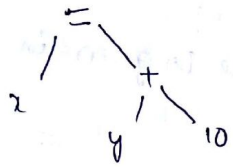
UNIT-II CONTEXT FREE GRAMMARS AND PARSING

Syntax Analysis:- A parser/or syntax analysis is a process which takes the i/p string w and produces either a parse tree (syntactic structure) (or) generates the syntactic errors.

1. To identify language construct present in a given i/p program. If the parser determines the i/p to be valid one, it o/p's a representation of the i/p in the form of parse tree.
2. If the i/p is grammatically incorrect, the parser declares the detecting syntax error in the i/p. This case, no parse tree can be produced.

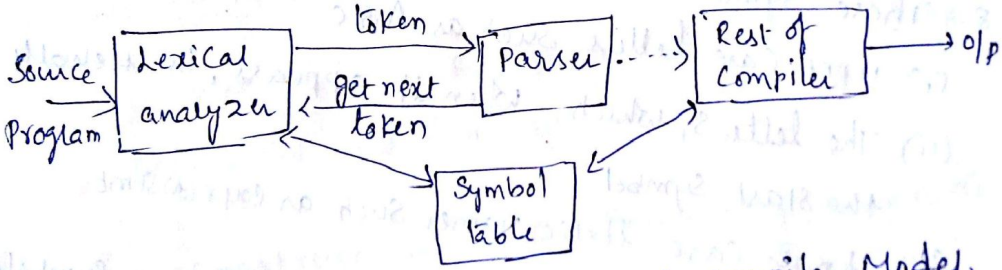
Ex:- $x = y + 10$



Parse tree

The ROLE OF THE PARSER

- In our compiler model, the parser obtains a string of token from the lexical analyzer.
- The parser collects sufficient number of tokens
- and builds a parse tree.



Position of parser in compiler model.

- Thus by building the parse tree, parser smartly finds the syntactical error if any.
- Now we will focus on first issue of parser i.e Specifica of input.
- As we know that Specification of i/p can be done by

CONTEXT - FREE GRAMMARS

- context free grammar (CFG or just grammar) is denoted $G = (V, T, P, S)$, where V and T are finite sets of variables and terminals.
  1. V is a set of non terminals
  2. T is a set of terminals
  3. S is a start symbol
  4. P is a set of Production rules.

The production rule is are is given in following form.

Non terminal $\longrightarrow (V \cup T)^*$.

Notational Conventions

1. These Symbols are terminals
   (i) Lowercase letters such as a, b, c --
   (ii) operator symbol such as +, - etc
   (iii) Punctuation Symbol such as Comma, parentheses --
   (iv) the digits 0, 1 --- 9.
   (v) Boldface strings such as id or if.
2. These Symbols are non terminals.
   (i) uppercase letters such as A, B, C.
   (ii) The letter S, which, when it appears, is usually the start Symbol.
   (iii) Lower case italic names such as expr or Stmt.
3. uppercase letters, such X, Y, Z, represent grammar Symbols. i.e either nonterminals or terminals.
4. Lowercase letters, ~~α, β,~~ u, v, z represent String of terminals.
5. Lowercase Greek letters α, β, γ, for example, represent String of grammar symbols. Thus a generic Production could be written as $A \to \alpha$, indicating that there is a

a single non terminal A on the left of the arrow (left side of production) and string of grammar symbols α to the right of the arrow (right side of production).

6. If $A \to \alpha_1, A \to \alpha_2 \dots A \to \alpha_k$ are all production with A on the left (we call them A-productions), we may write $A \to \alpha_1 | \alpha_2 | \dots | \alpha_k$. we call $\alpha_1, \alpha_2 \dots \alpha_k$ the alternative for A.

7) unless otherwise stated, the left side of first production is the start symbol.

Ex:- The grammar with the following product's define simple arthimatic expressions.

expr → expr op expr	op → +
expr → (expr)	op → −
expr → − expr	op → *
expr → id	op → /
	op → ↑

Sol:- In this grammar, the terminal symbols are

id + − * / ↑ ( )

The Non terminals Symbols are expr and op, and expr is the start Symbols.

Ex:- construct a C.F.G generating equal no of a's and b's over $\Sigma = \{a, b\}$

Sol:- $S \to A$
$A \to ab | aAb | ba | bAa | abA | baA)$

(or)

$S \to asb | bsa | ab | ba | \in$

(M)

$S \to aB$

$S \to bA$

$B \to aBB$

$A \to bAA$

$A \to a/as$

$B \to b/bS$

(N)

$S \to A$

$A \to ab/aAb/ba$

$bAa/AA.$

Ex:- (2) Find the CFG to recognize the set of odd palindrom over an alphabet $\Sigma = \{a, b\}$.

Any:- Let $G = (\{S\}, \{a, b\}, P, S)$ be C.F.G, where P is a set of Production rules given as follows.

Sol:- $P = \{ S \to asa$

$S \to bsb$

$S \to a$

$S \to b$

$S \to \epsilon$

$\}$

the string abaaba can be derived as

S	$S \to asa$
asa	$S \to bsb$
absba	$S \to asa$
abasaba	$S \to \epsilon$
aba ϵ aba	
abaaba	

is required string which is actually Palindrome.

Derivation Tree (or) Parse tree

The derivations in a C.F.G can be represent using tree. Such a tree representing derivations is called a derivation tree.

A derivation tree is also called a parse tree for a CFG = (V, T, P, S) is satisfying following conditions.

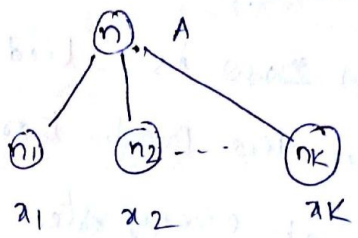
(i) Every vertex has a label which is a variable(A) terminal

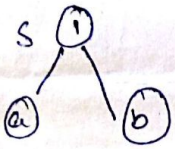
(ii) The root is labelled with the starting symbol of C.F.G say 'S'.

(iii) The label of an internal vertex is a variable

(iv) Every leaf node is labeled with a terminal symbol

(v) If the vertices $n_1, n_2 \cdots n_k$ written with labels $x_1, x_2 \cdots x_k$ are the sons of the vertex n with the label A, the $A \to x_1 \cdots x_k$ is a production in P.



Ex:- 

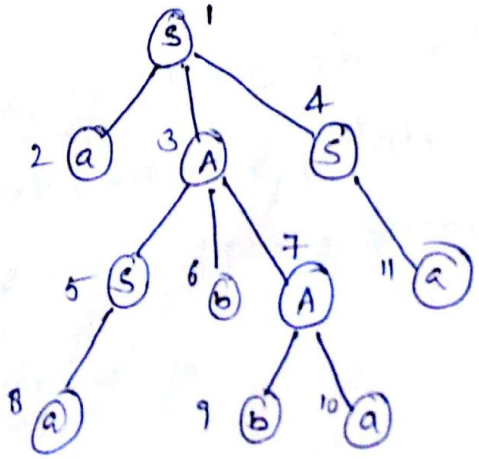
Hence $S \to ab$

Ex:- Construct a derivation tree for the following string aabbaa

glammar $G = (\{S, A\}, \{a, b\}, P, S)$

$P = \{ S \to aAS/a$
$A \to SbA/ba \}$
$\}$

Sol:-



There are two types of derivation

(1) Left most derivation

(2) Right most derivation

Left most derivation :-

A derivation $A \overset{*}{\Rightarrow} w$ is called a left most derivation if you apply the production only to the left most variable at every step.

Right most derivation :

A derivation $A \overset{*}{\Rightarrow} w$ is called Right most derivation if you apply the production only to the right most variable at every step.

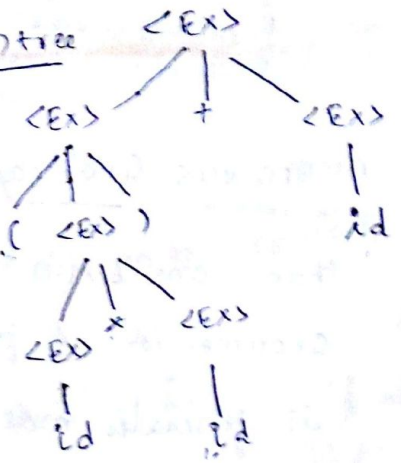
productions using leftmost derivation and rightmost derivation for the given string where V = {<Ex>}, T = {x, +, (,), id}

P = { <Ex> → <Ex> * <Ex>
<Ex> → <Ex> + <Ex>
<Ex> → (<Ex>)
<Ex> → id. }

Sol:- LMD

<Ex> → <Ex> * <Ex>
<Ex> → (<Ex>) + <Ex>
<Ex> → (<Ex> * <Ex>) + <Ex>
<Ex> → (id * <Ex>) + <Ex>
<Ex> → (id * id) + <Ex>
<Ex> → (id * id) + id.

