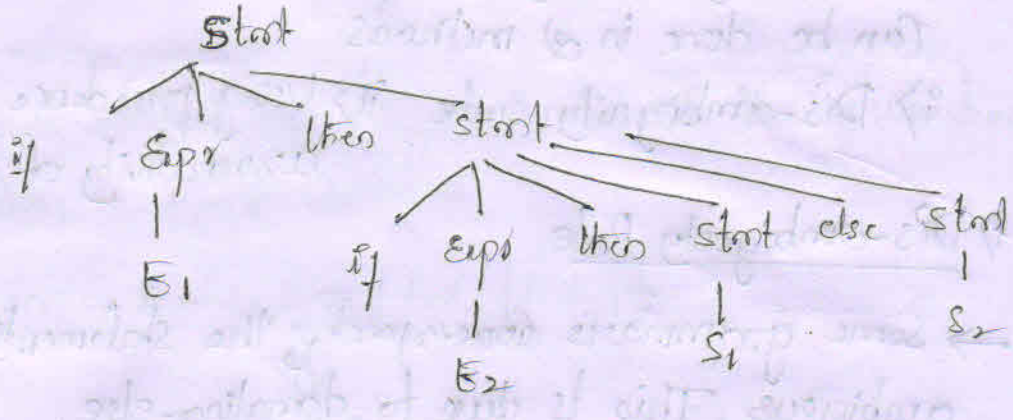
The Grammar is Ambigous

Since we got more than 1 LMD and more than 1 RMD for this Grammar

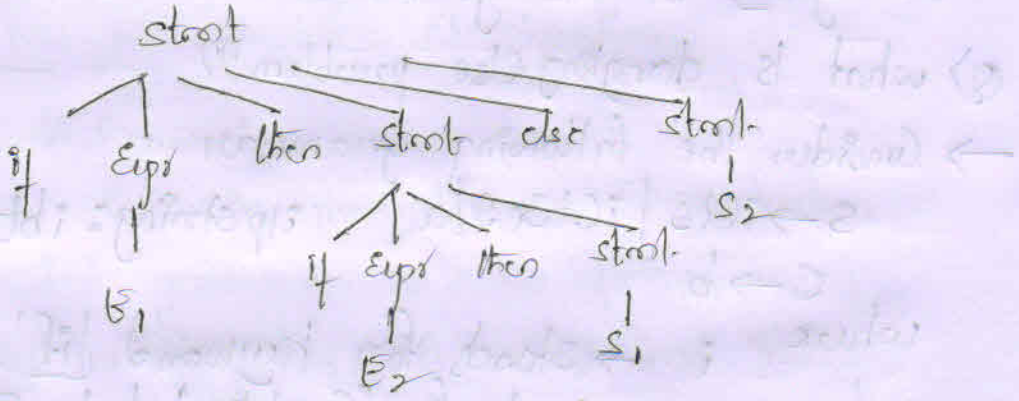
10. $S \rightarrow iEtS \mid iEtSeS \mid a \qquad E \rightarrow b$

$\Rightarrow$ If E_1 Then If E_2 Then S_1 Else S_2

1)



2)



The given grammar is ambiguous ∵ it has two different parse trees

ii) $R \rightarrow R|'R|RR|R*|(R)|a|b|c \Rightarrow a|b*c$

Lmd 1

$R \rightarrow R|R$
$\overset{lm}{\Rightarrow} a|R$
$\overset{lm}{\Rightarrow} a|RR$
$\overset{lm}{\Rightarrow} a|R*R$
$\overset{lm}{\Rightarrow} a|b*R$
$\Rightarrow a|b*c$

Lmd 2

$R \rightarrow RR$
$\overset{lm}{\Rightarrow} R|RR$
$\Rightarrow a|RR$
$\Rightarrow a|R*R$
$\Rightarrow a|b*R$
$\Rightarrow a|b*c$

The given grammar is ambiguous ∵ it has Lmds.

Eliminating ambiguity

Can be done in 2 methods

i) Dis-ambiguity rule ii) Using precedence & associativity of operators

i) Dis-ambiguity Rule

→ Some grammars corresponding the statements are ambigous, This is due to dangling-else.

The dangling else problem can be eliminated and thus ambiguity of the grammar can also be eliminated

Q) what is dangling else problem??

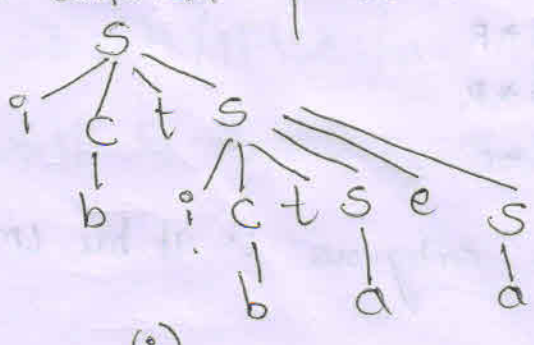
→ Consider the following grammar

$S \rightarrow iCts \mid iCtSeS \mid a$ i/p string: ibtibtaea

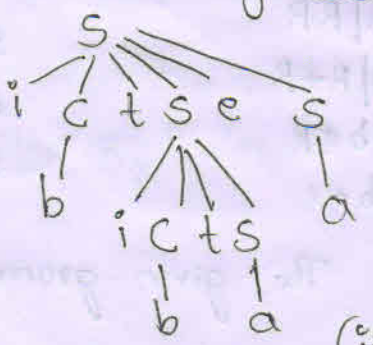
$C \rightarrow b$

where
- i → stands for keyword 'if'
- C → stands for 'Condition' to be satisfied, and C is nonterminal
- t → stands for keyword 'then'
- s → stands for 'Statement' for nonterminal
- e → stands for keyword 'else'
- a → stands for other statement
- b → stands for other statement

Since the above grammar is ambigous, we get two different parse tree for the string ibtibtaea



(i)



(ii)

Since there are 2 parse tree for the same string ibtibtaea the given grammar is ambigous.
observe the following points
→ The first parse tree associates else with 2nd statement
→ The second parse tree associates else with first if stmt
The ambiguity wheather to associate else with first if statement / second if-statements is called dangling-else problem.

Eg) Q) Eliminate ambiguity from the following ambigous grammar:

$S \rightarrow iCts \mid iCtSes \mid a$

$C \rightarrow b$

→ In all programming languages when if-statements are nested, the first parse tree is preferred. So, the general rule is "Match each else with closest unmatched then". This rule can be directly incorporated into grammar and ambiguity can be eliminated as shown below:

Stp1) The matched stmt M is an if-else statement where the statement S before else and after else keyword is matched. This can be expressed as:

$M \rightarrow iCtMeM$

Stp 2): An Unmatched statement U is the one consisting of:

a) Simple if-statement where the statement S is matched statement / Unmatched statement. ∴ The equivalent production is ⇒ $U \rightarrow iCtS$

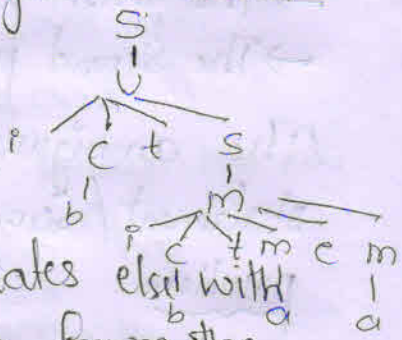
b) If-else statement where the statement before else is matched and statement after else is Unmatched.
∴ The equivalent production is: $U \rightarrow iCtMeU$

Step 3: The matched statement M and Un-matched statement U can obtained using the statement S as shown below:

$S \rightarrow M/U$

So, the final grammar which is un-ambigous is shown below:

$S \rightarrow M/U$
$M \rightarrow iCtMeM/a$
$U \rightarrow iCtS$
$U \rightarrow iCtMeU \quad c \rightarrow b$



observe that the above grammar associates else with closest then and eliminates ambiguity from the grammar

Eliminating ambiguity using precedence and Associativity

This method is explained using the following example:

eg: 8) Convert the ambigous grammar into Unambigous grammar:

$E \rightarrow E*E|E-E$
$E \rightarrow E\wedge E|E/E$
$E \rightarrow E+E$
$E \rightarrow (E)|id$

The grammar can be converted into unambigous grammar using the precednce of operators as ~~shown~~ well as associativity operators as shown below:

Step 1: Arrange the operators in increasing order of the precedence along with the associativity as shown below:

operators	Associativity	non-terminal used
+, −	LEFT	E
*, /	LEFT	T
^	RIGHT	P

Since there are three levels of precedence, we associate three non-terminals : E, T and P. Also an extra non-terminal F, generating basic units in an arithmatic expression

Step 2 : The basic units in expression are id (identifier) and paranthesized expressions, the production corresponding to this can be written as:

$$F \rightarrow (E) | id$$

Step 3: The next highest priority operator is ^ and it is right associative. So, the production must start from the non-terminal P and it should have right recursion as shown below:

$$P \rightarrow F \wedge P | F$$

Step 4: The next highest priority operators are * and / and they are left associative, So, the production must start from the non-terminal T and it should have left recursion as shown below:

$$T \rightarrow T * P | T / P | P$$

Step 5: The next highest priority operators are + and - and they are left associativity. So, the production must start from the non-terminal E and it should have left recursion as shown below:

$$E \rightarrow E + T | E - T | T$$

Step 6 : The final grammar which is unambigous grammar can be written as shown below:

$$E \rightarrow E + T | E - T | T$$
$$T \rightarrow T * P | T / P | P$$
$$P \rightarrow F \wedge P | F$$
$$F \rightarrow (E) | id$$

8) Convert the following Ambigous grammar into Unambigous grammar by considering * and − operators lowest priority and they are left associative, / and + operators have the highest priority and are right associative and ^ operator have the highest priority and are right associativity and ^ operator has preedence in between and It is left associativity.

$E \to E+E$
$E \to E-E$
$E \to E \wedge E$
$E \to E*E$
$E \to E/E$
$E \to (E) \mid id$

→ The grammar can be converted into unambigous grammar using the preedence of operators as well as associativity operators as shown below:

step1: Arrange the operators in increasing order of the preceedence along with associativity as shown below:

preedree	operators	Associativity	non-terminal used
(lowest)	*, −	LEFT	E
	^	LEFT	P
(highest)	/, +	RIGHT	T

Since there are three levels of preedree we associate three non-terminals: E, P and T. Also use an extra non-terminal F generating basic units in an arithmatic expression

step2: The basic units in expression are id (identifier) and parenthesized expressions. The production corresponding to this can be written as:

$F \to (E) \mid id$

Step 3: The next highest priority operators are + and / they are right associative, So, the production must start from the non-terminal T and it should be right recursive in RHS of the production as shown below:

$T \rightarrow F+T \mid F/T \mid F$

Step 4: The next highest priority operator is ^ and it is left associative. So, the production must start from the non-terminal P and it should be left recursive in RHS of the production as shown below:

$P \rightarrow P \wedge T \mid T$

Step 5: The next highest priority operators are * and - and they are left associative. So, the production must start from the non-terminal E and it should be left recursive in RHS of the production as shown below:

~~$P \rightarrow P \wedge T \mid T$~~ $E \rightarrow E+P \mid E-P \mid P$

~~Step 6: The next highest priority operators are * and - and they~~

Step 6: The final grammar which is unambiguous can be written as shown below:

$E \rightarrow E+P \mid E-P \mid P$

$P \rightarrow P \wedge T \mid T$

$T \rightarrow F+T \mid F/T \mid F$

$F \rightarrow (E) / id$

Q) Define Ambiguity ? Show that the following grammar is ambigous

$R \rightarrow R'|'R \mid RR \mid R* \mid (R) \mid a \mid b \mid c$ for input string $a|b*c$

Give an unambigous grammar for the above grammar such that precedence order from lowest to highest are Concatenation, *, |, (), identifier and all are left to right associativity

Ans: The grammar is said to be ambigous if it has more than one LMD / more than one RMD

(or)

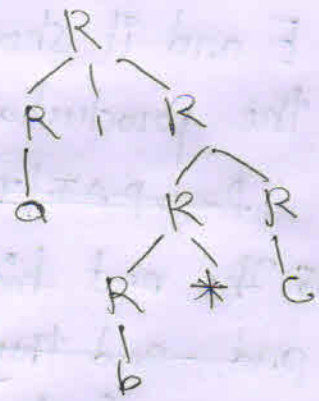
If there are two different parse trees for the input string by applying LMD / by applying RMD

→ i/p string: a|b*c

LMD 1	LMD 2
$R \rightarrow R'|'R$	$R \rightarrow RR$
$\Rightarrow a'|'R$	$\Rightarrow R'|'RR$
$\Rightarrow a'|'RR$	$\Rightarrow a'|'RR$
$\Rightarrow a'|'R*R$	$\Rightarrow a'|'R*R$
$\Rightarrow a'|'b*R$	$\Rightarrow a|b*R$
$\Rightarrow a'|'b*C$	$\Rightarrow a|b*C$

It has two LMD's ∴ the given grammar is Unambigous

parse tree:



2)

→ Unambigous grammar

1. Arrange the operators in the ascending order with the precedence and associativity

operators	Associativity	non-terminal used
•	LEFT	R
*	LEFT	S
|	LEFT	T

2. The basic units in expression are (R) and a, b, c we use additional non-terminal U for generating those $U \rightarrow (R) | a | b | c$

3. The next highest priority operator | and it is left associative. So the production must start from the non-terminal T and it must be left recursive is RHS of the production

$$T \rightarrow T'|'U|U$$

4. The next highest priority operator is * and is left associative. So the production must start from non-terminal S and it is a unary operator

$$S \rightarrow T*|T$$

5. The next highest priority operator is Concatenation. and is left associative. So the production must start from the non-terminal R and it should have left recursion as

$$R \rightarrow RS|S$$

Step 6: The final grammar which is unambigous can be written as

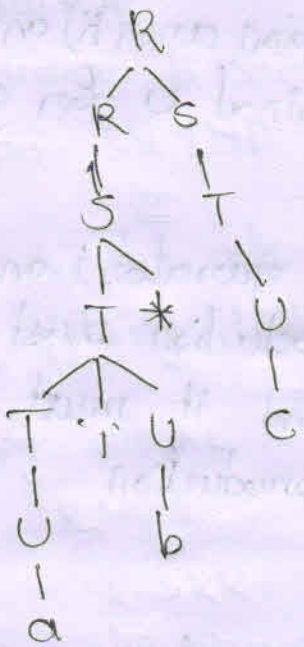
$R \rightarrow RS|S$

$S \rightarrow T*|T$

$T \rightarrow T'|'U|U$

$U \rightarrow (R)|a|b|c$

The parse tree for the i/p a|b*c is



Left Recursion:

defn:

If Non-terminal Symbol and the 1st Symbol of the production are same, then it is left recursion.

