UNIT-II

Syntax Analysis :-

* Introduction.
* Role of syntax Analysis.

Introduction :-

→ It is the Second Phase of the compilation.

- It checks for the syntax of the language.
- syntax analyser takes the tokens from the lexical analyser and groups them in some Programming structure called "syntax tree (or) Parse tree".
- If any syntax cannot be recognized then the syntax Error will be generated.

Definition :-

- A Parsing (or) syntax analysis is a Process which takes the I/P string "w". and produce either a Parse tree (or) generates the syntactic error.
- It is also called "Parser"

Ex:- consider the source Program statement A = B+10

Here the lexical analyser reads the above statement and broken it into the set of tokens like ~~'A' is identifier~~

A — is identifier

:= — Assignment operator

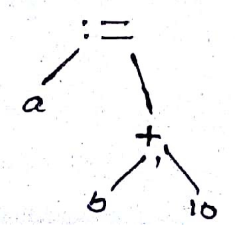
B — Identifier

+ — Assignment operator

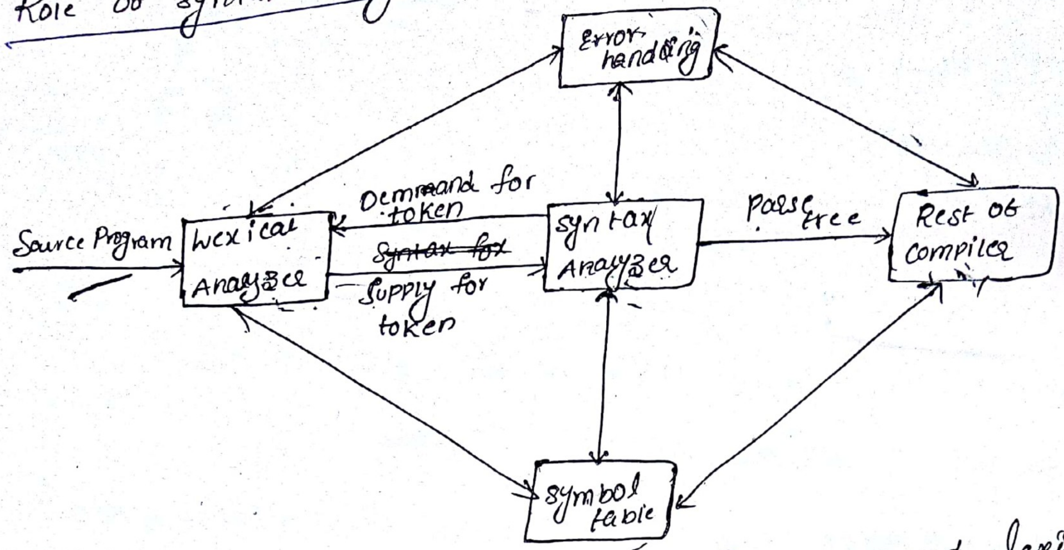
10 — number (or) constant.

- Now the syntax analyser collect the above tokens from lexical analyser and arrange them into a structure is called "Parse tree (or) Syntax tree".

⇒ a := b+10.



Role of Syntax Analyzer :- (or) Parser :-



In the Process of compilation the Parser and lexical analyser work together that means.

, when the Parser requires String of tokens it invokes lexical analyzer.

2. In turn the lexical Analyzer supply tokens to Syntax Analyzer.

3. The Parser collects sufficient number of tokens and build a Parse tree.

→ It finds Syntatical errors at the time of construction of Parse tree.

→ This errors are recovery ~~require~~ by Error handler.

Context free Grammar :-

* Introduction * Derivation and Parse tree * Ambiguous Grammar

→ Introduction:-

~~Where~~ A context free Grammar "G" is a four tuples like

$G = (V, T, P, S)$ where

V ⟶ set of non terminal symbols.

T ⟶ set of Terminal symbols.

P ⟶ set of Production ~~form~~ rules of the form α, B where $\alpha \in V$, B ...

$S \rightarrow$ start symbol.

Ex:- let the language $L = a^n b^n, n \geq 1$

~~$L = a^n b^n$~~

minimum string = ab

$S \rightarrow a^n b^n$

$\rightarrow a\, a^{n-1}\, b^{n-1}\, b$

$\rightarrow a a\, a^{n-2}\, b^{n-2}\, b b$

P: $S \rightarrow a S b$

$S \rightarrow a S b$

$S \rightarrow a b$

from the above $G = (V, T, P, S)$ ~~where~~ is a context free Grammar

Where $V \rightarrow$ set of $\{S\}$

$T \rightarrow$ set of $\{a, b\}$

$P \rightarrow \{aSb, ab\}$

$S \rightarrow$ start symbol $\{S\}$

→ Derivation and Parse tree :-

Derivation from 'S' means generation of a string 'W' form 'S'. for constructing derivation two things are important.

1. Choice of non-terminal. form several others.
2. choice of rule from production rules for corresponding non. terminal.

→ Definition of Derivation tree :-

Let $G' = (V, T, P, S)$ we a context free grammar.

The Derivation tree is a tree which can be constructed by following properties.