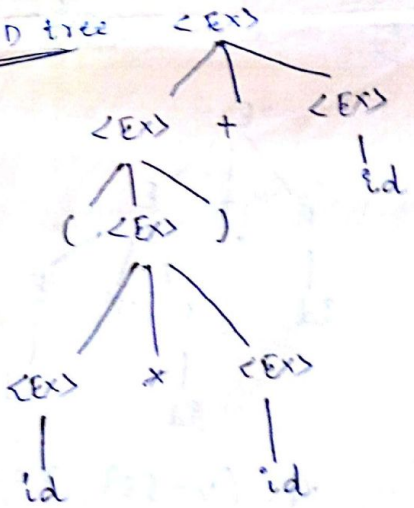
L.M.D tree



R.M.D

<Ex> → <Ex> + <Ex>
→ <Ex> + id
→ (<Ex>) + id
→ (<Ex> * <Ex>) + id
→ (<Ex> * id) + id
→ (id * id) + id

R.M.D tree

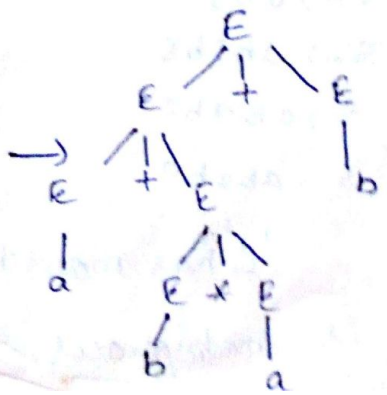


Ex:-(2) Consider Grammar given below.

E → E+E | E−E | E*E | E/E | a|b obtain leftmost and right most derivation for the string a+b*a+b.

Sol:- Leftmost derivation

E → E+E
E → E+E+E
E → a+E+E
E → a+E*E+E
E → a+b*E+E



$E \to a+b*a+b$

$E \to a+b*a+b$

R.M.D :

$E \to E*E$

$E \to E*E+E$

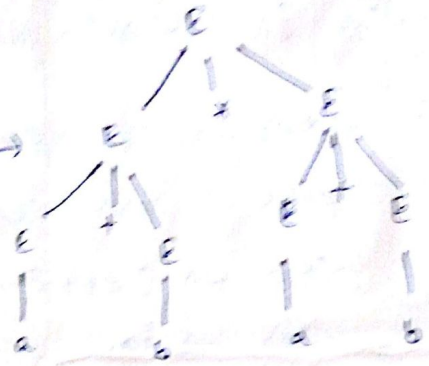
$E \to E*E+b$

$E \to E*a+b$

$E \to E+E*a+b$

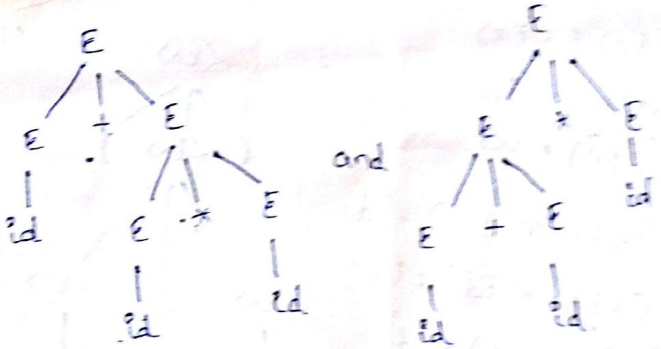
$E \to E+b*a+b$

$E \to a+b*a+b$



Ambiguous Grammar :- If any grammar consists of more than one L.M.D & R.M.D then it is called Ambiguous Grammar (G). A grammar G is said to be ambiguous if it generates more than one parse tree for the sentence of language L(G).

Ex:- $E \to E+E \mid E*E \mid id$ then for string id+id*id



Ex:- $G = (V=\{S\}, T=\{a,b\}, P, S\}$ where $P: \{S \to SbS \mid a\}$

string is ababa

Sol: LMD1 :-

$S \to SbS$

$S \to abS$

$S \to abSbS$

$S \to ababS$

$S \to ababa$

LMD2 :

$S \to SbS$

$S \to SbSbS$

$S \to abSbS$

$S \to ababS$

$S \to ababS$

'G' has more than one L.M.D's so G is ambiguous.

CFG from Given FA

- Let $M = (\{q_0, q_1, \ldots q_n\}, \Sigma, \delta, q_0, F)$ be a DFA. The equivalent grammar G can be constructed from this DFA such that Productions should correspond to transistions.
- The derivations can be terminated by a production rule giving terminals.
- For such production rule, the transistions terminating at some final state is encountered.

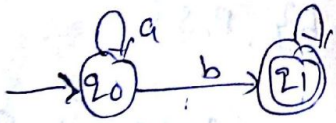
Let $G = (\{A_0, A_1, \ldots A_n\}, \Sigma, P, A_0)$

where P the set of production rules can be defined by following rules.

1. $A_i \rightarrow aA_j$ is a production rule if $\delta(q_i, a) = q_j$, where $q_j \notin F$
2. $A_i \rightarrow aA_j$ and $A_i \rightarrow a$ are production rule if $\delta(q_i, a) = q_j$ where $q_j \in F$

thus the given grammar is accepted by DFA Mechine

Ex:- construct regular grammar for given D.F.A



Sol:-

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

$V = \{A_0, A_1\}$ $T = \{a, b\}$

$A_0 \rightarrow aA_0$

$A_0 \rightarrow bA_1$

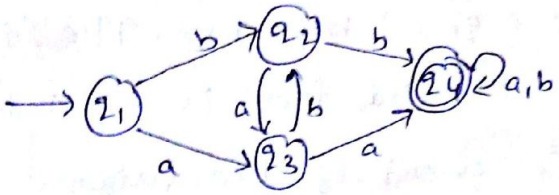
$A_0 \rightarrow b$

$A_1 \rightarrow aA_1$

$A_1 \rightarrow a$

$A_1 \rightarrow bA_1 | b$

Ex:- Construct a R.F.G for give F.A



Sol: Let G: ($\{A_1, A_2, A_3, A_4\}$, $\{a, b\}$, P, A_1)

$A_1 \rightarrow bA_2$

$A_1 \rightarrow aA_3$

$A_2 \rightarrow bA_4$

$A_2 \rightarrow b$

$A_2 \rightarrow aA_3$

$A_3 \rightarrow bA_2$

$A_3 \rightarrow aA_4$

$A_3 \rightarrow a$

$A_4 \rightarrow aA_4$

$A_4 \rightarrow a$

$A_4 \rightarrow bA_4$

$A_4 \rightarrow b$

Ex:- Ambiguous

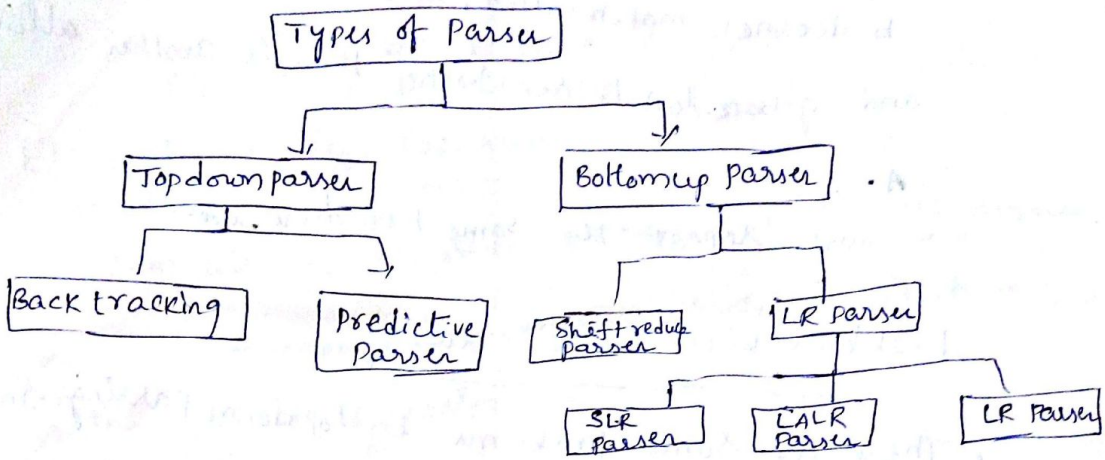
Let $G = \{V = \{S, A\}, T = \{a, b\}, P, S\}$