General form: $A \longrightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 / \beta$

$$\Downarrow$$

$$A \longrightarrow \beta A'$$
$$A' \longrightarrow \alpha_1 A' / \alpha_2 A' / \varepsilon$$

Algorithm for left Recursion Elimination

Algorithm Left_Recursion

i/p: Grammar G_r with no cycles or ε-production

o/p: An equivalent grammar with no left-recursion

Method: Apply the algorithm to G_r. Note that the resulting non-left-recursive grammar may have ε-productions.

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$
2. For(each i from 1 to n){
3. for (each j from 1 to i-1){
4. replace each production of the form $A_i \longrightarrow A_j\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma / \delta_2\gamma / \ldots / \delta_k\gamma$ where $A_j \longrightarrow \delta_1 / \delta_2 / \ldots / \delta_k$ are all current A_j-productions
5. }
6. Eliminate the Immediate left recursion among the A_i-production
7. }

Example on Removing / Eliminating left Recursion

1. $E \rightarrow \underbrace{E}_{A} \underbrace{+T}_{\alpha} | \underbrace{T}_{\beta}$

$\Downarrow$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$ ← Epsilon

(here $E \rightarrow E+T | T$ (both are Same) ∴ The grammar Contains left recursion)

2. $T \rightarrow \underbrace{T}_{A} \underbrace{*F}_{\alpha} | \underbrace{F}_{\beta}$ $\Rightarrow$ $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' / \varepsilon$

$\Downarrow$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

3. $S \rightarrow \underbrace{S}_{A} \underbrace{(S)S}_{\alpha} | \underbrace{\varepsilon}_{\beta}$

$\Downarrow$

$S \rightarrow S'$

$S' \rightarrow (S)SS' / \varepsilon$

4. $S \rightarrow Ss+ | ss* | a$

$\Downarrow$

$S \rightarrow aS'$

$S' \rightarrow S+S' | S*S' | \varepsilon$

5. $E \rightarrow E+T | T$

$T \rightarrow T*F | F$

$F \rightarrow (E) / id$

$\Downarrow$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) / id$

Left factoring

General form: $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 \text{ --- } / \alpha\beta_n / \gamma$

$$\Downarrow$$

$A \rightarrow \alpha A' / \gamma$

$A' \rightarrow \beta_1 / \beta_2 \text{ --- } / \beta_n$

Algorithm for left factoring

Algorithm left_factoring

i/p: Grammar G

o/p: An equivalent left-factored grammar

method: For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

If $\alpha \neq \varepsilon$ – i,e there is a nontrivial common prefix – replace all of the A-productions

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \text{ --- } / \alpha\beta_n / \gamma$,

where γ represents all alternatives that do not begin with α, by

$A \rightarrow \alpha A' / \gamma$

$A' \rightarrow \beta_1 / \beta_2 / \text{ --- } / \beta_n$

Here A' is new non-terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix

Examples on left factoring

1. $S \to \underbrace{SS+}_{\alpha\ \beta_1} \mid \underbrace{SS*}_{\alpha\ \beta_2} \mid \underbrace{a}_{\gamma}$

$\Downarrow$

$S \to SSS' \mid a$

$S' \to + \mid *$

> The common terminals we have to take as α and the remaining term we have to take as β_1, β_2 -- So on in each production

2. ~~$S \to 0S' \mid \epsilon$~~ ~~$S' \to 1$~~

$S \to \underbrace{0S1}_{\alpha\ \beta_1} \mid \underbrace{01}_{\alpha\ \beta_2}$

$\Downarrow$

$S \to 0S' \mid \epsilon$

$S' \to S1 \mid 1$

3. $S \to iEts \mid iEtses \mid a$

$E \to b$

$\Downarrow$

$S \to \underbrace{iEts}_{\alpha\ \beta_1} \mid \underbrace{iEts}_{\alpha}\underbrace{es}_{\beta_2} \mid \underbrace{a}_{\gamma}$

$\Downarrow$

$S \to iEtss' \mid a$

$S' \to \epsilon \mid es$

$E \to b$

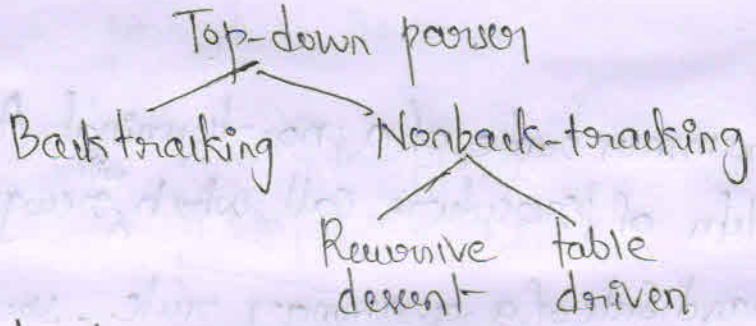
> Since β_1 is empty we'll take β_1 as ϵ

Top down parsers :

→ dfn: Is a parser of an i/p string of token by tracing out the steps in a left most derivation, it denies the string from the start symbol

→ It is termed as topdown because the parse tree is traversed in a preorder way that is from the root node to the leaf node

→ It has various types.

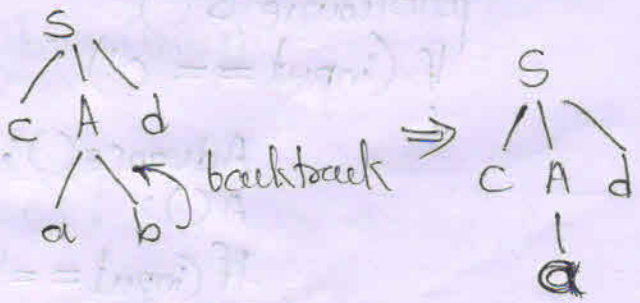


i) Backtracking

→ Backtracking tries different possibilities for parsing an i/p string by backing up an arbitrary amount in the i/p if any possibilities fails

→ These are more powerfull but very much slower as they require exponential time to parse, hence they're not ~~available~~ suitable for practical compilers

eg: S → CAd
A → abla
i/p string → Cabd
i/p string → Cad



ii) Non-backtracking parsers:

i) Recursive descent ii) Table driven

i) Recursive descent parser:

→ Recursive descent parsers are more Versatile & Suitable for handwritten parsers

→ It helps to study the method for parsing and serves as basis for topdown parsers

S → cAd

A → ab | a

here, the grammar rule of a non-terminal A is give as a defn of procedure call which will recognize A

a) The right hand side of a grammar rule, specifies the structure of the code for the procedure.

b) The sequence of terminals on the right hand side corresponds to the i/p matches while the sequence of non-terminals are calls with the corresponding procedure

NT = {S, A}

T = {a, b, c, d}

```
procedure S()
If (input == 'c')
{
        Advance();
        A();
        If (input == 'a')
        {
```

```
        Advance();
        return (true);
    }
    else
    {
        return(false); }
    }
    else {
        return(false); }
    }

procedure A()
{
    isave = in-ptr;
    If (input == 'a')
    {
        Advance();
        If (input == 'b')
        {
            Advance();
            return (true);
        }
        else {
            in-ptr = isave
            If (input == 'a')
            {
                Advance();
                return (true);
            }
        }
        return (false);
    }
```

isave:

saves the ilp pointer position before each alternate production to facilitate backtracking whenever a terminal is encountered the ilp pointer

Advances the next position if alternate phase the in pointer retraces to the previous position to trace the next alternate

Advance():

advance is a procedure that is written to advance the ilp pointer to the next position on a successfull Completion of the parsing action the parser returns a true value

drawbacks of Recursive descent parsings

1. left recursion → It has the ~~procedure~~ production of the form $A \rightarrow A\alpha$

the parsers goes into infinite loop

eg: A → Abla

ilp → abb

The device string abb there is an ambiguity as to how many times the nonterminals has to be expanded

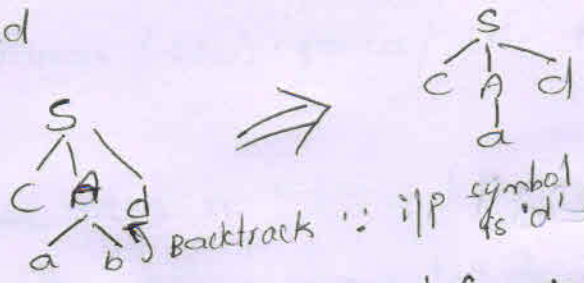
2. Backtracking: It occurs where there is more than one alternate in the production to be tried while parsing the ilp string

S → cAd

A → abla ilp: cabd

no backtr

i/p string : cad



:: i/p symbol is 'd' but ptr is pointing to b

3. It is very difficult to identify the posn of the errors

Example: Recursive descent-

Q) write a recursive descent parser for the following grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

i/p: id+id*id

→ → procedure E()

```
{ if (input=='T')
    T();
  Eprime();
}
procedure T()
{ F();
  Tprime();
procedure F()
{ if(input=="(")
  { Advance();
    E();
    if (input==')')
      return(True);
    else
      return(False);
  }
```

```
elseif(input == "id")
{
    Advance();
    return(True);
}
else
    return(False);
}
procedure Eprime()
{
    If(input == "+")
    {
        Advance();
        T();
        Eprime();
        return(True);
    }
    return(False);
}
procedure Tprime()
{
    If(input == "*")
    {
        Advance();
        F();
        Tprime();
        return(True);
    }
    else
        return(False);
}
```

ii) Table driven:

→ Table driven is also called as predictive parsing

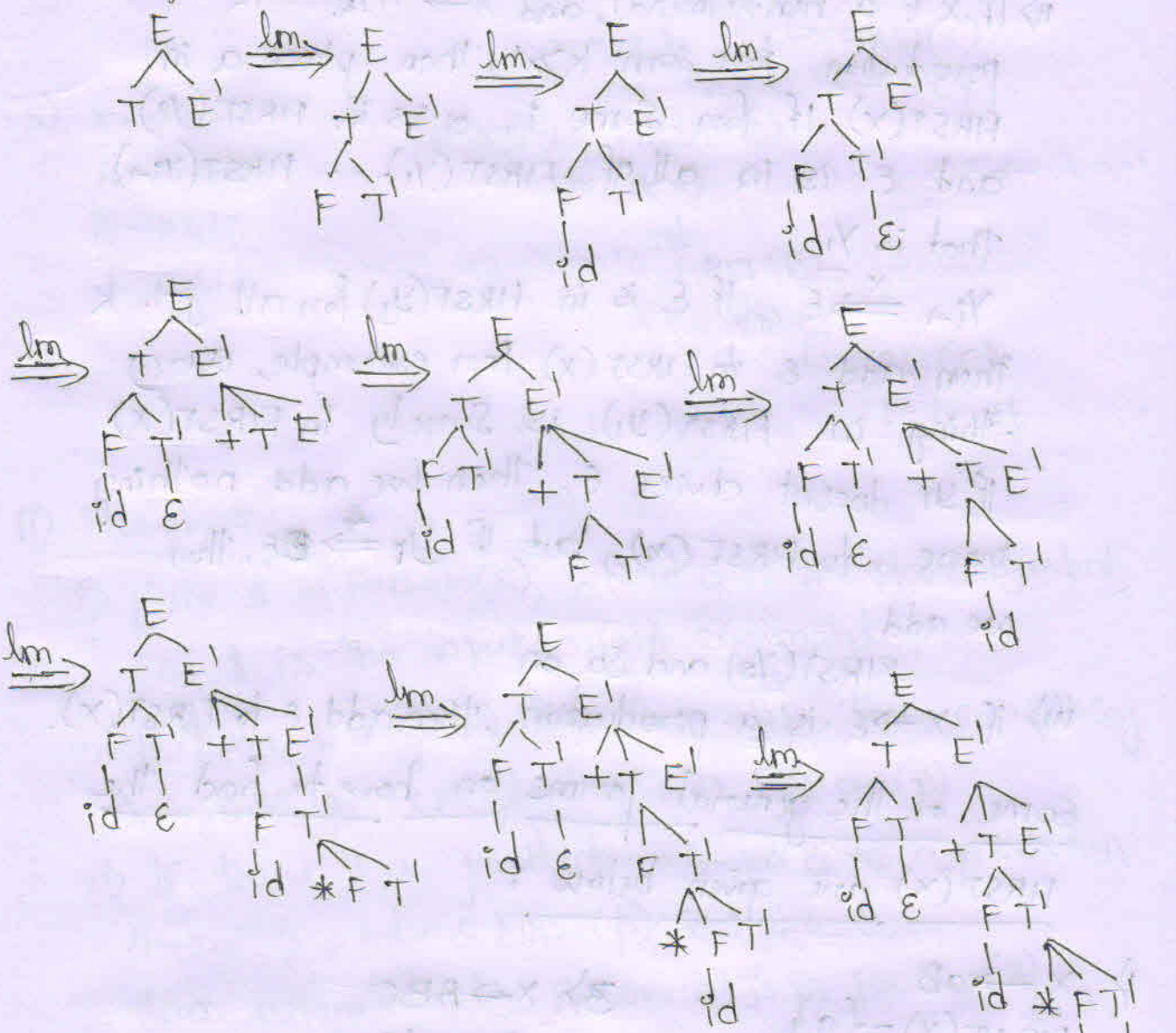
→ predictive parser is a recursive decent parser, which ~~production~~ has the capability to predict which production is to be used to replace the i/p string

→ The predictive parser does not suffer from backtracking

Predictive Parsers / LL(1) parsers / Table driven parsers.