1. The root node has label 'S'.
2. Every vertex can be derived from $\{V \cup T \cup \epsilon\}$

3. If there exist a vertex 'A' with children $x_1, x_2, x_3 ---- x_n$ then there should be a production rule

$A \rightarrow x_1\, x_2\, x_3 ------ x_n$

4. The leaf nodes are from set T and interial nodes are from ~~set~~ 'V' {V}.

→ 1. Left Most Derivation (LMD)

In left most derivation the left most non terminal is replaced by a terminal (or) non-terminal in each step begining with a start symbol.

→ Right Most Derivation (RMD):-

In Right most derivation the Right most non terminal in each step is replaced by a terminal (or) non terminal begining with a start symbol.

Ex:- Consider the grammar given bellow

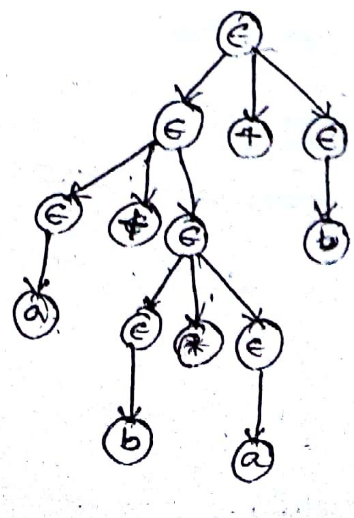
$E \rightarrow E+E$
$E \rightarrow E-E$
$E \rightarrow E*E$
$E \rightarrow E/E$
$E \rightarrow a/b$

Optain 1, Left Most Derivation 2, Right Most Derivation 3, Parse tree. for the input string a+b*a+b.

ⓐ Left Most Derivation:-

$E \rightarrow E+E$
$\rightarrow E+E+E$
$\rightarrow a+E+E$
$\rightarrow a+E*E+E$
$\rightarrow a+b*E+E$
$\rightarrow a+b*a+E$
$\rightarrow a+b*a+b$

Parse tree:-



Q, **Right Most Derivation:-** a+b * a+b.

$E \rightarrow E+E$

$\rightarrow E+E+E$

$\rightarrow E+E+b$

~~$\rightarrow E+a+b$~~

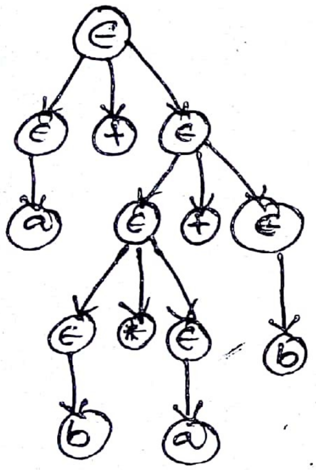
$\rightarrow E+E*E+b$

$\rightarrow E+E*a+b$

$\rightarrow E+b*a+b$

$\rightarrow a+b*a+b.$

Parse tree:-



Q, consider the grammar given below $S \rightarrow (L) \mid a$
$L \rightarrow S, S \mid S.$

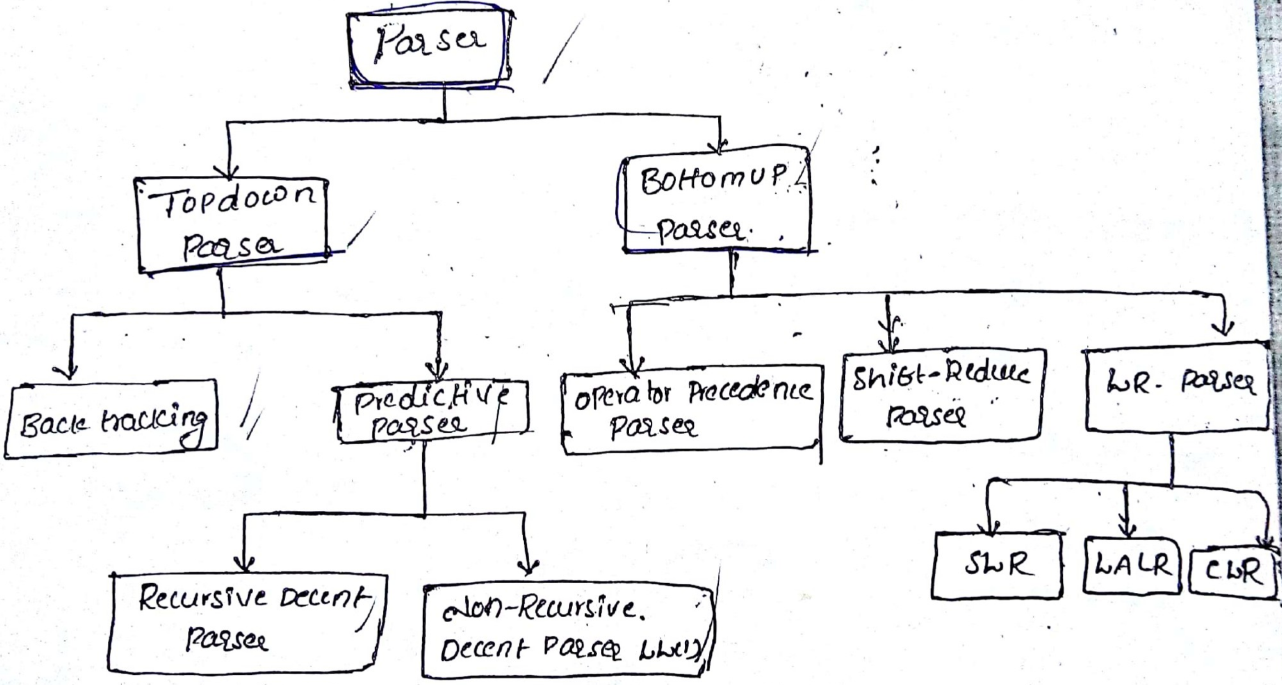
inputstring is (a, (a;a)).