$P = \{ S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba \}$

String is aa bb aa

• Basic Parsing Techniques



Basic parsing techniques

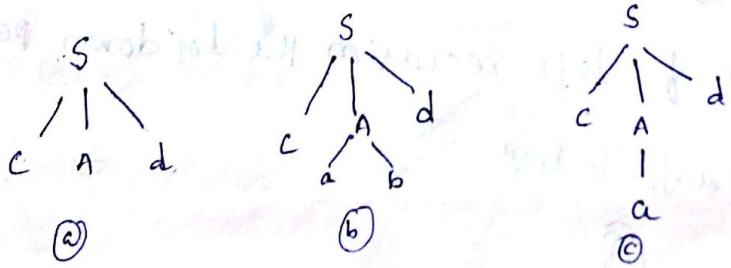
Top-Down Parsing :-

- Parse tree is generated from top to bottom (i.e root to leaves)
- The derivation terminates when the required input string terminates.

Ex:- Consider the grammar

$S \rightarrow cAd$
$A \rightarrow ab/a$ and input string is $w = cad$.

Sol:



(a) first leftmost leaf, labeled c, matches the first symbol of w we know move next i/p pointer to a. the second symbol of w, and consider next leaf, labeled A. We can then expand A using the first alternative for A to obtain the

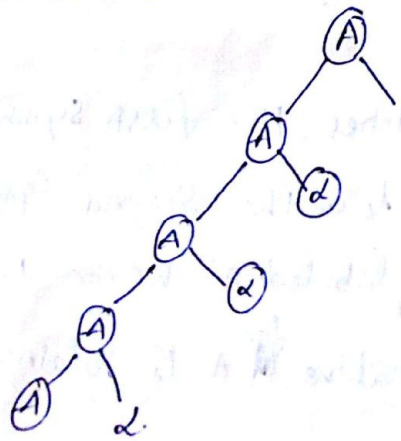
- In fig(b) generate a, b. So a match with a and b does not match with d, so we report error, and goback to A to see whether there is another alternative A.
- and repeat the same procedure above

Problems with Top down parsing :-

- There are some problems in top down parsing. In order to implement the parsing need to eliminate these problems
- So let us discuss these problems and how to remove them.

1) left recursion :-

- A grammar is left recursive if it has a non terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for string α.
- Here $\overset{+}{\Rightarrow}$ means deriving the input in one or more steps.
- If left recursion is present in the grammar then it creates serious problems.
- Because of left recursion the top down parser can enter in infinite loop.



Thus expansion of A causes further expansion of A only

- and, due to generation of $A, A\alpha, A\alpha\alpha, \ldots$
- This causes major problem in top down parsing

So elimination of left recursion is a must.

Elimination of left Recursion :-

- To eliminate left recursion we need to modify the grammar

Let, G be a context free grammar having a production rule with left recursion.

$$A \rightarrow A\alpha$$
$$A \rightarrow \beta$$

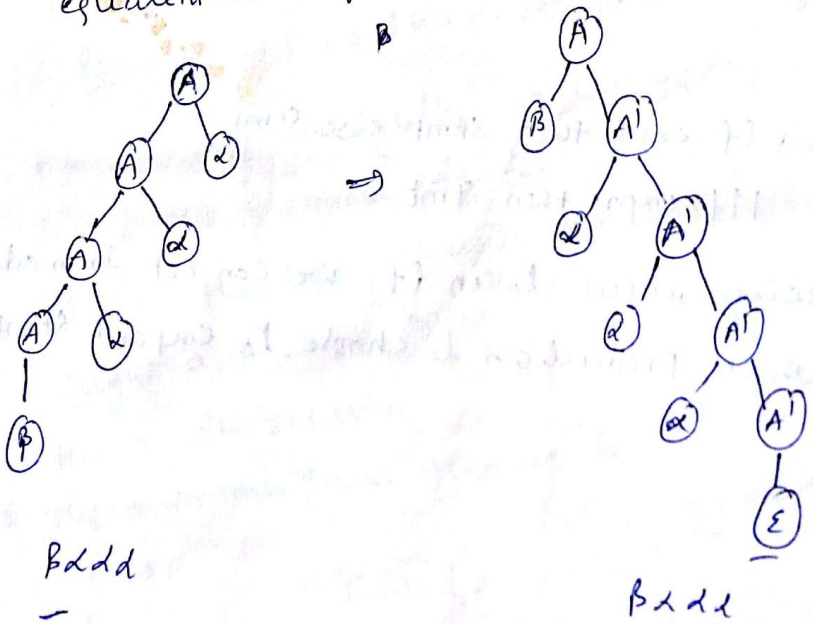
- The we eliminate left recursion by rewriting the production rule as

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'$$
$$A' \rightarrow \in$$

- without changing the set of strings derivable from A.

This rule by itself suffices in many grammars.

- We can also verify, whether the modified grammar is equalent to original or not. Eg string :- $\beta\ \alpha\ \alpha\ \alpha$ ---



$\beta\alpha\alpha\alpha$ $\beta\alpha\alpha\alpha$

Elimination of left recursion

Ex:- consider the grammar

$E \rightarrow E+T \mid T$

Sol:- We production in the form $A \rightarrow A\alpha \mid \beta$

$A = E \quad \alpha = +T \quad \beta = T$

Then the rule becomes,

$E \rightarrow T E'$

~~$E' \rightarrow \alpha E'$~~

$E' \rightarrow +T E'$

$E' \rightarrow \epsilon$

Ex:- $T \rightarrow T*F \mid F$

Sol:- $A = T \quad \alpha = *f \quad \beta = F$

$T \rightarrow F T'$

$T' \rightarrow *F T'$

$T' \Rightarrow \epsilon$

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

(2) Left factoring :-

The basic idea is that when it is not clear which of two alternative productions to use to expand a non terminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

Ex:-

Stmt → if expr then Stmt else Stmt

| if expr then Stmt.

on seeing input token if, we can not immediately tell which productions to choose to expand Stmt.

- If $A \to \alpha\beta_1 | \alpha\beta_2$ are two A-productions

  above grammar is left factored as

  $A \to \alpha A'$

  $A' \to \beta_1 | \beta_2$

- If $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \dots | \alpha\beta_n | \gamma$

  So left factored as

  $A \to \alpha A' | \gamma$

  $A' \to \beta_1 | \beta_2 | \dots | \beta_n$

- Ex:- $S \to iEtS | iEtSeS | a$

  $E \to b$

  So left factored grammar becomes

  $S \to iEtSS' | a$

  $S' \to eS | \epsilon$

  $E \to b$

Ambiguity Elimination

- The ambiguous grammar is not desirable in top down parsing. Hence we need to remove the ambiguity from the grammar if it is present.

  Ex:- $E \to E+E | E*E | id$ is an ambiguous grammar.

- For removing ambiguity we will apply one rule (i) If the grammar has left associative operators (+, -, *, /) then induce the left recursion and if the grammar has right associative operator (exponential operator) then induce the right recursion.

