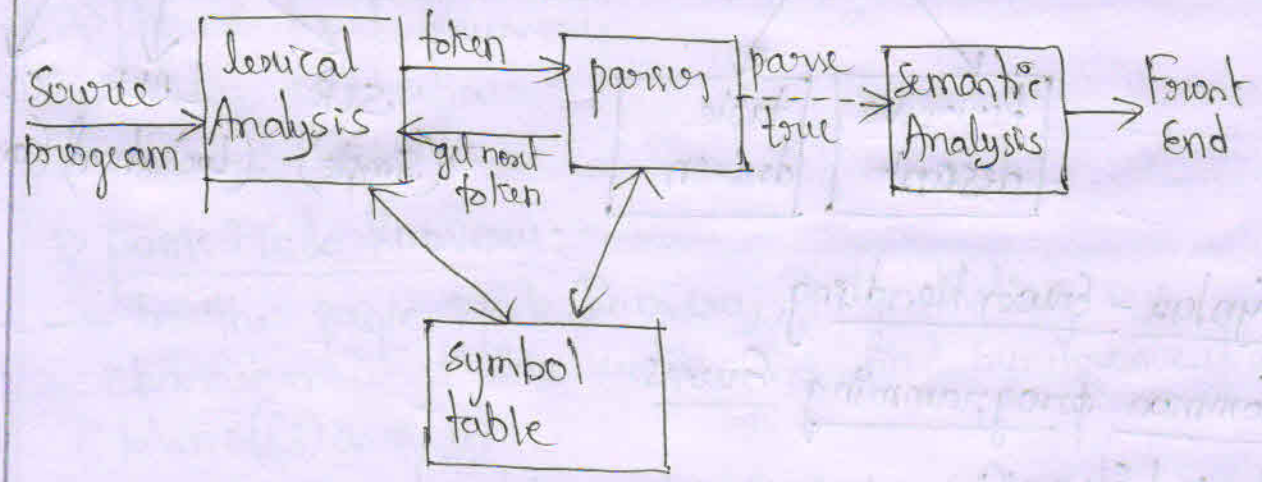
UNIT-2 Syntax Analysis -1

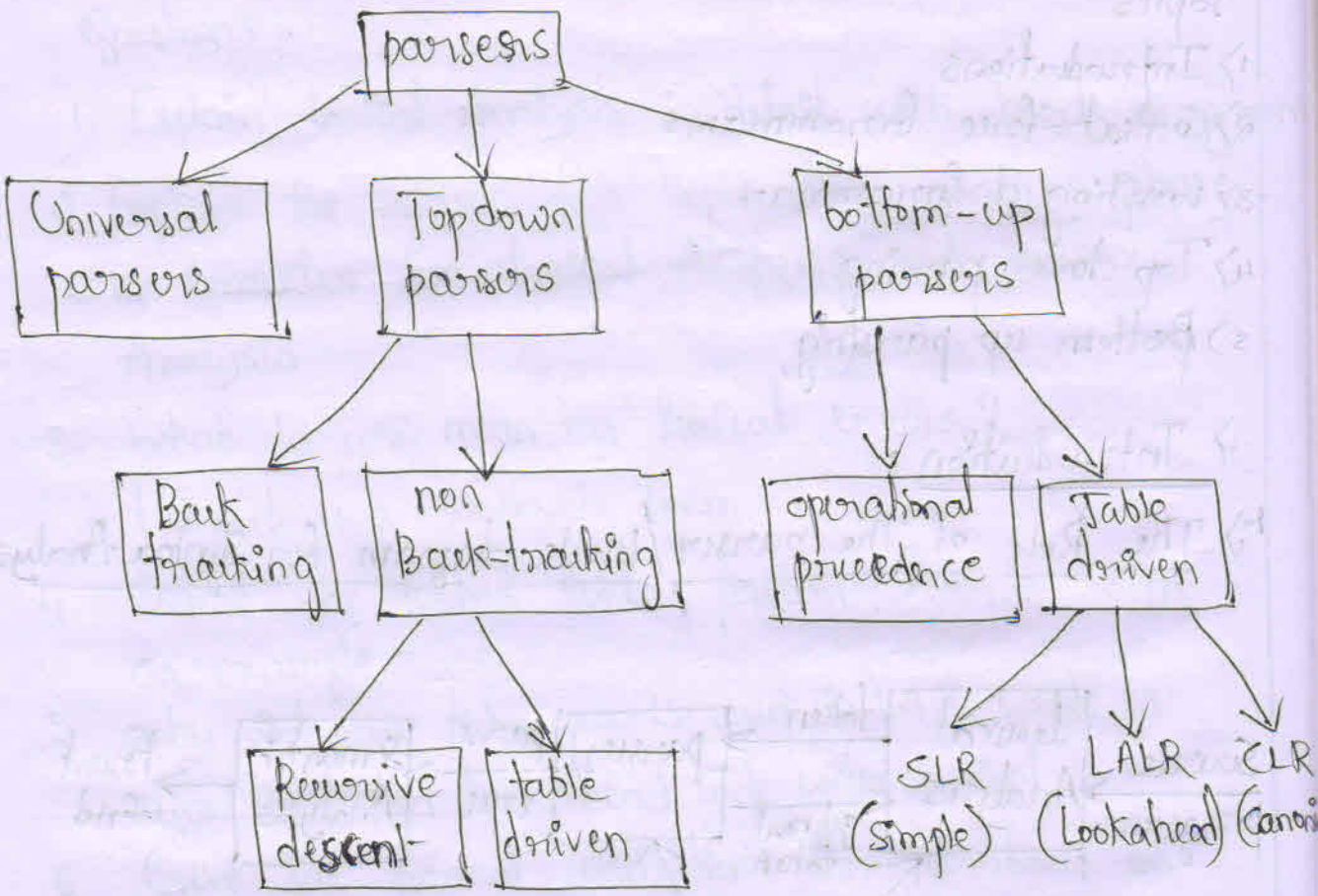
Topics
1) Introductions
2) Context-free Grammars
3) writing a Grammar
4) Top down parsing
5) Bottom up parsing

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 b/n 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 untill 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: difined 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 CFG and RE

