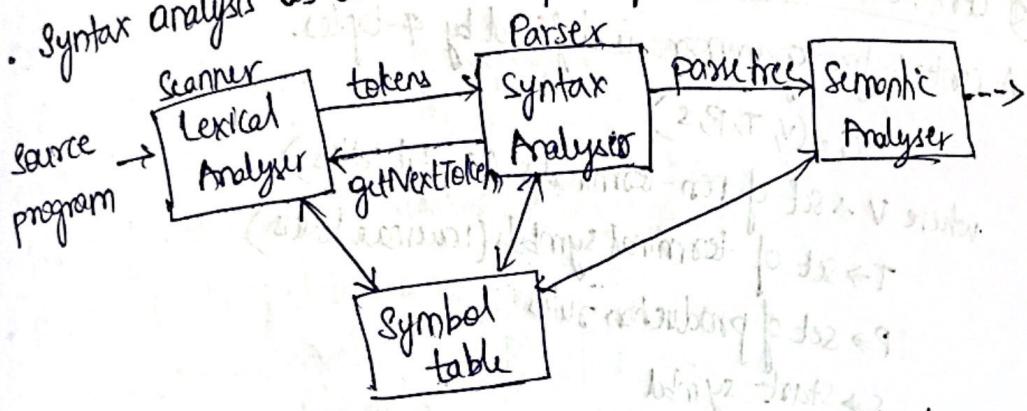


## Unit-2

### Syntax Analysis

#### ① Role of Syntax Analyser (Parser):

- Syntax analysis is also called parsing.



- The lexical analyzer reads the source program character by character and produces a token.
- The symbol table stores the information about tokens (tokenName, token attribute).
- Syntax Analyzer (Parser) receives token and sends request (getNextToken) to Lexical Analyzer. The lexical analyzer responds to the request by sending another token.
- Parser for any grammar that takes tokens produced by the lexical analyzer as input, produce either a parse tree or an error message as output.
- The Syntax analyzer checks whether tokens are following the language syntax rules or not.
- If they aren't following the language syntax rules, then error handler reports those errors to the user.

If the tokens follow the language syntax rules then it will generate a parse tree / syntax tree.

## ② Context-Free Grammars (CFG) and Writing Grammar

A context-free grammar is defined by 4-tuples.

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

where  $V \rightarrow$  set of non-terminals (Capital letters)

$T \rightarrow$  set of terminal symbols (Lowercase letters)

$P \Rightarrow$  set of production rules

$S \rightarrow$  start symbol

A production rule is represented in form of

$$A \rightarrow \alpha$$

where A is non-terminal

Conventions:

- Terminals: Terminals are symbols from which strings are formed.
  - Lowercase letters i.e., a, b, c
  - Operators i.e., +, -, \*
  - Punctuation symbols i.e., comma, parenthesis
  - Digits i.e., 0, 1, 2, ..., 9
  - Boldface letters i.e., id, if
- Non-terminals: Are variables that denote a set of strings.
  - Uppercase letters i.e., A, B, C
- Start symbol: Start symbol is the head of the production stated first in the grammar.

• Production: Production is of the form LHS  $\rightarrow$  RHS (or) head + body, where head contains only one non-terminal and body, contains a collection of terminals and non-terminals.

Derivation:

- A derivation is the process of applying a sequence of production rules to derive a string.
- Process of derivation always starts from start symbol.
- Derivation is classified into two types:
  1. Left-most derivation
  2. Right-most derivation

leftmost derivation:

On the left-most derivation, the input is scanned and replaced with the production rule from left to right so, in left most derivatives we read the input string from left to right.

Example: Production rules:

$$S = S + S$$

$$S = S - S$$

$$S = a/b/c$$

input:  $a - b + c$

the leftmost derivation is

$$S = S + S$$

$$S = S - S + S$$

$$S = a - S + S$$

$$S = a - b + S$$

$$S = a - b + c$$

## Right-most derivation:

In right-most derivation, the input is scanned and replaced with the production rule from right to left. So in right-most derivation we read the input string from right to left.

Example: Production rules

$$S = S + S$$

$$S = S - S$$

$$S = a - b + c$$

Input:  $a - b + c$

The right-most derivation is:

$$S = S - S$$

$$S = S - S + S$$

$$S = S - S + C$$

$$S = S - b + C$$

$$S = a - b + C$$

## Parse tree / derivation tree:

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parse, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It's the graphical representation of symbol that can be terminals or non-terminals.

Parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Yield of the tree: The yield of the tree is the collection of leaf nodes from left to right.

Qn: Consider the following grammar.

$$S \rightarrow 0A \mid 1B \mid 011$$

$$A \rightarrow 0S \mid 1B \mid 1$$

$$B \rightarrow 0A \mid 1S$$

Construct left-most derivation and parse tree for the following.

(i) 0101

0101 (LMD)

$$S \rightarrow 0A$$

$$S \rightarrow 01B$$

$$S \rightarrow 010A$$

$$S \rightarrow 0101$$

(ii) 1100101 (LMD)

$$S \rightarrow 1B$$

$$S \rightarrow 11S$$

$$S \rightarrow 110A$$

$$S \rightarrow 1100S$$

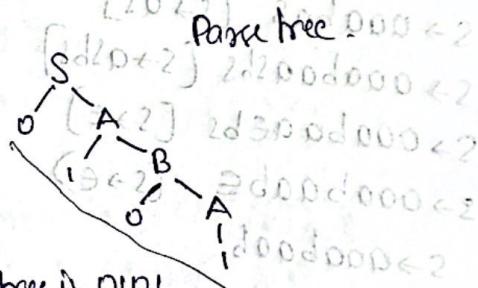
$$S \rightarrow 11001B$$

$$S \rightarrow 110010A$$

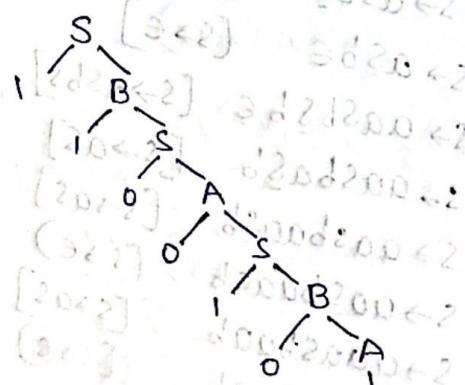
$$S \rightarrow 1100101$$

(ii) 1100101

Yield of tree is 0101



Parse tree



Yield of parse tree:

1100101

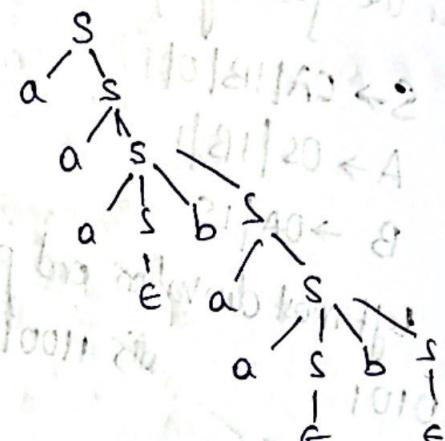
Ex: Design LND & RMD for the string aaabaab.  
 grammar is  $S \rightarrow aS / aSbS / \epsilon$

Check whether the string aaabb is accepted or not.

Q: aaabaab

LND:

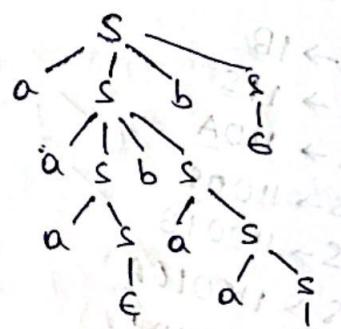
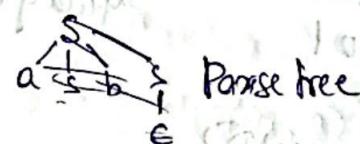
- $S \rightarrow aS$  [ $S \rightarrow aS$ ]
- $S \rightarrow aS$  [ $\vdash S \rightarrow aS$ ]
- $S \rightarrow aaSbS$  [ $\vdash S \rightarrow aSbS$ ]
- $S \rightarrow aaa\epsilon bS$  [ $S \rightarrow \epsilon$ ]
- $S \rightarrow aaabS$  [ $S \rightarrow aS$ ]
- $S \rightarrow aaabaasS$  [ $S \rightarrow aSbS$ ]
- $S \rightarrow aaabaaeS$  [ $S \rightarrow \epsilon$ ]
- $S \rightarrow aaabaae\epsilon$  [ $S \rightarrow \epsilon$ ]
- $S \rightarrow aaabaaab$



aaabaab

RMD:

- $S \rightarrow aSbS$  [ $S \rightarrow aSbS$ ]
- $S \rightarrow aS\epsilon$  [ $S \rightarrow \epsilon$ ]
- $S \rightarrow aSbS\epsilon$  [ $S \rightarrow aSbS$ ]
- $S \rightarrow aSbaS$  [ $S \rightarrow aS$ ]
- $S \rightarrow aSbaaS$  [ $S \rightarrow aS$ ]
- $S \rightarrow aSbaaeb$  [ $S \rightarrow \epsilon$ ]
- $S \rightarrow aaSbaab$  [ $S \rightarrow aS$ ]
- $S \rightarrow aaaebaab$
- $S \rightarrow aaabaaab$



aaabaab

Q) check whether the string  $aabb$  is accepted or not.

$$S \rightarrow aAB/bA/\epsilon$$

$$A \rightarrow aAb/\epsilon$$

$$B \rightarrow bb/\epsilon$$

sif:

$$\begin{array}{ll} S \rightarrow aAB & [S \rightarrow aAB] \\ S \rightarrow aaABB & [A \rightarrow aAb] \\ S \rightarrow aacbBB & [A \rightarrow \epsilon] \\ S \rightarrow aabbB & [B \rightarrow bB] \\ S \rightarrow aabb\epsilon & [B \rightarrow \epsilon] \\ S \rightarrow aabb \text{ is accepted.} & \end{array}$$

