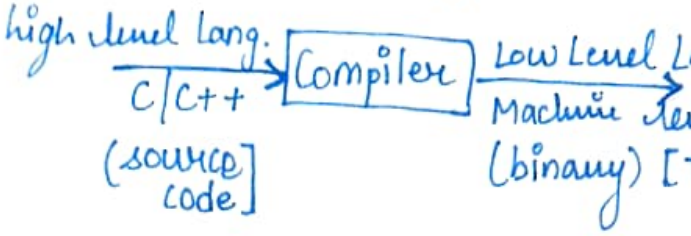
Compiler Design

What is Compiler ?

⇒ Compiler is a software with Input = high level language and output = low level language.

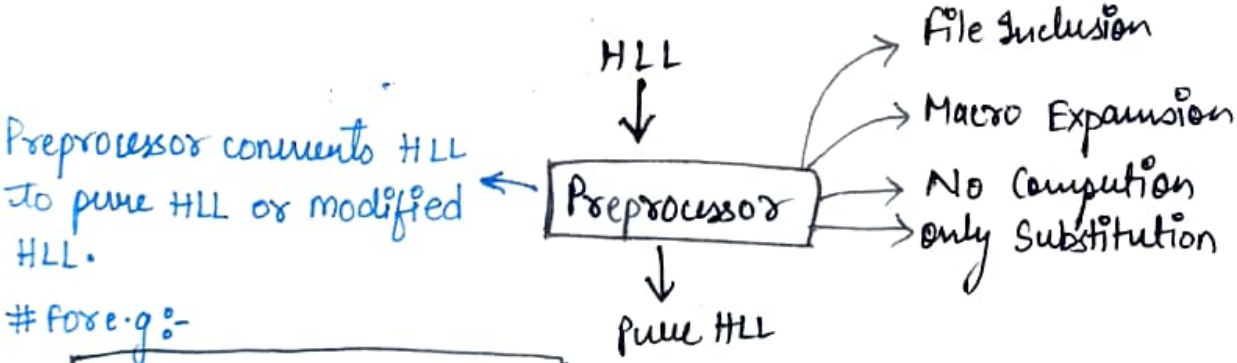


⇒ Machine level language is understood by machine → processor

⇒ Compiler is a translator.

Process of translation of HLL To LLL :-

Language Processing System

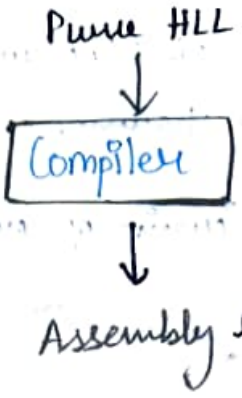


for e.g :-

```
#include<stdio.h>
#define sqr(a) a*a
main(){
    int b, a = 3;
     b = 10+sqr(a);
    Printf("%d", b);
    }
```

The Printf() function is defined in #include<stdio.h> which is Preprocessor directive.

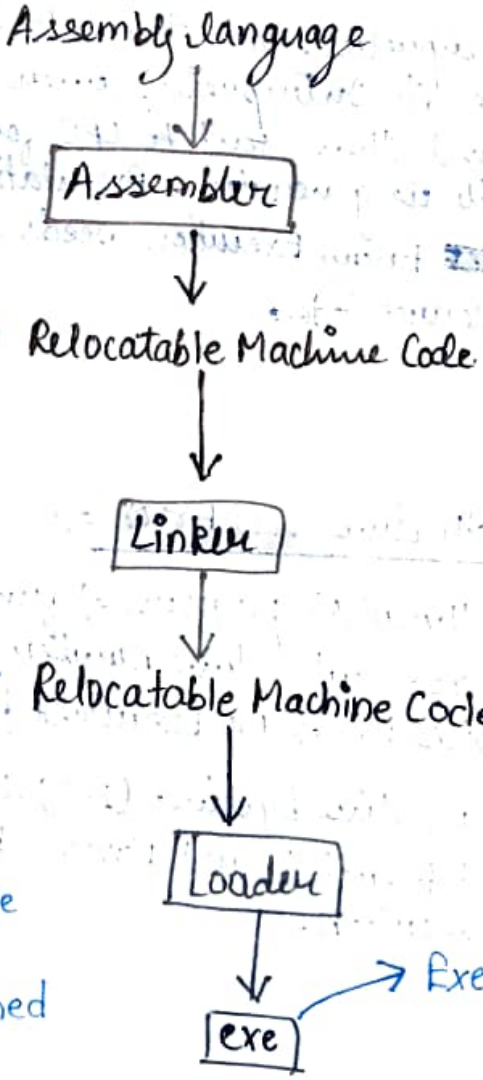
Preprocessor replaces all the instances of printf() and other predefined function with actually defined functions in preprocessor directive and Remove all #include statements from source code to convert HLL to Pure HLL without any preprocessor directive. This process is called file inclusion. #define is also replaced. This is called macro expansion.



Assembly language is a language written with symbols and Registors.