* The unambiguous grammar is

  $E \to E+T$

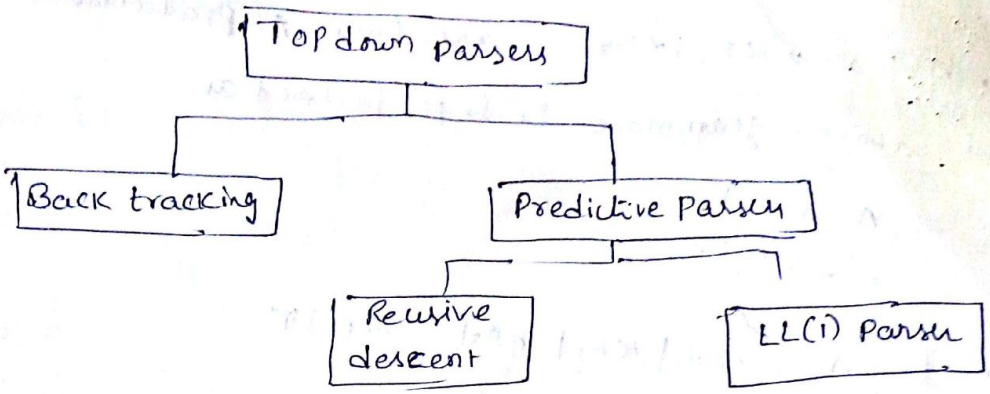
  $E \to T$

  $T \to T*F$

  $F \to id.$

Left associative: (+, -) operators which are evaluated from left to right

Right asso:



Types of top down Parsers

- **Back tracking :-**

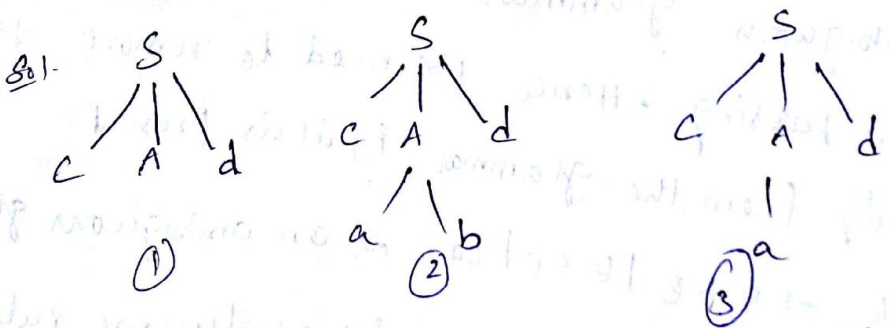
- Back tracking is a technique in which for expansion of non terminal symbol we choose one alternative and if some mismatch occures then we try another alternative if any.

Ex:- grammar

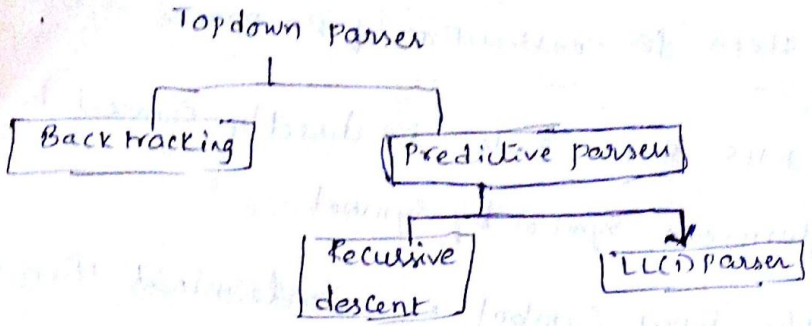
$S \rightarrow cAd$

$A \rightarrow ab / a$ and i/p String w = cad.

Sol.



- If for a non terminal there are multiple production rules begining with the same i/p symbol then to get the correct derivation we need to try all these alternatives
- In back tracking we need to move some levels upward inorder to check the possibilities



Back tracking :-

- Back tracking is a technique in which for expansion of non terminal symbol we choose one alternative and if some mismatch occures then we try another alternative if any.
- The back tracking is powerful than Predictive Parsing. but it is slower and it requires exponential time in general.

Predictive Parser :- Predictive Parser tries to Predict the next construction using one (or) more lookahead Symbols from input String.

Recursive Descent Parser:

- A parser that uses collection of recursive Procedures for parsing the given input string is called Recursive descent parser. A Procedure is associated with each nonterminal of a grammar.

Basic steps for construction of RD parser

The RHS of the rule is directly converted into Program code symbol by symbol

1) The input symbol is a nonterminal then a call to the procedure corresponding to non terminal is made.

2. If the input symbol is terminal then it is matched with the lookahead from input.

3. If the production rule has many alternatives then all these alternatives has to be combined into a single body of procedure

4. The parser should be activated by a procedure corresponding to the start symbol.

Pseudocode: 2.4 (P.NO: 45)

Ex:- E → num T
T → * num T | ∈

```
Procedure E
{ if lookahead = num then
  { Match (num);
    T;      /* calling procedure T */
  }
  else
  error
 if lookahead = $
  { declare Succes
  }
  else
   error;
}
Procedure T
{ if lookahead = '*';
  { Match('*')
    if ( lookahead = 'num'
```

```
    {
        match(num);
        T;
    }
    else
        error
}
else NULL
}
procedure match(token t)
{   if lookahead = t
    lookahead = next_token;
    else
    error
}
Procedure error
{ print ("error");
}
```

Pseudo Code for recursive descent parser

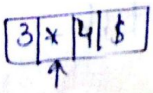
① we can see above code that the procedure for the non terminal are written and procedure body is simply mapping of RHS of the corresponding rule.

· consider input string is 3*4 we will parse this string using above grammar given procedures

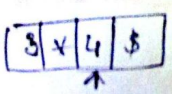
· The parser will be activated by calling procedure E.

· First input character 3 is matching with num the procedure match(num) will be invoked and then the lookahead will point to next token. And a call to procedure T is given.

E → num T

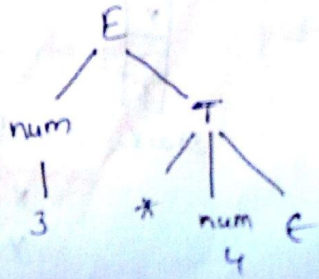


T → * num T



T → * num T

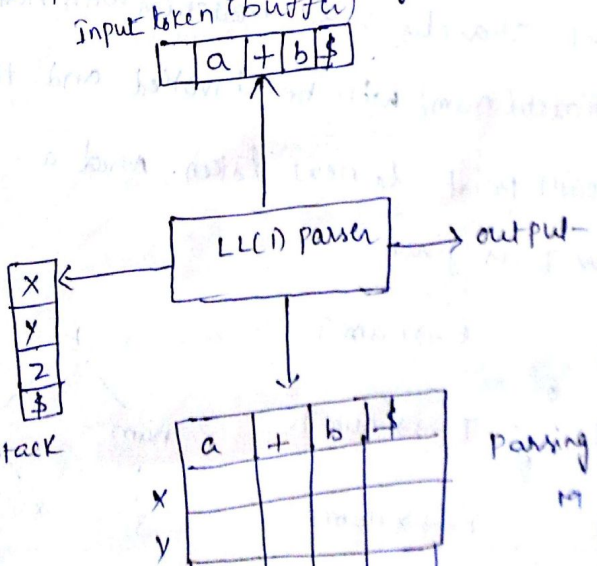
④ Declare Success



- A match with 'a' is found hence lookahead = next token.
- Now 'F' is matching with num hence again the procedure for match(num) is fulfilled. Then procedure for T is involved and T is replaced by ε.
- As lookahead points to $ we exit by reporting success.

predictive LL(1) parser :- (or) Table driven Parsing Algo:

- Top down parsing algorithm is of non recursive type.
- It is possible to build a non recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.
- In this type of parsing a table is built.
- LL(1) means
  1st L the input is scanned from left to right
  2nd L means it uses leftmost derivation for input string
  and 1 means it uses only one input symbol (lookahead) to predict the parsing process.



- Predictive parser has an input buffer, a stack, a parsing table and an output stream.
- The input buffer contains the string to be parsed, followed by $, a symbol is used as a right endmarker to indicate the end of the input string.
- Stack contains sequence of grammar symbols with $ in bottom, indicating bottom of stack. in reverse.
- Initially stack contains the start symbol of the grammar on top of $.
- The parsing table is a two dimensional array M[A,a] where A is a nonterminal and a is a terminal or the symbol $.
- The parser behaves is follows

The program consider X is symbol on top of stack and a is current input symbol. These two symbols determine the action of the parser.

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

Ex:- If $M[X, a] = \{X \to UVW\}$ the parser replaces X on top of stack by WVU (U on top).

As output assume that the parser just prints the production used; any other code could be executed here.

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

Construction of Predictive LL(1) Parser

The construction of a predictive parser aided by two functions associated with a grammar G. These functions FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G.

Following steps to Construction of Predictive LL(1) parser

1. Computation of FIRST and FOLLOW function.
2. Construct Predictive Parsing table using FIRST and FOLLOW functions.
3. Parse the input string with the help of predictive parsing table.

FIRST function

- FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S in derivation of α.