→ writing a context-free Grammar:

* Lexical Analyser Vs syntax Analyser
* classification of Parsing techniques.
* Problems with Parsing Techniques.

1. Back tracking.
2. Left Recursion.
3. Left factoring.
4. Ambiguity.

→ Classification of Parsing techniques:-



Top down Parser:-

- The Process of constructing a syntax tree (or) Parse tree from root node to leaf node is called "Topdown Parser".
- Top down Parser classified into two types.

1. Back tracking 2. Predictive Parser.

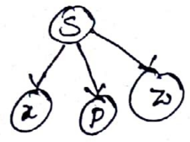
→ Back tracking:-

Back tracking is an technique in which for ~~Expanst.~~ Expansion of non terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

Ex:- Consider the grammar $S \rightarrow xPw$
$P \rightarrow yw|y$

Now we obtain an input string xyw from the above Grammar.

$\rightarrow$ 1

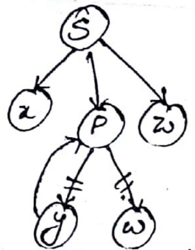


Here we take the first production

$S \rightarrow xPw$ then the corresponding parse tree is $S \rightarrow xPw$.

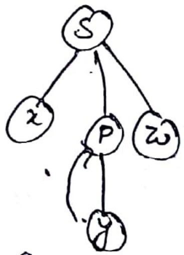
$\rightarrow$ 2. Now Non-terminal symbol 'p' is replace by the first alternative of it that is $P \rightarrow yw$ then the parse tree is

$S \rightarrow xPw$
$P \rightarrow xyww$



This string doesn't derived the Given i/p string. So ~~remove~~ we move backword to 'p'. and remove the corresponding branches from it. and apply another alternative of it. then the corresponding tree is

$S \rightarrow xPw$
$P \rightarrow xyw$



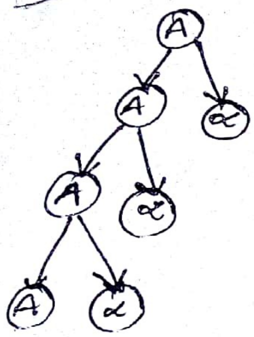
This parsing technique increases lot of over head in implementation of parse tree. So we need to eliminate the backtracking by modifying the grammar.

→ left Recursion:-

A left Recursive grammar is a grammar which contains the production rule is like $A \rightarrow A\alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$.

. If left Recursion is present in the grammar then the top down parser can enter into infinity loop. like

$A \rightarrow A\alpha$



***
→ Elimination of left Recursion :-

To Eliminate left Recursion we need to modified the grammar.

Let 'G' = (V, T, P, S) be a CFG with the productions rule having left recurssion. $\boxed{\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow \beta \end{array}}$ then we eliminate the left Recursion by rewriting the production rule has

$$\boxed{\begin{array}{ll} 1. & A \rightarrow \beta A' \\ 2. & A' \rightarrow \alpha A' \\ 3. & A' \rightarrow \epsilon \end{array}}$$

$$\boxed{\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow \beta \end{array}}$$

Ex:- consider the grammar

$E \rightarrow E+T$ } $A \rightarrow A\alpha$

$E \rightarrow T$ } $A \rightarrow \beta$ $A \rightarrow A\alpha$. $A = E$, $\alpha = +T$,

$T \rightarrow T*F$ } $A \rightarrow A\alpha$

$T \rightarrow F$ } $A \rightarrow \beta$

$F \rightarrow (E)$

$F \rightarrow id$

Eliminate left reccursion from the given grammar.

A: Step 1:- $E \rightarrow E+T$ $T \rightarrow T*F$ $F \rightarrow (E)$

$E \rightarrow T$ $T \rightarrow F$ $F \rightarrow id$.

we can map this grammar production rule with the rule $A \rightarrow A\alpha$, $A \rightarrow \beta$. where $A = E$

$\alpha = +T$

$\beta = T$

know we can eliminate left Recursion

1. $A \rightarrow \beta A'$
   ↳ $E \rightarrow TE'$
2. $A' \rightarrow \alpha A'$
   ↳ $E' \rightarrow +TE'$
3. $A' \rightarrow \epsilon$
   ↳ $E' \rightarrow \epsilon$

∴ After Eliminating left recursion the new Production rules are.

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \epsilon$

~~T → FT'~~ similarly for the rules

$T \rightarrow T*F$

$T \rightarrow F$

we can map this grammar rule with the rule

$A \rightarrow A\alpha$

$A \rightarrow \beta$

where $A = T$

$\alpha = *F$

$\beta = F$

know we can eliminate left recursion.

1. $A \rightarrow \beta A'$
   $\rightarrow T \rightarrow FT'$
2. $A' \rightarrow \alpha A'$
   $\rightarrow T' \rightarrow *FT'$
3. $A' \rightarrow \epsilon$
   $\rightarrow T' \rightarrow \epsilon$

Therefore the grammar without left recursion is

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T \rightarrow \epsilon$

$F \rightarrow (E)$

$F \rightarrow id.$

Q. Consider the Grammar

$A \rightarrow ABd$

$A \rightarrow Aa$

$A \rightarrow a$

$B \rightarrow Bc$

$B \rightarrow b.$

remove left recursion?