CFG	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 parantheis and so, on.	2. Usefull for describing the structure of construct / lexical construct such as Identifiers, keywords etc
3. CFG's are Combined using pushdown automata	3. Regular Expressions are Combined using Finite Automata
4. CFG 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. CFG are more powerfull	6. RE are less powerfull as compared to CFG
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 | bterm

   bterm $\rightarrow$ bterm and bfactor | bfactor

   bfactor $\rightarrow$ not bfactor | (bexpr) | true | false

   $\Rightarrow$ not (true | 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) $S \rightarrow SS+ \mid SS* \mid a \Rightarrow aa+a*$

$S \xrightarrow{lm} SS*$ $\qquad$ $S \xrightarrow{rm} SS*$

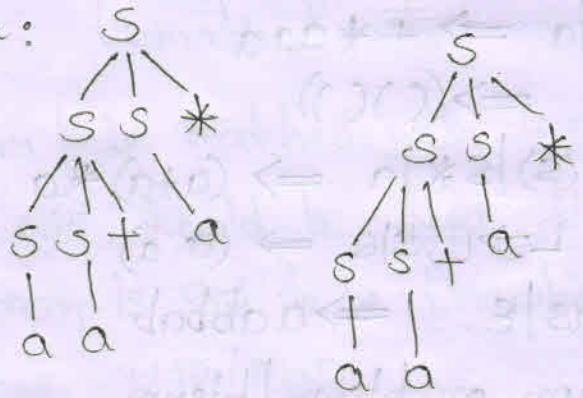
$\Rightarrow SS+S*$ $\qquad$ $\Rightarrow Sa*$

$\Rightarrow as+S*$ $\qquad$ $\Rightarrow SS+a*$

$\Rightarrow aa+S*$ $\qquad$ $\Rightarrow Sa+a*$

$\Rightarrow aa+a*$ $\qquad$ $\Rightarrow aa+a*$

parse tree:



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

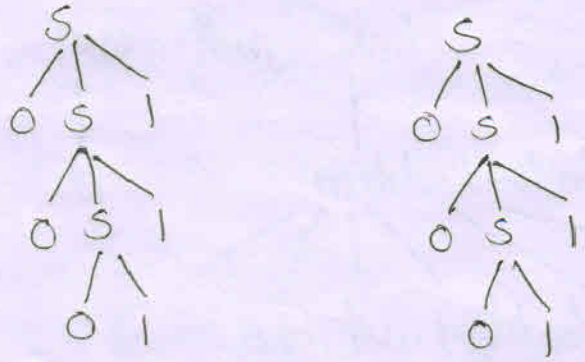
2) $S \rightarrow 0S1 \mid 01 \Rightarrow 000111$

$S \xrightarrow{lm} 0\underline{S}1$ $\qquad$ $S \xrightarrow{rm} 0S1$

$\Rightarrow 00\underline{S}11$ $\qquad$ $\Rightarrow 00S11$

$\Rightarrow 00S111$ $\qquad$ $\Rightarrow 000111$

parse tree:



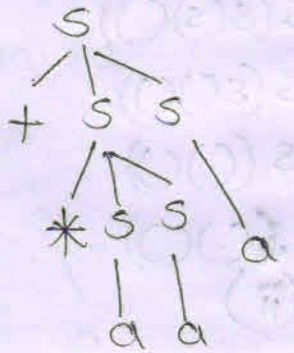
The Grammar Is Unambigous
because It has only one LMD and Only one RMD

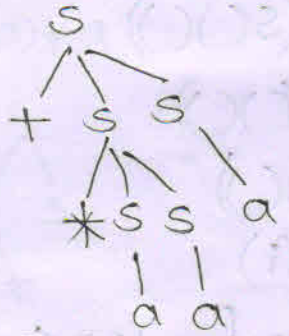
3)

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

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

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





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

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

$S \xrightarrow{lm} S(S)S$
$\Longrightarrow (\underline{S})S$
$\Longrightarrow (S(S)S)S$
$\Longrightarrow (\underline{S})S)S$
$\Longrightarrow ()S(S)S)S$
$\Longrightarrow ()(S)S)S$
$\Longrightarrow ()()S)S$
$\Longrightarrow ()()S)$
$\Longrightarrow ()()$
(i)

$S \xrightarrow{rm} \underline{S}(S)S$
$\xrightarrow{lm} (S)S$
$\xrightarrow{lm} (\underline{S}(S)S)S$
$\xrightarrow{lm} (S(S)S(S)S)S$
$\Longrightarrow ((S)S(S)S)S$
$\Longrightarrow ()S(S)S)S$
$\Longrightarrow ()(S)S)S$
$\Longrightarrow ()()S)S$
$\Longrightarrow ()()S)$
$\Longrightarrow ()()$
(ii)

RMD:

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

$\Longrightarrow S(S)$

$\Longrightarrow S(S(S)S)$

$\Longrightarrow S(S(S)S(S))$

$\Longrightarrow S(S(S)S())$

$\Longrightarrow S(S(S)())$

$\Longrightarrow S(S()())$

$\Longrightarrow S(()())$

$\Longrightarrow (()())$

(i)

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

$\Longrightarrow S(S)$

$\Longrightarrow S(S(S)S)$

$\Longrightarrow S(S(S))$

$\Longrightarrow S(\underline{S}())$

$\Longrightarrow S(S(S)S())$

$\Longrightarrow S(S(S)())$

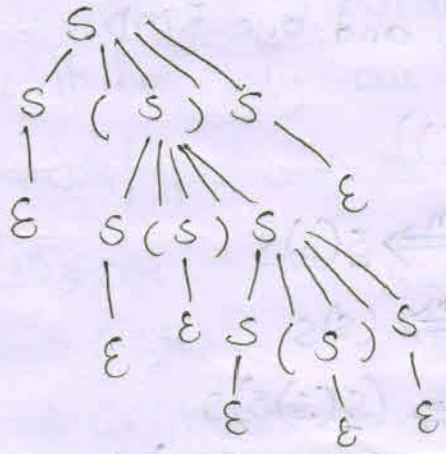
$\Longrightarrow S(S()())$

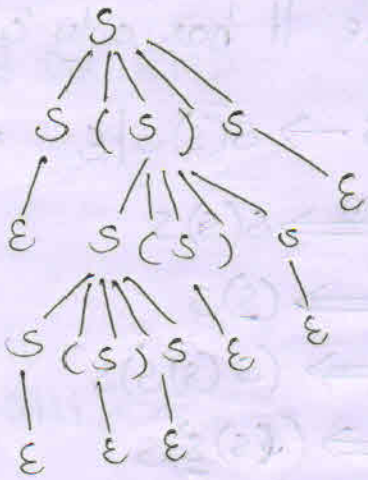
$\Longrightarrow S(()())$

$\Longrightarrow (()())$

(ii)

parse tree for LMD (i) and (ii)