Q) Why do we need a FIRST and FOLLOW set

Consider the below given top down approach for the example:



here, T' has two options i.e $T' \to *FT'$ and $T' \to \varepsilon$ we are in ambiguity situation to replace with which RHS of the production.

If we have n alternate productions for the same Non-terminal then we have to backtrack (n-1) times

So, It is very tedious process,

So, we find FIRST & FOLLOW set.

FIRST AND FOLLOW SETS:

i) FIRST:

i) If X is a terminal, then FIRST(X) = {X}

ii) If X is a nonterminal, and $X \to Y_1 Y_2 \dots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1) --- FIRST(Y_{i-1}) that is Y_1

$Y_{i-1} \xRightarrow{x} \varepsilon$, If ε is in FIRST(Y_j) for all $j = 1 \dots k$ then add ε to FIRST(X), For example, everything in FIRST(Y_1) is surely in FIRST(X) If Y_1 doesnt device ε, then we add nothing more to FIRST(X). but if $Y_1 \xRightarrow{*} \varepsilon$, then we add

FIRST(Y_2) and so on

iii) If $X \to \varepsilon$ is a production, then add ε to FIRST(X).

Some of the general forms on how to find the FIRST(X) are given below:

1) $X \to aB$

FIRST(X) = {a}

$X \to \varepsilon$

FIRST(X) = ε

2) $X \to ABC$

$A \to a$

$B \to b$

$C \to c$

FIRST(X) = FIRST(A)

= {a}

3) $X \to ABC$

$A \to a/\varepsilon$

$B \to b$

$C \to c$

FIRST(X) = FIRST(A)

= {a, ε}

↓

FIRST(B)

= {a, b}

4) $X \to ABC$
$A \to a/\varepsilon$
$B \to b/\varepsilon$
$C \to c$

FIRST(X) = FIRST(A)
$= \{a, \varepsilon\}$
↓
FIRST(B)
$\{b, \varepsilon\}$ ↳ FIRST(C)
$= \{a, b, c\}$ $\{c\}$

5) $X \to ABC$
$A \to a/\varepsilon$
$B \to b/\varepsilon$
$C \to c/\varepsilon$

FIRST(X) = FIRST(A)
$= \{a, \varepsilon\}$
↓
FIRST(B)
$\{b, \varepsilon\}$
↳ FIRST(C)
↳ $\{c, \varepsilon\}$

$\therefore = \{a, b, c, \varepsilon\}$

ii) FOLLOW :

i) place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker

ii) If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B)

iii) If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

How to find FOLLOW :

1) $A \to \underset{\alpha}{X} \underset{B}{B} \underset{\beta}{CD}$
$C \to c$

follow(B) = {FIRST(CD)}
$= \{c\}$

2) $A \to \underset{\alpha}{X} \underset{B}{B} \underset{\beta}{C D}$

Since $\beta = CD$
$\therefore$ FOLLOW(B) = {FIRST(β)}
= FIRST(CD}
$= \{c, d\}$

3) $A \to \overset{\alpha}{X}\overset{B}{B}\overset{\beta}{CD}$

$C \to c/\varepsilon$

$D \to d/\varepsilon$

FOLLOW(B) = first(β)

= first(CD)

= {c, d} + FOLLOW(A)

4) $A \to \underset{\alpha}{X}\underset{B}{B}\underset{\beta}{}$ (Since $\beta = \varepsilon$, here)

follow(B) = FALLOW(left most nonterminal)

= FALLOW(A)

5) $A \to \overset{\alpha}{X}\overset{B}{B}\overset{\beta}{CDBe}$ — ①

$C \to c/\varepsilon$ — ②

$D \to d$

FALLOW(B) = FIRST(CDBe)

= (C.D) ——— ①

+

FALLOW(B) = FIRST(e)

= {e}

∴ FAL(B) = ① + ②

FAL(B) = {c, d, e}

Find the FIRST and FOLLOW set for the following grammars

1. $E \to TE'$

$E' \to +TE'/\varepsilon$

$T \to FT'$

$T' \to *FT'/\varepsilon$

$F \to (E)/id$

2. $S \to iEts/iEtses/a$

$E \to b$

3) $S \to ,GH;$
$G \to aF$
$F \to bF \mid \varepsilon$
$H \to KL$
$K \to m \mid \varepsilon$
$L \to n \mid \varepsilon$

4) $S \to aB \mid ac \mid sd \mid se$
$B \to bBc \mid f$
$C \to g$

5) $S \to aBDh$
$B \to ec$
$C \to bc \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

6) $D \to EF$
$E \to g \mid \varepsilon$ → epsilon
$F \to f \mid \varepsilon$

6. $S \to (L) \mid a$
$L \to L,S \mid S$

7) $S \to L = R \mid R$
$L \to *R \mid id$
$R \to L$

8. $S \to AaAb \mid BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

9) $S \to aABb$
$A \to c \mid \varepsilon$
$B \to d \mid \varepsilon$

10. stmt-Sequence → stmt stmtSequence'
stmt_sequence' → ; stmt_sequence | ε
stmt → S

11) $S \to asbs \mid bsas \mid \varepsilon$

12) $S \to a \mid \uparrow \mid (T)$
$T \to T,S \mid S$

13) $S \to As \mid b$
$A \to SA \mid a$

Answer:

1) $E \to TE'$

$E' \to +TE' / \varepsilon$

$T \to FT'$

$T' \to *FT' / \varepsilon$

$F \to (E) / id$

	E	E'	T	T'	F
FIRST	(	+	(	*	(
	id	ε	id	ε	id
FOL	$	$	+	+	*
	)	)	$	$	
			)	)	

here FOLLOW(E) = $F \to (\underset{\alpha}{E})\underset{\beta}{}$

FOLLOW(E) = FIRST())

= {)}

FOLLOW(E') = $E \to \underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{}$ and $E' \to +\underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{}$

FOL(E') = FIRST(β) FOL(E') = FIRST(β)

↓ ε ↓ ε

∴ FOL(E') = FOL((E) ∴ FOL(E') = FOL(E

So, on FOLLOW(T) = $E' \to \underset{\alpha}{+}\underset{B}{T}\underset{\beta}{E'}$ $E \to \underset{\alpha}{T}\underset{B}{E'}\underset{\beta}{}$

= FIRST(β) = FIRST(β)

= FIRST(E') = FIRST(E')

= {+} = {+}

Note: we should not write ε in the FOLLOW set-

2) S → iEts | iEtses | a
E → b

	S	E
FIRST	i a	b
FOLLOW	$ e	t

FOLLOW(S) = $ → iEtS
α B β

S → iEtses
α B β

FOLLOW(S) = FIRST(β)
= FOLLOW(S) ∵ ε

FOLLOW(S) = FIRST(β)
= FIRST(es)
= {e}

3) S → ,GrH;
Gr → aF
F → bF | ε — Epsilon
H → KL
K → m | ε
L → n | ε

	S	Gr	F	H	K	L
FIRST	,	a	b ε	m n ε	m ε	n ε
FOLLOW	$	m n ;	m n ;	;	n ;	;

4) $S \to aB \mid aC \mid Sd \mid Se$
$B \to bBC \mid f$
$C \to g$

	S	B	C
FIRST	a	b f	g
FOLLOW	$ d e	$ d e c	$ d e

5) $S \to aBDh$
$B \to eC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

	S	B	C	D	E	F
FIRST	a	e	b ε	g f ε	g ε	f ε
FOLLOW	$	g f h	g f h	h	h f	h

6) $S \to (L) \mid a$
$L \to L,S \mid S$

	S	L
FIRST	(a	(a
FOLLOW	$) ,	) ,

8) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

	S	A	B
FIRST	a b	ε	ε
FOLLOW	$	a b	b a

9) $S \rightarrow aABb$

$A \rightarrow C \mid \varepsilon$

$B \rightarrow d \mid \varepsilon$

	S	A	B
FIRST	a	(ε	d ε
FOLLOW	$	d b	b

10) $S \rightarrow a \mid \uparrow \mid (T)$

12) $T \rightarrow T, S \mid S$

	S	T
FIRST	a ↑ (	a ↑ (
FOLLOW	$) ,	))

(3) $S \rightarrow As \mid b$

$A \rightarrow SA \mid a$

	S	A
FIRST	b	a
	a	
FOLLOW	$	a
	a	b
	b	

(4) $S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

	S	L	R
FIRST	*	*	*
	id	id	id
FOLLOW	$	=	$
		$	=

10) stmt_sequence → stmt stmt_sequence'

stmt_sequence' → ; stmt_sequence | ε

stmt → s

	stmt_sequence	stmt_sequence'	stmt
FIRST	s	;	s
		ε	
FOLLOW	$	$	;
			$

(18) $S \to asbs \mid bsas \mid \varepsilon$

	S
FIRST	a
	b
	ε
Follow	$
	a
	b

Follow(S) $\Rightarrow$ $S \to \underset{\alpha}{as}\underset{B}{b}\underset{\beta}{s}$ $\quad S \to \underset{\alpha}{bs}\underset{B}{as}\underset{\beta}{}$ $\quad S \to \cancel{\varepsilon}$

$= FIRST(\beta)$ $\quad = FIRST(\beta)$

$= FIRST(b\$)$ $\quad = FIRST(as)$

$Follow(S) = \{b\}$ $\quad = \{a\}$

$FOL(S) \to \underset{\alpha}{as}\underset{B}{b}\underset{\beta}{s}$ $\quad Fa(S) \to \underset{\alpha}{bs}\underset{B}{as}\underset{\beta}{}$

$= FIRST(\beta)$ $\quad = FIRST(\beta)$

$\beta \to \varepsilon$ $\quad \beta \to \varepsilon$

$FOL(S) = FOL(S)$ $\quad FOL(S) = FOL(S)$

Top-down Parsers

predictive parsing table / LL(1) grammar / Table Driven predictive parsers

→ lookahead symbol

LL(1) grammar

↓ → Left most derivation

Scan the i/p from left to right

Steps:

1) Eliminate left recursion from the grammar
2) perform left factoring
3) find the FIRST and FOLLOW sets
4) Construct the predictive parsing table
5) check whether the given i/p string is accepted / not

Algorithm for Constructing predictive parsing table

INPUT : Grammar G

OUTPUT : parsing table M

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following

1. For each terminal α in FIRST(A), add $A \rightarrow \alpha$ to $M[A, \alpha]$
2. If ε is in FIRST(α), Then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ε is in FIRST(α) & \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well

Predictive parsing Algorithm

INPUT : A string w and a parsing table m for a grammar G.

OUTPUT : If w is in L(G) and LMD of w; otherwise an error condition

Input:

			a	+	b	$

METHOD: Initially, the parser is in a configuration with w$ in the i/p buffer and the start symbol S on top of the stack, above $. The pgm in fig. uses the predictive parsing table M to procedure a predictive parser for the i/p

set 'ip' to point to the first symbol of w;
set 'x' to the top stack symbol;
while (x ≠ $) { /* stack is not empty */)
If (x is a) pop the stack & a advance ip:
else if (x is a terminal) error();
else if (M[x, a] is an error entry) error();
else if (M[x, a] = x → y1, y2 --- yk) {
output the production X → y1, y2 --- yk;
pop the stack;
push yk, yk-1 --- y1 onto the stack, with y1 on-top
}
Set x to the top stack symbol;
}

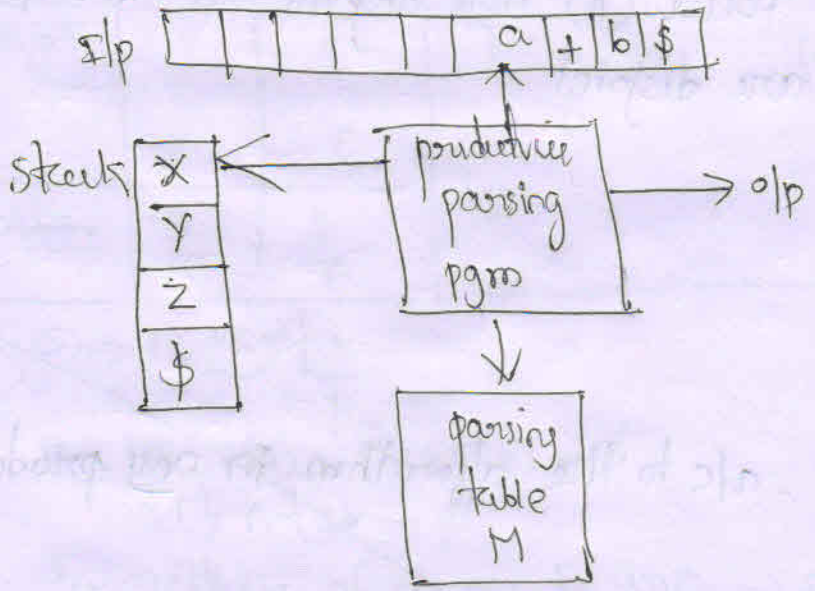


fig: model of a table driven predictive parser

Checking wheather the given grammar is LL(1) or not without using parsing table

A grammar is LL(1) iff whenever, $A \to \alpha | \beta$ are two distinct productions of G, the following conditions hold

i) For no terminal 'a' do with α and β derive strings beginning with a $\Rightarrow$ FIRST(α) and FIRST(β) are disjoint.

ii) Atmost one of α and β can derive the empty string
$\Rightarrow$ either FIRST(α) $\to \varepsilon$ or FIRST(β) $\to \varepsilon$ but not both.

iii) If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(B)

$\Rightarrow$ FIRST(α) and FOLLOW(A) are disjoint or FIRST(β) and FOLLOW(A) are disjoint

General forms:

① $A \to \underset{\alpha}{a}\underset{\beta}{B} | \underset{\alpha}{a}\underset{\beta}{c}$

FIRST(α) = {a}

FIRST(β) = {a}

are not disjoint, a/c to the algorithm in any production

eg: $A \to aB | bC$

FIRST(α) = {a}

FIRST(β) = {b} are disjoints

2) $A \to Bc \mid CD$
$B \to b \mid \varepsilon$
$C \to c \mid \varepsilon$

either $FIRST(\beta) \Rightarrow \varepsilon$ or
$FIRST(\alpha) \Rightarrow \varepsilon$
but not both

