UNIT-2
SYNTAX ANALYZER

Context free Grammar :-

A context-free grammar consists of sets of non terminals, terminals, set of productions, a single start symbol from the set of non-terminals. A context-free grammar 'G' is described as

$$G = (V_N, T, P, S)$$

Nonterminal → (VUT)*

→ N → finite set of nonterminal symbol
T → finite set of terminal symbol
P → finite set of productions

Eg:- Consider the following grammar

A → BcD
B → c/d
D → d

N = {A, B, D}
T = {c, d}

P:
A → BcD
B → c/d
D → d

S : {A}

→ **Derivations :-**

Derivations can be made into two ways depending upon the non-terminals to be replaced. they are

i) Left Most Derivation (LMD)
ii) Right Most Derivation (RMD)

i) **LMD:-**

In this LMD, a production is applied to the left most non-terminal, then the derivation is called LMD.

ii) RMD:-

In this RMD, a production is applied to the Rightmost non-terminal then the derivation is called RMD

Problem

i) Consider the grammar $E \rightarrow E+E \mid E*E \mid (E) \mid E \mid id$ and derive the string id+id*id.

LMD

$E \rightarrow \underline{E}+E$

$E \rightarrow id+\underline{E}$

$E \rightarrow id+\underline{E}*E$

$E \rightarrow id+id*\underline{E}$

$E \rightarrow id+id*id$

RMD

$E \rightarrow E*\underline{E}$

$E \rightarrow \underline{E}*id$

$E \rightarrow E+\underline{E}*id$

$E \rightarrow \underline{E}+id*id$

$E \rightarrow id+id*id.$

ii) Consider the grammar $E \rightarrow E+E \mid E*E \mid a \mid b$ & derive the string a+a*b

LMD

$E \rightarrow \underline{E}+E$

$E \rightarrow a+\underline{E}$

$E \rightarrow a+\underline{E}*E$

$E \rightarrow a+a*\underline{E}$

$E \rightarrow a+a*b$

RMD

$E \rightarrow E*\underline{E}$

$E \rightarrow \underline{E}*b$

$E \rightarrow E+\underline{E}*b$

$E \rightarrow \underline{E}+a*b$

$E \rightarrow a+a*b.$

→ Ambiguity:-

A Grammar 'G' is said to 'ambiguious if it generates more than one parse tree for the Sentence in given grammar

Eg:- Show that the following grammar is ambigious

$S \rightarrow aS \mid Sa \mid a$ and derive the string aaa

LMD

$S \rightarrow \underline{S}a$

$S \rightarrow \underline{S}aa$

$S \rightarrow aaa$

RMD

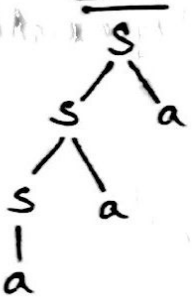
$S \rightarrow a\underline{S}$

$S \rightarrow aa\underline{S}$

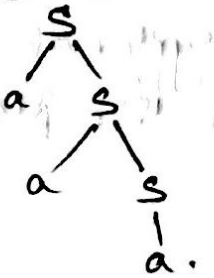
$S \rightarrow aaa$

for the string 'aaa' two different derivations trees are possible so it is ambigious grammar

LMD



RMD



ii) Show that the grammar is ambigious $S \rightarrow OS1 | SS | \epsilon$

→ string 01

LMD

$S \rightarrow OS1$

$S \rightarrow O\epsilon1$

$S \rightarrow 01$

RMD

$S \rightarrow SS$

$S \rightarrow SOS1$

$S \rightarrow \epsilon OS1$

$S \rightarrow 01$

Role of Parser :-

Introduction to Syntax Analysis | Parser :-

→ The Syntax analysis is 2nd phase in compilation

→ The Syntax analyzer (parser) basically checks for the Syntax of language

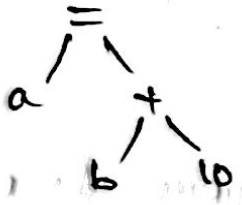
Definition of Parser :-

A parsing or Syntax Analysis is a process which takes input string "w" and produces either a parse tree (syntatic structure) or generates the syntatic errors

Eg :- a = b + 10

→ The above programming statement is first given to lexical Analyzer. The lexical Analyzer will divide it into group of tokens.