Sol:- $A \rightarrow ABd$ $A \rightarrow Aa$ $B \rightarrow Bc$

$A \rightarrow a$ $A \rightarrow a$ $B \rightarrow b$

1. $A \rightarrow aA'$ 1. $A \rightarrow aA'$ 1. $B \rightarrow bB'$
2. $A' \rightarrow BdA'$ 2. $A' \rightarrow aA'$ 2. $B' \rightarrow cB'$
3. $A' \rightarrow \epsilon$ 3. $A' \rightarrow \epsilon$ 3. $B' \rightarrow \epsilon$.

→ **Left factoring :-**

1. A Grammar may not be suitable for recursive decent Parsing Even if there is no left recursion.
2. for example consider the grammar $S \rightarrow iEtS \mid iEtSeS \mid a$, $E \rightarrow b$
3. A useful method for manipulating the grammar into a form suitable for recursive decent Parsing is left factoring.

Left-factoring :-

The Process of factoring out the common Prefix of alternatives are 'n' number of 'A' Production let $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \cdots \mid \alpha\beta_n$ rules. and 'α' is not equal to null. after left factoring the grammar will become

1. $A \rightarrow \alpha A'$
2. $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots \mid \beta_n$

Ex :- consider the Grammar $S \rightarrow iEtS \mid iEtSeS \mid a$, $E \rightarrow b$. do the left factoring on above Grammar.

Sol:- consider the Production rule with common Prefix Part. then

$S \rightarrow \underbrace{iEtS}_{\alpha} \mid \underbrace{iEtS}_{\alpha}\underbrace{eS}_{\beta_2}$ to Now Perform left factoring.

after the new Productions are

We can map this Grammar rules with the Rules

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

where $A = S$, $\alpha = iEtS$, $\beta_1 = \epsilon$, $\beta_2 = eS$ After left factoring the new Productions rules are

1. $A \rightarrow \alpha A'$
   $S \rightarrow iEtS\ S'$
2. $A' \rightarrow \beta_1 \mid \beta_2$
   $S' \rightarrow \epsilon \mid eS$.
3. $S \rightarrow a$
   $E \rightarrow b$.

∴ The grammar after left factoring is

$S \rightarrow iEtSS'$

$S' \rightarrow \epsilon \mid eS.$

$S \rightarrow a$

$E \rightarrow b.$

2, To the left factoring in the following Grammar.

$A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b.$

Sol:- $A \rightarrow a\overset{\beta_1}{AB} \mid a\overset{\beta_2}{A} \mid a.$

where $A = A$

$\alpha = a$

$\beta_1 = AB$

$\beta_2 = aA$

$\beta_3 = \epsilon$

After left factoring the new production rule is.

$A \rightarrow \alpha A'$

$A \rightarrow a A'$

2, $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

$A' \rightarrow AB \mid aA \mid \epsilon.$

3, $bB \mid b$

$A = B$

$\alpha = b$

$\beta_1 = B$

$\beta_2 = \epsilon.$

∴ the grammar after left factoring is.

$A \rightarrow aA'$

$A' \rightarrow AB \mid aA \mid \epsilon.$

$A \rightarrow a$

$B \rightarrow b.B'$

$B' \rightarrow B \mid \epsilon.$

→ **Ambiguity:-**

A Grammar which has more than one left most derivation (or) Right Most Derivation (or) Parse tree for the same input string is called "Ambiguous grammar."

Ex:- Consider the grammar which has more than one left most Derivation for the inputstring $a+a+a.$

$S \rightarrow S+S$

$S \rightarrow a.$

$A \rightarrow A\alpha$

$A \rightarrow \beta.$

1, LMD:-

$S \rightarrow S+S$

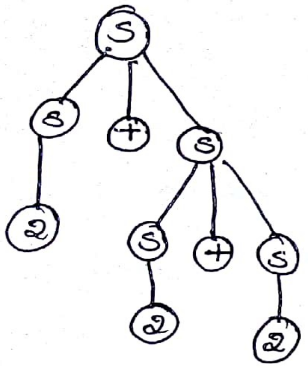
$\rightarrow a+S$

$\rightarrow a+S+S$

$\rightarrow a + a + S$ $\qquad A \rightarrow \alpha A$

$\rightarrow a + a + a.$ $\qquad A \rightarrow B$

Parse tree :-



ea (doesn't current cal)

a, LMD

Draw back :-

The computer may be confused at the time of computing mathematical Expressions due to the grammar is Ambiguous.