Q) Consider the grammar

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

$$L \rightarrow L, S | S$$

a) what are the non-terminals, terminals & start symbol

b) find LMD, RMD & parse trees for the following

$$(i) (a, a)$$

$$(ii) (a, (a, a))$$

Q) what

sop: (a) Non-terminals = { $S, L$ }

terminals = { $, ), , a, , ,$ }

start symbol =  $S$

(b) LMD:  $(a, a) \quad S \rightarrow (L)$

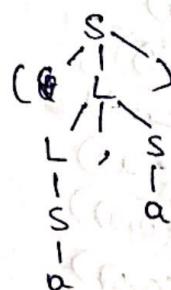
$$S \rightarrow (, S)$$

$$S \rightarrow (S, S)$$

$$S \rightarrow (a, S)$$

$$S \rightarrow (a, a)$$

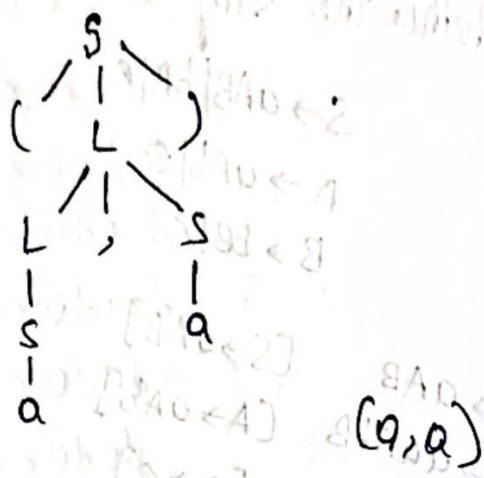
$$(a, a), a$$



$$(a, a)$$

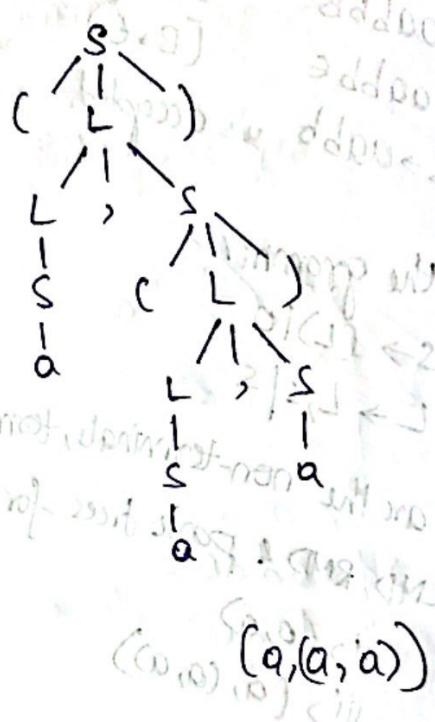
PMD:  $(a, a)$

- $S \rightarrow (L)$
- $S \rightarrow (L, S)$
- $S \rightarrow (L, a)$
- $S \rightarrow (S, a)$
- $S \rightarrow (a, a)$



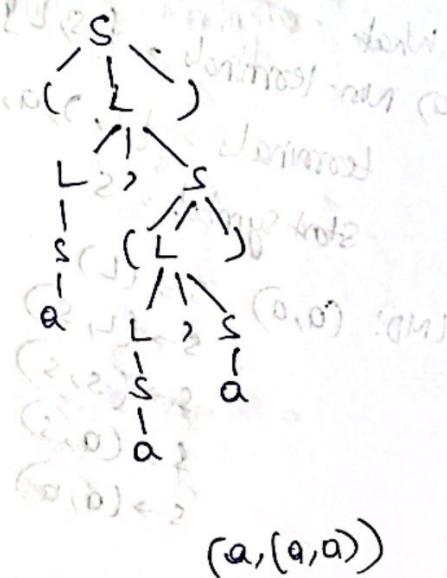
(i) LMD:  $(a, (a, a))$

- $S \rightarrow (L)$
- $S \rightarrow (L, S)$
- $S \rightarrow (S, S)$
- $S \rightarrow (a, S)$
- $S \rightarrow (a, (L))$
- $S \rightarrow (a, (L, S))$
- $S \rightarrow (a, (S, S))$
- $S \rightarrow (a, (a, S))$
- $S \rightarrow (a, (a, a))$



RMD:  $(a, (a, a))$

- $S \rightarrow (L)$
- $S \rightarrow (L, S)$
- $S \rightarrow (L, (L))$
- $S \rightarrow (L, (L, S))$
- $S \rightarrow (L, (L, a))$
- $S \rightarrow (L, (S, a))$
- $S \rightarrow (L, (a, a))$
- $S \rightarrow (S, (a, a))$
- $S \rightarrow (a, (a, a))$



## Ambiguity

A grammar  $G$  is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

$$E: E \rightarrow E+E \mid E * E \mid CE \mid id$$

Here  $id$  is terminal symbol!

sof : story:  $id + id * id$

$$E \rightarrow E+E$$

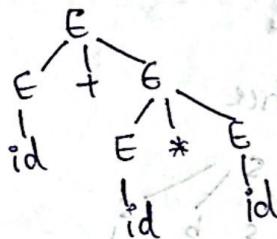
$$LMD: E \rightarrow E+E$$

$$E \rightarrow id + E * E$$

$$E \rightarrow id + id * E$$

$$E \rightarrow id + id * id$$

Possible tree!



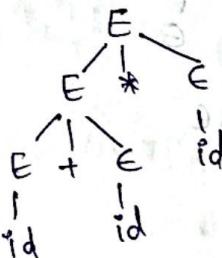
$$RMD: E \rightarrow E * E$$

$$E \rightarrow E * id$$

$$E \rightarrow E + E * id$$

$$E \rightarrow E + id * id$$

$$E \rightarrow id + id * id$$



gives ambiguous grammar.

Ex: Show that the following grammar is Ambiguous

$$S \rightarrow aSbS$$

$$S \rightarrow bSaS$$

$$S \rightarrow \epsilon$$

Sof. LMD!

$$S \rightarrow aSbS$$

$$S \rightarrow abSabS$$

$$S \rightarrow abc aSbS$$

$$S \rightarrow abaebS$$

$$S \rightarrow abab\epsilon$$

$$S \rightarrow abab$$

RMD

$$S \rightarrow aSbS$$

$$S \rightarrow a\epsilon bS$$

$$S \rightarrow ab aSbS$$

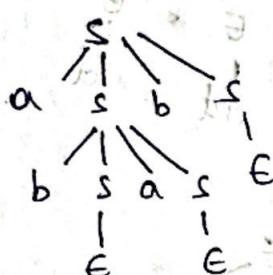
$$S \rightarrow abaebS$$

$$S \rightarrow abab\epsilon$$

$$S \rightarrow abab$$

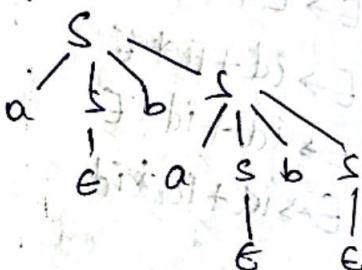
Two parse trees  $\Rightarrow$  Ambiguous grammar  
one possible

LMD parse tree



$\Rightarrow abaeab\epsilon$   
abab

RMD parse tree



$\Rightarrow aeb aeb\epsilon$   
abab

Ex: prove that the following grammar is Ambiguous

$$S \rightarrow AB$$

$$B \rightarrow a\bar{b}$$

$$A \rightarrow a\bar{a}$$

$$A \rightarrow \bar{a}$$

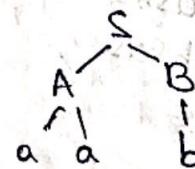
$$B \rightarrow b$$

Sof. LMD!

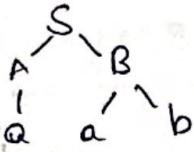
$$S \rightarrow AB$$

$$S \rightarrow a\bar{a}B$$

$$S \rightarrow a\bar{a}b$$



$$\begin{aligned} S &\rightarrow AB \\ S &\rightarrow aB \\ S &\rightarrow aaB \end{aligned}$$



Two parse trees are possible for the string aab. Hence, it's an ambiguous grammar.

### Left Recursion:

- A production of grammar is said to have left recursion if the leftmost variable of its LHS is same as variable of its RHS.

$$5: A \rightarrow A\alpha | B$$

- There are two types of left recursion:

(i) Direct left recursion

(ii) Indirect left recursion

$$5: B \rightarrow Ba$$

$$\begin{aligned} 6: S &\rightarrow Aa \\ A &\rightarrow Sm \end{aligned}$$

### Removal of left recursion:

- The top down parsers cannot accept the grammar having left recursion (i.e. they do not allow left recursive grammar).
- So, we have to remove left recursion but preserve the language generated by grammar.

- Recursion: Recursion could be Left Recursion (LR) or Right Recursion (RR).

### Recursion

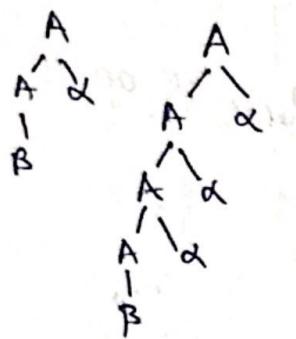
LR

$$A \rightarrow A\alpha | B$$

RR

$$A \rightarrow B\alpha A | B$$

LR:  $A \rightarrow A\alpha|\beta$



RR:  $A \rightarrow \alpha A |\beta$



Lang =  $\beta\alpha^*$

$\Rightarrow \{\beta, \beta\alpha, \beta\alpha\alpha, \dots\}$

Lang =  $\alpha^*\beta$

$\Rightarrow \{\beta, \alpha\beta, \alpha\alpha\beta, \dots\}$

g) We write it terms of function

( $\{A()\}$ ) (conflict loop)

$A()$

$\alpha;$

}

Left recursion

-  $A()$

$\alpha;$

$A();$

Right recursion

• There is a problem in left recursion, the recursive function calls itself infinite times.

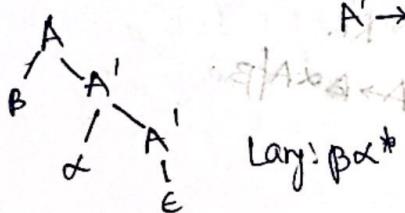
• In right recursion, ( $\alpha \rightarrow \text{Base condition}$ )  $\alpha$  is evaluated, hence it doesn't lead to infinite loop.

• Hence, we have to remove left recursion.

LR:  $A \rightarrow A\alpha|\beta$

Now, we want  $\beta\alpha^*$   $\rightarrow A \rightarrow \beta A'$  { A will generate  $\beta$  followed by  $A'$  }

$A' \rightarrow \alpha A' / \epsilon$  { now A' will generate  $\alpha^*$  }



Lang:  $\beta\alpha^*$

Therefore,  $A \rightarrow \beta A' \quad A' \rightarrow \alpha A' | E$  =  $A \rightarrow A\alpha | \beta$

This right recursive grammar is equivalent to this left recursive grammar.

So, to eliminate left recursion, we will follow these rules.