→ The syntax analyzer takes the tokens as input and generates a tree like structure called parse tree



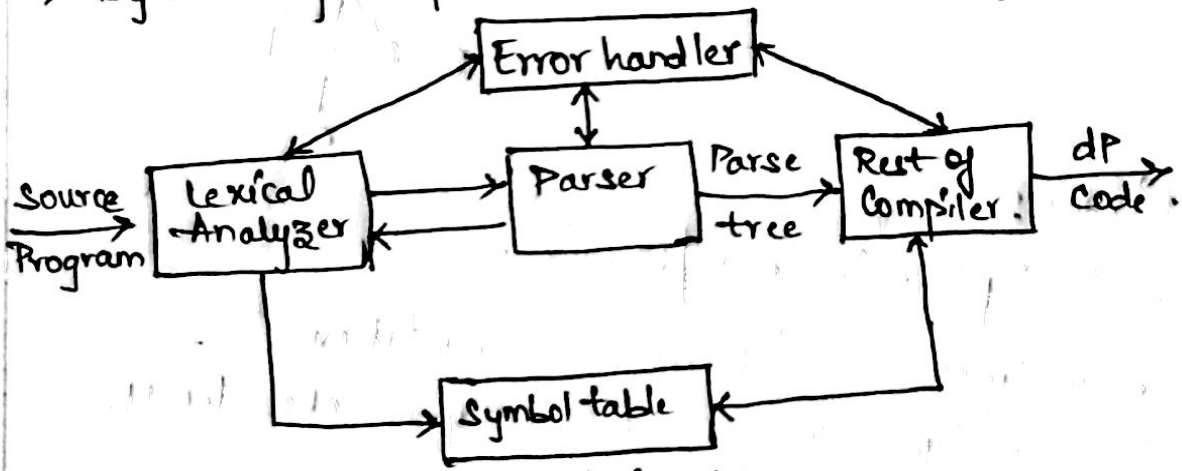
Role of Parser:

In this process of compilation the parser & lexical analyzer works together i.e when parser requires string of tokens it invokes lexical Analyzer

→ The lexical Analyzer supplies tokens to syntax Analyzer (Parser)

→ The parser collects sufficient no. of tokens and builds the Parse tree

→ By building the parse tree Parser find the Syntatic errors



Classification of Parsing Techniques:-

The parsing methods commonly used in compilers are classified as

1) top-down * 2) bottom-up parsing

→ The Top-down parsers built the parse tree from the root node. (top) to the leaves (bottom)

→ Bottom-up parsers built the parse tree from the leaves to the root node.

→ In both the cases the input to the parser is scanned from left to right.

Parse Tree :-

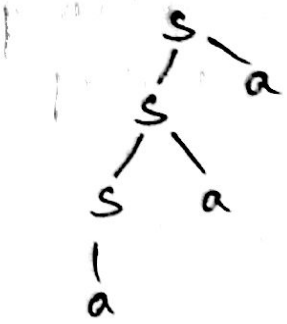
Parse Tree is the graphical representation for a derivation, the nodes of a parse tree are describe as follows:

→ The root is labelled with the start symbol of the grammar 'S'.
→ Interior nodes are labelled with a non-terminal
→ The children of a node are related to their parent by some Production in 'P'. The parent is the left hand side symbols of the production. The children are right hand side of the production and follow left to right ordering of production.
→ leaves are labled with a terminal or elipson
→ A left most derivation corresponds to a preorder traversal of a parsetree
→ A Right most derivation corresponds to post order traversal of a parse tree in reverse.