$x = a + b * c$ ⟶

```
MUL R2 ← R0, R1
ADD R2 ← R2, R1
MOVE X, R2
```



Relocatable machine Code corresponding to multiple source codefile.

Linker links all the Relocateable machine Code of necessary files and Convert them into a single file of needed relocatable machine Code.
All source code file combined to single file.

Roles of Compiler :-

(i) Converts source code to target code if no error.
(ii) If Error, Compilation error is shown.

Assembler, Linker, Loader are Cousins of Compiler.

Compiler V/S Interpreter :-

Compiler → Compiler converts the source code into Executable file if no error is present. The Exe. file is now stored in hard disk or secondary memory. Now Execution can be performed multiple / Infinite times using the EXE. file without the involvement of Compiler.

Interpreter :- Interpreter reads the code line by line. first line goes to Interpreter, Conversion in Executable format and then sent to CPU for Execution. No EXE file is generated. Execution takes place while Reading. Every Execution needs the Interpretor and source code.

Erroneous Code :-

If 6th, 10th, 100th line has Error ⇒

Compiler Reads the whole program ⇒ finds Error and displays Compilation Error with line number of Erroneous Code. If error is present ⇒ No Executable file is generated.

Interpretor reads line by line. (1-5) lines with no error is Translated and Executed and Error shown in 6th line only. No further Execution.

Compiler based languages are faster than Interpreter based languages.

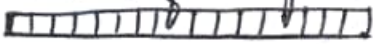
Space taken by Compiler based language in main memory is greater than Interpretor based languages ⇒ because in case of Compiler ⇒ whole programs goes to main memory and in case of Interpretor ⇒ only one line goes to main memory.

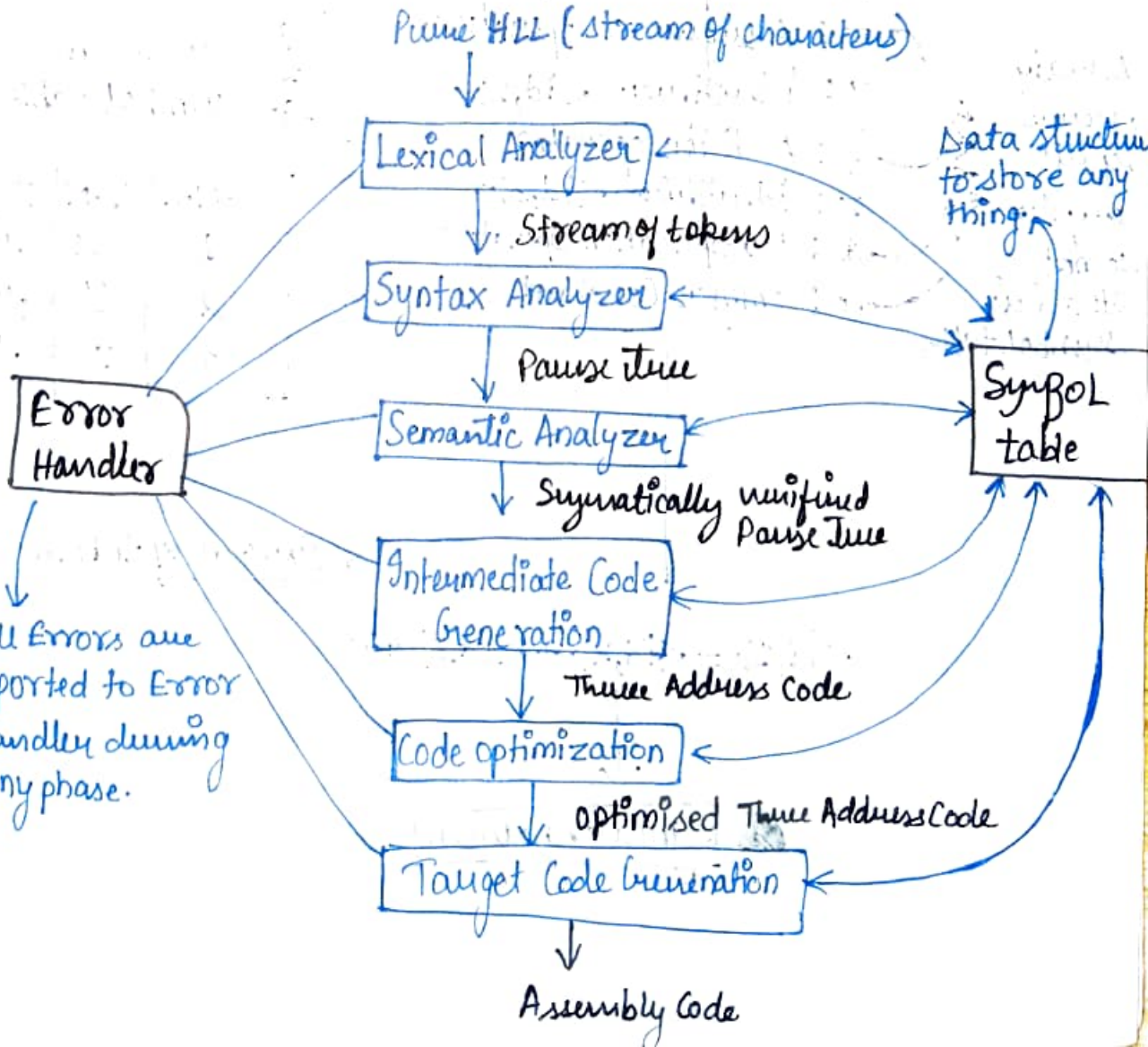
Compiler based languages have difficult debugging than Interpretor based languages.