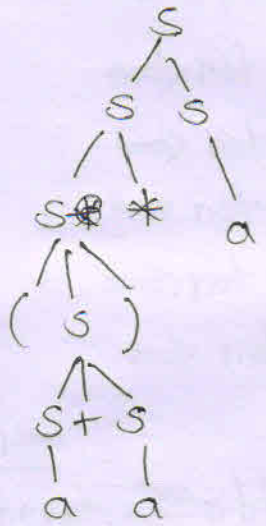


The Grammar is ambigous
Since It has 2 LMD and 2 RMD

5) $S \to S+S \mid SS \mid (S) \mid S* \mid a \Rightarrow (a+a)*a$

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

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

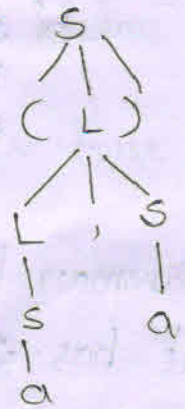


The Grammer Is Unambigous
because It has only one RMD and LMD

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

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

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

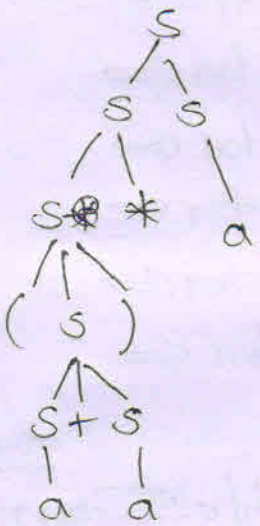


The Grammar Is Unambigous
because It has only one LMD and RMD

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

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

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



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

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

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

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

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

7) $S \rightarrow asbS \mid bsaS \mid \varepsilon \Rightarrow aabbab$

LMD:

$S \xrightarrow{lm} a\underline{S}bS$
$\Rightarrow aa\underline{S}bSbS$
$\Rightarrow aab\underline{S}bS$
$\Rightarrow aabbsaSbS$
$\Rightarrow aabbaSbS$
$\Rightarrow aabbabS$
$\Rightarrow aabbab$

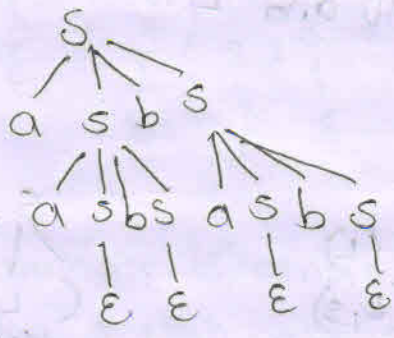
$S \xrightarrow{lm} asbS$
$\Rightarrow aaSbSbS$
$\Rightarrow aabSbS$
$\Rightarrow aabsbS$
$\Rightarrow aabbaSbS$
$\Rightarrow aabbabS$
$\Rightarrow aabbab$

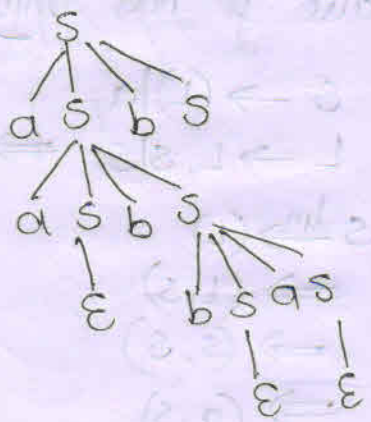
RMD:

$S \xrightarrow{rm} aSbS$
$S \Rightarrow asb$
$\Rightarrow aasbSb$
$\Rightarrow aasbbSaSb$
$\Rightarrow aasbbSab$
$\Rightarrow aaSbbSab$
$\Rightarrow aasbbab$
$\Rightarrow aabbab$

$S \xrightarrow{rm} asbS$
$\xrightarrow{rm} aSbasbS$
$\xrightarrow{rm} asbaSb$
$\Rightarrow asbab$
$\Rightarrow aasbSbab$
$\Rightarrow aasbbab$
$\Rightarrow aabbab$

parse tree:





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

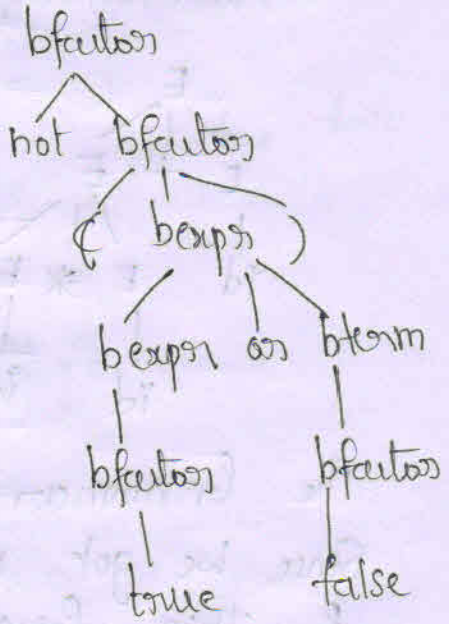
8) bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false
not (true or false)

LMD:

bexpr $\xrightarrow{lm}$ bterm
⟹ bfactor
⟹ not bfactor
⟹ not (bexpr)
⟹ not (bexpr or bterm)
⟹ not (bterm or bterm)
⟹ not (bfactor or bterm)
⟹ not (true or bterm)
⟹ not (true or bfactor)
⟹ not (true or false)

RMD:

bexpr $\xrightarrow{rm}$ bterm
⟹ bfactor
⟹ not bfactor
⟹ not (bexpr)
⟹ not (bexpr or bterm)
⟹ not (bexpr or bfactor)
⟹ not (bexpr or false)
⟹ not (bterm or false)
⟹ not (bfactor or false)
⟹ not (true or false)