$S \rightarrow aS|Sa|a$

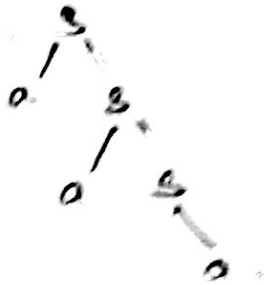
LMD

$S \rightarrow Sa$
$S \rightarrow Saa$
$S \rightarrow aaa$



RMD

$S \rightarrow aS$
$S \rightarrow aaS$
$S \rightarrow aaa$



Brute-Force Approach :-

Top down parsing with full backup is a brute force method of parsing. In general terms, this methods operate as follows.

→ Given a particular non-terminal that is to be expanded, the first production for this non terminal is applied.

→ Within this newly Expanded string, the next (left most) non-terminal is selected for Expansion and its first production is applied.

→ This step 2 is repeated for all subsequent non-terminals that are selected until such time as this step2 cannot or shouldnot be continued.

→ This termination may be due to two reasons :

1. No more non-terminals may be present, in which string is Successfully passed.
2. It may result from an incorrect Expansion which does not match the appropriate segment of the Source string

→ Illustrate brute-force technique through a suitable Example.

Sol:- Consider the following

$S \rightarrow aAd \mid aB$

$A \rightarrow b \mid c$

$B \rightarrow ccd \mid ddc$

where 'S' is start symbol

→ The following are the sequence of syntax trees generated during the parse of the string 'accd' by using brute force parsing technique.

Step1:- start with start Symbol i.e 'S' in given grammar

S

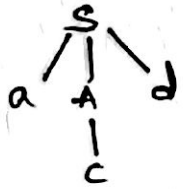
Step2:- Select the 1st production for 'S'

Symbol 'a' is matched with string to be parsed.

Step3:- choose the 1st production for A then we get

their is mismatch b/w the second symbol 'c' of the input string and the second symbol 'b' in the sentential form 'abd'

Step4:-
At this point we must backup the previous production application for 'A' must be deleted & replaced with next choice

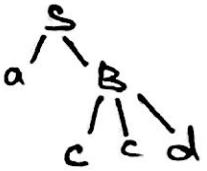


Now the left most two characters of the given input string is matched.

Step5:-
When the third symbol 'c' of the input string is compared with the last symbol 'd' of the current sentential form again a mismatch occurs.

The previously choosen production for 'A' must be deleted. Since there are no more rules for 'A' which can be selected we must also delete a production for 'S' and next production of S is selected

Step6:- now replaced 'B' with 1st production



∴ The given input string is obtained.

LEFT RECURSION:-

If there is a production $A \to A\alpha | \beta$ then to eliminate the left recursion in given productions replace with following productions.

$$A \to \beta A'$$
$$A' \to \alpha A' | \epsilon$$

Eg:- Consider the grammar.

$$E \to E+T | T$$
$$T \to T*F | F$$
$$F \to (E) | id$$

eliminate the left recursion.

Sol:- Non-terminals = (E, T, F)

Here the non-terminals E & T has left recursion.

consider,

$$\underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+T} | \underset{\beta}{T}$$
$$E \to T E'$$
$$E' \to +TE' | \epsilon$$

consider,

$$\underset{A}{T} \to \underset{A}{T} \underset{\alpha}{*F} | \underset{\beta}{F}$$
$$T \to FT'$$
$$T' \to *FT' | \epsilon$$

After eliminating the left recursion, the grammar becomes

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

LEFT FACTORING:-

If there are productions $A \to \alpha\beta_1 | \alpha\beta_2$ replace with

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2$

Eg:- Consider the following grammar.