a, LMD :-

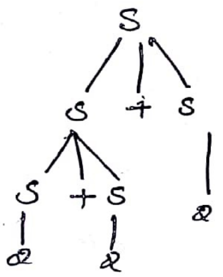
$S \rightarrow S+S$

$\rightarrow S+S+S$

$\rightarrow a+S+S$

$\rightarrow a+a+S$

$\rightarrow a+a+a$



for Removing ambiguity

1. If the grammar as left associative. operators$(+, -, *, /, \div)$ then induce left Recursion

$A \rightarrow A\alpha$

$A \rightarrow B$

2. If the grammar as Right associative Operator $(=, \uparrow,)$ then induce the Right Recursion.

$A \rightarrow \alpha A$

$A \rightarrow \beta$

Ex:- Consider the Grammar

$S \rightarrow S+S$

$S \rightarrow S*S$

$S \rightarrow a.$ And the i/p string is $a+a*a.$

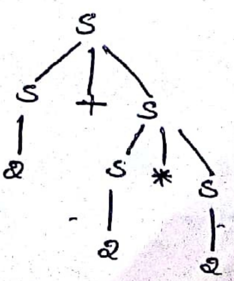
LMD :- $S \rightarrow S+S$

$\rightarrow a+S$

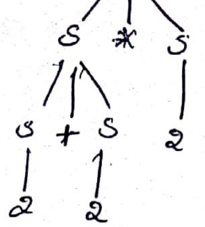
$\rightarrow a+S*S$

$\rightarrow a+a*S$

$\rightarrow a+a*a.$



2, S → S+S
→ S+S*S
→ 2+S*S
→ 2+2*S
→ 2+2*2



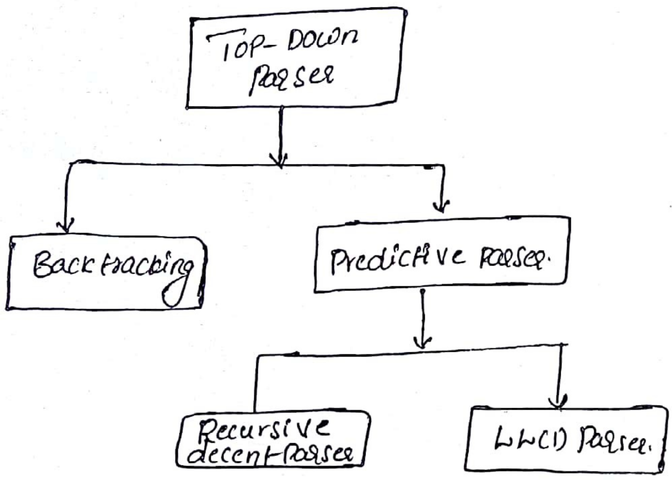
→ 8

X.

S → S+S
S → S* S
S → 2

S → S+T
S → T
T → T* F
T → F
F → 2

→ Top down Parser:-



Recursive decent Parser:

A parser that uses collection of recursive procedures for parsing the given input string is called "Recursive Decent Parser".

- In this type of parser the CFG is used to build the recursive procedures.
- The RHS of the Production rule is directly converted to a Program.
- For each non-terminal a separate procedure is ~~return~~ written and body of the procedure is RHS of the corresponding non-terminal.

STEPS for Construction of Recursive decent Parser:-

The RHS of the Production rule is directly converted into program code symbol by symbol.

Step 1 :- If the i/p symbol is non-terminal then a call to the procedure corresponding to that non-terminal symbol.

Step 2 :- If the i/p symbol is terminal then it is matched with the locate symbol from the input.

Step 3 :- If the production rule as many alternatives then all this alternatives as to be combined in to a single body of procedure.

Step 4 :- The parser should be activated by a procedure corresponding to start symbol.

Ex :- construct a recursive decent parser for the following grammar.

$E \to E+T$
$E \to T$
$T \to TF$
$T \to F$
$F \to F^* / a / b$

Note :-

The recursive decent parser is works on a CFG with out left ~~recu~~ reccursion.

Sol :- The Given grammar is
$E \to E+T$
$E \to T$
$T \to TF$
$T \to F$
$F \to F^* / a / b.$

The above grammar. constant left recursion. So, before constructing RDP we should eliminate left recursion from the given grammar.

Elimination of left Recursion :-

$\boxed{A \to A\alpha,\ A \to \beta \Rightarrow A \to \beta A',\ A' \to \alpha A',\ A' \to \epsilon}$

1. $E \to E+T$
$E \to T$
$\Downarrow$
$E \to TE'$
$E' \to +TE'$
$E' \to \epsilon$

2. $T \to TF$
$T \to F$
$T \to FT'$
$T' \to FT'$
$T' \to \epsilon$

3. $F \to F^*$
$F \to a$
$\Downarrow$
$F \to aF'$
$F' \to {*}F'$
$F' \to \epsilon$

4. $F \to F^*$.
$F \to b$
$\Downarrow$
$F \to bF'$
$F' \to {*}F'$
$F' \to \epsilon$

∴ The resultant Grammar with out left Recurssion is

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow \epsilon$

$T \rightarrow FT'$

$T' \rightarrow FT'$

$T' \rightarrow \epsilon$

$F \rightarrow aF'$

$F' \rightarrow *F'$

$F' \rightarrow \epsilon$

$F \rightarrow bF'$

→ Construction of Recursive decent Parser:-

```
Procedure E()
{
    T();
    E'();
    if (lookahead == $)
    Printf ("\n String accepted");
    Else
    Printf ("\n String rejected");
}
Procedure E'()
{
    if (lookahed == '+')
    {
        match (+);
        T();
        E'();
    }
    Else
    {
        null;
    }
}
Procedure T()
{
    F();
    T'();
}

procedure T'()
{   F();{ if (true)
}T'();
```

```
    Else
    { null ;
    }
}
procedure F()
  {
      if (lookahead =='a')
      { match (a);
         F'();
      }
       Else
       {
           if (lookahead = 'b')
           {
               match (b);
               F'();
           }
       }
  }
Procedure F'()
  {
      if (lookhead =='*')
      {
       match (*);
         F'();
      }
       Else
       {
          null ;
       }
  }
procedure match (char c)
 {  if (lookahead =='c')
    { lookahead ++;
 }
```