3) i) $A \to a \mid B$
$B \to cAa \mid \varepsilon$
$FIRST(\alpha) = \{a\}$
$FOLLOW(A) = \{\$, a\}$ are not disjoint

ii) $A \to B \mid a$
$B \to cAa \mid \varepsilon$
$FIRST(\beta) = \{a\}$ and $FOLLOW(A) = \{\$, a\}$
are not disjoint

Examples:

1. $S \to iEtss' \mid a$
$S' \to es \mid \varepsilon$
$E \to b$

	S	S'	E
FIRST	i	e	b
	a	ε	
FOLLOW	\$	\$	t
	e	e	

α β
$S \to iEtss' \mid a$

a. $FIRST(\alpha) \cap FIRST(\beta) = \phi$
$\{i\} \cap \{a\} = \phi$

b. neither of α or β are ε

I. $S \to \underbrace{iEtss'}_{\alpha} \mid \underbrace{a}_{\beta}$

a. $FIRST(\alpha) \cap FIRST(\beta) = \phi$
$\{i\} \cap \{a\} = \phi$

b. neither of α or β are $\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$, then $FIRST(\alpha) \cap FOLLOW(A) = \phi$
$FIRST(es) \cap FOLLOW(S') = \phi$
$\{e\} \cap \{\$, e\} \neq 0$

∴ The given grammar is not LL(1). Condition fails

$S' \to \overset{\alpha}{e}S \mid \overset{\beta}{\varepsilon}$

a) $FIRST(\alpha) \cap FIRST(\beta) = \phi$

$\{e\} \cap \{\varepsilon\} = \phi$

b) The given grammar

$\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c) $\beta \Rightarrow \varepsilon$, then $FIRST(\alpha) \cap FOL(A) = \phi$

$FIRST(eS) \cap FOL(S') = \phi$

$\{e\} \cap \{\$e\} \neq \phi$

Condition fails

∴ The given grammar is not LL(1)

2) $S \to S(S)S \mid \varepsilon \Rightarrow S \to S'$

$S' \to (S)\,SS' \mid \varepsilon$

	S	S'
FIRST	(	(
	ε	ε
FOLLOW	$	$
	)	)
	(	(

1. $S \to S'$

not required because we dont have β production

2. $S' \to \overset{\alpha}{(S)}\,SS' \mid \overset{\beta}{\varepsilon}$

a. $FIRST(\alpha) \cap FIRST(\beta) \neq \phi$

$\{(\} \cap \{\varepsilon\} = \phi$

b. only $\beta \Rightarrow \varepsilon$ and $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$, then

$FIRST(\alpha) \cap FOLLOW(A) = \phi$

$\{(\} \cap \{\$\,(\,)\} \neq \phi$

∴ The given grammar is not LL(1)

3. $S \to SS+ \mid SS* \mid a \Rightarrow S \to aS'$

$S' \to S+S' \mid S*S' \mid \varepsilon$

⇓ left factoring

final production
$S \to aS'$
$S' \to SS'' \mid \varepsilon$
$S \to +S' \mid *S'$

	S	S'	S''
FIRST	a	a	+
		ε	*
FOLLOW	$	$	$
	+	+	+
	*	*	*

1. $S \to aS'$
   not required
2. $S'' \to +S' \mid *S'$
   a. $FIRST(+S') \cap FIRST(*S') = \phi$
      $\{+\} \cap \{*\} = \phi$
   b. neither α or $\beta \Rightarrow \varepsilon$
      all Condition are Satisfied
3. $S' \to SS'' \mid \varepsilon$
   a. $FIRST(SS'') \cap FIRST(\varepsilon)$
      $\{a\} \cap \{\varepsilon\} = \phi$
   b. $\beta \Rightarrow \varepsilon$ but not α
   c $\beta \Rightarrow \varepsilon$ then
      $FIRST(SS'') \cap FOLLOW(S') = \phi$
      $\{a\} \cap \{\$ + *\} = \phi$

   all Conditions are Satisfied
   ∴ The grammar is LL(1)

4) $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

$\Rightarrow$

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

	E	E'	T	T'	F
FIRST	(id	+ ε	(id	* ε	(id
FOLLOW	$)	$)	+ $)	+ $)	+ * $)

i) $E \to TE'$

ii) $E' \to +E' \mid \varepsilon$

a. $FIRST(+TE) \cap FIRST(\varepsilon) = \phi$
$\{+\} \cap \{\varepsilon\} = \phi$

b. $\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$ then $FIRST(+TE) \cap FOL(E') = \phi$
$\{+\} \cap \{\$\} = \phi$

iii) $T \to FT'$

iv) $T' \to *FT' \mid \varepsilon$

a. $FIRST(*FT') \cap FIRST(\varepsilon) = \phi$
$\{*\} \cap \{\varepsilon\} = \phi$

b. $\beta \Rightarrow \varepsilon$ but $\alpha \not\Rightarrow \varepsilon$

c. $\beta \Rightarrow \varepsilon$ then $FIRST(*FT') \cap FOL(T') = \phi$
$\{*\} \cap \{+\$)\} = \phi$

v) $F \to (E) \mid id$

a. $FIRST((E)) \cap FIRST(id) = \phi$
$\{(\} \cap \{id\} = \phi$

b. neither of them are not $\Rightarrow \varepsilon$

checking wheather the given grammar is LL(1) or not with Constructing the predictive parsing table

1. $E \to E+T \mid T$
   $T \to T*F \mid F$
   $F \to (E) \mid id$

   i/p : id+id*id

i) Remove left Recursion

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

ii) Remove left factoring
→ here, not required

iii) find FIRST and FOLLOW set

	E	E'	T	T'	F
FIRST	(	+	(	*	(
	id	ε	id	ε	id
Follow	$	$	+	+	*
	)	)	$	$	+
			)	)	$
					)

iv) Construct the parsing table

	(	id	)	+	*	$
E	$E \to TE'$	$E \to TE'$				
E'			$E' \to \varepsilon$	$E' \to +TE'$		$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$				
T'			$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$	$T' \to \varepsilon$
F	$F \to (E)$	$F \to id$				

v)

Stack	Input	Action
$E\$$	$id+id*id\$$	
$TE'\$$	$id+id*id\$$	push $E \to TE'$
$FT'E'\$$	$id+id*id\$$	push $T \to FT'$
$idT'E'\$$	$id+id*id\$$	push $F \to id$
$T'E'\$$	$+id*id\$$	matched 'id'
$E'\$$	$+id*id\$$	$T' \to \varepsilon$
$+TE'\$$	$+id*id\$$	push $E' \to +TE'$
$TE'\$$	$id*id\$$	matched '+'
$FT'E'\$$	$id*id\$$	push $T \to FT'$
$idT'E'\$$	$id*id\$$	push $F \to id$
$T'E'\$$	$*id\$$	matched 'id'
$*FT'E'\$$	$*id\$$	push $T' \to *FT'$
$FT'E'\$$	$id\$$	matched '*'
$idT'E'\$$	$id\$$	push $F \to id$
$T'E'\$$	$\$$	matched 'id'
$E'\$$	$\$$	$T' \to \varepsilon$
$\$$	$\$$	$E' \to \varepsilon$

∴ The grammar is accepted the ip string
i.e, The ip is passed successfully

The grammar is LL(1) ∵ no multiple entries

2) $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

i/p: If E_1 then if E_2 then S_1 else S_2

If b then if b then a else a

i) No left Recursion

ii) Remove left factoring

$S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

$E \to b$

iii) FIRST and FOLLOW set

	S	S'	E
FIRST	i a	e ε	b
FOLLOW	$ e	$ e	t

iv) parsing table

	$	i	e	b	a	t
S		$S \to iEtSS'$				
S'	$S' \to \varepsilon$		$S' \to eS$ $S' \to \varepsilon$			
E				$E \to b$		

The grammar is not LL(1) because it has multiple entries for the same terminal in a table

v)

Stack	Input	Action
S $	~~iEtEtSeS~~ ibtibtaea $	
iEtSS' $	ibtibtaea $	$S \to iEtSS'$
EtSS' $	btibtaea $	matched 'i'
btSS' $	btibtaea $	$E \to b$

Stack	input	Action
$tss'\$$	$ibtaea\$$	matched 'b'
$ss'\$$	$ibtaea\$$	matched 't'
$iEtss's'\$$	$ibtaea\$$	$S \to iEtSS'$
$Etss'S'\$$	$btaea\$$	matched 'i'
$btss'S'\$$	$btaea\$$	$E \to b$
$tss'S'\$$	$taea\$$	matched 'b'
$ss'S'\$$	$aea\$$	matched t
$as'S'\$$	$aea\$$	$S \to a$
$s'S'\$$	$ea\$$	matched 'a'

↳ ambiguity wheather to push $S' \to eS$ or $S' \to \varepsilon$

⇒ i/p: if E then S else S

ibtaea

Stack	input	Action
$S\$$	$ibtaea\$$	$S \to iEtSS'$
$iEtss'\$$	$ibtaea\$$	matched 'i'
$Etss'\$$	$btaea\$$	push $E \to b$
$btss'\$$	$btaea\$$	
$tss'\$$	$taea\$$	match 'b'
$ss'\$$	$aea\$$	match t
$as'\$$	$aea\$$	$S \to a$
$s'\$$	$ea\$$	match a

↳ ambiguity, whether to push $S' \to eS$ or $S' \to \varepsilon$

3) $S \to SS+ | SS* | a$ ilp: aa+a*

i) Remove left recursion

$S \to aS'$

$S' \to S+S | S*S' | \varepsilon$

ii) Remove left factoring

$S \to aS'$

$S' \to SS'' | \varepsilon$

$S'' \to +S' | *S'$

iii) find FIRST and FOLLOW set

	S	S'	S''
FIRST	a	a	+
		ε	*
FOLLOW	$	$	$
	+	+	+
	*	*	*

iv) find the predictive parsing table

	$	a	+	*
S		$S \to aS'$		
S'	$S' \to \varepsilon$	$S' \to SS''$	$S' \to \varepsilon$	$S' \to \varepsilon$
S''			$S'' \to +S'$	$S'' \to *S'$

The given grammar is LL(1) because there are no multiple production.

v) parse the ilp string

stack	input	action
S$	aa+a*$	
aS'$	aa+a*$	$S \to aS'$
S'$	a+a*$	match 'a'
SS''$	a+a*$	$S' \to aS'$
aS'S''$	a+a*$	$S \to aS'$
S'S''$	+a*$	match 'a'
S''$	+a*$	$S' \to \varepsilon$

Stack	input	Stack
+s'$	+a*$	s'' → +s'
s'$	a*$	match '+'
ss''$	a*$	push s' → ss''
as's''$	a*$	push s → as'
s' s''$	*$	mpatche 'a'
s''$	*$	push s' → ε
*s'$	*$	push s'' → *s'
s'$	$	match *
$	$	push s' → ε

The i/p string is Successfully parsed

4) S → 0S1|01 i/p string: 000111

i) Remove left recursion
→ not needed

ii) Remove left factoring
S → 0S'
S' → S1|1

iii) FIRST and FOLLOW set

	S	$
FIRST	0	0
		1
FOLLOW	$	$
	1	1

iv) Construct the predictive parsing table

	$	0	1
S		S → 0S'	
S'		S' → S1	S' → 1

The grammar is LL(1), Since it does not have any multiple production

↓ parse the input storing

Stack	input	action
$S\$$	$000111\$$	
$0S'\$$	$000111\$$	push $S \to 0S'$
$S'\$$	$00111\$$	match '0'
$S1\$$	$00111\$$	push $S' \to S1$
$0S'1\$$	$00111\$$	$S \to 0S'$
$S'1\$$	$0111\$$	match '0'
$S11\$$	$0111\$$	$S' \to S1$
$0S'11\$$	$0111\$$	push $S \to 0S'$
$S'11\$$	$111\$$	match 0
$111\$$	$111\$$	$S' \to 1$
$11\$$	$11\$$	match 1
$1\$$	$1\$$	match 1
$\$$	$\$$	match 1

The input storing is successfully parsed

5) $S \rightarrow +SS|*SS|a$ i/p: +*aaa

i) Remove left recursion ⇒ not required

ii) Remove left-factoring ⇒ Not required

iii) Construct FIRST and FOLLOW sets

	S
FIRST	+
	*
	a
FOLLOW	$
	+
	*
	a

iv) Construct the predictive parsing table

	$	+	*	a
S		+SS	*SS	a

The grammar is LL(1). Since It Contains no more than 1 production

v) i/p string: +*aaa

Stacks	input	Action
S$	+*aaa$	
+SS$	+*aaa$	S→+SS
SS$	*aaa$	match '+'
*SSS$	*aaa$	S→*SS
SSS$	aaa$	match *
aSS$	aaa$	push S→a
SS$	aa$	match a
aS$	aa$	push S→a
S$	a$	match a
a$	a$	push S→a
$	$	match a

i/p is Successfully parsed

6) $S \to S(S)S \mid \varepsilon$ i/p: (()())

i) Remove left Recursion

$S \to \varepsilon S'$

$S' \to (S)SS' \mid \varepsilon$

ii) Remove left factoring

→ not needed

iii) Find FIRST and FOLLOW

	S	S'
FIRST	a	+, ε
	(	a
		(
		*
FOLLOW	$ +	$
	) a	)
	(	+
	*	a, (, *

iv) Construct the predictive parsing table

	$	(	a	)	+	*
S		$S \to (S)S'$	$S \to aS'$			
S'	$S' \to \varepsilon$	$S' \to SS'$	$S \to SS'$	$S' \to \varepsilon$	$S' \to +SS'$	$S' \to *S'$
		$S' \to \varepsilon$	$S' \to \varepsilon$		$S' \to \varepsilon$	$S' \to \varepsilon$

The grammar is not LL(1)

v) input : (a+a)*a

stack	input	Action
S$	(a+a)*a	
(S)S'$	(a+a)*a	$S \to (S)S'$
S)S'$	a+a)*a	match (
aS')S'$	a+a)*a	push $S \to aS'$
S')S'$	+a)*a	match a