$A \to aAB | aA | a$

$B \to bB | b$

consider,

$A \to \underset{\alpha}{a}\underset{\beta_1}{AB} | \underset{\alpha}{a}\underset{\beta_2}{A} | \underset{\alpha}{a}$ $\quad \beta_3$

$A \to aA'$

$A' \to AB | A | \epsilon$

consider,

$B \to \underset{\alpha}{b}\underset{\beta_1}{B} | \underset{\alpha}{b}$ $\quad \beta_2$

$B \to bB'$

$B' \to B | \epsilon$

∴ Now the grammar becomes

$A \to aA'$

$A' \to AB | A | \epsilon$

$B \to bB'$

$B' \to B | \epsilon$

RECURSIVE DESCENT PARSER:-

→ A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent (RD) Parser.

→ In this type of parser the CFG is used to build the recursive routines.

Basic steps for construction of RD parser:-

1. If the input symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.
2. If the input symbol is terminal then pointer has to be advanced on matching of the input symbol.
3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Construct Recursive descent parser for the following grammar.

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to F* \mid a \mid b$

First eliminate left recursion in the given productions.

$A \to A\alpha \mid \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

consider, $\underset{A}{E} \to \underset{A}{E}\underset{\alpha}{+T} \mid \underset{\beta}{T}$

$E \to T\,E'$

$\underset{A}{T} \to \underset{A}{T}\underset{\alpha}{*F} \mid \underset{\beta}{F}$

$T \to FT'$

$E' \rightarrow +TE' \mid \epsilon$ $\quad\quad$ $T' \rightarrow *FT' \mid \epsilon$

consider $F \rightarrow F* \mid a$ $\quad\quad$ consider, $F \rightarrow F* \mid b$

$F \rightarrow aF'$ $\quad\quad$ $F \rightarrow bF'$

$F' \rightarrow *F' \mid \epsilon$ $\quad\quad$ $F' \rightarrow *F' \mid \epsilon$

After eliminating the left recursion, the grammar becomes

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow aF' \mid bF'$

$F' \rightarrow *F' \mid \epsilon$

```
procedure E()
  begin
    T();
    E prime();
  end
procedure Eprime()
  begin
  if input symbol = '+' then
  begin
      Advance();
      T();
```

```
    E prime ();
   end
   else NULL
  end
procedure T()
 begin
  F();
  T prime();
 end
procedure T prime()
  begin
   if input symbol='*' then
   begin
     Advance();
      F();
     T prime();
   end
   else NULL
 end
procedure F()
  begin
   if input symbol='a' then
   begin
      Advance();
       F prime();
   end
   else if input symbol='b' then
```

```
    begin
        Advance();
        F prime();
    end
    end
Procedure F prime()
  begin
  if input symbol = '*' then
   begin
     Advance();
     F prime();
   end
   else NULL
end.
```

Construct Recursive descent Parser for the following grammar

$E \rightarrow T + E / T$

$T \rightarrow V * T / V$

$V \rightarrow id$

```
Ed:- procedure E()
        begin
          T();
          if input symbol = '+' then
          begin
```

```
        Advance()
          E();
      end
    end
procedure T()
    begin
      V();
      if input symbol = '*' then
      begin
        Advance();
          T();
      end
    end
procedure V()
    begin
      if input symbol = 'id' then
      begin
        Advance();
      end
    end.
```

PREDICTIVE PARSING (OR) NON RECURSIVE PREDICTIVE PARSER- The predictive parser has an input, a stack, a parsing table and an output. The input contains the string to be parsed, followed by $, the right end marker. The stack contains a sequence of grammar

symbols, preceded by $, the bottom-of-stack marker. Initially, the stack contains the start symbol of the grammar preceded by $. The parsing table is a two-dimensional array M[A, a], where A is a non-terminal, and a is a terminal or the symbol $.