Ex:  $E \rightarrow E + T / T$  // Remove Left recursion

We will compare it with  $A \rightarrow A\alpha | \beta$

$$\begin{array}{c} E \rightarrow E + T / T \\ \hline A \quad A \alpha \quad \beta \end{array}$$

Ans:  $\begin{array}{c} E \rightarrow TE' \\ E' \rightarrow \epsilon + TE'/E \end{array} \quad =$

Ex: Eliminate left recursion

(i)  $S \rightarrow SOSIS / OI$

sol: We know that,

$$A \rightarrow A\alpha | \beta = \begin{array}{c} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | E \end{array}$$

$$\begin{array}{c} S \rightarrow SOSIS / OI \\ \hline A \quad A \alpha \quad \beta \end{array}$$

$$\begin{array}{c} S \rightarrow OI S' \\ S' \rightarrow OSIS' / E \end{array}$$

(ii)  $S \rightarrow (L) | \chi$  {this is not left recursive, as it's left starts with 'L')}

$$\begin{array}{c} L \rightarrow L, S | S \\ \hline A \quad A \alpha \quad \beta \end{array}$$

(Left recursion)

sol:

$$\begin{array}{c} L \rightarrow SL' \\ L' \rightarrow \epsilon / , SL' \end{array}$$

$$\begin{array}{c} S \rightarrow (L) | \chi \\ L \rightarrow SL' \\ L' \rightarrow \epsilon / , SL' \end{array}$$

(iii)  $A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \dots$   
 $| \beta_1 | \beta_2 | \beta_3 | \dots$

iv)  $A \rightarrow \beta_1 A' | \beta_2 A' | \beta_3 A' | \dots$   
 $A' \rightarrow | \epsilon | \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | \dots$   
 $\downarrow$   
 $A \rightarrow A\alpha_1 | \beta$   
 $A \rightarrow \beta A'$   
 $A' \rightarrow \alpha_1 A' | \epsilon$

(v)  $E \rightarrow E + T | T - \textcircled{1}$   
 $T \rightarrow T * F | F \textcircled{2}$   
 $F \rightarrow \text{id} \textcircled{3}$

so, for the 1<sup>st</sup> production

$E \rightarrow E + T | T$   
 $\bar{A} \quad \bar{A} \quad \bar{\alpha} \quad \bar{F}$

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

we know that

$A \rightarrow A\alpha_1 | \beta$   
 $\downarrow$   
 $A \rightarrow BA'$   
 $A' \rightarrow \alpha_1 A' | \epsilon$

similarly for 2<sup>nd</sup> production

$T \rightarrow T * F | F$   
 $\bar{A} \quad \bar{A} \quad \bar{\alpha} \quad \bar{B}$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT'| \epsilon$

3<sup>rd</sup> production - Ok

Final Answer:

$E \rightarrow TE'$   
 $E' \rightarrow +TE'| \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT'| \epsilon$   
 $F \rightarrow \text{id}$

IV)  $S \rightarrow Aa$  [produced left recursion]  
 $A \rightarrow Sb/C$

- s):  
g) we replace with 'A' with  $Sb/C$  and 'S' with  $A$   
g) we replace 'S' production:  
 $A \rightarrow Aab/C$  [left recursive]

We know that,  $A \rightarrow A\alpha | \beta \equiv A \rightarrow \beta A'$

$$A' \rightarrow \alpha A' | \epsilon$$

$S \rightarrow Aa$  and conduct  
 $A \rightarrow Sb/C$  [left recursive]

$$\frac{A \rightarrow Aab/C}{A \quad A \alpha \quad \beta}$$

Final Ans:  $A \rightarrow CA'$   
 $A' \rightarrow abA' | \epsilon$

$$\boxed{\begin{array}{l} S \rightarrow Aa \\ A \rightarrow CA \\ A' \rightarrow abA' | \epsilon \end{array}}$$

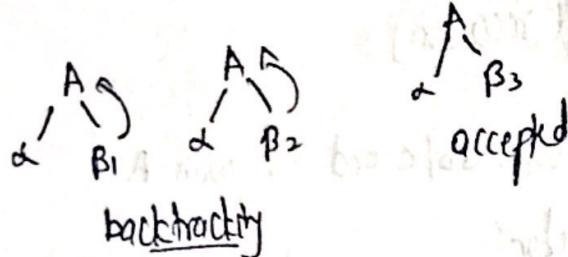
### Left Factoring

Sometimes, it is not clear which production to choose / to expand a non-terminal because multiple productions begin with the same (terminal / non-terminal) lookahead. This type of grammar is called Non Deterministic grammar (ND).

Grammar containing left factoring.

Ex:  $A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3$ ,  
common prefixes

Suppose we have to access  $\alpha \beta_3$

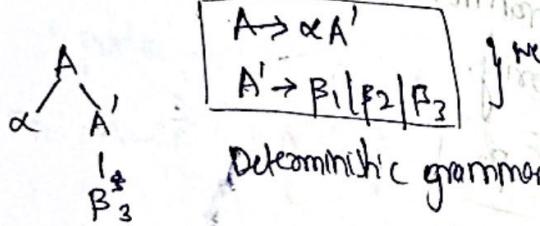


- The backtracking is happening because of the common prefix. One or more productions in the PSLs are having something common in the prefixes.
- This is also called common prefix problem or non-deterministic grammar.

Reason for backtracking:

- Backtracking occurs because we are making our decision only after seeing  $\alpha$ . (i.e. which production to choose without seeing full input).
- If input is  $\alpha\beta_3$ , we should make the decision, if  $\beta_3$  is available then only we will go onto that production.
- So, for

$$A \xrightarrow{\alpha} \beta_1 \mid \beta_2 \mid \beta_3 \quad \text{Non-Deterministic Grammar}$$



we have postponed  
the decision making

The procedure we used to convert Non-Deterministic grammar to deterministic grammar is called left factoring.

## Problem with Non-Deterministic Grammar:

The problem was most of the top down parser will have to backtrack.

So, we do not need Non-Deterministic grammar.

Ex:  $S \rightarrow iEtSes \mid iEtS/a$  // Remove left factoring  
 $E \rightarrow b$  or Remove common prefix problem

Sf:  $S \rightarrow iEtSS' \mid a$   
 $S' \rightarrow eS \mid \epsilon$   
 $E \rightarrow b$

(ii)  $S \rightarrow assbs \mid asasb \mid abb \mid b$

Sf:  $S \rightarrow as' \mid b$   
 $S' \rightarrow ssbs \mid sasb \mid bb$  } again common prefix problem  
 $\downarrow$

$S' \rightarrow ss'' \mid bb$   
 $S'' \rightarrow sbs \mid asb$

$\therefore A \rightarrow aA$  } not a  
 $B \rightarrow aB$  } common  
prefix problem

Find Ans:

$S \rightarrow as' \mid b$   
 $S' \rightarrow ss'' \mid bb$   
 $S'' \rightarrow sbs \mid asb$

(iii)  $S \rightarrow bssas \mid bssasb \mid bsb \mid a$

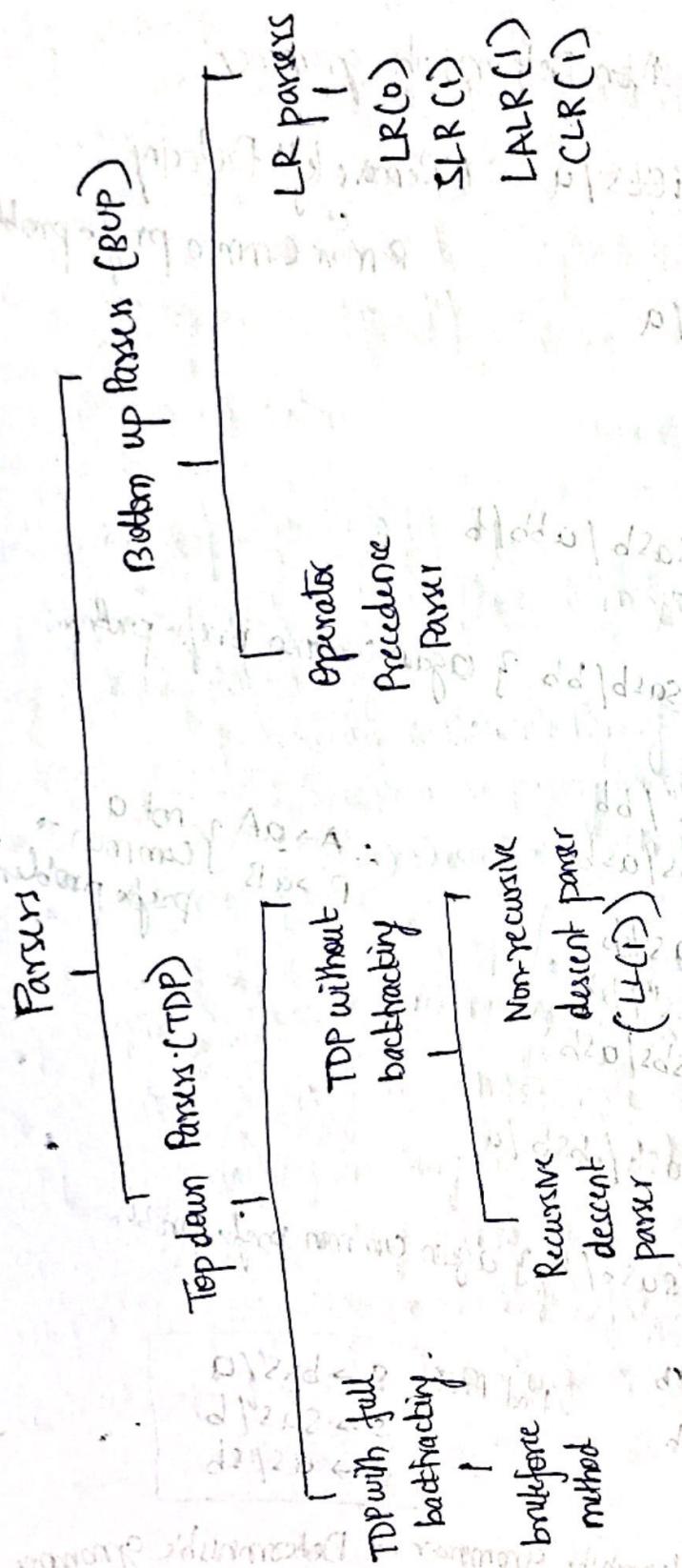
Sf:  $S \rightarrow bss' \mid a$   
 $S' \rightarrow saas \mid sasb \mid b$  } again common prefix problem  
 $\downarrow$

$S' \rightarrow saas'' \mid b$   
 $S'' \rightarrow as \mid sb$

Find Ans:

$S \rightarrow bss' \mid a$   
 $S' \rightarrow saas'' \mid b$   
 $S'' \rightarrow as \mid sb$

Converted Non-Deterministic grammar  $\rightarrow$  Deterministic grammar



Parser)  
Parser is that phase of compiler which takes input in the form of sequence of tokens and produces output in the form of parse tree. Parser is also known as Syntax Analyser.

Type of Parser's

Parser is mainly classified into 2 categories:

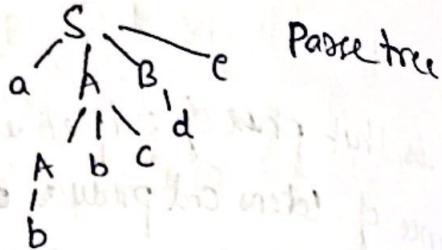
1. Top-down Parser
2. Bottom up Parser

### ③ Top-down Parser:

Top down Parser generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals ie (starts from the start symbol and ends with the terminals).  
It uses left-most derivation.  
The process of constructing the parse tree which starts from the start symbol and goes down to the leaf is Top-down parsing.  
- Top down parsers constructs from the grammar which is free from ambiguity and left recursion.  
- Top down parsers uses leftmost derivation to construct a parse tree.  
- It allows a grammar which is free from Left factoring.