6

LL(1) Parser :

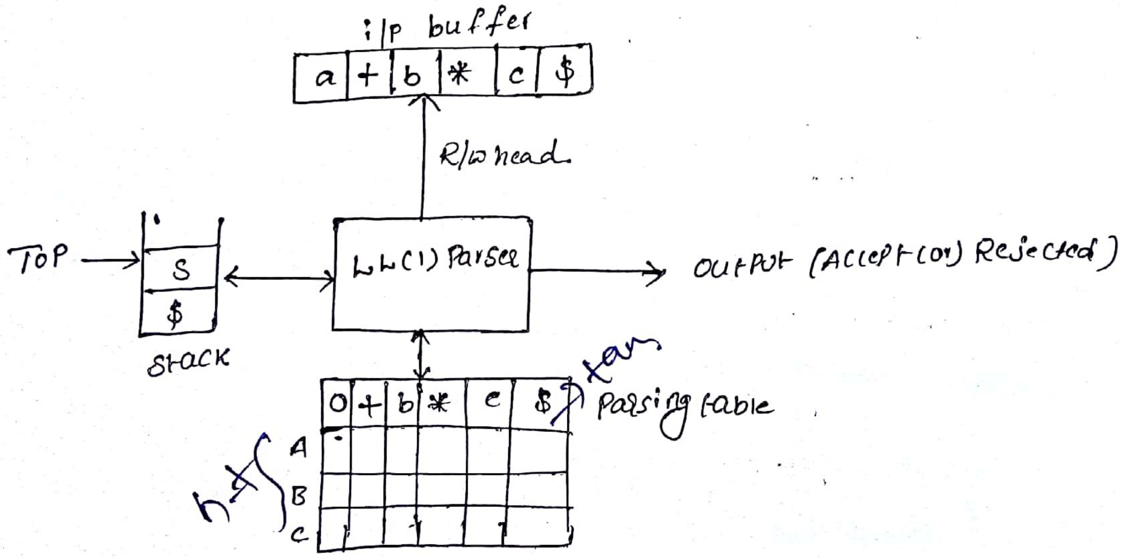
* Introduction.
* Model of LL(1) Parser.
* Construction of LL(1) Parser.

Note :- LL1 Parser is works on only the CFG with out left Recursion.

ⓐ Introduction :

* Top-down Parser.
* Non-Recursive Parser.
* Predictive parser.
* LL(1) means
  - L ⟶ Reads the given i/p string from left side to right side.
  - L ⟶ Parse the given i/p string by using "left most derivation".
  - 1 ⟶ Reads only one lookahead i/p symbol at a time.
* It constants a "LL(1) Predictive Parsing table".

ⓑ Model of LL(1) Parser :-



It contains three data structures like

1. Input buffer. 2. stack 3. Parsing table.

• Input buffer :- LL1 Parser uses input buffer to store the i/p tokens.

• Stack :- LL1 Parser uses stack to hold the left sentential form. i.e; the symbols in the R.H.S of rule are placed (Pushed) into the stack in reverse order that is from right to left.

• Parsing table :- It is a two dimensional array contains collection of rows and columns. rows represents non-terminals. columns represent terminals. The table can be represented by M[A,a] where A is a non terminal.

a is a current i/p symbol.

→ Construction of LL(1) Parser :-

Steps :-

1. Computation of FIRST and FOLLOW functions.
2. Construction of LL(1) Parser table using FIRST and FOLLOW.
3. Construction of LL(1) Parsing Algorithm using Parser table.

1. Computation of FIRST and FOLLOW functions :-

1. FIRST function :- FIRST is a set of Terminal symbols that are FIRST symbols appearing at RHS of Production rule.

Rule for Computing first function :-

1. If the terminal symbol 'a' then FIRST (a) = {a}
2. If there is a rule '~~x~~' ~~derives~~ $X \rightarrow \epsilon$ then FIRST (X) = {ϵ}
3. for the rule $A \rightarrow x_1 x_2 x_3 \ldots x_n$ then FIRST (A) = {FIRST (x_1) $\cup$ FIRST (x_2) $\cup$ FIRST (x_3) $\cup \ldots \cup$ FIRST (x_n)}.

~~Note:~~ ~~LL, Par.~~

Ex: Compute FIRST function on the following grammar.

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

The grammar without left recursion is

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

Production rule	FIRST
$E \rightarrow TE'$	FIRST (T) = {(, id}
$E' \rightarrow +TE' / \epsilon$	{+, ϵ}
$T \rightarrow FT'$	FIRST (F) = {(, id}
$T' \rightarrow *FT' / \epsilon$	{*, ϵ}
$F \rightarrow (E) / id$	{(, id}

2, compute FIRST function for the following grammar $S \to (L)/a$

$L \to L, S/S$

$S \to (L) | a$

$L \to L, S | S$

$\Downarrow$

$L \to SL'$

$L' \to , SL'$

$L' \to \epsilon$

The Grammar without left recursion is

$S \to (L) | a$

$L \to SL'$

$L' \to , SL'$

$L' \to \epsilon$

production rule	FIRST
$S \to (L) \| a$	$\{ (, a \}$
$L \to S L'$	FIRST(S) = $\{ (, a \}$
$L' \to , SL' \| \epsilon$	$\{ ,, \epsilon \}$

3, compute FIRST function on the following grammar.