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

$\Rightarrow id + id * id$

LMD

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

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

$E \xrightarrow{lm} E*E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

RMD

$E \xrightarrow{rm} E+E$

$\Rightarrow E+E*E$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

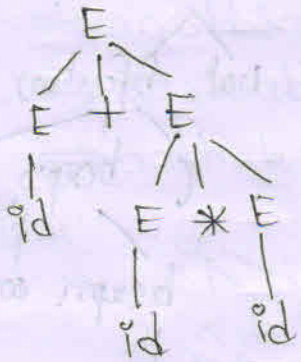
$E \xrightarrow{rm} E*E$

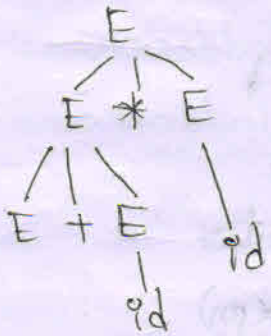
$\Rightarrow E*id$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$





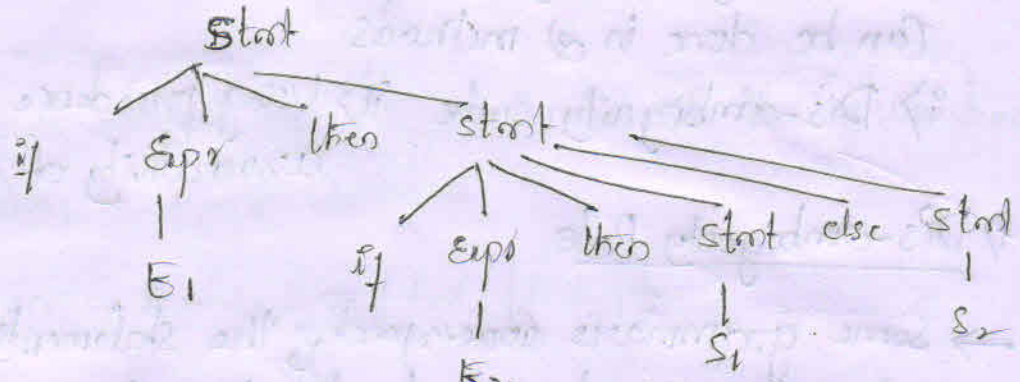
The Grammar is Ambigous

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

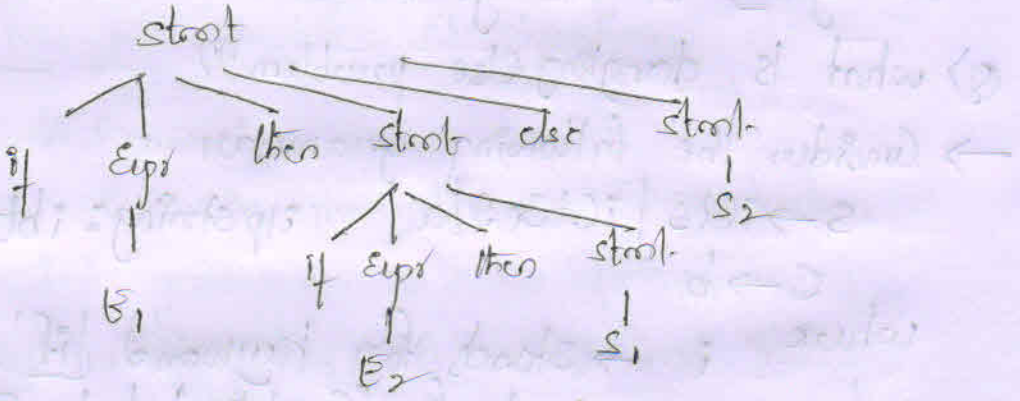
10. $S \rightarrow iEtS \mid iEtSeS \mid a \qquad E \rightarrow b$

$\Rightarrow$ 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

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

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??

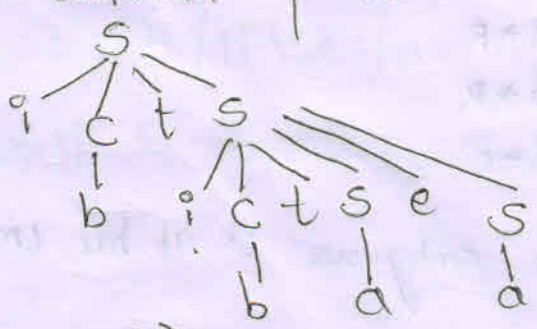
→ Consider the following grammar

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

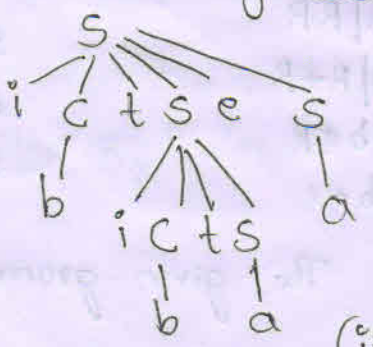
$C \rightarrow b$

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 storing 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:

Step 1) 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:

$M \rightarrow iCtMeM$

Step 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 ⇒ $U \rightarrow iCtS$

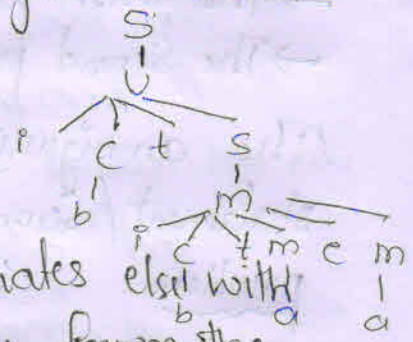
b) If-else statement where the statement before else is matched and statement after else is Unmatched.
∴ The equivalent production is: $U \rightarrow iCtMeU$

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

<u>Eliminating ambiguity using precedence and Associativity</u>

This method is explained using the following example:

eg: Q) 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 precedence 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

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

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

<u>Step3</u>: 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:

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

<u>Step4</u>: 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:

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

Step5: 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:

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

Step6: 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 associativative and ^ operator have the highest priority and are right associativity and ^ operator has precedence in between and It is left associativity.