Ex:  $S \rightarrow aABC$   
 $A \rightarrow Abc/b$   
 $B \rightarrow d$

String: abbcde



Yield of parse tree: abbcde

$S \rightarrow aABC$  [ $S \rightarrow aABC$ ]  
 $S \rightarrow aAbcBe$  [ $A \rightarrow Abc$ ]  
 $S \rightarrow abbcBe$  [ $A \rightarrow b$ ]  
 $S \rightarrow abbcde$  [ $B \rightarrow d$ ]

∴ left most derivation

### Classification of Top-Down Parsing:

1. With Backtracking : Brute Force technique

2. Without Backtracking :

1. Recursive Descent Parsing

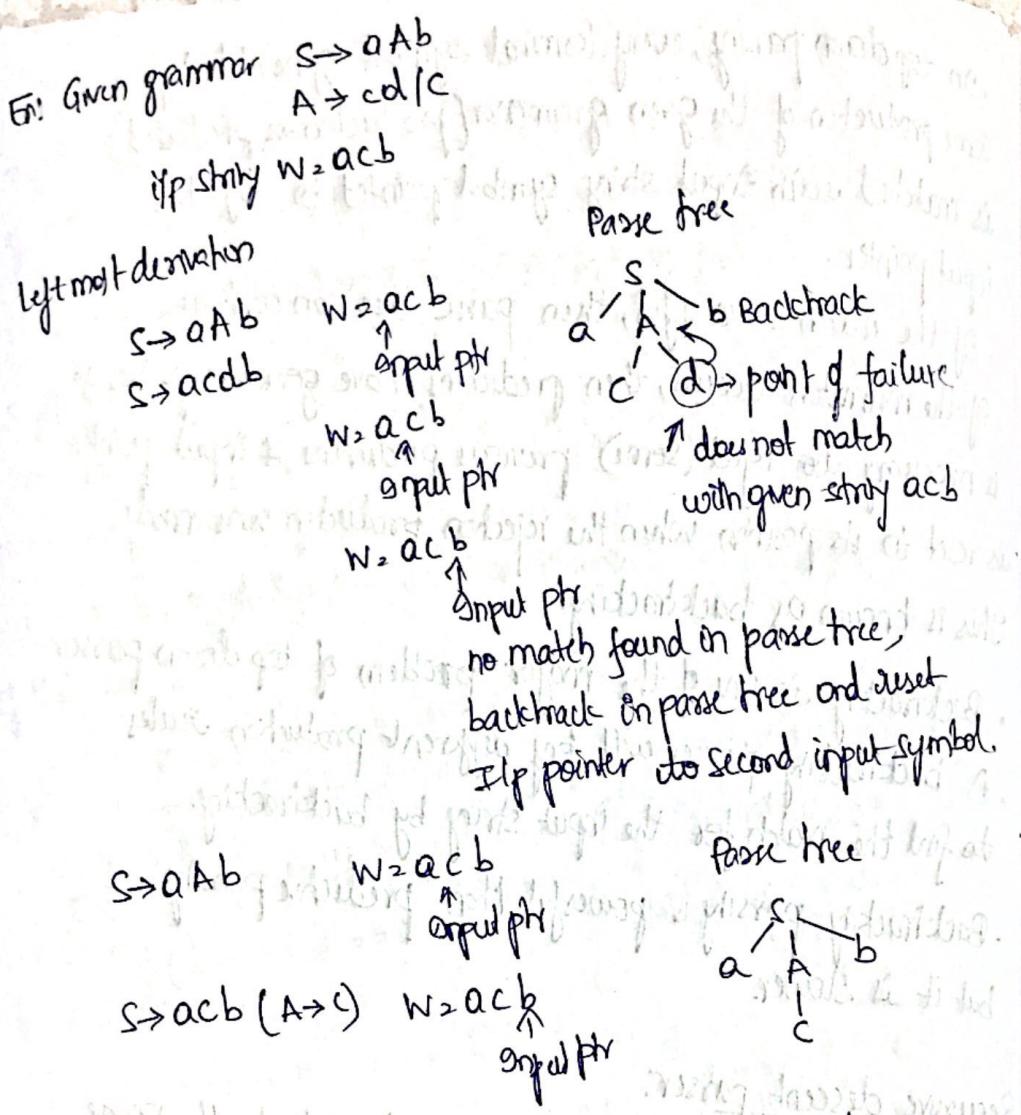
2. Non-Recursive parsing (or) Predictive Parsing (or)

LL(1) Parsing (or) Table Driver Parsing

### Top Down Parser with Backtracking (Brute Force Method):

Here, full backtracking is used to create a parse tree until the correct/given string is generated at leaves.

In worst case, when string is not given in given language all possible combinations are checked before the failure to construct a parse tree for given string is recognised.



As the leaf 'c' matches the second symbol of input string and next leaf 'b' matches third symbol of the input string  $w$ , then the parser will stop and announce successful completion of parsing.

As yield of parser is 'acb' which matches with 'ip  $w = abc$ ', hence string is generated by grammar.

- In top down parsing, every terminal symbol generated by some production of the given grammar (production is predicted) is matched with input string symbol pointed to by the input pointer.
- If the match is successful, then parse tree can continue.
- If the mismatch occurs, then predictions have gone wrong. So, it is necessary to reject (some) previous predictions & input pointer is reset to its position when the rejection production was made. This is known as backtracking.
  - Backtracking is one of the major problem of top down parser.
  - A backtracking parser will try different production rules to find the match for the input string by backtracking.
  - Backtracking parser is powerful than predictive parser but it is slower.

### Recursive descent parser:

- It is a top-down parsing technique that constructs the parse tree from top and if it is read from left  $\rightarrow$  right.
- A procedure/ function is associated with each non-terminal of the grammar.
- A top down parser that uses collection of recursive procedures for <sup>parsing</sup> the given input string is called as Recursive Descent Parser (RDP).

In RDP, the CPG is used to build the recursive routines.

The RHS of the production rule is directly converted to a program which is the body of the corresponding non-terminal of LTLs.

### Basic steps for constructing of Recursive Descent parser:

- Step 1: Write procedure for start variable production of given grammar. The RHS of the production is directly converted into program code symbol by symbol.
- Step 2: If the input symbol is Non-terminal, then call procedure of that Non-Terminal.

Ex:  $E \rightarrow \text{num}T$   
 $T \rightarrow * \text{num}T / E$

$E()$   
{ if (lookahead == num)  
 { match(num);  
 T();  
 }  
 if (lookahead == '\$')  
 { print ("success");  
 }  
 else  
 print ("error");  
 }

$T()$   
{ if (lookahead == '\*')  
 { match('\*');  
 }

\* here num is terminal,  
 T is non-terminal \*

```

if (lookahead == num)
{
    match(num);
    T();
}
else
    point("error");
}

else
    return; // E
}

```

(07)

```

match(char t) /* t is token */
{
    if (lookahead == t)
    {
        lookahead = getch();
        /* lookahead = next token */
    }
    else
        point("error");
}

```

(07)

main()

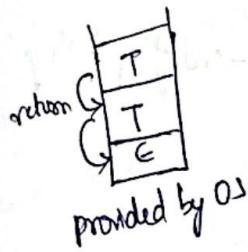
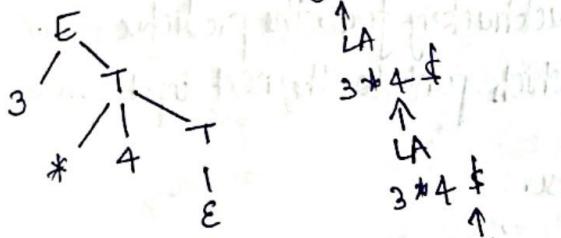
{ E(); }

Step 3: If the input symbol is terminal then it is matched with the lookahead from the input.  
 The lookahead pointer has to be advanced on matching the input symbol.

Step 4: If the production rule has many alternatives then all these alternatives to be combined into a simple body of the Non-terminal procedure.

Repeat above steps for all non-terminals present in Grammar.

Ex. Let input string be  $w = 3 * 4 \$$



### Advantages of Recursive Descent Parser without backtracking

- gets easy to construct
- Overhead associated with backtracking is eliminated.

### Drawbacks / Disadvantages:

- We can not write Recursive descent parser for all types of CPG.
- It can be implemented only for those languages which supports recursive procedure calls.

## LL(1) Parser:

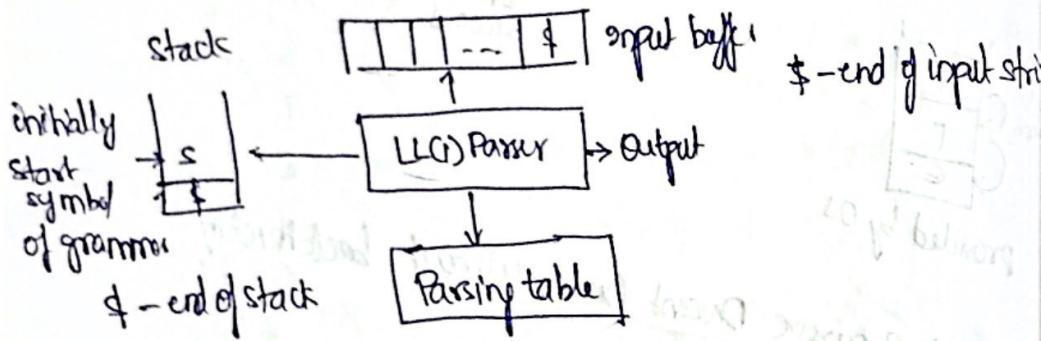
- LL(1) Parser is a top-down parser of non-recursive type.

LL(1)

Input is scanned from left to right	It was leftmost derivation for input string	It was only one input symbol (lookahead) to predict the parsing process.
-------------------------------------	---	--

- To make the parser backtracking free, the predictive parser uses a lookahead pointer, which points to the next input symbol.

## Block diagram of LL(1) Parser:



## Data structures used by LL(1) Parser

- (1) Input Buffer
- (2) Stack
- (3) Parsing table

- Predictive parsers can be constructed for a class of grammars called LL(1).
- No left recursive or ambiguous grammar can be LL(1).
- Predictive parser has the capability to predict which production is to be used to replace the input string.

- Predictive parser does not suffer from backtracking.
- To remove backtracking, the predictive parser uses a lookahead pointer which points to the next input symbol.
- Predictive parser uses a stack & parsing table to parse the input stored in input buffer & generate a parse tree from top to bottom. (top down approach).
- Both the stack & the input contains an end symbol \$, where
  - \$ in stack - represent empty stack
  - \$ in input - represent input is consumed.
- The parsing program reads top of the stack & a current input symbol & with the help of these two symbols, the parsing action is determined.
- The parser consults the table  $M[A, a]$
- ( $A \rightarrow \text{top of stack}$ ,  $a - \text{input}$ ) each time while taking the actions. Hence, this type of parsing method is called table driven parsing algorithm.