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

The given grammar doesn't contain left recursion.

Production rule	FIRST
$S \to AaAb \| BbBa$	$\{a, b\}$
$A \to \epsilon$	$\{\epsilon\}$
$B \to \epsilon$	$\{\epsilon\}$

4, compute FIRST grammar.

$S \to aAB$ (or) bA (or) ϵ.

$A \to aAb$ (or) ϵ

$B \to bB$ (or) ϵ

Production rule	FIRST
$S \to aAB \| bA \| \epsilon$	$\{a, b, \epsilon\}$
$A \to aAb \| \epsilon$	$\{a, \epsilon\}$
$B \to bB \| \epsilon$	$\{b, \epsilon\}$

FOLLOW function:-

FOLLOW(A) is a set of terminal symbols that appear immediately to the right of (A). i.e;

$$FOLLOW(A) = \{a \mid S \rightarrow \alpha A a \beta\}$$

where α, β is grammar symbols.
a. is terminal symbols

Rules for Computing follow:-

1. FOLLOW (S) = {\$} where S is the start symbol.
2. If there is a Production rule $A \rightarrow \alpha B \beta$ then if
   FOLLOW (B) = FIRST (β) accept 'ε' in FIRST (β).
3. If there is a Production rule ~~$A \rightarrow B\beta$ FOLLOW (B) = FIRST (β)~~ if FIRST (β) contains ε.
   $A \rightarrow B\beta$ FOLLOW (B) = FIRST (β) ∪ FOLLOW (A)
4. If there is a Production rule of the form $A \rightarrow \alpha B$ then
   FOLLOW (B) = FOLLOW (A).

Ex:- find follow function on the following grammar.
$S \rightarrow Bb \mid cd$
$B \rightarrow aB \mid \epsilon$
$C \rightarrow cC \mid \epsilon$

Sol:-
FOLLOW (S) = {\$}
FOLLOW (B) = {b}
FOLLOW (C) = {d}

Ex:- compute ~~first~~ FIRST and follow functions on the following Grammar.

$S \rightarrow ABCDE$
$A \rightarrow a \mid \epsilon$
$B \rightarrow b \mid \epsilon$
$C \rightarrow c$
$D \rightarrow d \mid \epsilon$
$E \rightarrow e \mid \epsilon$

4

Sol:-
$S \rightarrow ABCDE$
FIRST (S) = FIRST (A)
= {a ε}

$A \rightarrow a$
$A \rightarrow \epsilon$
FIRST (A) = {a} ∪ {ε}
= {a, ε}

$B \rightarrow b$
$B \rightarrow \epsilon$
FIRST (B) = {b} ∪ {ε}
= {b, ε}

$C \rightarrow c$

$C \rightarrow \epsilon$

$FIRST(C) = \{c\} \cup \{\epsilon\}$

$= \{c, \epsilon\}$

$D \rightarrow d$

$D \rightarrow \epsilon$

$FIRST(D) = \{d\} \cup \{\epsilon\}$

$= \{d, \epsilon\}$

$E \rightarrow e$

$E \rightarrow \epsilon$

$FIRST(E) = \{e\} \cup \{\epsilon\}$

$= \{e, \epsilon\}$

$S \rightarrow ABCDE$.

$S \rightarrow aBCDE$

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

$S \rightarrow \epsilon BCDE$.

$S \rightarrow BCDE$.

$FIRST(S) = FIRST(B)$

$= \{b, \epsilon\}$

$S \rightarrow BCDE$.

$S \rightarrow bCDE$.

$FIRST(S) = \{b\}$

$S \rightarrow \epsilon CDE$

$S \rightarrow CDE$

$FIRST(S) = FIRST(C)$

$= \{c\}$

$\therefore FIRST(S) = \{a\} \cup \{b\} \cup \{c\}$

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

→ Computation of Follow:-

$S \rightarrow ABCDE$

$A \rightarrow a | \epsilon$

$B \rightarrow b | \epsilon$

$C \rightarrow c$

$D \rightarrow d | \epsilon$

$E \rightarrow e | \epsilon$

$FOLLOW(S) = \{\$\}$

$FOLLOW(A) = \{b, c\}$

$FOLLOW(B) = \{c\}$

$FOLLOW(C) =$

$S \to ABCDE$

FOLLOW(A) = FIRST(B)

$= \{b, \epsilon\}$

1, $S \to A\underline{B}CDE$

$S \to A\underline{b}CDE$

FOLLOW(A) = $\{b\}$

2, $S \to A\underline{B}CDE$

$S \to A\underline{\epsilon}CDE$

$S \to ACDE$

FOLLOW(A) = FIRST(C)

$= \{c\}$

$S \to ABCDE$

FOLLOW(B) = FIRST(C)

$= \{c\}$

$S \to AB\underline{C}DE$

FOLLOW(C) = FIRST(D)

$= \{d, \epsilon\}$

(1) $S \to ABC\underline{D}E$

$S \to AB\underline{C}dE$

FOLLOW(C) = $\{d\}$

(2) $S \to ABCDE$

$S \to ABC\underline{\epsilon}E$

$S \to ABCE$

→ Construction of LL(1) Parse table :

Algorithm :-

~~construction~~ for the rule $A \to \alpha$ of grammar 'G'.

step1 :- for each 'a' in FIRST(α) create entry $M[A, a] = A \to \alpha$

where 'a' is a terminal symbol.

step2: for 'ε' in FIRST(α) create entry $M[A, b] = A \to \alpha$.

where 'b' is a terminal symbol. In FLLOW(A).

step3:- If ε in FIRST(α) and \$ is in FLLOW(A) then create entry

in the table $M[A, \$] = A \to \alpha$.

step4:- All the remaining entries in the table M are maked as

~~synta~~ "SYNTAX ERRORS".