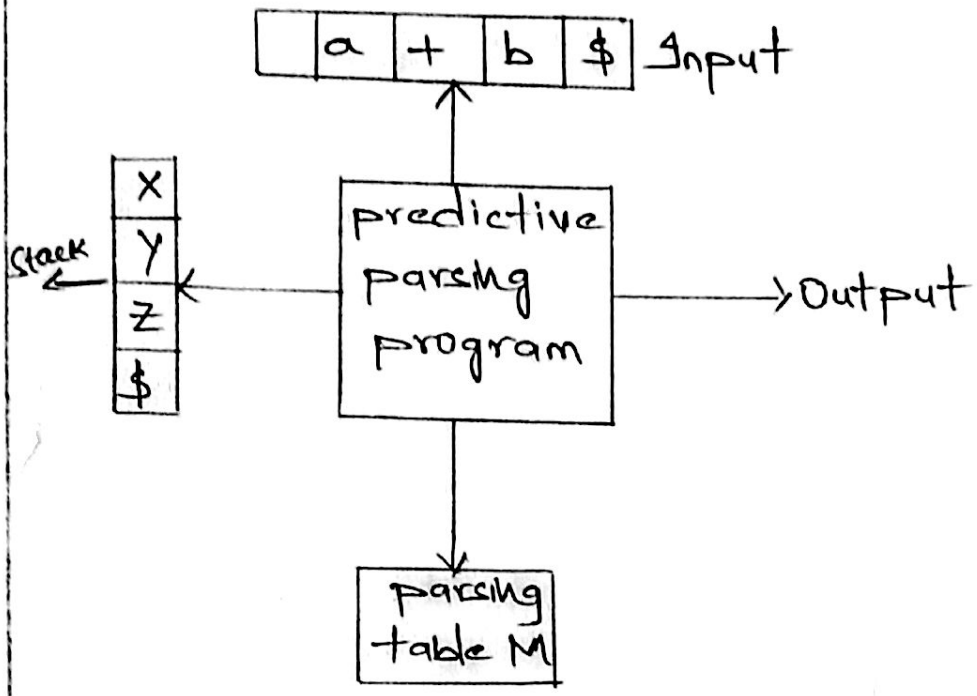


Fig: Predictive Parser.

The parser is controlled by a program that behaves as follows. The program considers X, the symbol on top of the stack and 'a', the current input symbol. These two symbols determine the action of the parser. They are:-

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next

input symbol.

3. If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.

(i) If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU.

(ii) If M[X, a] = error, the parser calls an error recovery routine.

The behaviour of the parser can be described interms of its configurations, which give the stack contents and the remaining input. Initially, the parser is in configuration.

Stack	Input
$S	w$

where 'S' is the start symbol of the grammar and 'w' is the string to be parsed.

PREPROCESSING STEPS REQUIRED FOR PREDICTIVE PARSING

FIRST Rules:-

1. If 'a' is a terminal then FIRST(a) = {a}
2. If 'A' is a non-terminal and 'A' derives A → α is a production then add 'a' to FIRST(A).

3. If $A \to \epsilon$ is a production then add 'ϵ' to FIRST(A).

FOLLOW Rules:-

1. If 'S' is a start symbol, place $ in FOLLOW(S).
2. If there is a production $A \to \alpha B \beta$ then everything in FIRST(β) without 'ϵ' is to be placed in FOLLOW(B).
3. If there is a production $A \to \alpha B \beta$ (or) $A \to \alpha B$ and FIRST(β) = {ϵ} then FOLLOW(A) = FOLLOW(B) (or) FOLLOW(B) = FOLLOW(A) that means everything in FOLLOW(A) is in FOLLOW(B).

1. Consider the grammar.

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$ construct FIRST & FOLLOW

Sol:- First eliminate the left recursion in given grammar.

$A \to A\alpha \mid \beta$

$\Downarrow$

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

consider, $\underset{A}{E} \to \underset{A}{E} \underset{\alpha}{+T} \mid \underset{\beta}{T}$

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$\underset{A}{T} \to \underset{A}{T} \underset{\alpha}{*F} \mid \underset{\beta}{F}$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