### First() and Follow():

- $\text{First}(A)$ : It is a set of terminal(s) that begin in strings derived from  $A$ .
- Ex:  $S \rightarrow aAB$   
 $A \rightarrow b$   
 $B \rightarrow C$
- $\text{first}(S) = \{a\}$ ,  $\text{first}(A) = \{b\}$   
 $\text{first}(B) = \{C\}$
- Ex:  $A \rightarrow abc \mid def \mid ghi$   
 $\text{first}(A) = \{a, d, g\}$

- Follow(A): set of terminals that appear immediately to the right of A. e.g.  $A \rightarrow BCD$
- follow of start symbol is always  $\$$ .
- for  $A \rightarrow \alpha \beta B$   
 $\text{follow}(\beta) \supset \text{follow}(A)$

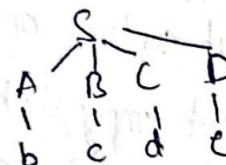
Ex:  $S \rightarrow ABCD$

$$A \rightarrow b$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$



$$\text{first}(S) = \text{first}(A) = b$$

$$\text{first}(B) = c$$

$$\text{first}(C) = d$$

$$\text{first}(D) = e$$

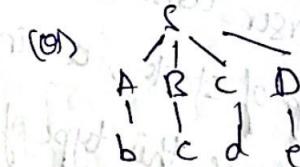
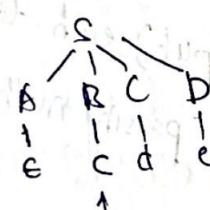
Ex:  $S \rightarrow ABCD$

$$A \rightarrow b | \epsilon$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$



$$\text{so: } \text{first}(S) = \{b, c\} \text{ as } A \rightarrow b | \epsilon$$

## First and Follow rules:

First:

1. If  $A \rightarrow \alpha\beta$ ,  $\alpha$  [Combination of terminal & Non-terminal]

then  $\text{First}(A) = \{\alpha\}$

2. If  $A \rightarrow \epsilon$ , then  $\text{First}(A) = \{\epsilon\}$

3. If  $A \rightarrow BC$  then  
 $\text{First}(A) = \text{First}(B)$  if  $\text{First}(B)$  doesn't contain epsilon  
 or  $\text{First}(B)$  contains epsilon ( $\epsilon$ ) then  $\text{First}(A) = \text{First}(B) \cup \text{First}(C)$

Follow:

1. If '\$\\$' is start symbol, then  $\text{Follow}(S) = \{\$\}$

2. If  $A \rightarrow \alpha B \beta$  then  $\text{Follow}(B) = \text{Search in RHS}$

- Follow(B) =  $\text{First}(B)$  if  $\text{First}(B)$  doesn't contain  $\epsilon$ .

In Follow, take care of place  $\epsilon$ .

3. If  $A \rightarrow \alpha B$  then  $\text{Follow}(B) = \text{Follow}(A)$

4. If  $A \rightarrow \alpha B \beta$  where  $B \rightarrow \epsilon$

then  $\text{Follow}(B) = \text{Follow}(A)$

Ex: First & Follow examples:

$S \rightarrow ABCDE$   
 $A \rightarrow a/E$   
 $B \rightarrow b/E$   
 $C \rightarrow C$   
 $D \rightarrow d/E$   
 $E \rightarrow e/E$

First

$\{a, b, c\}$   
 $\{a, \epsilon\}$   
 $\{b, \epsilon\}$   
 $\{c\}$   
 $\{d, \epsilon\}$   
 $\{e, \epsilon\}$

Follow

$\{\$\}$   
 $\{b, c\}$   
 $\{c\}$   
 $\{d, e, \$\}$   
 $\{e, \$\}$   
 $\{\$\}$

First

$S \rightarrow Bb|cd$   
 $B \rightarrow aB|\epsilon$   
 $C \rightarrow CC|\epsilon$

$\{a, b, \epsilon, d\}$   
 $\{a, \epsilon\}$   
 $\{c, \epsilon\}$

Follow

$\{\$\}$   
 $\{b\}$   
 $\{d\}$

First

$S \rightarrow ACB|CBB|Ba$   
 $A \rightarrow da|BC$   
 $B \rightarrow g|\epsilon$   
 $C \rightarrow h|\epsilon$

$\{d, g, h, \epsilon, b, a\}$   
 $\{d, g, h, \epsilon\}$   
 $\{g, \epsilon\}$   
 $\{h, \epsilon\}$

Follow

$\{\$\}$   
 $\{b, g, \$\}$   
 $\{f, a, b, h, g\}$   
 $\{g, \$, b, h\}$

## Construction of LL(1) Parser:

steps involved:

1. Elimination of left recursion
2. Elimination of left factoring
3. Calculation of first and follow
4. Construction of parse table
5. Check whether input string is accepted by parser or not.

### Example:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Step 1: Elimination of left recursion

If production is of form  $A \rightarrow A\alpha \beta$  (left recursive)

Replace it with  $A \rightarrow \beta A'$   
 $A' \rightarrow \alpha A' \mid \epsilon$

$$\begin{array}{c} E \rightarrow E + T \mid T \\ \hline A & A \alpha & \beta \end{array}$$

$$\begin{array}{c} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \end{array}$$

$$\begin{array}{c} T \rightarrow T * F \mid F \\ \hline A & A \alpha & \beta \end{array}$$

$$\begin{array}{c} T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \end{array}$$

$F \rightarrow (E) \mid \text{id}$  (Not left recursive)

After elimination of left recursion, productions are:

$$\begin{array}{c} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid \text{id} \end{array}$$

Step 3: Elimination of left factory:

$$A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | \dots$$

[left factory]

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2 | \beta_3$$

There is no left factory in the production.

Step 4: Calculation of first and follow:

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

$$\text{First}(E) = \{ (, id) \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T) = \{ (, id) \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

$$\text{First}(F) = \{ (, id) \}$$

$$\text{Follow}(E) = \{ \$, ) \}$$

$$\text{Follow}(E') = \{ \$, ) \}$$

$$\text{Follow}(T) = \{ \$, ), + \}$$

$$\text{Follow}(T') = \{ \$, ), + \}$$

$$\text{Follow}(F) = \{ *, +, \$, ) \}$$

Step 4: Construction of parse table

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

## Rules for construction of parse table:

$A \rightarrow \alpha$

1. If the production is of  $A \rightarrow \alpha$

Calculate  $\text{First}(\alpha)$ , &

↳ it produces terminal symbol @

then, add  $A \rightarrow \alpha$  to  $M[A, \alpha]$

2. If  $\text{First}(\alpha)$  prod contains  $\epsilon$  (or)  $A \rightarrow E$  (epsilon)

then, we have to calculate  $\text{Follow}(A)$

↳ produces terminal b

then, add  $A \rightarrow E$  to  $M[A, b]$

3. The remaining entries of the parse table are filled with  
error error.

1.  $E \rightarrow TE'$

$\text{First}(TE') = \{c, \text{id}\}$

Add  $E \rightarrow TE'$  to  $M[E, c]$   
 $M[E, \text{id}]$

2.  $E' \rightarrow +TE' / \epsilon$

(a)  $\text{First}(+TE') = \{+\}$

Add  $E' \rightarrow +TE'$  to  $M[\epsilon, +]$

(b)  $E' \rightarrow \epsilon$

$\text{Follow}(E') = \{\$, +\}$

Add  $E' \rightarrow \epsilon$  to  $M[E', \$]$   
 $M[E', +]$

3.  $T \rightarrow PT'$

$\text{First}(PT') = \{c, \text{id}\}$

Add  $T \rightarrow PT'$  to  $M[T, c]$   
 $M[T, \text{id}]$

4.  $T' \rightarrow *FT' / \epsilon$

(a)  $\text{First}(*FT') = \{*\}$

Add  $T \rightarrow *FT'$  to  $M[T', *]$

(b)  $\text{Follow}(T') = \{\$, +\}$

Add  $T \rightarrow \epsilon$  to  $M[T, \$]$   
 $M[T', \$]$   $M[T, +]$

5.  $F \rightarrow (\epsilon) / \text{id}$

(a)  $\text{First}((\epsilon)) = \{\epsilon\}$

Add  $F \rightarrow (\epsilon)$  to  $M[F, \epsilon]$

(b)  $\text{First}(\text{id}) = \{\text{id}\}$

Add  $F \rightarrow \text{id}$  to  $M[F, \text{id}]$

Steps: Check whether the input string is accepted or not.

Let's consider i/p: id + id \$

Stack	Input String	Action
\$ E	id + id \$	$E \rightarrow TE'$
\$ E' T	id + id \$	$T \rightarrow FT'$
\$ E' T' F	id + id \$	$F \rightarrow id$
\$ E' T' id	. id + id \$	pop (When stack top symbol = input string) & move pointer to one position to right
\$ E' T'	id + id \$	$T' \rightarrow E$
\$ E'	+ id \$	$E' \rightarrow +TE'$
\$ E' T +	+ id \$	pop (finishes id)
\$ E' T	id \$	$T \rightarrow FT'$
\$ E' T' F	id \$	$F \rightarrow id$
\$ E' T' id	id \$	pop
\$ E' T'	\$	$T' \rightarrow E$
\$ E'	\$	$E' \rightarrow E$
\$	\$	

After performing of entire string, if the stack and Input string constitutes '\$', The string is accepted.

#### ④ Bottom-up Parsing

In bottom-up parsing, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol of grammar.

The process of bottom-up parsing stops successfully as soon as we reach the start symbol.

- It uses reverse right-most derivation.

- Parse tree is constructed from bottom to top. i.e from leaves to root.

- In bottom-up parsing, parser tries to identify the RHS of production rule and replace it by corresponding LHS. This activity is called as Reduction.

- So, the prime task in bottom-up parsing is to find the production that can be used for reduction.

Example:  $E \rightarrow E+E$  Input string: id+id+id  
 $E \rightarrow id$

RND:  $E \rightarrow E+E$

$E \rightarrow E+E+E$

$E \rightarrow E+E+id$

$E \rightarrow E+(id+id)$

$E \rightarrow id+id+id$

Bottom-up Parsing: id+id+id  
 id+id+E ( $E \rightarrow id$ )  
 id+E+E ( $E \rightarrow id$ )  
 id+E ( $E \rightarrow E+E$ )  
 E+E ( $E \rightarrow id$ )  
 E ( $E \rightarrow E+E$ )

Handle & Handle Parsing!

Handle: handle is a substring which matches with right side of induction, then it's replaced with LHS Non-terminal.

$$\text{Ex: } S \rightarrow aABe \\ A \rightarrow AbClb \\ B \rightarrow d$$

Input string "abbcde"

S1 Input string abbcde

Handles dummy passivity of abbcde

Right Sentential Form

abbcde  
aAbcde  
aAde  
aABe

	Handle	Reducing production
	b	$A \rightarrow b$
	Abc	$A \rightarrow AbC$
	d	$B \rightarrow d$
	aABe	$S \rightarrow aABe$

S

Ex2: Bottom-up parsing uses right-most derivation in reverse order.  
is called handle pruning.