→ Ambigous, whether to parse $S' \to +SS'$ or $S' \to \varepsilon$

7) $S \to S+S \mid SS \mid (S) \mid S* \mid a$

i/p: (a+a)*a

i) Remove left Recursion

$S \to aS' \mid (S)S'$

$S' \to +SS' \mid SS' \mid *S' \mid \varepsilon$

6) $S \to S(S)S \mid \varepsilon$ i/p: (()())

i) Remove left Recursion

ii) $S \to \varepsilon S'$

$S' \to (S)SS' \mid \varepsilon$

ii) Remove left factoring

→ not needed

iii) find FIRST and FOLLOW set

	S	S'
FIRST	(, ε	(, ε
FOLLOW	$,), (	$,), (

iv) parsing table

	$	(	)
S	$S \to \varepsilon$	$S \to S$, $S \to \varepsilon$	$S \to \varepsilon$
S'	$S' \to \varepsilon$	$S' \to (S)SS'$, $S' \to \varepsilon$	$S' \to \varepsilon$

→ multiple productions

The grammar is not LL(1). Since it has a multiple ~~transaction~~ productions

v) parse the i/p string: (()())

Stack	Input	Action
S$	(()())$	$S \to S'$ → ambiguity
S'$	(()())$	$S' \to (S)SS'$
(S)SS'$	(()())$	match '('
S)SS'$	()())$	
S')SS'$	()())$	push $S \to S'$

7) $S \to (L) \mid a$
$L \to L,S \mid S \quad \Rightarrow (a,a)$

step i) Remove left Recursion

$L \to \underset{A}{\$}, \underset{\alpha}{S} \mid \underset{\beta}{S}$

$L \to SL'$
$L' \to ,SL' \mid \varepsilon$
$S \to (L) \mid a$

ii) Remove left ~~Recursion~~ factoring
not required

iii) write FIRST and FOLLOW set

	S	L	L'
FIRST	(	(	,
	a	a	ε
FOLLOW	$	)	)
	,		
	)		

iv) write the predictive parsing table

	(	a	)	,	$
S	S→(L)	S→a			
L	L→SL'	L→SL'			
L'		L'→ε		L'→,SL	

v)

Stack	Input	action
S$	(a,a) $	
(L)$	(a,a) $	S→(L)
L)$	a,a) $	match (
SL')$	a,a) $	L→SL'
aL')$	a,a)$	S→a
L')$	,a)$	match a
,SL')$	,a)$	L'→,SL'
SL')$	a)$	match ,
aL')$	a)$	S→a
L')$	)$	match a
)$	)$	L'→ε
$	$	match)

The grammar i/p is successfully parsed

8) S→ S+S| SS| (S)|S* |a ⇒ (a+a) *a

i) Remove left recursion

S→ S+S |SS| (S) |S*| a

S→ (S)S' |a S'

S' → ~~(S)S'|aS'~~ +SS' |SS'|*S'|ε

ii) remove left factoring
not required

iii) Find FIRST and FOLLOW Set

	S	S'
FIRST	(	+
	a	a
		*
		ε

	S	S'
FOLLOW	$	$
	)	)
	+	+
	(	(
	a	a
	*	*

iv) write predictive parse table

	(	a	)	+	*	$
S	S→(S)S'	S→aS'				
S'	S→+SS'			S'→+SS'		
	S→SS'					
	S→*S'					
	S→ε					

∴ The given grammar is not LL(1)

g) S→aSbS | bSaS | ε ⇒ aabbab

i) Remove left factoring – not required

ii) Remove left Recursion – not required

iii) Write FIRST and FOLLOW set

	S
FIRST	a
	b
	ε

	S
FOLLOW	b
	a
	$

iv) write a predictive parsing table

	a	b	$
S	S→aSbS	S→bSaS	S→ε
	S→ε	S→ε	

v.

Stack	Input	Action
S$	aabbab$	
asbs$	aabbab$	S→asbs
sbs$	abbab$	match a
asbsbs$	abbab$	S→asbs
sbsbs$	bbab$	match a
bsasbsbs$	bbab$	S→bSaS
SaSbsbs$	bab$	match b
bSaSasbsbs$	bab$	S→bSaS
SaSasbsbs$		

↳ ambigous

vi) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false
i/p: not (true or false)

→ i) Remove left Recursion
bexpr → ~~bterm~~ bexpr or bterm | bterm
bexpr' → or bterm bexpr' | ε → bterm → bterm and bfactor | bfactor
bterm → ~~bfactor bterm'~~ bfactor bterm'
bterm' → and bfactor bterm' | ε
bfactor → not bfactor | (bexpr) | true | false

ii) No left ~~recursion~~ factoring

iii) Find the FIRST and Follow Set

	bexpr	bexpr'	bterm	bterm'	bfactor
FIRST	not (true false	or ε	not (true false	and ε	not (true false
FOLLOW	$)	$)	or) $	or $)	and or) $

predictive parsing table :

	(	not	)	or	and	true	false	$
bexpr	btermbexpr'	btermbexpr'				btermbexpr'	btermbexpr'	
bexpr'			ε	orbtermbexpr				ε
bterm	bfactorbterm'	bfactorbterm'				bfactorbterm'	bfactorbterm'	
bterm'			ε	ε	andbfactorbterm			ε
bfactor	(bexpr)	not bfactor				true	false	

Stack	Input	Action
bexpr $	not (true or false)$	bexpr → bterm bexpr'
bterm bexpr'$	not (true or false)$	bterm → bfactor bterm'
bfactor bterm' bexpr'$	not (true or false)$	bfactor → not bfactor
not bfactor bterm' bexpr'$	not (true or false)$	bfactor → (bexpr)
(bexpr) bterm' bexpr'$	not (true or false)$	~~match (~~
bexpr) bterm' bexpr'$	true or false)$	match (
bterm bexpr') bterm' bexpr'$	true or false)$	bexpr → bterm bexpr'
bfactor bterm' bexpr') bterm' bexpr'$	true or false)$	bterm → bfactor bterm'
bterm' bexpr') bterm' bexpr'$	or false)$	bfactor → bterm'
bexpr') bterm' bexpr'$	or false)$	bterm' → ε
bterm bexpr') bterm' bexpr'$	false)$	bexpr' → bterm bexpr
bfactor bterm' bexpr') bterm' bexpr'$	false)$	bterm → bfactor bterm'
bexpr') bterm' bexpr'$	) $	bfactor → ε
bterm' bexpr') bterm' bexpr'$	) $	bterm' → ε & bexpr' → ε & match)
bterm' bexpr'$	$	
$	$	

Error recovery in predictive parsers:

1. panic mode Recovery: In the blank entries of LL follow set of all NT, place 'syn

Stack	i/p	Table Entry	Action
NT	T	blank	skip the terminal from the i/p
NT	T	Synch	Remove NT from the S except start symbol
T	T	match	pop the terminal from stack & input

Example:

1. $E \to E+T/T$
   $T \to T*F/F$
   $F \to (E)/id$
   i/p:)id*+id

$E \to TE'$
$E' \to +TE'/\varepsilon$
$T \to FT'$
$T' \to *FT'/\varepsilon$
$F \to (E)/id$

	E	E'	T	T'	F
FIRST	(	+	(	*	(
	id	ε	id	ε	id
Follow	$	$	+	+	*
	)	)	$	$	$
			)	)	)
					+

predictive parsing table

	(	id	)	+	*	$
E	$E \to TE'$	$E \to TE'$	Synch			
E'			$E' \to \varepsilon$	$E' \to +TE'$		$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	~~Synch~~ ~~$T \to \varepsilon$~~	~~Synch~~ ~~$T \to *FT'$~~		Synch
T'			$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$	$T' \to \varepsilon$
F	$F \to (E)$	$F \to id$	Synch	Synch	Synch	Synch

Stack	input	Action
E $	)id * +id $	Since it is the start symbol skip the i/p [E,)] = Synch
E $	id * +id $	E → TE'
TE' $	id * +id $	T → FT'
FT'E' $	id * +id $	F → id
idT'E' $	id * +id $	match id
T'E' $	* +id $	T' → *FT'
*FT'E' $	* +id $	match *
FT'E' $	+id $	[F, +] = Synch, remove NT from stack
T'E' $	+id $	T' → ε
E' $	+id $	E' → +TE'
+TE' $	+id $	match +
TE' $	id $	T → FT'
FT'E' $	id $	F → id
idT'E' $	id $	match id
T'E' $	$	T' → ε
E' $	$	E' → ε
$	$	match $

5) Show that the following grammar is ambiguous.
E → E+E | E-E | E*E | E/E | (E) | id and i/p: id + id * id
Give an unambiguous grammar such that precedence order from lowest or highest are +, -, *, /, (), id and all are left-to right associative.

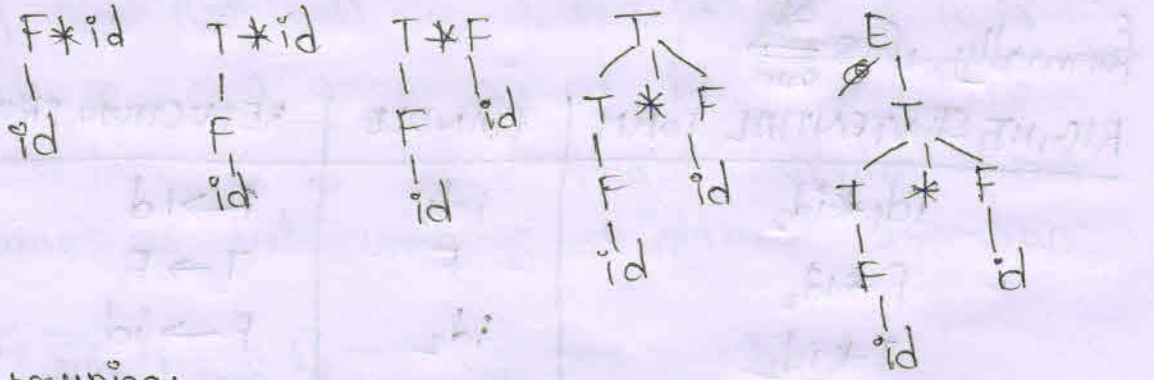
UNIT-3

Bottom up parsers:

Introduction:

→ A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top)

→ eg: A bottom-up parses for 'id * id



Handle pruning:

Bottom up parsing during a left to right scan of the input constructs a right most derivation in reverse. Informally a 'handle' is a substring that matches the body of the production, and whose reduction represents one step along the reverse of the right-most derivation

Example:

adding subscripts to the tokens id for clarity, the handles during the parse of $id_1 * id_2$ a/c to the expressions grammar → $\left\{ \begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{array} \right\}$ are shown in the figure.

Although T is the body of the production $E \to T$, the symbol T is not a handle in the sentential form $T * id_2$. If T were indeed replaced by E, we would get a string $E * id_2$, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle

~~formally, if $S \overset{*}{\underset{rm}{\Longrightarrow}}$~~

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
F	$T * F$	$E \to T * F$

Handles during a parse of $id_1 * id_2$ (a)

Formally, if $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta w$, as in figure (b), then production $A \to \beta$, in the position following α is a handle of $\alpha \beta w$. Alternatively, a handle of a right sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right sentential form in a rightmost derivation of γ

Notice that the string w to the right of the handle must contain only terminal symbols. For convenience, we prefer to the body β rather than $A \to \beta$ as a handle. Note we "a handle" rather than "the handle", because

the grammar could be ambigous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambigous, then every right-sentential form of the grammar has Exactly one handle.

A right-most derivation in reverse can be obtained by "handle pruning". That is, we start with a string of terminals w to be parsed. If w is a sentence of a grammar at hand, then let $w = \gamma_n$, where γ_n is the n^{th} right sentential form of some as yet unknown rightmost derivation.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

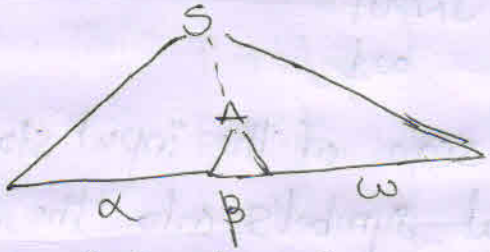


figure (b). A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of relevant production $A_n \to \beta_n$ to obtain the previous right sentential form γ_{n-1}. Note that we do not know how handles are to be found, but we shall see methods of doing so shortly

We the repeat this process. That is we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right sentential form γ_{n-2}. If by continuing this process we produce a right sentential form consisting only of the start symbol S, the we halt & that is successfull.

Shift-Reduce parsing:

Shift-reduce parsing is a form of bottom up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

→ we use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather on the left as we did for top-down parsing.

Initially, the stack is empty, and string w is on the input, as follows:

STACK	INPUT
$	w$

During left to right scan of the input string, the parser shifts zero/more input symbols onto the stack until it is ready to reduce a string β of the grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and input is empty.