If $\alpha \Rightarrow \epsilon$ then ϵ is also in FIRST(α)

following are the rules used to compute the FIRST function

1. If the terminal symbol is a then FIRST(a) = {a}.
2. If there is a rule $X \rightarrow \epsilon$ then FIRST(X) = {ε}
3. For the rule $A \rightarrow X_1 X_2 X_3 \cdots X_K$ FIRST(A) = (FIRST(X_1) ∪ FIRST(X_2) ∪ FIRST(X_3) ··· FIRST(X_K))

Ex:- Grammar

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

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

Sol:- then FIRST(E) = {(, id)
FIRST(T) = {(, id}
FIRST(F) = {(, id}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}

FOLLOW function :-

- FOLLOW(A) is defined as the set of terminals symbols that appear immediately to the right of A
- following are the rules used to compute the FOLLOW function

1. For the start symbol S 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 every thing in FOLLOW(A) is in FOLLOW(B).

Ex:- $E \to E+T/T$
$T \to T*F/F$
$F \to (E)/id$

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

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

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

2. Construction of predictive parsing tables

Input: Grammar G.

Output: parsing table M

1. For each production $A \to \alpha$ of the grammar do step 2 and step 3
2. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A, a]
3. If ϵ is in FIRST(α), add $A \to \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and $ is in FOLLOW(A), add $A \to \alpha$ to M[A, $]
4. Make each undefined entry of M be error.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FIRST (E) = {(, id}

FIRST (T) = {(, id}

FIRST (F) = {(, id}

FIRST (E') = {+, ε}

FIRST (T') = {*, ε}

① FOLLOW (E) = {), $}

② $E \to TE'$ is applied to $A \to \alpha B \beta$ ($\alpha = A \to \alpha B \beta$) where FIRST of B contains ε then every thing in FOLLOW(A) is in FOLLOW(B) ↳ rule ③

∴ FOLLOW(E) = FOLLOW(E') = {), $}

③ FOLLOW (T) = {

$E \to TE'$ ∴ T follow E' ⇒ {), $}

$E' \to +TE'$

$A \to \alpha B \beta$ then everything in FIRST(β) except ε placed in FOLLOW(B) — rule ②

∴ FIRST(E') = {+}

∴ ~~FIR~~ FOLLOW(T) = {+,), $}

∴ $T \to FT'$ apply rule 3

$A \to \alpha \beta$ ⇒ everything in FOLLOW(A) is in FOLLOW(B)

∴ FOLLOW(T) = FOLLOW(T') = {+,), $}

④ FOLLOW(F) $T \to FT'$

F follow T' ⇒ {+,), $}

⇒ $T' \to *FT'$ apply rule 2

$T' \to * FT'$

$A \to \alpha B \beta$ then every thing in FIRST(β) except ϵ placed in FOLLOW(B)

$\therefore$ FIRST(T') = {*, ϵ}

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

09-281, 86, 75, 59, 89
67, 64, 71, 47
46, 49, 66, 65,
2e-1, 76, 77

(2) Construction of Predictive Parsing table.

NON TERMINAL	INPUT SYMBOL id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

(3) Parsing input string with the help of parsing table. id + id * id

STACK	INPUT	OUTPUT
$E	id + id * id $	~~$E \to TE'$~~
$ E T T'	id + id * id $	$E \to TE'$
$ E T T' F	id + id * id $	$T \to FT'$
$ E T T' id	id + id * id $	$F \to id$
$ E T T'	+ id * id $	
$ E T'	+ id * id $	$T' \to \epsilon$
$ E' T +	+ id * id $	$E' \to +TE'$
$ E' T	id * id $	
$ E' T' F	id * id $	$T \to FT'$
$ E' T' id	id * id $	$F \to id$
$ E' T'	* id $	
$ E' T' F *	* id $	$T' \to *FT'$
$ E' T' F	id $	
$ E' T' id	id $	$F \to id$
$ E' T'	$	
$ E'	$	~~$T' \to \epsilon$~~ $T' \to \epsilon$

$S \to (L) / a$
$L \to L, S / S$
$S/P = (a, a)$

SP 56633

13/02
47, 50, 53, 54,
55, 63, 65, 68

BOTTOM UP PARSING

Bottom up parsing:-

Parse tree is generated from input string and the derivation is terminated when starting symbol is derived from input string.

Ex1:- Consider a grammar $S \to aABe$, $A \to Abc|b$, $B \to d$ and input string is abbcde.

Sol:- Given CFG is

$S \to aABe$
$A \to Abc|b$
$B \to d$

b,d – substring

$\Rightarrow a \underrightarrow{b} b\ cde$
$\Rightarrow a \underrightarrow{Abc}\ de$
$\Rightarrow a\ \underrightarrow{Ade}$
$\Rightarrow aABe$
$\Rightarrow S$

Ex2:- Consider a grammar $E \to E+E|E*E|id$, consider a input string id+id*id.

Sol:-

Right sentential form (input buffer)	Handle	Production
id + id*id	id	E→ id
E+ id *id	id	E→ id
E+E *id	id	E→ id
E+ E*E	E*E	E→ E*E
E+E	E+E	E→ E+E.
E		

Types of Bottom up Parsing Techniques:-

1) Shift Reduce parser
2) Operator precedence parsing
3) LR-Parser

1.) **Shift-Reduce Parser:-** One of the parsing technique to construct parse tree from bottom to root node. There are some basic keywords in this shift, reduce, accept, error.

Shift:- If terminal symbol is in input buffer we shift it to top of stack.

Reduce:- The input string (or) substring or terminal, we move is handle. We can reduce handle with appropriate non-terminal symbol. This occurs when top of stack is handle and input buffer has string.

Accept:- If top of stack is non-terminal i.e; starting symbol and i/p buffer has $ or entry then it is said to be accept i.e; successful.

Error:- If shift, reduce, accept are doesnot performed then 'error' is existed.

Ex:- The CFG is $E \to E+E \mid E*E \mid id$, Use shift reduce parser of input string id + id * id.

stack	i/p buffer	action
$	id+id * id$	shift (id)
$id	+Id* id$	reduce E→id
$ E	+id * id$	
$ E+	id *id $	shift (+)
$E+id	*id $	shift id
$E+E	* id $	Reduce E→id

stack	i/p buffer	action
$E+E *	id$	shift *
$E+E*id	$	shift id
$E+E*E	$	reduce E→id
$ E+E	$	reduce E→E*E
$E	$	reduce E→E+E
$E	$	accept

2) Operator Precedence Parser:-

Basing on operator's precedence, we are parsing here. It follows 2 rules.

i.) No ε on RHS of production (A→ε)

ii.) No immediate non-terminals are in RHS of production (A→XYZ)

If ε or immediate non-terminals are in CFG we have to rewrite this CFG.

Ex:- CFG be E→ EAE | (E)
A→ + | - | * | ↑ | ----

Sol:- After rewriting CFG then,

E→ E+E | E*E | E-E | E↑E
E→ (E).

- There will be a precedence table for operators. At first input string is scanned from left t right. At first operator of higher precedence is identified and then back tracking for operator lowest precedence is identified basing on precedence table.

• Let the precedence table for above example is

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

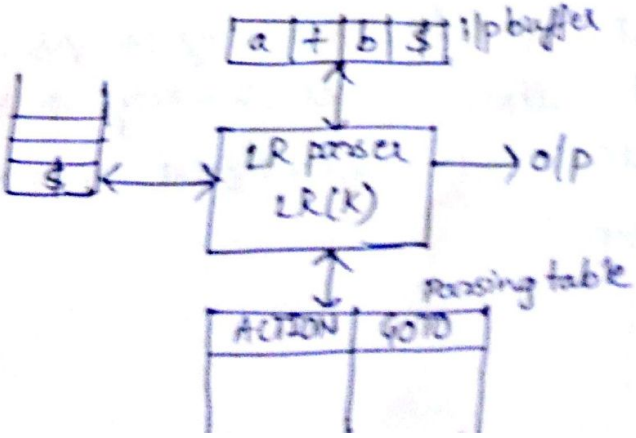
let the input string be id+id*id, now place $ as endmarker and starting of i/p string as follows

$id+id*id$.

$ id + id * id $
$ <·id·> + <·id·> * <·id·> $
$ E + <·id·> * <·id·> $
$ E + E * <·id·> $ (reduce E → id)
$ E + E * E $
$ + * $ (after eliminating all non-terminals).
$ <·+ <·* ·> $ (reduce E → E*E)
$ <·+ $
$ <·+·> $ (reduce E → E+E)
$ $

3) LR(K) parser:-

L - scanning input from left to right
R - applying RMD of i/p string in reverse order
K - number of i/p symbols (assume K=1)



GOTO (I, Grammar) = location
(I → number of states)

- GOTO is used to move from one state to another state.
- There are several parsing techniques in LR(k) parser. They are
  i.) simple LR (SLR)
  ii.) Canonical LR (CLR)
  iii.) Look ahead LR (LALR)
- Dot(·) operator is used for above parsing techniques.