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

Input string: "id \* id"

Right Sentential Form

id \* id

\* id

T \* id

T \* F

T

E

Handle

id

F

id

T \* F

T

Reducing production

$F \rightarrow id$

$T \rightarrow F$

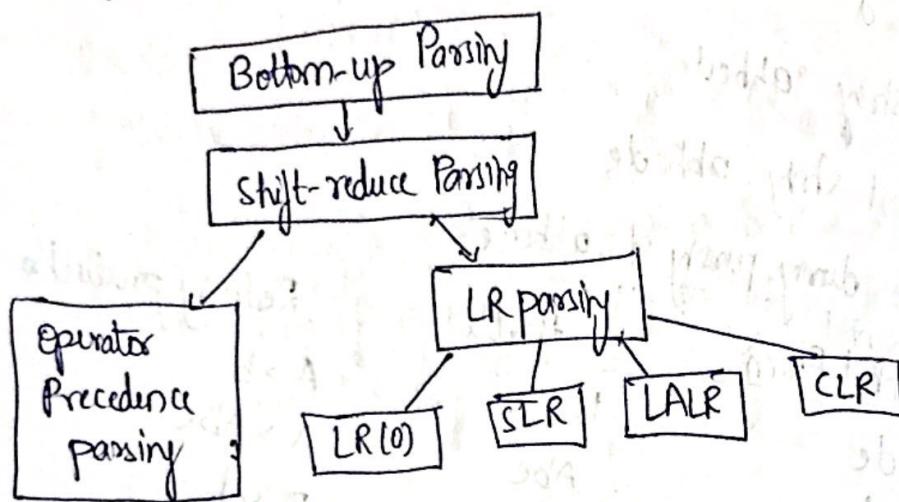
$F \rightarrow id$

$T \rightarrow T * F$

$E \rightarrow T$

## LR Parser:

LR parsing is one type of



## Shift-reduce Parser:

- Shift-reduce parser attempts for the construction of parse tree in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves (bottom) to the root (up). A more general form of shift-reduce parser is LR parser.
- This parser requires some data structures i.e.
  - An input buffer for storing the input string.
  - A stack for storing and accessing the production rules.

## Basic operations:

- Shift: This involves moving of symbol from input buffer onto the stack.
- Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production is done.

- RHS of the production is popped out of stack and LHS of production rule is pushed onto the stack.
- Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it means successful parsing is done.
  - Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

$$\text{Ex! } E \rightarrow E+T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid \text{id}$$

Input string: id \* id

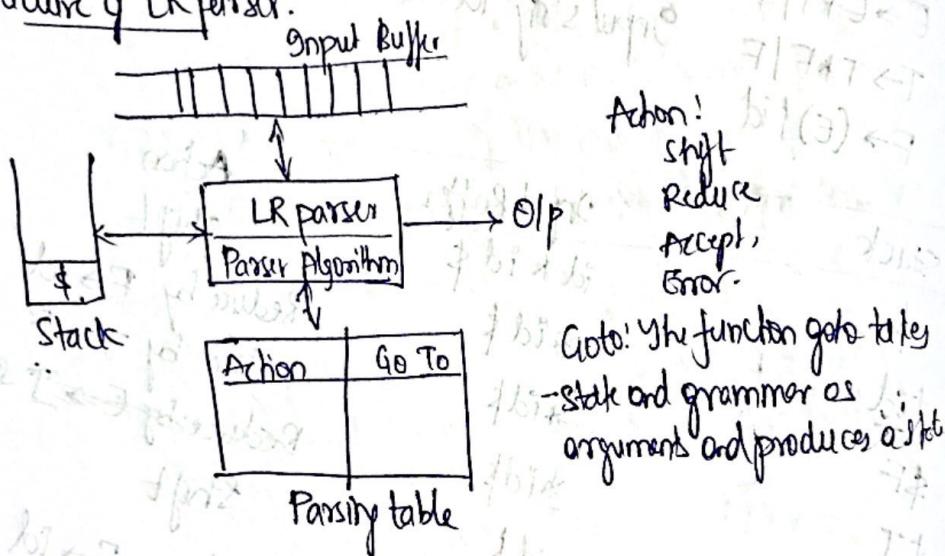
Stack	Input Buffer	Action
\$	id * id \$	shift
\$ id	* id \$	Reduce by $F \rightarrow \text{id}$
\$ F	* id \$	Reduce by $T \rightarrow F$
\$ T	* id \$	Reduce by <del><math>E \rightarrow T</math></del> shift
\$ T *	id \$	shift
\$ T * id	\$	Reduce $F \rightarrow \text{id}$
\$ T * F	\$	Reduce $T \rightarrow T * F$
\$ T	\$	Reduce $T \rightarrow E$
\$ E	\$	Accepted

- Shift-reduce conflict:** It occurs if the parser has a choice to select both shift action and reduce action. But only one can be selected.
- Reduce-reduce conflict:** It occurs if the parser has more than one reduction is possible.

## ⑤ LR-Parsy:

- LR parsy is one type of bottom up parsy. It's used to parse the large class of grammars.
- In the LR parsy, "L" stands for left-to-right scanning of the input.
- "R" stands for constructing a rightmost derivation in reverse.
- $LR(1)$ , '1' is the no. of input symbols of the lookahead used to make number of parsing decision.

Structure of LR parser:



- There are three widely used algorithms available for constructing an LR parser.

• SLR(1) - Simple LR

• CLR(1) - Canonical LR parser

• LALR(1) - Look ahead LR parser

LR(0):

- An LR(0) item is a production  $A \rightarrow \alpha$  with dot at some position on the right side of the production.
- LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.

on the LR(0), we place the reduce node in the entire row.

(Example:

$$S \rightarrow AA$$

$$A \rightarrow aAb$$

~~Augmented Grammar: Add Augment production and insert '!' symbol at the first production position of every production~~

$$S' \rightarrow S$$

$$S \rightarrow .AA$$

$$A \rightarrow .aA$$

$$A \rightarrow .b$$

steps to solve LR(0) sums:

1. Augment the given grammar

2. Draw the canonical collection of LR(0) item.

3. Number the production

4. Create the parse table

5. Stack implementation

6. Draw parse tree.

Ex:  $E \rightarrow BB$

$B \rightarrow CB/d$

Step 1: Augment the given grammar

$E \rightarrow E$

$E \rightarrow BB$

$B \rightarrow CB$

$B \rightarrow d$

Step 2: Draw Canonical collection of LR(0) item

$I_0 : E' \rightarrow .E$

$E \rightarrow .BB$

$B \rightarrow .CB$

$B \rightarrow .d$

goto( $I_0, E$ ):  $E' \rightarrow E.$   $I_1$

goto( $I_0, B$ ):  $E \rightarrow B.B$

$I_2:$   $B \rightarrow .CB$   
 $B \rightarrow .d$

goto( $I_0, C$ ):  $B \rightarrow C.B$

$I_3:$   $B \rightarrow .CB$   
 $B \rightarrow .d$

goto( $I_0, d$ ):  $B \rightarrow d.$   $I_4$

goto( $I_3, B$ ):  $B \rightarrow CB.$   $I_6$

goto( $I_3, C$ ):  $B \rightarrow C.B$   
 $B \rightarrow .CB$   $I_3$   
 $B \rightarrow .d$

goto( $I_3, d$ ):  $B \rightarrow d.$   $I_4$

~~goto( $I_2, B$ )~~

$A \rightarrow .B$

$A \rightarrow .A$

$d \rightarrow A$

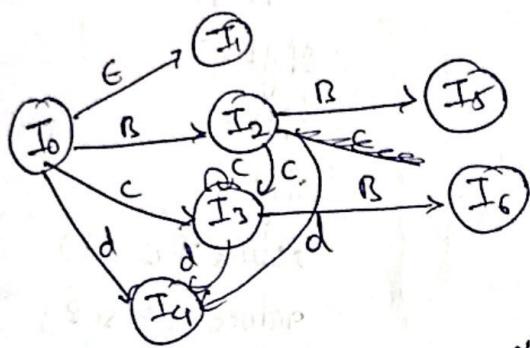
goto( $I_2, B$ ):  $E \rightarrow BB.$   $I_5$

goto( $I_2, C$ ):  $B \rightarrow C.B$

$I_3:$   $B \rightarrow .CB$   
 $B \rightarrow .d$

goto( $I_2, d$ ):  $B \rightarrow d.$   $I_4$

Step 3: Designing of finite automata



Step 4: Create Possibility table Number the productions

$$E' \rightarrow E - 0$$

$$E \rightarrow BB - 1$$

$$B \rightarrow CB - 2$$

$$B \rightarrow d - 3$$

Step 5: Create Possibility table:

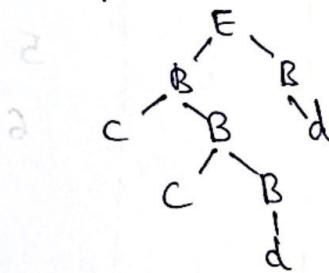
State	Action		Action \$	Goto		
	c	d		E	Goto	B
I0	s3	s4	Accepted			2
I1						5
I2	s3	s4				6
I3	s3	s4	r3			
I4	r3	r1	r1			
I5	r1	r2	r2			
I6	r2					

Steps: Stack implementation, Input: ccdd \$

Stack	Input	Action
\$0	ccdd \$	Shift c
\$0C3	cd \$	Shift c
\$0C3C3	dd \$	Shift d
\$0C3C3d4	d \$	Reduce 3 ( $B \rightarrow d$ )
\$0C3C3B6	d \$	Reduce 2 ( $B \rightarrow CB$ )
\$0C3B6	d \$	Reduce 2 ( $B \rightarrow CB$ )
\$0B2	d \$	Shift d
\$0B2d4	\$	Reduce ( $B \rightarrow d$ )
\$0B2B5	\$	Reduce 1 ( $E \rightarrow RB$ )
\$0E1	\$	Accepted

The string is accepted.

Step 6: Draw parse tree



If the grammar is not LR(0), there are multiple entries in stack cell.

⑥ SLR (Simple LR) parser:

(i) Consider the following grammar

$$S \rightarrow AS \mid b$$

$$A \rightarrow SA \mid a$$

Construct SLR parse table for the grammar. Show the action of the parser for the input string "abab".

of steps involved are:

1. Calculate canonical collection of LR(0) items.

2. Designing of finite Automata

3. Construction of SLR parse table

4. Parsing the input string abab

Augmented Grammar:

In augmented grammar, a new production is added.

$$S' \rightarrow S$$

$$S \rightarrow AS$$

$$S \rightarrow b$$

$$A \rightarrow SA$$

$$A \rightarrow a$$

[i.e. start symbol  $\rightarrow$  start symbol]

After writing the Augmented Grammar, above steps ~~are to be~~ followed:

Step 1: Calculating canonical collection of LR(0) items.

if there is a (.) dot symbol at R.H.S of production, then it's called item.

if there is a Non-terminal after (.) dot symbol, then we have to write the production of the non-terminal.

$I_0 : S' \rightarrow .S$   
 $S \rightarrow .AS$   
 $S \rightarrow .b$   
 $A \rightarrow .SA$   
 $A \rightarrow .a$

[After(.) Non-terminal  $\Rightarrow$  so A products, need to work]

$\text{goto}(I_0, S) :$   
 'S' is observed on RHS in two productions; whenever goto is applied, (.)dot position is shifted to one position right.