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

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

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

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

Since there are three levels of precedence 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:

$F \rightarrow (E) | id$

Step 3: The next highest priority operators are + and / they are right associative, So, the production must start from the nonterminal 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:

$$\cancel{P \rightarrow P \wedge T \mid T} \quad E \rightarrow E+P \mid E-P \mid P$$

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

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

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

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

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

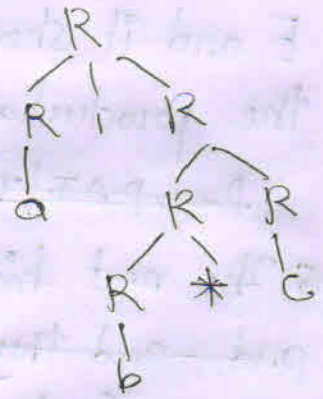
or

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

parse tree:



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

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

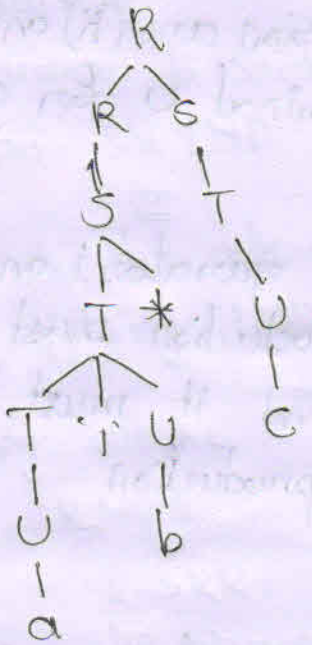
$R \rightarrow RS|S$

$S \rightarrow T*|T$

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

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

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: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \beta$

$$\Downarrow$$

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$$

Algorithm for left Recursion Elimination

Algorithm Left_Recursion

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

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

Method: Apply the algorithm to G. 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 $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ 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. $E \rightarrow \underbrace{E}_{A}\underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

$\Downarrow$

$E \rightarrow TE'$

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

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

2. $T \rightarrow \underbrace{T}_{A}\underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$ $\Rightarrow$ $\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}}$

$\Downarrow$

$T \rightarrow FT'$

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

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

$\Downarrow$

$S \rightarrow S'$

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

4. $S \rightarrow SS+ \mid SS* \mid a$

$\Downarrow$

$S \rightarrow aS'$

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

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

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$\Downarrow$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Left factoring

General form: $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 --- | \alpha\beta_n | \gamma$

$$\Downarrow$$

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 --- | \beta_n$$

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

$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | --- | \alpha\beta_n | \gamma$,

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

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | --- | \beta_n$$

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

$\Downarrow$

$S \to SSS' \mid a$

$S' \to + \mid *$

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. ~~$S \to 0S' \mid \varepsilon$~~ $S \to \underbrace{0S}_{\alpha}\underbrace{1}_{\beta_1} \mid \underbrace{0}_{\alpha}\underbrace{1}_{\beta_2}$

$\Downarrow$

$S \to 0S' \mid \varepsilon$

$S' \to S1 \mid 1$

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

$E \to b$

$\Downarrow$

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

$\Downarrow$

$S \to iEtss' \mid a$

$S' \to \varepsilon \mid es$

$E \to b$

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 derives 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.

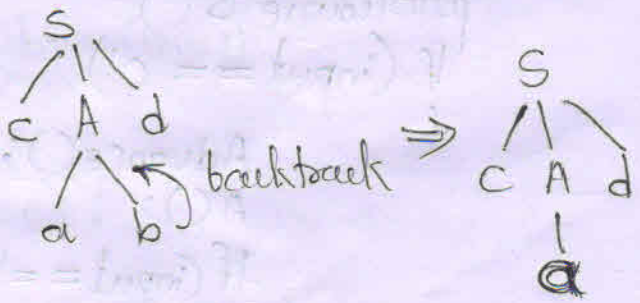
- Top-down parser
  - Backtracking
  - Nonback-tracking
    - Recursive descent
    - table driven

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) <u>Recursive descent parser</u>:

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

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

$S \rightarrow cAd$

$A \rightarrow ab|a$

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 == 'd')
        {
```

```
        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);
    }
```

iscave:

Saves the i/p pointer position before each alternate production to facilitate backtracking whenever a terminal is encountered the i/p 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 i/p 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: $A \rightarrow Abla$

i/p → abb

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

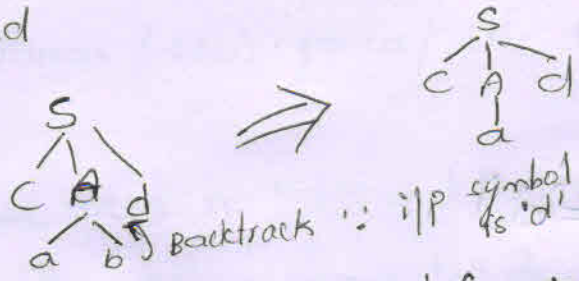
2. Backtracking: It occurs where there is more than one alternate In the production to be tried while parsing the i/p string

$S \rightarrow cAd$

$A \rightarrow abla$ i/p: cabd

no backtr

i/p string : cad



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

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

<u>Example</u>: Recursive descent-

Q) Write a recursive descent parser for the following grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$ i/p: id+id*id

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

$F \rightarrow (E) \mid id$

→ → 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

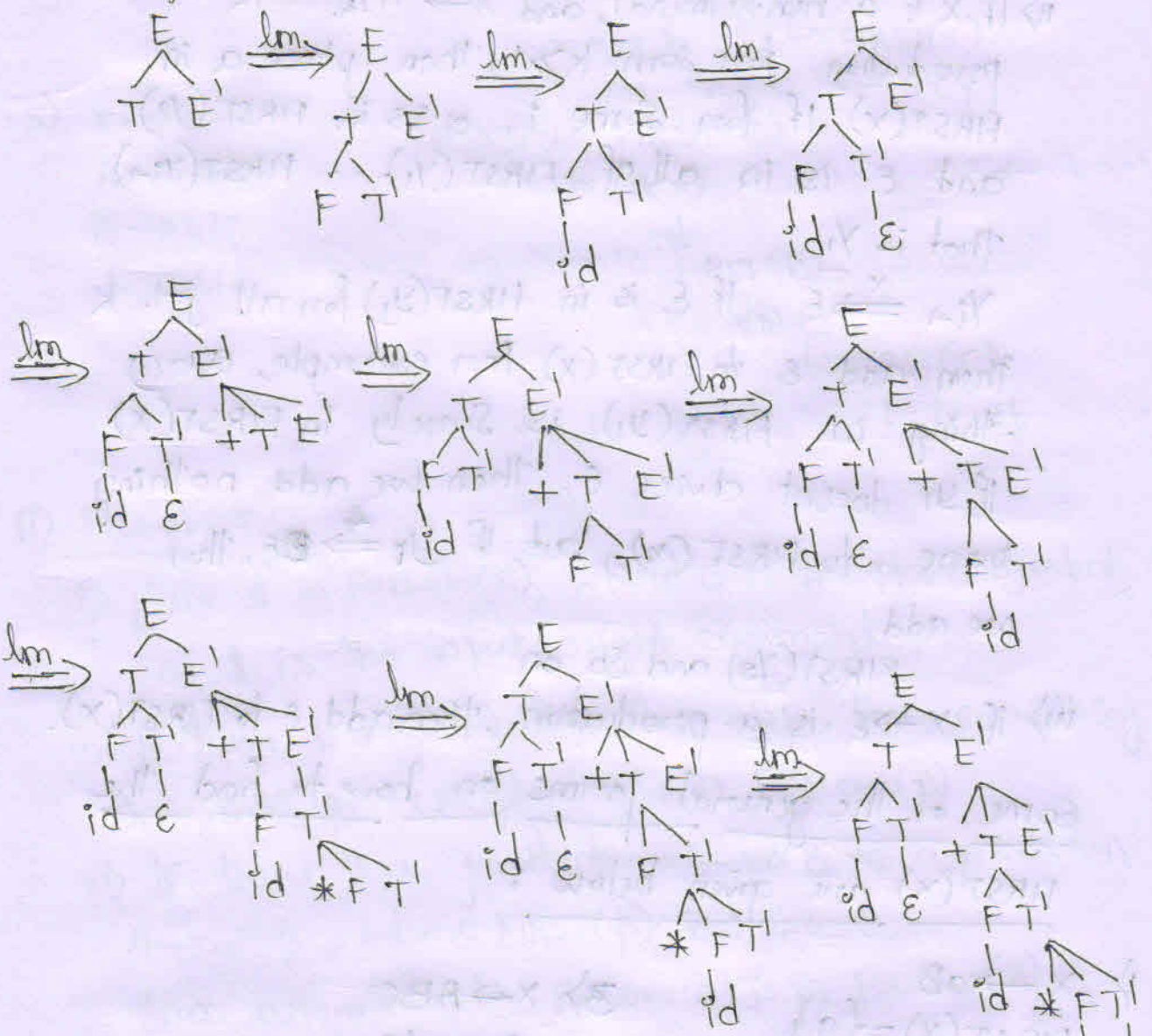
→ predictive parser is a recursive decent parser, which ~~production~~ has the capability to predict which production is to be used to replace the i/p string

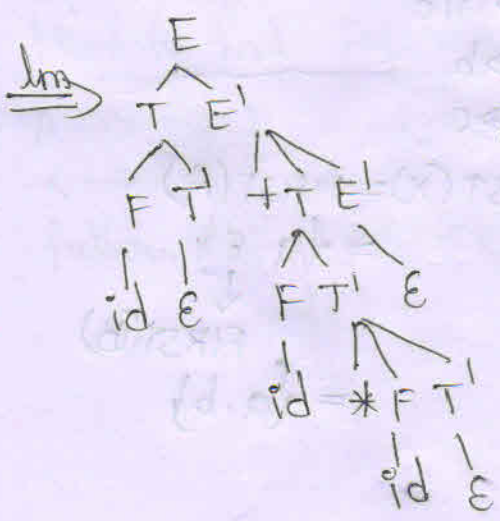
→ The predictive parser does not suffer from backtracking

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

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

$FIRST(Y_2)$ and so on

iii) If $X \rightarrow \varepsilon$ 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) $X \rightarrow aB$

$FIRST(X) = \{a\}$

$X \rightarrow \varepsilon$

$FIRST(X) = \varepsilon$

2) $X \rightarrow ABC$

$A \rightarrow a$

$B \rightarrow b$

$C \rightarrow c$

$FIRST(X) = FIRST(A)$

$= \{a\}$

3) $X \rightarrow ABC$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b$

$C \rightarrow c$

$FIRST(X) = FIRST(A)$

$= \{a, \varepsilon\}$

$\downarrow$

$FIRST(B)$

$= \{a, b\}$

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

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

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

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

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

ii) <u>FOLLOW</u> :

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

<u>How to find FOLLOW</u> :

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

follow(B) = {FIRST(CD)}
= {C}

2) $A \to \underbrace{X}_{\alpha} \underbrace{B}_{B} \underbrace{CD}_{\beta}$

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

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

$C \rightarrow c \mid \varepsilon$

$D \rightarrow d \mid \varepsilon$

$FOLLOW(B) = first(\beta)$

$= first(CD)$

$= \{c, d\} + FOLLOW(A)$

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

$fallow(B) = FALLOW(\text{left most nonterminal})$

$= FALLOW(A)$

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

$C \rightarrow c \mid \varepsilon$ $\quad \alpha \; B \; \beta$ — ②

$D \rightarrow d$

$FALLOW(B) = FIRST(CDBe)$

$= (C, D)$ — ①

$+$

$FALLOW(B) = FIRST(e)$

$= \{e\}$

$\therefore FAL(B) = ① + ②$

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

Find the FIRST and FOLLOW set for the following grammars

1. $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

2. $S \rightarrow iEts \mid iEtses \mid a$

$E \rightarrow b$

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

~~4) $D \to EF$~~
~~$E \to g \mid \varepsilon$~~ → 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 stmt_sequence'
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) $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

here $\text{FOLLOW}(E) = F \rightarrow \underset{\alpha}{(}\underset{B}{E}\underset{\beta}{)}$

$\text{FOLLOW}(E) = \text{FIRST}(\,)$

$= \{\,)\,\}$

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

$\text{FOL}(E') = \text{FIRST}(\beta)$ $\qquad$ $\text{FOL}(E') = \text{FIRST}(\beta)$

$\downarrow \varepsilon$ $\qquad$ $\downarrow \varepsilon$

$\therefore \text{FOL}(E') = \text{FOL}(E)$ $\qquad$ $\therefore \text{FOL}(E') = \text{FOL}(E$

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

$= \text{FIRST}(\beta)$ $\qquad$ $= \text{FIRST}(\beta)$

$= \text{FIRST}(E')$ $\qquad$ $= \text{FIRST}(E')$

$= \{+\}$ $\qquad$ $= \{+\}$

<u>Note</u>: we should not write <u>ε</u> in the <u>FOLLOW</u> set-

2) $S \to iEts \mid iEtses \mid a$

$E \to b$

	S	E
FIRST	i a	b
FOLLOW	\$ e	t

$FOLLOW(S) = S \to iEt\underset{\beta}{s}$ $\quad S \to \underbrace{iEt}_{\alpha}\underbrace{ses}_{\beta}$

$FOLLOW(S) = FIRST(\beta)$

$= FOLLOW(S)$ ε

$FOLLOW(S) = FIRST(\beta)$

$= FIRST(es)$

$= \{e\}$

3) $S \to ,GH;$

$G \to aF$

$F \to bF \mid \varepsilon$ — Epsilon

$H \to KL$

$K \to m \mid \varepsilon$

$L \to n \mid \varepsilon$

	S	G	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 b	a b

7) $S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$

	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	; ε	S
FOLLOW	$	$	; $

1) $S \to aSbS \mid bSaS \mid \varepsilon$

	S
FIRST	a
	b
	ε
Follow	$\$$
	a
	b

$$\text{Follow}(S) \Longrightarrow S \to \underbrace{a}_{\alpha}\underbrace{S}_{B}\underbrace{bS}_{\beta} \qquad S \to \underbrace{b}_{\alpha}\underbrace{S}_{B}\underbrace{aS}_{\beta} \qquad \cancel{S \to \varepsilon}$$

$$= \text{FIRST}(\beta) \qquad = \text{FIRST}(\beta)$$

$$= \text{FIRST}(bS) \qquad = \text{FIRST}(aS)$$

$$\text{Follow}(S) = \{b\} \qquad = \{a\}$$

$$\text{FOL}(S) \to \underbrace{aSb}_{\alpha}\underbrace{S}_{B}\underbrace{\ }_{\beta} \qquad \text{Fol}(S) \to \underbrace{bSa}_{\alpha}\underbrace{S}_{B}\underbrace{\ }_{\beta}$$

$$= \text{FIRST}(\beta) \to \varepsilon \qquad = \text{FIRST}(\beta) \to \varepsilon$$

$$\text{FOL}(S) = \text{FOL}(S) \qquad \text{FOL}(S) = \text{FOL}(S)$$

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 wheather 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;
}
```