Actions in shift-reduce parsing:

1. Shift: shift the next input symbol onto the top of stack
2. Reduce: The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what non terminal to replace the string

3. Accept: Announce successfull Completion of parsing

4. Error: Discover a syntax error and can an error recovery routine

Find the handles for the given RSF and Construct shift-reduce parser:

1. $E \to E+T \mid T$
   $T \to T*F \mid F$
   $F \to (E) \mid id$

i/p: id+id
id+id*id

→ RMD

$E \to E+T$
$\Rightarrow E+F$
$\Rightarrow E+id$
$\Rightarrow T+id$
$\Rightarrow F+id$
$\Rightarrow id+id$

RSF	Handle	Action
id_1+id_2	id_1	$F \to id$
$F+id_2$	F	$T \to F$
$T+id_2$	T	$E \to T$
$E+id_2$	id_2	$F \to id$
$E+F$	F	$T-F$
$E+T$	$E+T$	$E \to E+T$
E		

Stack	RSF	Action
$	$id_1+id_2\$$	shift id_1
$\$id_1$	$\$id_2\$$	reduce $F \to id$
$\$F$	$+id_2\$$	reduce $T \to F$
$\$T$	$+id_2\$$	reduce $E \to T$
$\$E$	$+id_2\$$	shift +
$\$E+$	$+id_2\$$	shift id_2
$\$E+id_2$	$\$$	reduce $F \to id_2$
$\$E+F$	$\$$	reduce $T \to F$
$\$E+T$	$\$$	reduce $E \to T$
$\$E$		Success

ip: id+id*id

RMD:

$E \to E+T$
$\Rightarrow E+T*F$
$\Rightarrow E+T*id$
$\Rightarrow E+F*id$
$\Rightarrow E+id*id$
$\Rightarrow T+id*id$
$\Rightarrow F+id*id$
$\Rightarrow id+id*id$

RSF	Handle	Action
$id_1+id_2*id_3$	id_1	$F \to id$
$F+id_2*id_3$	F	$T \to F$
$T+id_2*id_3$	T	$E \to T$
$E+id_2*id_3$	id_2	$F \to id_2$
$E+F*id_3$	F	$T \to F$
$E+T*id_3$	id_3	$F \to id_3$
$E+T*F$	$T+F$	$T \to F*F$
$E+T$	$E+T$	$E \to E+T$
E		

Stack	RSF	Action
$	$id_1+id_2*id_3\$$	shift id_1
$\$id_1$	$+id_2*id_3\$$	$F \to id_1$
$\$F$	$+id_2*id_3\$$	$T \to F$
$\$T$	$+id_2*id_3\$$	$E \to T$
$\$E$	$+id_2*id_3\$$	Shift +
$\$E+$	$id_2*id_3\$$	Shift id_2
$\$E+id_2$	$*id_3\$$	reduce $F \to id$
		$T \to F$
		shift *
		shift id_3
$\$E+T*id_3$	$\$$	reduce $F \to id_3$
$\$E+T*F$	$\$$	reduce $T \to T*F$
		reduce $F \to E+T$
$\$E$	$\$$	Success

2) $S \to 0S1 \mid 01$ i/p: 000111

$S \xrightarrow{rm} 0S1$

$\Rightarrow 00S11$

$\Rightarrow 000111$

RSF	Handle	Action
000111	01	$S \to 0S1$
00S11	0S1	$S \to 0S1$
0S1	0S1	$S \to 0S1$

Stack	RSF	Action
$	000111$	Shift 0
$0	00111$	shift 0
$00	0111 $	shift 0
$000	111 $	shift 1
$0001	11 $	reduce $S \to 01$
$00S	11$	shift 1
$00S1	1$	reduce $S \to 0S1$
$0S	1$	shift 1
$0S1	$	reduce $S \to 0S1$
$S	$	success

3) $S \to SS+ \mid SS* \mid a$ i/p: aaa*a++

$S \to SS+$

$\Rightarrow SSS++$

$\Rightarrow SSa++$

$\Rightarrow SSS*a++$

$\Rightarrow SSa*a++$

$\Rightarrow Saa*a++$

$\Rightarrow aaa*a++$

RSF	Handle	Action
aaa*a++	a	$S \to a$
Saa*a++	a	$S \to a$
SSa*a++	a	$S \to a$
SSS*a++	SS*	$S \to SS*$
SSa++	a	$S \to a$
SSS++	SS+	$S \to SS+$
SS+*	SS+	$S \to SS+$
S		

Stack	RSF	Action
$	aaa*a++$	shift a
$a	aa*a++$	reduce S→a
$S	aa*a++$	Shift a
$Sa	a*a++$	reduce S→a
$SS	a*a++$	Shift a
$SSa	*a++$	reduce S→a
$SSS	*a++$	Shift *
$SS*	a++$	reduce S→SS*
$SS	a++$	Shift a
$SSa	a++$	reduce S→a
$SSa	a++$	Shift +
$SS	+$	reduce S→SS+
$SS+	+$	Shift +
$S	$	reduce S→SS+
	$	Success

Types of conflicts in shift-reduce parsers

Conflicts during shifts-Reduce parsing:

There are CFG's for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents & the next input symbol

Types:

i) shift/reduce conflict:

→ Cannot decide whether to shift or to reduce
Called shift-reduce conflict

ii) redu/reduce conflict:

→ Cannot decide whether which of several reductions to make Called reduce-reduce conflicts

eg: Consider the grammar

1) E → E+E
   | E-E
   | NUM
   | ID

i/p: 2+3*4

No.	Stack	operation/grammar
1	2 NUM	shift 2
2	E	reduce E→NUM
3	E+	shift +
4	E+3	shift 3
5	E+E	reduce E→NUM
6	E	reduce E→E+E or shift *

ie Shift reduce conflict

eg 2) An ambiguous grammar can never be LR. for eg:
Consider the dangling-else grammar

stmt → if expr then stmt
| if expr then stmt else stmt
| other

If we have a shift-reduce parser in configuration

STACK	INPUT
...if expr then stmt	else ... $

→ we cannot tell whether if expr then stmt is the handle no matter what happens below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the else on the input, it might be correct to reduce if expr then stmt to stmt, or it might be correct to shift else then to look for another stmt to complete the alternative if expr then stmt else stmt

eg 3) (1) stmt → id (parameter_list)
(2) stmt → expr := expr
(3) parameter_list → parameter_list , parameter
(4) parameter_list → parameter
(5) parameter → id
(6) expr → id (expr_list)
(7) expr → id
(8) expr_list → expr_list , expr
(9) expr_list → expr

STACK	INPUT
... id (id	, id) ...

In the problem id on the top of the stack must be reduced, but by which production? The correct choice is production (5) if p is a procedure, but production (7) if p is an array. The stack does not tell which; information in the symbol table obtained from the declaration of p must be used

So, In this case we have the conflict that reduce id by parameter or expr. It is called as reduce-reduce conflict.

or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse. □

4.6 OPERATOR-PRECEDENCE PARSING

The largest class of grammars for which shift-reduce parsers can be built successfully – the LR grammars – will be discussed in Section 4.7. However, for a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property (among other essential requirements) that no production right side is ϵ or has two adjacent nonterminals. A grammar with the latter property is called an *operator grammar*.

Example 4.27. The following grammar for expressions

$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals. However, if we substitute for A each of its alternatives, we obtain the following operator grammar:

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id} \qquad (4.17)$$

We now describe an easy-to-implement parsing technique called operator-precedence parsing. Historically, the technique was first described as a manipulation on tokens without any reference to an underlying grammar. In fact, once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using the nonterminals on the stack only as placeholders for attributes associated with the nonterminals.

As a general parsing technique, operator-precedence parsing has a number of disadvantages. For example, it is hard to handle tokens like the minus sign, which has two different precedences (depending on whether it is unary or binary). Worse, since the relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language. Finally, only a small class of grammars can be parsed using operator-precedence techniques.

Nevertheless, because of its simplicity, numerous compilers using operator-precedence parsing techniques for expressions have been built successfully. Often these parsers use recursive descent, described in Section 4.4, for statements and higher-level constructs. Operator-precedence parsers have even been built for entire languages.

In operator-precedence parsing, we define three disjoint *precedence relations*, $\lessdot$, $\doteq$, and $\gtrdot$, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a <\cdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \cdot> b$	a "takes precedence over" b

We should caution the reader that while these relations may appear similar to the arithmetic relations "less than," "equal to," and "greater than," the precedence relations have quite different properties. For example, we could have $a <\cdot b$ and $a \cdot> b$ for the same language, or we might have none of $a <\cdot b$, $a \doteq b$, and $a \cdot> b$ holding for some terminals a and b.

There are two common ways of determining what precedence relations should hold between a pair of terminals. The first method we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ <\cdot *$ and $* \cdot> +$. This approach will be seen to resolve the ambiguities of grammar (4.17), and it enables us to write an operator-precedence parser for it (although the unary minus sign causes problems).

The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees. This job is not difficult for expressions; the syntax of expressions in Section 2.2 provides the paradigm. For the other common source of ambiguity, the dangling **else**, grammar (4.9) is a useful model. Having obtained an unambiguous grammar, there is a mechanical method for constructing operator-precedence relations from it. These relations may not be disjoint, and they may parse a language other than that generated by the grammar, but with the standard sorts of arithmetic expressions, few problems are encountered in practice. We shall not discuss this construction here; see Aho and Ullman [1972b].

Using Operator-Precedence Relations

The intention of the precedence relations is to delimit the handle of a right-sentential form, with $<\cdot$ marking the left end, $\doteq$ appearing in the interior of the handle, and $\cdot>$ marking the right end. To be more precise, suppose we have a right-sentential form of an operator grammar. The fact that no adjacent nonterminals appear on the right sides of productions implies that no right-sentential form will have two adjacent nonterminals either. Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 \cdots a_n \beta_n$, where each β_i is either ϵ (the empty string) or a single nonterminal, and each a_i is a single terminal.

Suppose that between a_i and a_{i+1} exactly one of the relations $<\cdot$, $\doteq$, and $\cdot>$ holds. Further, let us use \$ to mark each end of the string, and define $\$ <\cdot b$ and $b \cdot> \$$ for all terminals b. Now suppose we remove the nonterminals from the string and place the correct relation $<\cdot$, $\doteq$, or $\cdot>$, between each

pair of terminals and between the endmost terminals and the \$'s marking the ends of the string. For example, suppose we initially have the right-sentential form **id** + **id** * **id** and the precedence relations are those given in Fig. 4.23. These relations are some of those that we would choose to parse according to grammar (4.17).

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	

Fig. 4.23. Operator-precedence relations.

Then the string with the precedence relations inserted is:

$$\$ <\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot> * <\cdot \textbf{id} \cdot> \$ \tag{4.18}$$

For example, $<\cdot$ is inserted between the leftmost \$ and **id** since $<\cdot$ is the entry in row \$ and column **id**. The handle can be found by the following process.

1. Scan the string from the left end until the first $\cdot>$ is encountered. In (4.18) above, this occurs between the first **id** and +.
2. Then scan backwards (to the left) over any $\doteq$'s until a $<\cdot$ is encountered. In (4.18), we scan backwards to \$.
3. The handle contains everything to the left of the first $\cdot>$ and to the right of the $<\cdot$ encountered in step (2), including any intervening or surrounding nonterminals. (The inclusion of surrounding nonterminals is necessary so that two adjacent nonterminals do not appear in a right-sentential form.) In (4.18), the handle is the first **id**.

If we are dealing with grammar (4.17), we then reduce **id** to E. At this point we have the right-sentential form $E+$**id**$*$**id**. After reducing the two remaining **id**'s to E by the same steps, we obtain the right-sentential form $E+E*E$. Consider now the string \$+*\$ obtained by deleting the nonterminals. Inserting the precedence relations, we get

$$\$ <\cdot + <\cdot * \cdot> \$$$

indicating that the left end of the handle lies between + and * and the right end between * and \$. These precedence relations indicate that, in the right-sentential form $E+E*E$, the handle is $E*E$. Note how the E's surrounding the * become part of the handle.

Since the nonterminals do not influence the parse, we need not worry about distinguishing among them. A single marker "nonterminal" can be kept on

the stack of a shift-reduce parser to indicate placeholders for attribute values.

It may appear from the discussion above that the entire right-sentential form must be scanned at each step to find the handle. Such is not the case if we use a stack to store the input symbols already seen and if the precedence relations are used to guide the actions of a shift-reduce parser. If the precedence relation $<\cdot$ or $\doteq$ holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts; it has not yet found the right end of the handle. If the relation $\cdot>$ holds, a reduction is called for. At this point the parser has found the right end of the handle, and the precedence relations can be used to find the left end of the handle in the stack.

If no precedence relation holds between a pair of terminals (indicated by a blank entry in Fig. 4.23), then a syntactic error has been detected and an error recovery routine must be invoked, as discussed later in this section. The above ideas can be formalized by the following algorithm.

Algorithm 4.5. Operator-precedence parsing algorithm.

Input. An input string w and a table of precedence relations.

Output. If w is well formed, a *skeletal* parse tree, with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.

Method. Initially, the stack contains \$ and the input buffer the string w\$. To parse, we execute the program of Fig. 4.24. □

```
(1)  set ip to point to the first symbol of w$;
(2)  repeat forever
(3)      if $ is on top of the stack and ip points to $ then
(4)          return
         else begin
(5)          let a be the topmost terminal symbol on the stack
                 and let b be the symbol pointed to by ip;
(6)          if a <· b or a ≐ b then begin
(7)              push b onto the stack;
(8)              advance ip to the next input symbol;
             end;
(9)          else if a ·> b then          /* reduce */
(10)             repeat
(11)                 pop the stack
(12)             until the top stack terminal is related by <·
                     to the terminal most recently popped
(13)         else error()
         end
```

Fig. 4.24. Operator-precedence parsing algorithm.

Operator-Precedence Relations from Associativity and Precedence

We are always free to create operator-precedence relations any way we see fit and hope that the operator-precedence parsing algorithm will work correctly when guided by them. For a language of arithmetic expressions such as that generated by grammar (4.17) we can use the following heuristic to produce a proper set of precedence relations. Note that grammar (4.17) is ambiguous, and right-sentential forms could have many handles. Our rules are designed to select the "proper" handles to reflect a given set of associativity and precedence rules for binary operators.

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$. For example, if $*$ has higher precedence than $+$, make $* \cdot> +$ and $+ <\cdot *$. These relations ensure that, in an expression of the form $E+E*E+E$, the central $E*E$ is the handle that will be reduced first.

2. If θ_1 and θ_2 are operators of equal precedence (they may in fact be the same operator), then make $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if the operators are left-associative, or make $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if they are right-associative. For example, if $+$ and $-$ are left-associative, then make $+ \cdot> +$, $+ \cdot> -$, $- \cdot> -$, and $- \cdot> +$. If $\uparrow$ is right associative, then make $\uparrow <\cdot \uparrow$. These relations ensure that $E-E+E$ will have handle $E-E$ selected and $E \uparrow E \uparrow E$ will have the last $E \uparrow E$ selected.