$\text{goto}(I_0, S) : S' \rightarrow S.$   
 $I_1 : \begin{cases} A \rightarrow S.A \\ A \rightarrow .SA \\ A \rightarrow .a \\ S \rightarrow .AS \\ S \rightarrow .b \end{cases}$

[g] there is a non-terminal after(.)dot then we need to work production of that non-terminal]

[After(.)dot there is a non-terminal]

$\text{goto}(I_0, A) :$

$I_2 : \begin{cases} S \rightarrow A.S \\ S \rightarrow .AS \\ S \rightarrow .b \\ A \rightarrow .SA \\ A \rightarrow .a \end{cases}$

$\text{goto}(I_0, b) :$

$I_3 : S \rightarrow b. \quad [dot at end \Rightarrow \text{final item}]$

$\text{goto}(I_0, a)$

$I_4 : A \rightarrow a.$

goto(I<sub>1</sub>, A):

I<sub>5</sub>:

- A → SA.
- S → A.S
- S → .AS
- S → .b
- A → SA
- A → .a
- ~~A → .AA~~

goto(I<sub>1</sub>, S):

I<sub>6</sub>:

- A → S.A
- A → .SA
- A → .a
- S → .AS
- S → .b

goto(I<sub>1</sub>, a):

I<sub>4</sub>:

- A → a.

goto(I<sub>1</sub>, b):

I<sub>3</sub>:

- S → b.

goto(I<sub>2</sub>, S)

I<sub>7</sub>

- S → AS.
- A → S.A
- A → .SA
- A → .a
- ~~S → AS~~
- S → .AS
- S → .b

goto(I<sub>2</sub>, A):

I<sub>2</sub>:

- A → A.S
- S → .AS
- S → .b
- A → .SA
- A → .a

goto(I<sub>2</sub>, a):

I<sub>4</sub>: A → a.

goto(I<sub>2</sub>, b):

I<sub>3</sub>: S → b.

goto(I<sub>5</sub>, S):

I<sub>7</sub>:

- S → AS.
- A → S.A
- A → .SA
- A → .a
- S → .AS
- S → .b

goto(I<sub>5</sub>, A):

I<sub>2</sub>:

- S → AS.
- S → .AS
- S → .b
- A → .~~AS~~ A
- A → .a

goto(I<sub>5</sub>, a):

I<sub>4</sub>: A → a.

goto(I<sub>5</sub>, b):

I<sub>3</sub>: S → b.

goto (I<sub>6</sub>, A) :

I<sub>5</sub>:  $A \rightarrow SA$   
 $S \rightarrow A \cdot S$   
 $S \rightarrow \cdot AS$   
 $S \rightarrow \cdot b$   
 $A \rightarrow \cdot SA$   
 $A \rightarrow \cdot a$

goto (I<sub>6</sub>, S) :

I<sub>6</sub>:  $A \rightarrow S A$   
 $A \rightarrow \cdot SA$   
 $A \rightarrow \cdot a$   
 $S \rightarrow \cdot AS$   
 $S \rightarrow \cdot b$

goto (I<sub>6</sub>, a) :

I<sub>4</sub>:  $A \rightarrow a$ .

goto (I<sub>6</sub>, b) :

I<sub>3</sub>:  $S \rightarrow b$ .

goto (I<sub>7</sub>, A)

I<sub>5</sub>:  $A \rightarrow SA$ .  
 $S \rightarrow A \cdot S$   
 $S \rightarrow \cdot AS$   
 $S \rightarrow \cdot b$   
 $A \rightarrow \cdot SA$   
 $A \rightarrow \cdot a$

goto (I<sub>7</sub>, S) :

I<sub>6</sub>:  $A \rightarrow S A$   
 $A \rightarrow \cdot SA$   
 $A \rightarrow \cdot a$   
 $S \rightarrow \cdot AS$   
 $S \rightarrow \cdot b$

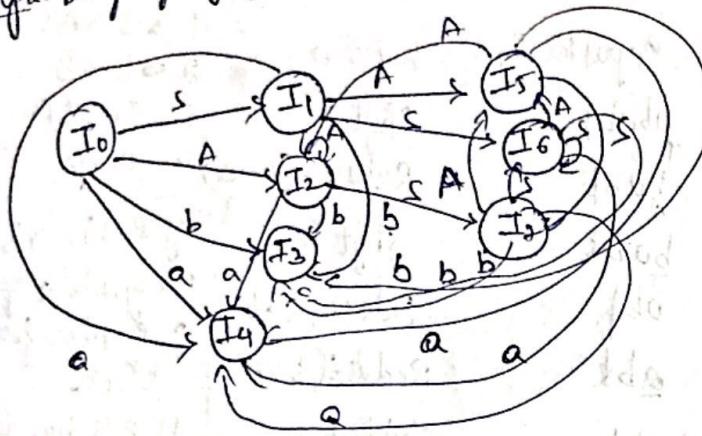
goto (I<sub>7</sub>, a) :

I<sub>4</sub>:  $A \rightarrow a$ .

goto (I<sub>7</sub>, b)

I<sub>3</sub>:  $S \rightarrow b$ .

Step 2: Designing of finite automata:



Step 3: Construction of SLR parse table

$$S \rightarrow AS | b$$

$$A \rightarrow SA | a$$

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

$$\text{Follow}(A) = \{a, b\}$$

$$\text{First}(S) = \{b, a\}$$

$$\text{First}(A) = \{a, b\}$$

Action

$\bar{I}$	a	b	$\$$	s	A
0	S4	S3	Accepted	6	5
1	S4	S3		7	2
2	S4	S3			
3	$r_2$	$r_2$			
4	$r_4$	$r_4$			
5	$S_4/r_3$	$S_3/r_3$		7	5
6	S4	S3		6	5
7	$S_4/r_1$	$S_3/r_1$	$r_1$	6	

$I_3: S \rightarrow b$  (similarly, do it for every final item)  
 Add  $r_2$  to the  $\text{follow}(S) = \{\$, a, b\}$

- Some cells contain multiple entries [shift & reduce].  
 Hence, it's not  $\text{SLR}(1)$ . [single entry  $\Rightarrow$   $\text{SLR}(1)$ ]

## Step 6: Parsing of string abab

Stack	Input Buffer	Action	
\$0	abab\$	Shift 4	$0 \Rightarrow S' \rightarrow \$$
\$0A4	bab\$	Reduce4 ( $A \Rightarrow a$ )	1. $S \Rightarrow A\$$
\$0A2	bab\$	Shift 3	2. $S \Rightarrow b$
\$0A2b3	ab\$	Reduce2 ( $S \Rightarrow b$ )	3. $A \Rightarrow SA$
\$0A2S7	ab\$	Reduce4 ( $S \Rightarrow AS$ )	4. $S \Rightarrow \$$
\$0S1	ab\$	Shift 4	If RHS has only one symbol, pop
\$0S1A4	b\$	Reduce4 ( $A \Rightarrow a$ )	2 symbols from stack.
\$0S1A5	b\$	Reduce3 ( $A \Rightarrow SA$ )	If RHS has two symbols, pop, 4 symbols
\$0A2	b\$	Shift 2	Apply '0' to A
\$0A2b3	\$	Reduce2 ( $S \Rightarrow b$ )	
\$0A2S7	\$	Reduce1 ( $S \Rightarrow AS$ )	
\$0S1	\$	Accepted	

$\therefore$  The input string is accepted by parser.

Difference between LR(0) and SLR(1):

The SLR(1) has the extra ability to help decide what action to take when there are conflicts. Because of this, any grammar that can be parsed by LR(0) parser can be parsed by an SLR(1) parser.

However, SLR(1) parser can parse a large number of grammars than LR(0).

## LALR(1) Parser:

- LALR refers to the lookahead LR. To construct the LALR(1) parser table, we use the canonical collection of LR(0) items. In LALR(1) parsing, the LR(0) items which have same production but different lookahead are combined to form a single set of items.
- LALR(1) parser is same as the CLR(1) parser, only difference is the parser table.

## CLR(1) Parser:

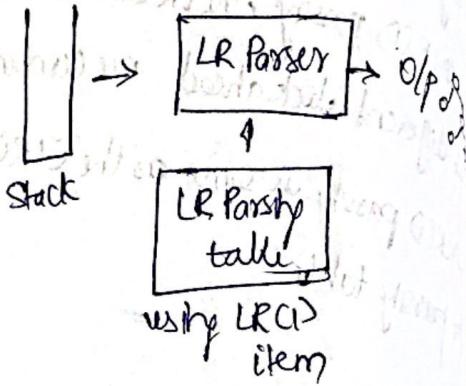
- CLR refers to canonical lookahead. CLR parser use the collection of LR(0) items to build the CLR(1) parser table.
- CLR(1) parser table produces the more number of states as compare to the LALR(1) parser.
- In CLR(1), we place the reduce node only in the lookahead symbol.
- Various steps involved in the CLR(1) Parser:
  - For the given input string, work a CPG.
  - Check the ambiguity of the grammar
  - Add Augment production in the given grammar.
  - Create Canonical collection of LR(0) items.
  - Draw a state flow diagram (DFA)
  - Construct a CLR(1) parser table

LR(1) item = LR(0) item + lookahead

$CLR(i)$  &  $LALR(i)$

$CLR(i) \downarrow$  take \$, \$ is lookahead of \$  
 $LRCDitem \Rightarrow LR(0) item + lookahead$

$LR(0) \Rightarrow$  Put reduce in full tree  
 $SLR(1) \Rightarrow$  Put reduce in follow of P



$CLR(1)$   $\Rightarrow$  Put reduce only on

$LALR(1)$  lookahead

Procedure to find lookahead:

$$E \rightarrow BB$$

$$B \rightarrow CB/0l$$

Augmented grammar

$$E' \rightarrow E$$

$$E \rightarrow BB$$

$$B \rightarrow CB$$

$$B \rightarrow d$$

$$E' \rightarrow \cdot E, \$$$

$$E \rightarrow \cdot BB, \$$$

$$B \rightarrow \cdot CB, C/d$$

$$B \rightarrow \cdot d, C/d$$

for the first production, lookahead is \$

$$\text{First}(B, \$) = \text{First}(B) = (C, d)$$

Ex: Construct LALR parser for

$S \rightarrow CC$   
 $C \rightarrow aC/b$  parse the string bb

step: Augmented grammar:

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow b$

Step 1: Calculation of LR(1) items.