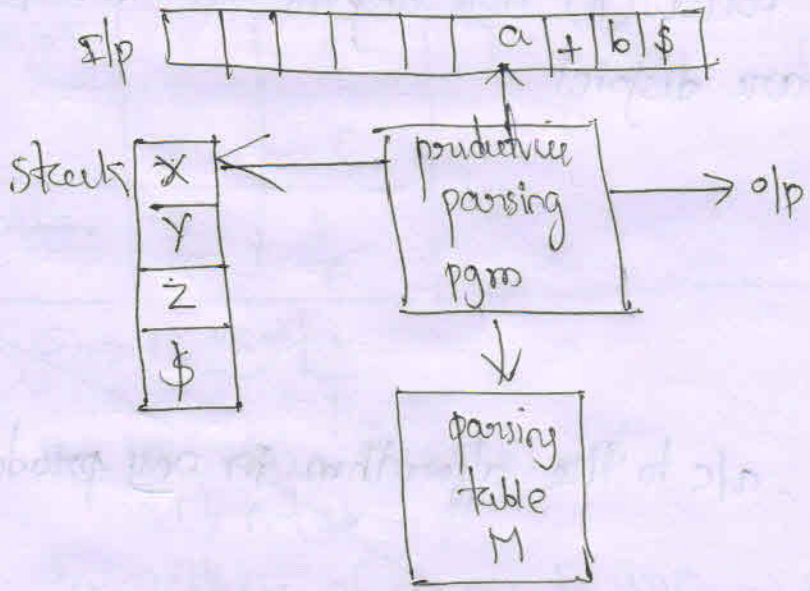


fig: model of a table driven predictive parser

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

A grammar is LL(1) iff whenever, $A \rightarrow \alpha | \beta$ 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) $\rightarrow \varepsilon$ or FIRST(β) $\rightarrow \varepsilon$ 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 $A \rightarrow \underset{\alpha}{aB} | \underset{\beta}{ac}$

FIRST(α) = {a}

FIRST(β) = {a}

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

eg: $A \rightarrow aB | bC$

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 a	e ε	b
FOLLOW	\$ e	\$ e	t

$S \to \overset{\alpha}{iEtss'} \mid \overset{\beta}{a}$

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

b. neither of α or β are ε

~~1. $S \to \underbrace{iEtss'}_{\alpha} \mid \underbrace{a}_{\beta}$~~

~~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 $FIRST(\alpha) \cap FOLLOW(A) = \phi$~~
~~$FIRST(es) \cap FOLLOW(S') = \phi$~~
~~$\{e\} \cap \{\$, e\} \neq 0$~~

~~$\therefore$ The given grammar is not LL(1).~~ Condition fails

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

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

$\therefore$ 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

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

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

$\therefore$ The given grammar is not LL(1)

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

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

$\Downarrow$ left factoring

final production $\begin{cases} S \to aS' \\ S' \to SS'' \mid \varepsilon \\ S \to +S' \mid *S' \end{cases}$

	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

   $\therefore$ The grammar is LL(1)

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

$\Rightarrow$

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

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

i) $E \rightarrow TE'$

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

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) $T \rightarrow FT'$

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

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

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

i/p: id+id*id

i) Remove left Recursion

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

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

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'
$E'\$$	$\$$	$T' \to \varepsilon$
$\$$	$\$$	$E' \to \varepsilon$

$\therefore$ The grammar is accepted the i/p string

i.e, The i/p is passed successfully

The grammar is LL(1) $\because$ no multiple entries

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

$E \rightarrow b$

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

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

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		$S \rightarrow iEtSS'$				
S'	$S' \rightarrow \varepsilon$		$S' \rightarrow eS$ $S' \rightarrow \varepsilon$			
E				$E \rightarrow b$		