The terminologies involved in this parsing techniques are

i.) LR(0) items :- $A \to \alpha$

we can place . (dot) operator in RHS of production in anyway.

$A \to \cdot\alpha$
$A \to \alpha\cdot$

Ex:- If $A \to AB$
$A \to .AB$
$A \to A.B$
$A \to AB.$
} These productions are called as LR(0) items.

2.) Augmented Grammar :-

It is a new grammar. Let a grammar is in form $G \to S$, where 'G' is grammar, starting symbol of G is 'S'. Then augmented grammar of G is $G' \to S' \to S$.

Ex:- Let $G \to A \to \alpha$
then G' is $A' \to A$
$A \to \alpha$

Kernel items :- which include the initial item, $S' \to .S$, and all items whose dots are not at the left end.

Ex:- $A \to AB$
$A' \to .A$
~~$A' \to .AB$~~
$A' \to A.B$
$A' \to AB.$

4.) Non-Kernel items:- These will be . (dot) operator on LHS (Left side) of RHS of production rule.

Ex:- $A \to \cdot\alpha$
$A \to \cdot AB$

5.) Closure operation:-

closure items is calculated on grammar 'G'.

(i) • Let initially 'G' has items in I. added to closure (I)
∴ closure (I) = set of items in G.

• If production is in form of $A \to \alpha \cdot B\beta$ is in closure (I) and $B \to \gamma$, now add $B \to \cdot\gamma$ to closure (I).

Ex:- $E' \to E$
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

$\longrightarrow$

$E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot id$

Ex2:- $A \to X \cdot YZ$
$Y \to a$

$\to$

$A \to X \cdot YZ$
$Y \to \cdot a$

• GOTO is a keyword having arguments of set of items and grammar symbol. Moving symbol to next location.

GOTO (set of items, grammar symbol)

GOTO $(A \to \alpha \cdot B\beta, B) = A \to \alpha B \cdot \beta$

1.) Simple LR:-

• constructing the canonical set of items
• constructing the parsing table
• parsing i/p string using parsing table.

constructing the canonical set of items:-

Ex:- $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E)$
$F \to id$

} be the grammar. The augmented grammar here is $E' \to E$.

- The augmented grammar for above grammar is, the closure of above grammar is

(Ast state $\leftarrow I_0$,

$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

calculating I_1 :-

I_1: GOTO (I_0, E)

$\boxed{E' \rightarrow E.}$
$E \rightarrow E.+T$

I_2: GOTO (I_0, T)

$T \rightarrow T.*F$
$\boxed{E \rightarrow T.}$

I_3: GOTO (I_0, F)

$\boxed{T \rightarrow F.}$

I_4: GOTO $(I_0, ()$

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

I_5: GOTO (I_0, id)

$\boxed{F \rightarrow id.}$ final state for I_0.

I_6: GOTO $(I_1, +)$

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

I_7: GOTO $(I_2, *)$

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

... the parsing table using set ...

I_8 : GOTO (I_4, E)

$F \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

i) GOTO $(I_4, T) \rightarrow I_2$

ii) GOTO $(I_4, F) \rightarrow I_3$

iii) GOTO $(I_4, ($ $) \rightarrow I_4$

iv) GOTO $(I_4, id) \rightarrow I_5$

I_9 : GOTO (I_6, T)

$E \rightarrow E + T \cdot$

$T \rightarrow T \cdot * F$

i) GOTO $(I_6, F) = I_3$

ii) GOTO $(I_6, ($ $) = I_4$

iii) GOTO $(I_6, id) = I_5$

I_{10} : GOTO (I_7, F)

$E \rightarrow T * F \cdot$

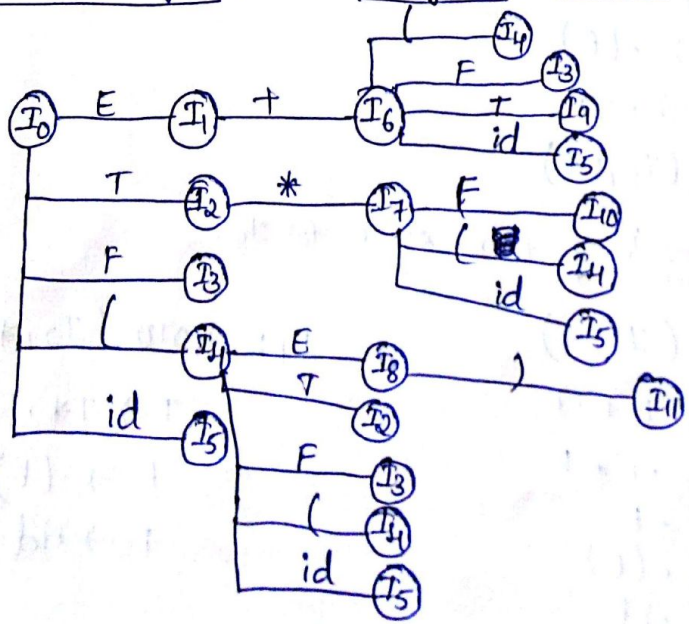
i) GOTO $(I_7, ($ $) \rightarrow I_4$

ii) GOTO $(I_7, id) \rightarrow I_5$

I_{11} : GOTO $(I_8,))$

$F \rightarrow (E) \cdot$

Data Flow Analysis (DFA) diagram based on the set of items:



Constructing the parsing table using set of items:-

ACTION Rules:-

1.) If $A \to \alpha \cdot a\beta$ is in I_i state and $GOTO(I_i, a) = I_j$ then $ACTION[I_i, a] = shift\ j$.

Ex:- if I_0 is state and $GOTO(I_0, a)$ (a should be only terminal symbol) then here grammar is $A \to \alpha \cdot a\beta$, $A[I_0, a] = S_1$ where $j=1$, S = shift

2.) If $A \to \alpha \cdot$ is in goto final state of any state. $A \to \alpha \cdot$ is I_i then $ACTION(I_i, a)$ (a → terminal symbol) = reduce the rule ($A \to \alpha$) where $a \in FOLLOW(A)$.

3.) If there is an augmented grammar $S' \to S.$ is in I_i state then $ACTION[I_i, \$] = accept$,

($S' \to S.$ is in I_i state then $ACTION[I_i, a]$ = reduce rule $A \to \alpha$, $a \in FOLLOW(A)$ where $FOLLOW(A) = \$$ then accept).

GOTO Rules:-

1.) $GOTO(I_i, E) = I_j$, then $GOTO[I_i, E] = j$.

$E' \to E$

$E \to E+T \to rule1\ (r_1)$

$E \to T \to rule2\ (r_2)$

$T \to T*F \to rule3\ (r_3)$

$T \to F \to rule4\ (r_4)$

$F \to (E) \to rule5\ (r_5)$

$F \to id \to rule6\ (r_6)$

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

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

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

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

Parsing table:-

states	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
I_1		S_6				accept			
I_2		r_2	S_7		r_2	r_2			
I_3		r_4	r_4		r_4	r_4			
I_4	S_5			S_4			8	2	3
I_5		r_6	r_6		r_6	r_6			
I_6	S_5			S_4				9	3
I_7	S_5			S_4					10
I_8					S_{11}				
I_9		r_1			r_1	r_1			
I_{10}		r_3	r_3		r_3	r_3			
I_{11}		r_5	r_5		r_5	r_5			

For rule-1:- $E \to E+T$ at I_9 [I_9, a] ; FOLLOW(A)

GOTO (I_9, +)

(I_9,))

[I_9, $]

For rule-2:- $E \to T$ at I_2, GOTO [I_2, +],

GOTO [I_2,)]

GOTO [I_2, $]

For rule-3:- $T \to T*F$ at I_{10}

GOTO [I_{10}, *], GOTO [I_{10}, +], GOTO [I_{10},)], GOTO [I_{10}, $]

For rule-4:- $T \to F$ at I_3

GOTO [I_3, +], GOTO [I_3, *], GOTO [I_3,)], GOTO [I_3, $]

For rule-5:- $F \to (E)$ at I_{11}

GOTO $[I_{11}, +]$, GOTO $[I_{11}, *]$, GOTO $[I_{11},)]$, GOTO $[I_{11}, \$]$

For rule-6:- $F \to id$ at I_5

GOTO $(I_5, +)$, GOTO $[I_5, *]$, GOTO $[I_5,)]$, GOTO $[I_5, \$]$.