I0:  $S' \rightarrow S, \$$  - 0  
 $S \rightarrow CC, \$$  - 1  
 $C \rightarrow aC, a/b$  - 2  
 $C \rightarrow .b, a/b$  - 3

goto(I0, \$):  
 $S' \rightarrow S., \$$

goto(I0, C):

I2:  $S \rightarrow C.C, \$$   
 $C \rightarrow aC, \$$   
 $C \rightarrow .b, \$$

goto(I0, a):

I3:  $C \rightarrow a.C, a/b$   
 $C \rightarrow .aC, a/b$   
 $C \rightarrow .b, a/b$

goto(I0, b):  
I4:  $C \rightarrow b., a/b$

goto(I2, C):  
I5:  $S \rightarrow CC., \$$

goto(I2, a):  
I6:  $C \rightarrow a.C, \$$   
 $C \rightarrow .aC, \$$   
 $C \rightarrow .b, \$$

goto(I2, b):  
I7:  $C \rightarrow b., \$$

goto(I3, C):

I8:  $C \rightarrow aC., a/b$

goto(I3, a):

I9:  $C \rightarrow a.C, a/b$   
 $C \rightarrow .aC, a/b$   
 $C \rightarrow .b, a/b$

$\text{goto}(I_3, b)$ :

$I_4: C \rightarrow b\cdot, a/b$

$\text{goto}(I_6, C)$ :

$I_9: C \rightarrow aC\cdot, \frac{1}{a}$

$\text{goto}(I_6, a)$ :

$I_6:$

- $C \rightarrow a\cdot C, \frac{1}{a}$
- $C \rightarrow \cdot aC, \frac{1}{a}$
- $C \rightarrow \cdot b, \frac{1}{a}$

$\text{goto}(I_6, b)$ :

$I_7: C \rightarrow b\cdot, \frac{1}{a}$

We can combine  $I_3 \cup I_6 \rightarrow I_{36}$  [Same production but different lookahead]

$C \rightarrow a\cdot C, a/b\frac{1}{a}$

$C \rightarrow \cdot aC, a/b\frac{1}{a}$

$C \rightarrow \cdot b, a/b\frac{1}{a}$

Similarly  $I_4 \cup I_7 \rightarrow I_{47}$

$C \rightarrow b\cdot, a/b\frac{1}{a}$

Similarly  $I_8 \cup I_9 \rightarrow I_{89}$

$C \rightarrow aC\cdot, a/b\frac{1}{a}$

Conclusion

Construction of LALR parse table.

	a	b	\$	s	c
I	S36	S47		1	2
0			Accepted		
1	S36	S47			5
2	S36	S47			89
36	S36	r3	r3		
42	r3	r3	r1		
5		r2	r2		
89	r2				

Stack implementation:

Stack

\$0

\$0b47

\$0C2

\$0C2b47

\$0C2CS

\$0S1

Alphabty

bb\$

b\$

b\$

\$

\$

\$

\$

Action

S47

r(C → b)

S47

r(C → b)

r(S → CC)

Accepted

The input strty is accepted by parser

3) Construct CLR parser table for the grammar

$$\begin{array}{l} S \rightarrow CC \\ E \rightarrow aC \\ C \rightarrow d \end{array}$$

parse dd\$

Augmented grammar:  $\begin{array}{l} S' \rightarrow S \rightarrow_0 \\ S \rightarrow CC \rightarrow_1 \\ C \rightarrow aC \rightarrow_2 \\ C \rightarrow d \rightarrow_3 \end{array}$

LR(1) items:

$$\begin{array}{l} I_0 : S' \rightarrow S \cdot, \$ \\ S \rightarrow .CC, \$ \\ C \rightarrow aC, a/d \\ C \rightarrow .d, a/d \end{array}$$

goto ( $I_0, \$$ )

$$I_1 : S \rightarrow S \cdot, \$$$

goto ( $I_0, C$ )

$$\begin{array}{l} I_2 : S \rightarrow C \cdot C, \$ \\ C \rightarrow .aC, \$ \\ C \rightarrow .d, \$ \end{array}$$

goto ( $I_0, a$ )

$$\begin{array}{l} I_3 : C \rightarrow aC \cdot, a/d \\ C \rightarrow .aC, a/d \\ C \rightarrow .d, a/d \end{array}$$

goto ( $I_0, d$ )

$$I_4 : C \rightarrow d \cdot, a/d$$

goto ( $I_2, C$ )

$$I_5 : S \rightarrow CC \cdot, \$$$

goto ( $I_2, a$ )

$$\begin{array}{l} I_6 : C \rightarrow E \cdot aC, \$ \\ C \rightarrow .aC, \$ \\ C \rightarrow .d, \$ \end{array}$$

goto ( $I_2, d$ )

$$I_7 : C \rightarrow d \cdot, \$$$

goto ( $I_3, C$ )

$$I_8 : C \rightarrow aC \cdot, a/d$$

goto ( $I_3, a$ )

$$I_9 : C \rightarrow aC \cdot, a/d$$

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x

y

z

?

;

,

.

+

\*

-

/

\*

=

<

>

;

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

:

&lt;p

Action			goto	
state	a	d	s	c
0	s3	s4	Accepted	
1		s7		5
2	s6			8
3	s3	s4		
4	g13	g13	g11	
5		s7		9
6	s6		g13	
7				
8	g12	g12		
9			g12	

stack implementation!

Stack

\$0

\$0d4

\$0C2

\$0C2d7

\$0C2C5

\$0S1

Input String

dd \$

d \$

d \$

\$

\$

\$

Action

shift 4

reduce (c→d)

shift 7

reduce (c→d)

reduce (s→cc)

Accepted

The input string is accepted.

## ⑥ Using Ambiguous Grammars:

Only operator precedence parser accepts ambiguous grammars.

Operator grammar:

- A grammar that is used to define mathematical operators is called an operator grammar (or) operator precedence grammar.
- A grammar is said to be operator precedence grammar if it has a properties:
  - (i) no R.H.S of any production has a  $\epsilon$  (epsilon)
  - (ii) no two non-terminals are adjacent on R.H.S.

Ex:  $E \rightarrow E+E \mid E * E \mid id$  // operator grammar

Ex:  $S \rightarrow SAS \mid a$       // not operator grammar  
 $A \rightarrow bsb \mid b$       [no two non-terminals can be adjacent]

We can convert it into operator grammar

$S \rightarrow Sbsbs \mid a \mid sbs$  // operator grammar

[ $A \rightarrow bsb \mid b$ ] is not required

there are 3 operator precedence relations:

- $a > b \rightarrow$  means 'a' has higher precedence than the terminal 'b'  
 'a' takes precedence over 'b'
- $a < b \rightarrow$  means 'a' yields precedence to 'b'  
 'a' has lower precedence than terminal 'b'

3)  $a \div b \Rightarrow$  means 'a has same precedence as b'.

Operator Precedence Parser:

Bottom up parser that interprets on operator grammar.

This parser is only used for operator grammar.  
Ambiguous grammar are not allowed in any parser except  
operator precedence parser

Ex:  $E \rightarrow E + E \mid E^* E \mid id$

Operator precedence relation table

	id	+	*	\$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	-
\$	<	<	<	-

defining precedence rules

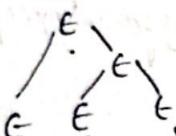
- Two ids will never be compared 'becoz they will never come side by side.
- Identifier will be given highest precedence compared to any other operator.
- \* has least precedence compared to any other operator.
- Using this operator precedence table, parser will parse the i/p.

$E \quad E \quad E$   
 $| id + id * id |$   
 id + id \* id  
 id + id \* id

if top of stack  
 of the precedence is higher  
 perform pop operation  
 else

\$	id	+	id	*	id	
----	----	---	----	---	----	--

stack



id + id \* id

if the precedence of operator is higher  
 perform push operation  
 else  
 (parse tree generated)

### Disadvantage of Operator Relation Table

when no. of entries are large, size of table will be  $O(n^2)$   
 for N operators  $\rightarrow O(n^2)$

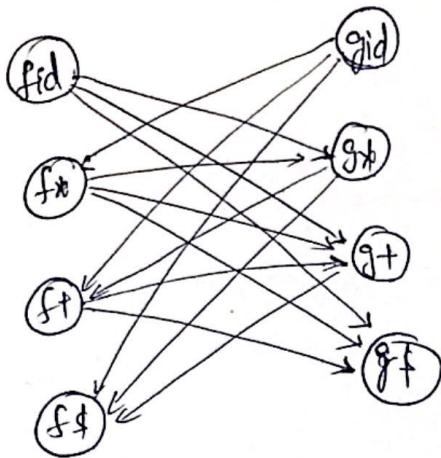
so, to decrease the size of the table, we use operator functions

for table:

$f$	$g$	id	+	*	\$
id	-	>	>	>	>
+	<	>	<	>	>
*	<	>	>	>	>
\$	<	<	<	-	-

Now we will construct a graph.

Now for every operator, we will have  $f_{id}, g_{id}, f_+, g_+, f_*, g_*$ , and  $f_{$}, g_{$}$



If we found any cycle in  
graph, we will stop there  
because then we cannot  
construct an operator  
function table

To make function table:

We have to find out for every node that what is the length  
of the longest path starting from that node.

$$\begin{aligned} \text{fid} &\rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$ \\ \text{gid} &\rightarrow f* \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$ \end{aligned}$$

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

Advantage:

- ⇒ size is less.
- ⇒  $O(2n)$  ~~less~~ Space complexity

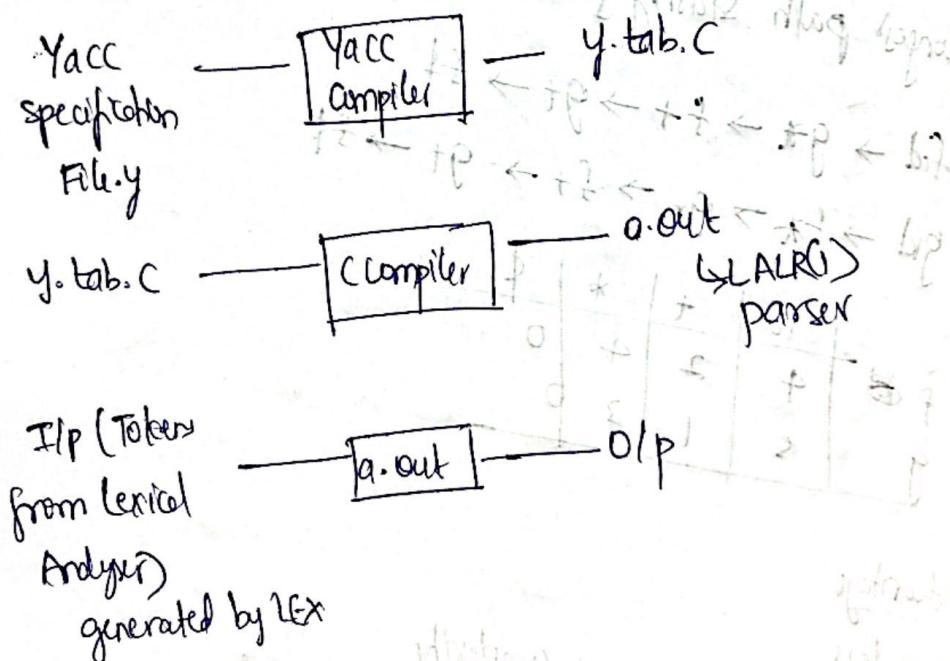
Disadvantage:

Error detection capability of function table is less compare  
to operator relation table.

## ⑦ Parser Generator (YACC in compiler design):

- It stands for Yet Another Compiler - Compiler
- It stands for Yet Another Compiler - Compiler
- It stands for Yet Another Compiler - Compiler Look-Ahead Left-to-Right (LALR)
- It's a tool for generating parser
- It takes I/p from the lexical analyzer and generates parse tree
- Syntax Analyzer / parser is the 3<sup>rd</sup> phase of the compiler which takes I/p as tokens and generates a parse tree.

### Working (3 steps) / Block diagram:



Working:

- 1) I/p to the Yacc compiler will be a file with .y extension.  
it will contain desired grammar in yacc format. YACC compiler  
will convert it into a C code in the form of y.tab.c file.
- 2) this y.tab.c file will be given as a input to the compiler  
and the output will be LALR (i.e a.out).
- 3) Tokens generated by the lexical analyser (using the lex tool)  
will be given as input to a.out; i.e. ~~Lexer~~ parser we will  
get the parse tree as O/p.