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

v)

Stack	Input	Action
S $	~~iEtEtS1eS2~~ ibtibtaea $	
iEtSS' $	ibtibtaea $	$S \rightarrow iEtSS'$
EtSS' $	btibtaea $	matched 'i'
btSS' $	btibtaea $	$E \rightarrow b$

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'

$\hookrightarrow$ ambiguity wheather to push $s' \to es$ or $s' \to \varepsilon$

$\Rightarrow$ i/p: If E then S else S
ibtaea

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

$\hookrightarrow$ ambiguity whether to push $s' \to es$ or $s' \to \varepsilon$

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

i) Remove left recursion

$S \to aS'$

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

ii) Remove left factoring

$S \to aS'$

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

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

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 i/p 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
$+s'\$$	$+a*\$$	$s'' \to +s'$
$s'\$$	$a*\$$	match '+'
$ss''\$$	$a*\$$	push $s' \to ss''$
$as's''\$$	$a*\$$	push $s \to as'$
$s's''\$$	$*\$$	mpatch 'a'
$s''\$$	$*\$$	push $s' \to \varepsilon$
$*s'\$$	$*\$$	push $s'' \to *s'$
$s'\$$	$\$$	match *
$\$$	$\$$	push $s' \to \varepsilon$

The i/p string is Successfully parsed

i/p string: 000111

4) $S \to 0S1 | 01$

i) Remove left recursion

→ not needed

ii) Remove left factoring

$S \to 0S'$

$S' \to S1 | 1$

iii) FIRST and FOLLOW Set

	S	$
FIRST	0	0
		1
FOLLOW	$	$
	1	1

iv) Construct the predictive parsing table

	$	0	1
S		$S \to 0S'$	
S'		$S' \to S1$	$S' \to 1$

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

v) parse the input string

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

The input string is Successfully parsed

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

i) Remove left recursion ⇒ not required

ii) Remove left factoring ⇒ Not required

iii) Construct FIRST and FOLLOW set

	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: $+*aaa$

Stack	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, (, *

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

i/p: (a+a)*a

i) Remove left Recursion

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

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

iv) Construct the predictive parsing table

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

The grammar is not LL(1)

v) input : (a+a)*a

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

→ Ambigous, whether to parse S'→+SS' or S'→ε

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

i) Remove left Recursion

$S \rightarrow \varepsilon S'$

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

ii) Remove left factoring

→ not needed

iii) find FIRST and FOLLOW set

	S	S'
FIRST	(ε	(ε
FOLLOW	$) (	$) (

iv) parsing table

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

→ multiple production

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

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

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

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

step i) Remove left Recursion

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

$L \to SL'$

$L' \to ,SL' \mid \varepsilon$

$S \to (L) \mid a$

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	$S \to (L)$	$S \to a$		
L	$L \to SL'$	$L \to SL'$		
L'		$L' \to \epsilon$	$L' \to ,SL$	

v)

Stack	Input	action
$S\$$	$(a,a)\$$	
$(L)\$$	$(a,a)\$$	$S \to (L)$
$L)\$$	$a,a)\$$	match (
$SL')\$$	$a,a)\$$	$L \to SL'$
$aL')\$$	$a,a)\$$	$S \to a$
$L')\$$	$,a)\$$	match a
$,SL')\$$	$,a)\$$	$L' \to ,SL'$
$SL')\$$	$a)\$$	match ,
$aL')\$$	$a)\$$	$S \to a$
$L')\$$	$)\$$	match a
$)\$$	$)\$$	$L' \to \epsilon$
$\$$	$\$$	match)

The grammar is successfully parsed

8) $S \to S+S \mid SS \mid (S) \mid S* \mid a$ $\Rightarrow (a+a)*a$

i) Remove left recursion

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

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

$S' \to \cancel{(S)S' \mid aS'}\ +SS' \mid SS' \mid *S' \mid \epsilon$

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→*S' S→ε			S'→+SS'		

∴ 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→ε	S→bSaS 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

10) 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 → ~~bfactor bterm'~~ bfactor bterm'
bterm' → and bfactor bterm' | ε
bfactor → not bfactor | (bexpr) | true | false

ii) No left ~~recur~~ 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	bterm bexpr'	bterm bexpr'				bterm bexpr'	bterm bexpr'	
bexpr'			ε	or bterm bexpr				ε
bterm	bfactor bterm'	bfactor bterm'				bfactor bterm'	bfactor bterm'	
bterm'			ε	ε	and bfactor bterm			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)
bfactor bterm' bexpr' $	(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'
bfactor bterm' bexpr) bterm' bexpr' $	) $	bfactor → ε
bterm' bexpr) bterm' bexpr' $	$	bterm' → ε & bexpr' → ε & match)
bterm' bexpr' $ $	$	

Error recovery in predictive parser:

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

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

Example:

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

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

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

predictive parsing table

	(	id	)	+	*	$
E	$E \to TE'$	$E \to TE'$	Synch			
E'			$E' \to \varepsilon$	$E' \to +TE'$		$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$	~~$T \to \varepsilon$~~ Synch	~~$T \to *FT'$~~ Synch		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
$E\$$	$id*+id\$$	$E \to TE'$
$TE'\$$	$id*+id\$$	$T \to FT'$
$FT'E'\$$	$id*+id\$$	$F \to id$
$idT'E'\$$	$id*+id\$$	match id
$T'E'\$$	$*+id\$$	$T' \to *FT'$
$*FT'E'\$$	$*+id\$$	match *
$FT'E'\$$	$+id\$$	$[F,+]$ = Synch, remove NT from stack
$T'E'\$$	$+id\$$	$T' \to \varepsilon$
$E'\$$	$+id\$$	$E' \to +TE'$
$+TE'\$$	$+id\$$	match +
$TE'\$$	$id\$$	$T \to FT'$
$FT'E'\$$	$id\$$	$F \to id$
$idT'E'\$$	$id\$$	match id
$T'E'\$$	$\$$	$T' \to \varepsilon$
$E'\$$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	match $

Q) Show that the following grammar is ambigous.
$E \to E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid id$ and i/p: $id+id*id$
Give an unambigous grammar such that precedence order from lowest or highest are +, −, *, /, ( ), id and all are left-to right associative.