~~LL(1)~~ means :- ~~Et~~ re

Ex:- construct

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

→ FIRST (E) = FIRST (T) = { (, id }

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

FIRST (E') = { +, ∈ }

FIRST (T') = { *, ∈ }

FIRST (F) = { (, id }

FOLLOW (E) = { $,) }

FOLLOW (E') = FOLLOW (E) = { $,) }

FOLLOW (T) = { +, $,) }

FOLLOW (T') = { +, $,) }

E → TE'

FOLLOW(T) = FIRST(E')
= { +, ∈ }

E → T∈

E → T

FOLLOW(T) = FOLLOW(E)
= { $,) }

∴ FOLLOW(T) = { +, $,) }

E' → T∈

E' → +T

FOLLOW(T) = FOLLOW(E')
= { $,) }

T* → * FT'

FOLLOW(F) = FIRST (T')
= { *, ∈ }

T' → * F ∈

T' → * F

FOLLOW(F) = FOLLOW (T') = { +, $,) }

T → FT'

FOLLOW(F) = FIRST(T')
= { *, ∈ }

T → F∈

T → F

FOLLOW (F) = FOLLOW(T)
= { +, $,) }

T → FT'

FOLLOW (T') = FOLLOW(T)
= { +, $,) }

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

LL(1) Parser table :-

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

→ E → TE'
↳ FIRST (TE') = FIRST(T)
= { (, id }

E' → +TE'
↳ FIRST (+TE') = FIRST(+)
= { + }

3, $E' \rightarrow \epsilon$

FIRST(ϵ) = {ε}

FOLLOW (E') = {$,)}

4, $T \rightarrow FT'$

FIRST(FT') = FIRST(F) = {(, id}

5, $T' \rightarrow *FT'$

FIRST$(*FT')$ = FIRST$(*)$ = {*}

6, $T' \rightarrow \epsilon$

FIRST(ϵ) = {ε}

FOLLOW (T') = {+, $,)}

7, $F \rightarrow (E)$

FIRST $((E)$ = {(}

$F \rightarrow id$

FIRST(id) = {id}.

WWCD Parsing algorithm:-

consider the i/p string ~~id + id * id~~. ((a),a)

Stack	input string	ACTION
$S	((a),a)$	S→(L)
$)L(	((a),a)$	pop
$)L	(a),a)$	L→SL'
$)L'S	(a),a)$	S→(L)
$)L')L(	(a),a)$	pop
$)L')L	a),a)$	L→SL'
$)L')L'S	a),a)$	S→a
$)L')L'a	a),a)$	pop
$)L')L'	),a)$	L'→ε
$)L')ε	),a)$	–
$)L')	),a)$	pop
$)L'	,a)$	L'→SL'
$)L'S,	,a)$	pop
$)L'S	a)$	S→a
$)L'a	a)$	pop
$)L'	)$	L'→ε
$)ε	)$	–
$)	)$	pop
$	$	accepted.

① construct LL(1) Parser table for the following grammar

$S \to (L) / a$

$L \to L,S / S$ and check the input string ((a),a).

is replaced or not by the LL(1) parser.

② construct LL(1) parser table for the following grammar.

$S \to iEtSS'$

$S \to a$

$S' \to eS / \epsilon$

$E \to b$.

Error Recovery in predictive Parser:-

→ An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol (or).

→ when non-terminal A on the top of the stack a is the next input symbol. and parsing table entry $M[A,a]$ is empty.

→ the process of reducing number of errors in the parser table is called error recovery.

→ LL(1) parser uses "panic mode" error recovery technique.

"Panic mode Error recovery".

It is based on the idea of skipping symbols on the i/p until a synchronizing token is selected.

synchronizing token:- It is a set of terminals obtained from follow of non-terminal in the given grammar.

Ex:- $FOLLOW(E) = \{\$,)\}$

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

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

$FOLLOW(T') = \{+, \$,)\}$

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

After Applying Panic mode error recovery technique modified LL(1) Parser table is.

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

Parsing Algorithm:-

* If the parser look up the entry M [A,a] as a blank then the i/p symbol 'a' skipped.
* If the entry is 'sync' then the non-terminal the top of the stack is poped.
* If the token on the top of the stack doesn't match the i/p symbol then we pop on the token from the stack.

Ex:- consider the i/p string +id ** id.

Stack	i/p string	Action
$E	+id**id$	skipped +
$E	id**id$	E→TE'
$E'T	id**id$	T→FT'
$E'T'F	id**id$	F→id
$E'T'id	id**id$	pop
$E'T'	**id$	T'→*FT'
$E'T'F*	**id$	pop
$E'T'F	*id$	syn, pop
$E'T'	*id$	T'→*FT'
$E'T'F*	*id$	pop
$E'T'F	id$	F→id
$E'T'id	id$	pop
$E'T'	$	T'→ε
$E'		
$	$	