EX:- write a grammar

$E \to E$ sub E sup E

$E \to E$ sub E

$E \to E$ sup E

$E \to \{E\}$

$E \to c$

- parsing the i/p string with the help of parsing table.

The input string is id * id + id

stack	i/p buffer	ACTION	GOTO	Parsing ACTION
$	id * id + id $			
$0	id * id + id $	A[0,id] = S_5		shift id
$0 id5	* id + id $	A[5,*] = r_6	G[0,F] = 3	reduce F → id
$0 F3	* id + id $	A[3,*] = r_4	G[0,T] = 2	reduce T → F
$0 T2	* id + id $	A[2,*] = S_7		shift *
$0 T2 * 7	id + id $	A[7,id] = S_5		Shift id
$0 T2 * 7 id5	+ id $	A[5,+] = r_6	G[7,F] = 10	reduce F → id
$0 T2 * 7 F10	+ id $	A[10,+] = r_3	G[0,T] = 2	reduce T → T * F
$0 T2	+ id $	A[2,+] = r_2	G[0,E] = 1	reduce E → T
$0 E1	+ id $	A[1,+] = S_6		shift +
$0 E1 + 6	id $	A[6,id] = S_5		shift id
$0 E1 + 6 id5	$	A[5,$] = r_6	G[6,F] = 3	reduce F → id
$0 E1 + 6 F3	$	A[3,$] = r_4	G[6,T] = 9	reduce T → F
$0 E1 + 6 T9	$	A[9,$] = r_1	G[0,E] = 1	reduce E → E
$0 E1	$	A[1,$] = accept		

ii.) Canonical LR:- CLR also called as LR parsers.

- construction of canonical set of items along with look ahead symbol.
- constructing parsing table
- parsing i/p string using parsing table.

→ difference between SLR and CLR is, in SLR we construct parsing table using FOLLOW, In CLR while doing set of items only we recognize lookahead symbol.

1.) closure :- $[A \to \alpha \cdot X\beta, a]$

↳ $X \to \gamma$ → add $[X \to \cdot\gamma, b]$

↳ $b \in FIRST(\beta a)$

2.) GOTO $[A \to \alpha \cdot X\beta, X]$

$A \to \alpha X \cdot \beta$

- If any rule in form of $[A \to \alpha \cdot X\beta, a]$ where 'a' is a look ahead symbol is in I_0 and X having rule $X \to \gamma$, then add $[X \to \cdot\gamma, b]$ to closure (I) where $b \in FIRST(\beta a)$
- GOTO of the rule $[A \to \alpha \cdot X\beta, X]$ is equivalent to $[A \to \alpha X \cdot \beta, X]$

Ex:- Let us consider a CFG,

$S' \to S$

$S \to CC$

$C \to ac/d$

Sol:-

$S' \to S$

$S \to CC$ ⇒ $S' \to S$ is augmented grammar, so add $ where $ is lookahead symbol since for starting symbol FOLLOW is $.

$C \to ac/d$

I_0: $S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot aC, a/d$

$C \to \cdot d, a/d$

$A = S', \alpha = \epsilon, X = S, \beta = \epsilon, a = \$$

$b = FIRST(c\$) = \$$

$A = C, \alpha = \epsilon, X = a, \beta = C, a = \$$

$b = FIRST(C) = a/d$

I_1: GOTO (I_0, S)

$S' \to S\cdot, \$$

I_2: GOTO (I_0, C)

$S \to C\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

$A = S,\ \alpha = C,\ X = C,\ \beta = \epsilon,\ a = \$$

$b = FIRST(\beta, a)$

$FIRST(\epsilon, \$) = \$$

I_3: GOTO (I_0, a)

$C \to a\cdot C, a/d$

$C \to \cdot aC, a/d$

$C \to \cdot d, a/d$

$A = C,\ \alpha = a,\ X = C,\ \beta = \epsilon,\ a = a/d$

$b = FIRST(\epsilon, a/d)$

$b = a/d$.

I_4: GOTO (I_0, d),

$C \to d\cdot, a/d$

I_5: GOTO (I_2, C)

$S \to CC\cdot, \$$

I_6: GOTO (I_2, a)

$C \to a\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

$A = C,\ \alpha = a,\ X = C,\ \beta = \epsilon,\ a = \$$

$b = FIRST(\epsilon, \$) = \$$

I_7: GOTO (I_2, d)

$C \to d\cdot, \$$

I_8: GOTO (I_3, C)

$C \to aC\cdot, a/d$

$C \to a\cdot C, a/d$

i) GOTO $(I_3, a) \to (I_3)$

ii) GOTO $(I_3, d) \to (I_4)$

$C \to d\cdot, a/d$

I_9: GOTO (I_6, C) $C \to aC\cdot, \$$

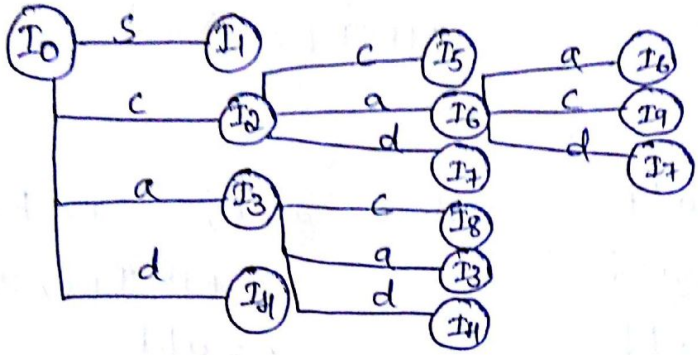
7) GOTO $(I_6, a) \rightarrow I_6$

$C \rightarrow a \cdot C, \$$

$C \rightarrow \cdot aC, \$$

$C \rightarrow \cdot d, \$$

Data flow analysis diagram:-



states	ACTION			GOTO	
	a	d	$	S	C
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Input string be aadd.

Stack	input string	ACTION	GOTO	Parsing Action
$	aadd$			
$0	aadd$	A[0,a]=S3		shift a
$0a3	add$	A[3,a]=S3		shift a
$0a3a3	dd$	A[3,d]=S4		shift d
$0a3a3d4	d$	A[4,d]=r3	G[3,C]=8	reduce C→d
$0a3a3C8	d$	A[8,d]=r2	G[3,C]=8	reduce C→a
$0a3C8	d$	A[8,d]=r2	G[0,C]=2	reduce C→a
$0C2	d$	A[2,d]=S7		shift d
$0C2d7	$	A[7,$]=r3	G[2,C]=5	reduce C→d
$0C2C5	$	A[5,$]=r1	G[0,S]=1	reduce S→C
$0S1	$	A[1,$]=accept		

iii) Look ahead LR (LALR):-

Here we merge identical closures with set of items also if look ahead symbols are different.

- Construction of canonical set of items along with look ahead symbol.
- constructing the parsing table
- parsing the i/p string by using table.

Ex:- Let grammar be S→CC
C→aC
C→d.

Sol:- Augmented grammar is

S'→S
S→CC
C→aC
C→d

I_0: $S' \to .S, \$$

$S \to .CC, \$$

$C \to .aC, a/d$

$C \to .d, a/d$

I_1: GOTO (I_0, S)

$\boxed{S' \to S., \$}$

I_2: GOTO (I_0, C)

$S \to C.C, \$$

$C \to .aC, \$$

$C \to .d, \$$

I_3: GOTO (I_0, a)

$C \to a.C, a/d$ $\quad A=C,\ \alpha=a,\ X=C,\ \beta=\epsilon,\ a=a/d$

$C \to .aC, a/d$ $\quad b = FIRST(\epsilon, a/d) = a/d$

$C \to .d, a/d$

I_4: GOTO (I_0, d) $\quad A=C,\ \alpha=d,\ X=\epsilon,\ \beta=\epsilon,\ a=a/d$

$\boxed{C \to d.}, a/d$ $\quad b = FIRST(\epsilon, a/d) = a/d.$

I_5: GOTO (I_2, C)

$\boxed{S \to CC.}, \$$ $\quad A=S,\ \alpha=CC,\ X=\epsilon,\ \beta=\epsilon,\ a=\$$

I_6: GOTO (I_2, a)

$C \to a.C, \$$ $\quad A=C,\ \alpha=a,\ B=C,\ \beta=\epsilon,\ a=\$$

$C \to .aC, \$$ $\quad b = FIRST(\epsilon, \$) = \$$

$C \to .d, \$$

I_7: GOTO (I_2, d)