After eliminating left recursion, the grammar becomes

$E \to TE'$

$E' \to +TE' | \epsilon$

$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to (E) | id$

Terminals :- +, *, (,), id

Non-terminals :- E, E', T, T', F

FIRST (+) = {+}

FIRST (*) = {*}

FIRST (() = {(}

FIRST ()) = {)}

FIRST (id) = {id}

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E') = {+, ϵ}

FIRST (T') = {*, ϵ}

FOLLOW :-

FOLLOW (E) = {$}

Consider,

$F \to (E)$

$(A \to \alpha B \beta)$

FIRST ()) = FOLLOW (E)

= {)}

∴ FOLLOW (E) = {), $}

consider,

$E \rightarrow TE'$

$(A \rightarrow \alpha B)$

$FOLLOW(E) = FOLLOW(E')$

$= \{), \$\}$

$\therefore$ $\boxed{FOLLOW(E') = \{), \$\}}$

consider,

$E' \rightarrow +TE'$

$(A \rightarrow \alpha B\beta)$

$FIRST(E') = FOLLOW(T)$

$= \{+\}$

$FOLLOW(E') = FOLLOW(T)$

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

$\therefore$ $\boxed{FOLLOW(T) = \{+,), \$\}}$

consider,

$T' \rightarrow *FT'$

$(A \rightarrow \alpha B\beta)$

$FIRST(T') = FOLLOW(F)$

$= \{*\}$

$FOLLOW(T') = FOLLOW(F)$

$= \{*, +,), \$\}$

$\therefore$ $\boxed{FOLLOW(F) = \{*, +,), \$\}}$

Symbol	FIRST	FOLLOW
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,), $}
F	{(, id}	{*, +,), $}

Parsing table:-

	+	*	id	(	)	$
E			E→TE'	E→TE'		
E'	E→+TE'				E'→ε	E'→ε
T			T→FT'	T→FT'		
T'	T'→ε	T'→*FT'			T'→ε	T'→ε
F			F→id	F→(E)		

2. Construct FIRST & FOLLOW for the given grammar.

S→aAB|bA|ε

A→aAb|ε

B→bB|ε

Sol:- terminals :- a, b

Non-terminals :- S, A, B

FIRST (a) = {a}

FIRST (b) = {b}

FIRST (S) = {a, b, ∈}

FIRST (A) = {a, ∈}

FIRST (B) = {b, ∈}

FOLLOW:-

FOLLOW (S) = {$}

consider,

S → aAB

(A → αBβ)

FIRST (B) = FOLLOW (A)

= {b}

FOLLOW (S) = FOLLOW (A)

= {b$}

∴ **FOLLOW (A) = {b, $}**

consider,

S → bA

(A → αB)

FOLLOW (S) = FOLLOW (A)

= {$}

consider,

A → aAb

(A → αBβ)

FIRST(b) = FOLLOW (A)

= {b}

∴ **FOLLOW(A) = {b, \$}**

Consider

$\underset{A}{S} \rightarrow \underset{\alpha}{aA}\ \underset{B}{B}$

$(A \rightarrow \alpha B)$

FOLLOW (S) = FOLLOW (B)

= {\$}

∴ **FOLLOW (B) = {\$}**

symbol	FIRST	FOLLOW
S	{a, b, ϵ}	{\$}
A	{a, ϵ}	{b, \$}
B	{b, ϵ}	{\$}

parsing table:-

	a	b	\$
S	S→aAB	S→bA	S→ϵ
A	A→aAb		A→ϵ
B		B→bB	B→ϵ

3) Construct the Predictive passer for the following grammar

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

Sol:- Given Grammar is having left recursion, so first eliminate the left recursion

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

$A \to A\alpha \mid \beta$ can be converted as $A \to \beta A'$, $A' \to \alpha A' \mid \epsilon$

we can write $L \to SL'$
$L' \to ,SL' \mid \epsilon$

Now the grammar is

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

To Compute FIRST

FIRST(S) = { (, a }
FIRST(L) = { (, a }
FIRST(L') = { , , ε }

To Compute FOLLOW

Follow(S) = { $ }

Now $S \to (L)$
$A \to \alpha B \beta$

FIRST( )) = Follow(L) = {) }

Now $L \to SL'$
$A \to \alpha B$

Follow(L) = Follow(L') = {) }

$L' \to ,SL'$
$A \to \alpha B \beta$

FIRST(L') = Follow(S) = { , }

Follow(L') = Follow(S) = { , ,) , $ }

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

Predictive Parsing table can be constructed as

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

LL(1)

A grammar whose parsing table has no multiple defined entries is said to be LL(1).