3. Make $\theta <\cdot$ **id**, **id** $\cdot> \theta$, $\theta <\cdot$ (, ($<\cdot \theta$,) $\cdot> \theta$, $\theta \cdot>$), $\theta \cdot>$ \$, and \$ $<\cdot \theta$ for all operators θ. Also, let

($\doteq$)	\$ $<\cdot$ (	\$ $<\cdot$ **id**
($<\cdot$ (	**id** $\cdot>$ \$	) $\cdot>$ \$
($<\cdot$ **id**	**id** $\cdot>$)	) $\cdot>$)

These rules ensure that both **id** and (E) will be reduced to E. Also, \$ serves as both the left and right endmarker, causing handles to be found between \$'s wherever possible.

Example 4.28. Figure 4.25 contains the operator-precedence relations for grammar (4.17), assuming

1. $\uparrow$ is of highest precedence and right-associative,
2. $*$ and $/$ are of next highest precedence and left-associative, and
3. $+$ and $-$ are of lowest precedence and left-associative,

(Blanks denote error entries.) The reader should try out the table to see that it works correctly, ignoring problems with unary minus for the moment. Try the table on the input **id** $*$ (**id** $\uparrow$ **id**) $-$ **id** $/$ **id**, for example. □

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig. 4.25. Operator-precedence relations.

Handling Unary Operators

If we have a unary operator such as ¬ (logical negation), which is not also a binary operator, we can incorporate it into the above scheme for creating operator-precedence relations. Supposing ¬ to be a unary prefix operator, we make $\theta <\cdot \neg$ for any operator θ, whether unary or binary. We make $\neg \cdot> \theta$ if ¬ has higher precedence than θ and $\neg <\cdot \theta$ if not. For example, if ¬ has higher precedence than &, and & is left-associative, we would group $E\&\neg E\&E$ as $(E\&(\neg E))\&E$, by these rules. The rule for unary postfix operators is analogous.

The situation changes when we have an operator like the minus sign − that is both unary prefix and binary infix. Even if we give unary and binary minus the same precedence, the table of Fig. 4.25 will fail to parse strings like **id*−id** correctly. The best approach in this case is to use the lexical analyzer to distinguish between unary and binary minus, by having it return a different token when it sees unary minus. Unfortunately, the lexical analyzer cannot use lookahead to distinguish the two; it must remember the previous token. In Fortran, for example, a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol.

Precedence Functions

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two *precedence functions* f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b,

1. $f(a) < g(b)$ whenever $a <\cdot b$,
2. $f(a) = g(b)$ whenever $a \doteq b$, and
3. $f(a) > g(b)$ whenever $a \cdot> b$.

Thus the precedence relation between a and b can be determined by a

numerical comparison between $f(a)$ and $g(b)$. Note, however, that error entries in the precedence matrix are obscured, since one of (1), (2), or (3) holds no matter what $f(a)$ and $g(b)$ are. The loss of error detection capability is generally not considered serious enough to prevent the using of precedence functions where possible; errors can still be caught when a reduction is called for and no handle can be found.

Not every table of precedence relations has precedence functions to encode it, but in practical cases the functions usually exist.

Example 4.29. The precedence table of Fig. 4.25 has the following pair of precedence functions,

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

For example, $* \lessdot \mathbf{id}$, and $f(*) < g(\mathbf{id})$. Note that $f(\mathbf{id}) > g(\mathbf{id})$ suggests that $\mathbf{id} \gtrdot \mathbf{id}$; but, in fact, no precedence relation holds between **id** and **id**. Other error entries in Fig. 4.25 are similarly replaced by one or another precedence relation. □

A simple method for finding precedence functions for a table, if such functions exist, is the following.

Algorithm 4.6. Constructing precedence functions.

Input. An operator precedence matrix.

Output. Precedence functions representing the input matrix, or an indication that none exist.

Method.

1. Create symbols f_a and g_a for each a that is a terminal or $.
2. Partition the created symbols into as many groups as possible, in such a way that if $a \doteq b$, then f_a and g_b are in the same group. Note that we may have to put symbols in the same group even if they are not related by $\doteq$. For example, if $a \doteq b$ and $c \doteq b$, then f_a and f_c must be in the same group, since they are both in the same group as g_b. If, in addition, $c \doteq d$, then f_a and g_d are in the same group even though $a \doteq d$ may not hold.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if $a \lessdot b$, place an edge from the group of g_b to the group of f_a. If $a \gtrdot b$, place an edge from the group of f_a to that of g_b. Note that an edge or path from f_a to g_b means that $f(a)$ must exceed $g(b)$; a path from g_b to f_a means that $g(b)$ must exceed $f(a)$.
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path

beginning at the group of f_a; let $g(a)$ be the length of the longest path from the group of g_a. □

Example 4.30. Consider the matrix of Fig. 4.23. There are no $\doteq$ relationships, so each symbol is in a group by itself. Figure 4.26 shows the graph constructed using Algorithm 4.6.

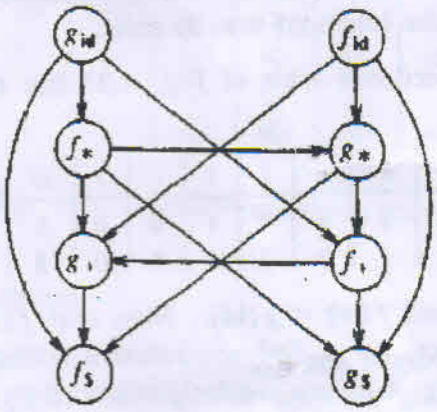


Fig. 4.26. Graph representing precedence functions.

There are no cycles, so precedence functions exist. As $f_\$$ and $g_\$$ have no out-edges, $f(\$) = g(\$) = 0$. The longest path from g_+ has length 1, so $g(+) = 1$. There is a path from g_{id} to f_* to g_* to f_+ to g_+ to $f_\$$, so $g(\mathbf{id}) = 5$. The resulting precedence functions are:

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

□

Error Recovery in Operator-Precedence Parsing

There are two points in the parsing process at which an operator-precedence parser can discover syntactic errors:

1. If no precedence relation holds between the terminal on top of the stack and the current input.[1]
2. If a handle has been found, but there is no production with this handle as a right side.

Recall that the operator-precedence parsing algorithm (Algorithm 4.5) appears to reduce handles composed of terminals only. However, while nonterminals

[1] In compilers using precedence functions to represent the precedence tables, this source of error detection may be unavailable.

are treated anonymously, they still have places held for them on the parsing stack. Thus when we talk in (2) above about a handle matching a production's right side, we mean that the terminals are the same and the positions occupied by nonterminals are the same.

We should observe that, besides (1) and (2) above, there are no other points at which errors could be detected. When scanning down the stack to find the left end of the handle in steps (10-12) of Fig. 4.24, the operator-precedence parsing algorithm, we are sure to find a $<\cdot$ relation, since \$ marks the bottom of stack and is related by $<\cdot$ to any symbol that could appear immediately above it on the stack. Note also that we never allow adjacent symbols on the stack in Fig. 4.24 unless they are related by $<\cdot$ or $\doteq$. Thus steps (10-12) must succeed in making a reduction.

Just because we find a sequence of symbols $a <\cdot b_1 \doteq b_2 \doteq \cdots \doteq b_k$ on the stack, however, does not mean that $b_1 b_2 \cdots b_k$ is the string of terminal symbols on the right side of some production. We did not check for this condition in Fig. 4.24, but we clearly can do so, and in fact we must do so if we wish to associate semantic rules with reductions. Thus we have an opportunity to detect errors in Fig. 4.24, modified at steps (10-12) to determine what production is the handle in a reduction.

Handling Errors During Reductions

We may divide the error detection and recovery routine into several pieces. One piece handles errors of type (2). For example, this routine might pop symbols off the stack just as in steps (10-12) of Fig. 4.24. However, as there is no production to reduce by, no semantic actions are taken; a diagnostic message is printed instead. To determine what the diagnostic should say, the routine handling case (2) must decide what production the right side being popped "looks like." For example, suppose *abc* is popped, and there is no production right side consisting of *a*, *b* and *c* together with zero or more nonterminals. Then we might consider if deletion of one of *a*, *b*, and *c* yields a legal right side (nonterminals omitted). For example, if there were a right side *aEcE*, we might issue the diagnostic

`illegal` *b* `on line` (line containing *b*)

We might also consider changing or inserting a terminal. Thus if *abEdc* were a right side, we might issue a diagnostic

`missing` *d* `on line` (line containing *c*)

We may also find that there is a right side with the proper sequence of terminals, but the wrong pattern of nonterminals. For example, if *abc* is popped off the stack with no intervening or surrounding nonterminals, and *abc* is not a right side but *aEbc* is, we might issue a diagnostic

`missing` *E* `on line` (line containing *b*)

Here E stands for an appropriate syntactic category represented by nonterminal E. For example, if a, b, or c is an operator, we might say "expression;" if a is a keyword like **if**, we might say "conditional."

In general, the difficulty of determining appropriate diagnostics when no legal right side is found depends upon whether there are a finite or infinite number of possible strings that could be popped in lines (10-12) of Fig. 4.24. Any such string $b_1 b_2 \cdots b_k$ must have $\doteq$ relations holding between adjacent symbols, so $b_1 \doteq b_2 \doteq \cdots \doteq b_k$. If an operator precedence table tells us that there are only a finite number of sequences of terminals related by $\doteq$, then we can handle these strings on a case-by-case basis. For each such string x we can determine in advance a minimum-distance legal right side y and issue a diagnostic implying that x was found when y was intended.

It is easy to determine all strings that could be popped from the stack in steps (10-12) of Fig. 4.24. These are evident in the directed graph whose nodes represent the terminals, with an edge from a to b if and only if $a \doteq b$. Then the possible strings are the labels of the nodes along paths in this graph. Paths consisting of a single node are possible. However, in order for a path $b_1 b_2 \cdots b_k$ to be "poppable" on some input, there must be a symbol a (possibly \$) such that $a \lessdot b_1$. Call such a b_1 *initial*. Also, there must be a symbol c (possibly \$) such that $b_k \gtrdot c$. Call b_k *final*. Only then could a reduction be called for and $b_1 b_2 \cdots b_k$ be the sequence of symbols popped. If the graph has a path from an initial to a final node containing a cycle, then there are an infinity of strings that might be popped; otherwise, there are only a finite number.

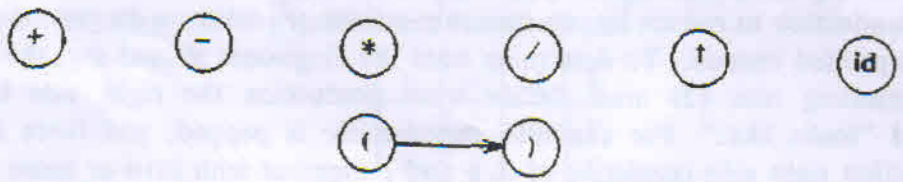


Fig. 4.27. Graph for precedence matrix of Fig. 4.25.

Example 4.31. Let us reconsider grammar (4.17):

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$$

The precedence matrix for this grammar was shown in Fig. 4.25, and its graph is given in Fig. 4.27. There is only one edge, because the only pair related by $\doteq$ is the left and right parenthesis. All but the right parenthesis are initial, and all but the left parenthesis are final. Thus the only paths from an initial to a final node are the paths $+$, $-$, $*$, $/$, **id**, and $\uparrow$ of length one, and the path from (to) of length two. There are but a finite number, and each corresponds to the terminals of some production's right side in the grammar. Thus the error checker for reductions need only check that the proper set of

nonterminal markers appears among the terminal strings being reduced. Specifically, the checker does the following:

1. If +, −, *, /, or ↑ is reduced, it checks that nonterminals appear on both sides. If not, it issues the diagnostic

```
missing operand
```

2. If **id** is reduced, it checks that there is no nonterminal to the right or left. If there is, it can warn

```
missing operator
```

3. If ( ) is reduced, it checks that there is a nonterminal between the parentheses. If not, it can say

```
no expression between parentheses
```

Also it must check that no nonterminal appears on either side of the parentheses. If one does, it issues the same diagnostic as in (2). □

If there are an infinity of strings that may be popped, error messages cannot be tabulated on a case-by-case basis. We might use a general routine to determine whether some production right side is close (say distance 1 or 2, where distance is measured in terms of tokens, rather than characters, inserted, deleted, or changed) to the popped string and if so, issue a specific diagnostic on the assumption that that production was intended. If no production is close to the popped string, we can issue a general diagnostic to the effect that "something is wrong in the current line."

Handling Shift/Reduce Errors

We must now discuss the other way in which the operator-precedence parser detects errors. When consulting the precedence matrix to decide whether to shift or reduce (lines (6) and (9) of Fig. 4.24), we may find that no relation holds between the top stack symbol and the first input symbol. For example, suppose a and b are the two top stack symbols (b is at the top), c and d are the next two input symbols, and there is no precedence relation between b and c. To recover, we must modify the stack, input or both. We may change symbols, insert symbols onto the input or stack, or delete symbols from the input or stack. If we insert or change, we must be careful that we do not get into an infinite loop, where, for example, we perpetually insert symbols at the beginning of the input without being able to reduce or to shift any of the inserted symbols.

One approach that will assure us no infinite loops is to guarantee that after recovery the current input symbol can be shifted (if the current input is \$, guarantee that no symbol is placed on the input, and the stack is eventually shortened). For example, given ab on the stack and cd on the input, if $a \lessdot c$[2]

[2] We use $\lessdot$ to mean $<\cdot$ or $\doteq$.

we might pop b from the stack. Another choice is to delete c from the input if $b \lessdot d$. A third choice is to find a symbol e such that $b \lessdot e \lessdot c$ and insert e in front of c on the input. More generally, we might insert a string of symbols such that

$$b \lessdot e_1 \lessdot e_2 \lessdot \cdots \lessdot e_n \lessdot c$$

if a single symbol for insertion could not be found. The exact action chosen should reflect the compiler designer's intuition regarding what error is likely in each case.

For each blank entry in the precedence matrix we must specify an error-recovery routine; the same routine could be used in several places. Then when the parser consults the entry for a and b in step (6) of Fig. 4.24, and no precedence relation holds between a and b, it finds a pointer to the error-recovery routine for this error.