$\boxed{C \to d., \$}$

I_8: GOTO (I_3, C) $\quad A=C,\ \alpha=aC,\ X=\epsilon,\ \beta=\epsilon,\ a=a/d$

$\boxed{C \to aC., a/d}$ $\quad b = FIRST(\epsilon, a/d) = a/d.$

i) GOTO $(I_3, a) = I_3 \to C \to a.C, a/d$

ii) GOTO $(I_3, d) = I_4 \to C \to d., a/d$

I_9: GOTO (I_6, C)

$C \to aC\cdot, \$$

GOTO $(I_6, a) \to (I_6)$

$C \to a\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

Here I_3, I_6 have same set of items, and I_4, I_7 have same set of items, where I_8, I_9 have same set of items, so we have to merge them, then

I_{36}: $C \to a\cdot C, a/d/\$$

$C \to \cdot aC, a/d/\$$

$C \to \cdot d, a/d/\$$

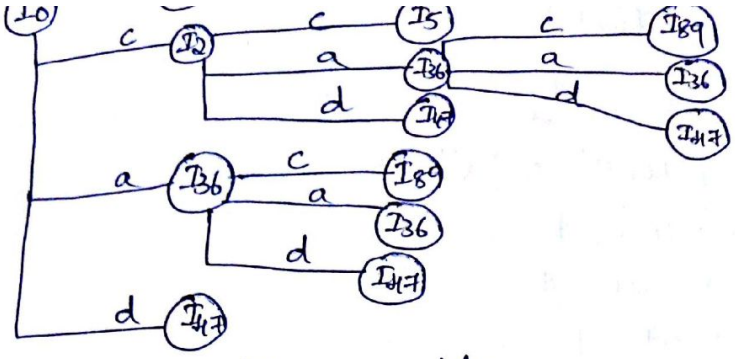
I_{47}: $C \to d\cdot, a/d/\$$

I_{89}: $C \to aC\cdot, a/d/\$$

- Now the number of states are 7, they are $I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}$.

Constructing a parsing table using these states:-

states	ACTION			GOTO	
	a	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		



Parsing the input string aadd:-

Parsing stack	i/p buffer	ACTION	GOTO	Parsing Action
$	aadd$			
$0	aadd$	A[0,a] = S36		shift a
$0a36	add$	A[36,a] = S36		shift a
$0a36a36	dd$	A[36,d] = S47		shift d
$0a36a36d47	d$	A[47,d] = r3	G[36,C] = 89	reduce C→d
$0a36a36C89	d$	A[89,d] = r2	G[36,C] = 89	reduce C→ac
$0a36C89	d$	A[89,d] = r2	G[0,C] = 2	reduce C→ac
$0C2	d$	A[2,d] = S47		shift d
$0C2d47	$	A[47,$] = r3	G[2,C] = 5	reduce C→d
$0C2C5	$	A[5,$] = r1	G[0,S] = 1	reduce S→C
$0S1	$	A[1,$] = accept		

Handling the ambiguous grammar:-

Ex:-

E → E+E
E → E*E
E → (E)
E → id

(SLR parser)

Sol:-

E → E+E
E → E*E
E → (E)
E → id

→

I0 :
E' → .E
E → .E+E
E → .E*E
E → .(E)
E → .id

I_1: GOTO (I_0, E)

$E' \to E.$ final state

$E \to E. + E$

$E \to E. * E$

I_2: GOTO $(I_0, ()$

$E \to (.E)$

$E \to .E+E$

$E \to .E*E$

$E \to .(E)$

$E \to .id$

I_3: GOTO (I_0, id)

$E \to id.$

I_4: GOTO $(I_1, +)$

$E \to E+.E$

$E \to .E+E$

$E \to .E*E$

$E \to .(E)$

$E \to .id$

I_5: GOTO $(I_1, *)$

$E \to E*.E$

$E \to .E+E$

$E \to .E*E$

$E \to .(E)$

$E \to .id$

I_6: GOTO (I_2, E)

$E \to (E.)$

$E \to E.+E$

$E \to E.*E$

i) GOTO $(I_2, () = I_2$

ii) GOTO $(I_2, id) = I_3$

I_7: GOTO (I_4, E)

$E \to E+E.$

$E \to E.+E$

$E \to E.*E$

i) GOTO $(I_4, () = I_2$

ii) GOTO $(I_4, id) = I_3$

I_8 : GOTO (I_5, E)

$E \to E * E.$

$E \to E . + E$

$E \to E . * E$

i) GOTO $(I_5, ()) = I_2$

ii) GOTO $(I_5, id) = I_3$

I_9 : GOTO $(I_6,))$

$E \to (E).$

states	ACTION						GOTO
	+	*	(	)	id	$	E
I_0			S_2		S_3		1
I_1	S_4	S_5				accept	
I_2			S_2		S_3		6
I_3	r_4	r_4		r_4		r_4	
I_4			S_2		S_3		7
I_5			S_2		S_3		8
I_6				S_9			
I_7	r_1 or S_4	r_1 or S_5		r_1		r_1	
I_8	r_2 or S_4	r_2 or S_5		r_2		r_2	
I_9	r_3	r_3		r_3		r_3	

$E' \to E$

$E \to E + E$ (r_1)

$E \to E * E$ (r_2)

$E \to (E)$ (r_3)

$E \to id$ (r_4)

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

parsing input string id * id + id

stack	i/p string	ACTION	GOTO	parsing action
$	id * id + id $			
$0	id * id + id $	A[0,id] = S3		shift id
$0id3	* id + id $	A[3,*] = r4	G[0,E] = 1	reduce E→id
$0E1	* id + id $	A[1,*] = S5		shift *
$0E1*5	id + id $	A[5,id] = S3		shift id
$0E1*5id3	+ id $	A[3,+] = r4	G[5,E] = 8	reduce E→id
$0E1*5E8	+ id $	A[8,+] = r2	G[0,E] = 1	reduce E→E*E
$0E1	+ id $	A[1,+] = S4		shift +
$0E1+4	id $	A[4,id] = S3		shift id
$0E1+4id3	$	A[3,$] = r4	G[4,E] = 7	reduce E→id
$0E1+4E7	$	A[7,$] = r1	G[0,E] = 1	reduce E→E+E
$0E1	$	A[1,$] = accept		

Let the input buffer be id + id * id

stack	i/p buffer	ACTION	GOTO	Parsing Action
$	id + id * id $			
$0	id + id * id $	A[0,id] = S3		shift id
$0id3	+ id * id $	A[3,+] = r4	G[0,E] = 1	reduce E→id
$0E1	+ id * id $	A[1,+] = S4		shift +
$0E1+4	id * id $	A[4,id] = S3		shift id
$0E1+4id3	* id $	A[3,*] = r4	G[4,E] = 7	reduce E→id
$0E1+4E7	* id $	A[7,*] = S5		shift *
$0E1+4E7*5	id $	A[5,id] = S3		shift id
$0E1+4E7*5id3	$	A[3,$] = r4	G[5,E] = 8	reduce E→id
$0E1+4E7*5E8	$	A[8,$] = r2	G[4,E] = 7	reduce E→E*E
$0E1+4E7	$	A[7,$] = r1	G[0,E] = 1	reduce E→E+E
$0E1	$	A[1,$] = accept		

- If action result is shift (s) then first shift that terminal to top of stack and then shift subscript of that s to top of stack.
- If action result is reduce (r), then take rule of particular rule (subscript), in that rule. Consider RHS term, pop that term from stack, then now consider top of that stack, and non-terminal of the rule to perform GOTO, now push non-terminal in LHS of rule to top of stack and then push result of GOTO to top of stack.

Hierarchy of the Parsing techniques:-

SLR	CLR	LALR
1) SLR is smallest in size	1) CLR is largest in size	1) LALR is same as SLR i.e., smallest in size
2) It is easiest method based on FOLLOW function	2) CLR is most powerful than SLR and LALR	2) It is applicable to wider class than SLR.
3) It displays less syntactical errors	3) It displays less syntactical features than that of LR parser.	3) It displays most syntactical features of a language, are expressed in LALR
4) Error detection is not immediate in SLR	4) Immediate error detection in CLR	4) Same as SLR, Error detection is not immediate in SLR.
It requires less time and space complexity	5) It requires more time and space complexity	5) Time and space complexity are more in LALR but efficient methods exis for constructing LALR parsers directly