The top-down parsing algorithm is of non-recursive type. The first 'L' means the input is scanned from left to right, the second 'L' means that uses left most derivation for input string. and the number '1' in the input symbol means it uses only one input symbol (Look a head) to predict the parsing process. The grammar which is ambiguious and has left-recursion is not an LL(1) grammar. when a parsing table has multiple defined entries, the remedy is to eliminate left recursion and left factoring the grammar.

[Continuation is predictive parsing theory by changing the name in table as LL(1) Grammar]

Consider the grammar

$S \rightarrow iCtSA \mid a$
$A \rightarrow eS \mid e$
$C \rightarrow b$

whether it is LL(1) grammar.

Sol Let the given grammar will be

$S \rightarrow iCtSA \mid a$
$A \rightarrow eS \mid \epsilon$
$C \rightarrow b$

Now Compute FIRST

$FIRST(S) = \{i, a\}$
$FIRST(A) = \{e, \epsilon\}$
$FIRST(C) = \{b\}$

Now Compute Follow

Let Follow(S) = {$}

Now $\underset{A}{S} \rightarrow \underset{\alpha}{iCt} \underset{B}{S} \underset{\beta}{A}$

FIRST(A) = Follow(S)
= {e}

Follow(S) = Follow(S)
= {e, $}

Now $\underset{A}{S} \rightarrow \underset{\alpha}{iCtS} \underset{B}{A}$

Follow(S) = Follow(A)
= {e, $}

Now $\underset{A}{A} \rightarrow \underset{\alpha}{e} \underset{B}{S}$

Follow(A) = Follow(S)
= {e, $}

Now $\underset{A}{S} \rightarrow \underset{\alpha}{iC} \underset{B}{t} \underset{\beta}{SA}$

FIRST(tSA) = Follow(C)
= {t}

	FIRST	Follow
S	{i, a}	{e, $}
A	{e, ε}	{e, $}
C	{b}	{t}

The Predictive Parsing table is given as

	a	b	e	i	t	$
S	S→a			S→iCtSA		
A			A→eS A→ε			A→ε
C		C→b				

As we have got multiple entries in M[A, e] given grammar is not LL(1) grammar.

Errors recovery in Predictive Parsing

→ Error is detected during Predictive parsing, when the terminal on top of stack does not match the next input symbol (or) nonterminal 'A' on top of the stack, 'a' is the next input symbol, and parsing table entry M[A,a] is empty.

→ Panic mode errors recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens.

→ If Passes lookup entry M[A,a] as blank then the input symbol 'a' is skipped.

→ If entry is synch then the non-terminal on the top of stack is popped.

→ If token on the top of the stack does not match the input symbol then we POP the token from the stack.

Consider the grammar

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

[For solution Refer Problem1 in pageno 10]

Sol For the above Grammar, Construct the Predictive Parsing table as follows

Nonterminals	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	Synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Synch		$T \rightarrow FT'$	Synch	Synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	Synch	Synch	$F \rightarrow (E)$	Synch	Synch

for the input +id * *id $

stack	input	Comments
$E	+id * *id $	skip +
$E	id * *id $	
$E'T	id * *id $	
$E'T'F	id * *id $	
$E'T'id	id * *id $	
$E'T'	* *id $	
$E'T'F*	* *id $	
$E'T'F	*id $	Error: M[F, *] = synch ∴ POP F
$E'T'	*id $	
$E'T'F*	*id $	
$E'T'F	id $	
$E'T'id	id $	
$E'T'	$	
$E'	$	
$	$	