Example 4.32. Consider the precedence matrix of Fig. 4.25 again. In Fig. 4.28, we show the rows and columns of this matrix that have one or more blank entries, and we have filled in these blanks with the names of error handling routines.

	id	(	)	$
id	e3	e3	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	e4
)	e3	e3	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	e2	e1

Fig. 4.28. Operator-precedence matrix with error entries.

The substance of these error handling routines is as follows:

e1: /* called when whole expression is missing */
insert **id** onto the input
issue diagnostic: "`missing operand`"

e2: /* called when expression begins with a right parenthesis */
delete) from the input
issue diagnostic: "`unbalanced right parenthesis`"

e3: /* called when **id** or) is followed by **id** or (*/
insert + onto the input
issue diagnostic: "`missing operator`"

e4: /* called when expression ends with a left parenthesis */
pop (from the stack
issue diagnostic: "`missing right parenthesis`"

Let us consider how this error-handling mechanism would treat the

erroneous input **id** +). The first actions taken by the parser are to shift **id**, reduce it to E (we again use E for anonymous nonterminals on the stack), and then to shift the +. We now have configuration

STACK	INPUT
$E +	)$

Since + $\cdot>$) a reduction is called for, and the handle is +. The error checker for reductions is required to inspect for E's to left and right. Finding one missing, it issues the diagnostic

```
missing operand
```

and does the reduction anyway.

Our configuration is now

$E	)$

There is no precedence relation between $ and), and the entry in Fig. 4.28 for this pair of symbols is e2. Routine e2 causes diagnostic

```
unbalanced right parenthesis
```

to be printed and removes the right parenthesis from the input. We are now left with the final configuration for the parser.

$E	$	□

4.7 LR PARSERS

This section presents an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, k is assumed to be 1. LR parsing is attractive for a variety of reasons.

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

handle - substring that match the right side of the prod

Shift Reduce Process:

Stack	RSF	Action
$	000111	shift 01
$01	0011	reduce S→01
$S	0011	

27/04/18: Conflict in shift reduce parsing:

1. Shift Reduce conflict — example:
2. Reduce-Reduce conflict

↓

2 productions with same production on the right

S→id
P→id

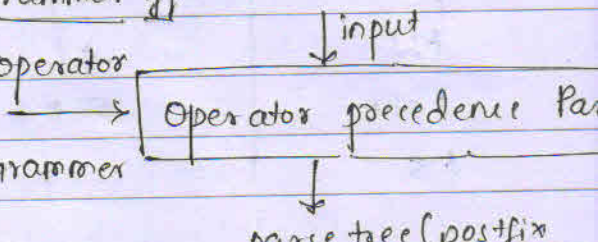
Stack	RSF	Action
$	i E + s e s $	
$ i E + s	e s $	→ shift else
$ S		reduce by

} Parser doesn't know either id should be reduced to S or P.

Operator precedence Parser:

Grammer G is operator grammer iff:

i) No ε production
ii) No two adjacent non-terminals.



Ex: E → EAE | id } not a OG
A → * | +

E → E+E | E*E | id ✓ OG.

Steps for operator precedence parsing:

Problems:

1. Check whether the given grammer is operator grammer or not, if possible try to convert.
2. Generate operator relation table
3. Parse the input string
4. Construct the parse tree.

PROBLEM:

1. Construct the operator precedence parser for the given grammer and parse the given input string.

$E \rightarrow EAE \mid id$

$A \rightarrow + \mid *$

(i) Converting to operator grammer

$E \rightarrow E+E \mid E*E \mid id.$

(ii) Operator Relation Table:

Assumption: identifier – highest precedence

* – left associative (·>) Right associative (<·)

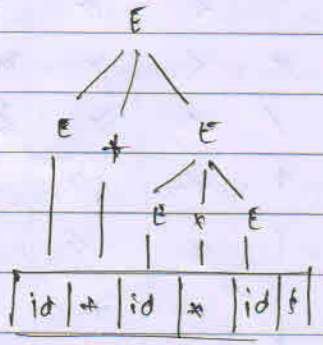
\+ – left associative.

$ – least precedence

	id	+	*	$
id	–	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	–

Accept.

(iv)



(iii) Parse the input string.

Stack	input	Relation	Action
$	id+id*id$	<·	push id
$id	+id*id$	·>	pop id
$	+id*id$	<·	push +
$+	id*id$	<·	push id
$+id	*id$	·>	pop id
$+	*id$	<·	push *
$+*	id$	<·	push id
$+*id	$	·>	pop id
$+*	$	·>	pop *
$+	$	·>	pop +
$	$	–	Accept

28/04/18

9. $E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid id$

input: id * (id ↑ id) - id / id

i) It is an OG

ii) Generate relation table

id - highest precedence + - → left associative

( ) - right associative $ - least precedence

↑ - right associative

*/ - left associative.

	id	+	-	*	/	↑	(	)	$
id	-	⋗	⋗	⋗	⋗	⋗	-	⋗	⋗
+	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
-	⋖	⋗	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
/	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
↑	⋖	⋗	⋗	⋗	⋗	⋖	⋖	⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	-
)	-	⋗	⋗	⋗	⋗	⋗	-	⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖	-	Accept

iii) Parse input:

Stack	input	Relation	Action
$	id * (id ↑ id) - id / id $	⋖	push id
$ id	* (id ↑ id) - id / id $	⋗	pop id
$	* (id ↑ id) - id / id $	⋖	push *
$ *	(id ↑ id) - id / id $	⋖	push (
$ * (	id ↑ id) - id / id $	⋖	push id
$ * (id	↑ id) - id / id $	⋗	pop id
$ * (	↑ id) - id / id $	⋖	push ↑
$ * (↑	id) - id / id $	⋖	push id
$ * (↑ id	) - id / id $	⋗	pop id
$ * (↑	) - id / id $	⋗	pop ↑
$ * (	) - id / id $	≐	push)
$ * ( )	- id / id $	⋗	pop), (
$ *	- id / id $	⋗	pop *

$-	id / id $	<·	push id
$-id	/ id $	·>	pop id
$ -	/ id $	<·	push /
$-/	id $	<·	push id
$-/ id	$	·>	pop id
$-/	$	·>	pop /
$-	$	·>	pop -
$	$	-	Accept

Algorithm: Operator precedence parsing algorithm:

Input: An input string w and a table of precedence relations.

Output: If w is well formed, a skeletal parse tree, with a placeholder non-terminal E labelling all interior nodes otherwise, an error indication.

Method: Initially the stack contains $ and the input buffer the input buffer the string w$. To parse, we execute the program.

1. Set input to point to the first symbol of w$
2. repeat forever
3. if $ is on top of stack and ip points to $ then
4. return
5. Let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by ip
6. if a <· b or a ·= b then begin
7. push b onto the stack
8. advance ip to next input symbol
   end;
9. else if a ·> b then
10. repeat
11. pop the stack over any of ·=
12. until the top stack terminal is related by <· to the terminal most recently popped
13. else error()

30/04/18

3. $S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

i/p : (a, (a,a))

a - highest (- left

$ least , - left

i) It is an OG

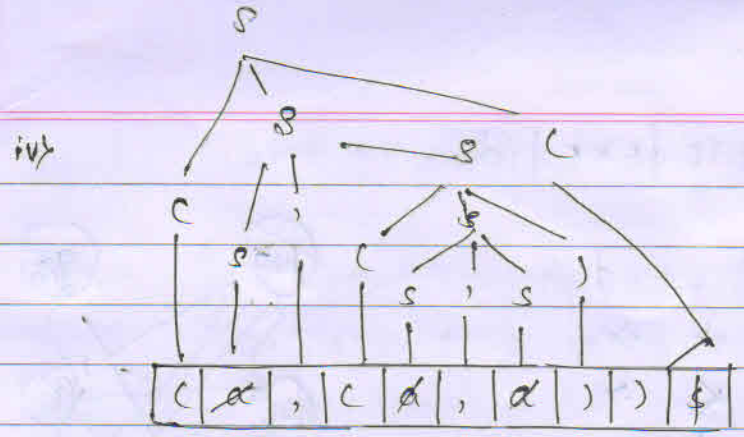
ii) Relation table:

	id	,	(	)	$
id	-	⋗	-	⋗	⋗
,	⋖	-	⋖	⋗	⋗
(	⋖	⋖	⋖	≐	-
)	-	⋗	-	⋗	⋗
$	⋖	⋖	⋖	-	Accept

Note: Do remember to put the relation operator based on associativity i.e which one is evaluated first. After parsing we get the parse tree

iii) Input is (a, (a,a))$

Stack	Input	Relation	Action
$	(a,(a,a))$	⋖	Push (
$(	a, (a,a))$	⋖	Push a
$(a	, (a,a))$	⋗	Pop a
$(	, (a,a))$	⋖	push ,
$(,	(a,a))$	⋖	push (
$(,(	a,a))$	⋖	push a
$(,(a	,a))$	⋗	pop a
$(,(	,a))$	⋖	~~pop~~ push ,
$(,(,	a))$	⋖	~~push~~ push a
$(,(,a	))$	⋗	~~pop a~~ pop
$(,(,	))$	⋗	~~pop ,~~ pop
$(,(	))$	≐	~~push)~~ push
$(,()	)$	⋗	~~pop ()~~ pop
$(,	)$	⋗	pop ,
$(	)$	≐	push)
$()	$	⋗	pop ()
$	$	Accept	



Drawbacks of operator relation table:

→ It is very difficult to handle tokens like '-' which has two precedence functions based on whether it is unary operator or binary operator.

→ Only small class of grammars can be passed

→ If we ever have 4 operators, then the no. of entries in the table are 4×4 = 16 entries i.e in general, if the number of operators are n, we need $O(n^2)$ entries. To overcome this we go for operator precedence functions.

Operator precedence functions:

- The passers doesnot store relation table instead they make use of precedence functions which map the terminal symbols to integers.
- It uses two functions i.e f_a and g_b for the symbols a and b.

(i) if $a \doteq b$, then there is an arrow (edge) from function f_a to function g_b

(ii) if $a \lessdot b$, then there is an edge from g_b to f_a

(iii) If $a \doteq b$ then $f_a = g_b$ are in the same group. Note that even if they are not related by $\doteq$ directly we group them together for example if $a \doteq b$ and $c \doteq b$ then f_a and f_c are in the same group, since they are both in the same group as g_b

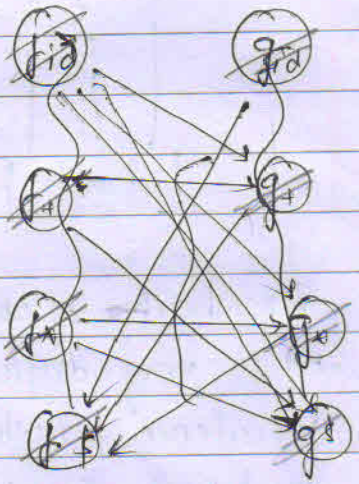
(iv) If the graph constructed has no cycle, then the precedence functions exists.

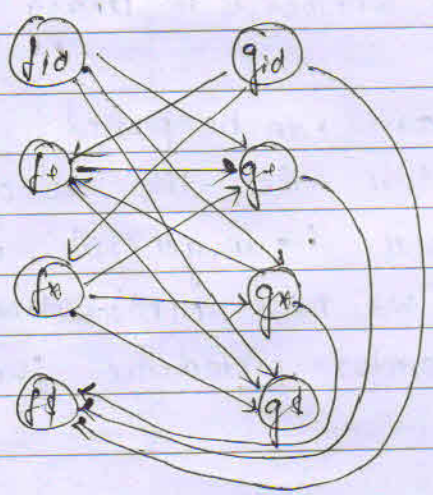
(v) Find the longest path in the function starting from terminal to $ i.e $f(a)$ to $ and $g(a)$ to $. using these

Drawbacks of operator relation table

Example: $E \to E+E \mid E*E \mid id$

	id	+	*	$
id	-	.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	-





The longest path is

fid → g* → f+ → g+ → f$

gid → f* → g* → f+ → g+ → f$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Advantages: lesser entries

disadvantages: For blank entries of relation table, we get non-blank entries in function table i.e we can't make out the errors during passing

H.W Construct an operator precedence parser for the given gram and parse an input string

(i) $E \to E+E \mid E*E \mid (E) \mid id$ ip: (id + id * id): it is an OG

(ii) $E \to E+T \mid T$

$T \to T*V \mid V$ input: a + b * c * d.

$V \to a \mid b \mid c \mid d$

(ii) i) It is an OG

ii) Relation table

	id	+	*	$
id	-	.>	.>	.>
+	<.	.>	<.	.>

input a + b * c * d $

Stack	Input	Relation	Action
$	a+b*c*d$	<·	push a
$a	+b*c*d$	·>	pop a
$	+b*c*d$	<·	push +
$+	b*c*d$	<·	push b
$+b	*c*d$	·>	pop b
$+	*c*d$	<·	push *
$+*	c*d$	<·	push c
$+*c	*d$	·>	pop c
$+*	*d$	·>	pop *
$+	*d$	<·	push *
$+*	d$	<·	push d
$+*d	$	·>	pop d
$+*	$	·>	pop *
$+	$	·>	pop +
$	$	Accept	

(i)

Relation table	id	+	*	(	)	$
id	-	·>	·>	-	·>	·>
+	<·	·>	<·	<·	·>	·>
*	<·	·>	·>	<·	·>	·>
(	<·	<·	<·	<·	=	-
)	-	·>	·>	-	·>	·>
$	<·	<·	<·	<·	-	Accept

Stack	input	Relation	Action
$	(id+id*id)$	<·	push (
$(	id+id*id)$	<·	push id
$(id	+id*id)$	·>	pop id
$(	+id*id)$	<·	push +
$(+	id*id)$	<·	push id
$(+id	*id)$	·>	pop id
$(+	*id)$	<·	push *
$(+*	id)$	<·	push id
$(+*id	)$	·>	pop id
$(+*	)$	·>	pop *
$(+	)$	·>	pop +
$(	)$	=	push)
$()	$	·>	pop
$	$	Accept	