Compiler based language ⟶ C/C++

Interpreter based language ⟶ Javascript.

Phases of Compiler :-





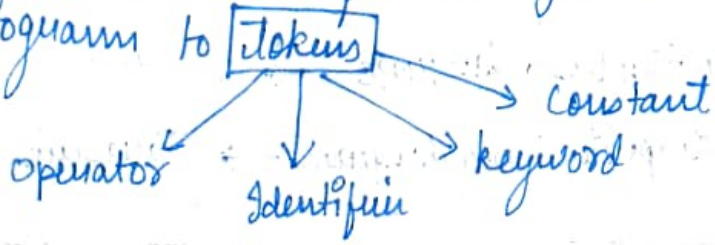
Consider the following program :-

```
float x,y,z;
x = y+z*60;
```

} High level Pure language or streams of characters

(1) Lexical Analyzer :-

⇒ It takes streams of characters as input and gives tokens.
⇒ It converts the program to tokens



tokens :-

x : identifier <id,1>
= : Assignment operator.
y : identifier <id,2>
+ : Addition operator
z : identifier <id,3>
* : Multiplication Operator
60 : Integer Constant

Already known by the Compiler so not stored in symbol table

stores in symbol table

S.No.	V.N	Type
1	x	float
2	y	float
3	z	float

<id,1> = <id,2> + <id,3> * 60 → stream of tokens

Syntax Analyser

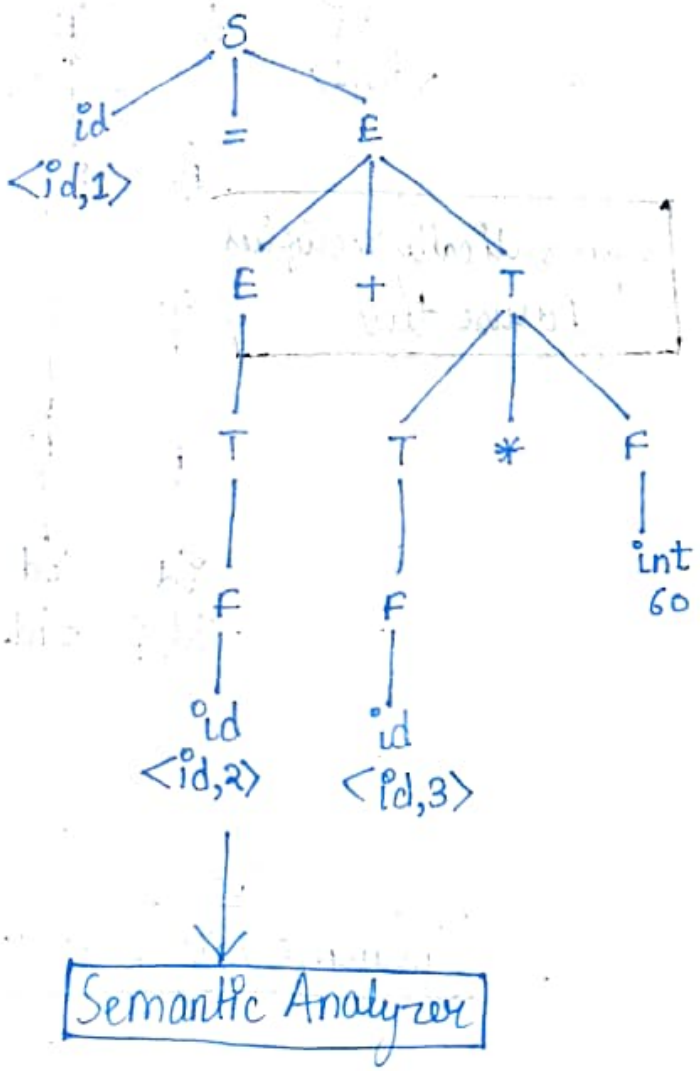
(2) Syntax Analyzer :-

Syntax Analyzer is also called Parser. It takes a stream of tokens and gives a parse tree as output.

TO Construct Parse Tree ⇒ we need grammer ⇒ grammar is set of Rules. The following grammer is considered for the current Input :-

$$S \rightarrow id = E$$
$$E \rightarrow E+T/T$$
$$T \rightarrow T*F/F$$
$$F \rightarrow id/int$$

Parse tree :-



(3) Symantic Analyzer :-

⇒ It converts the Parse tree into symatically Verified Parse Tree.

⇒ Symatic Analyzer stores all the identifiers and Variables into the Symbol table.

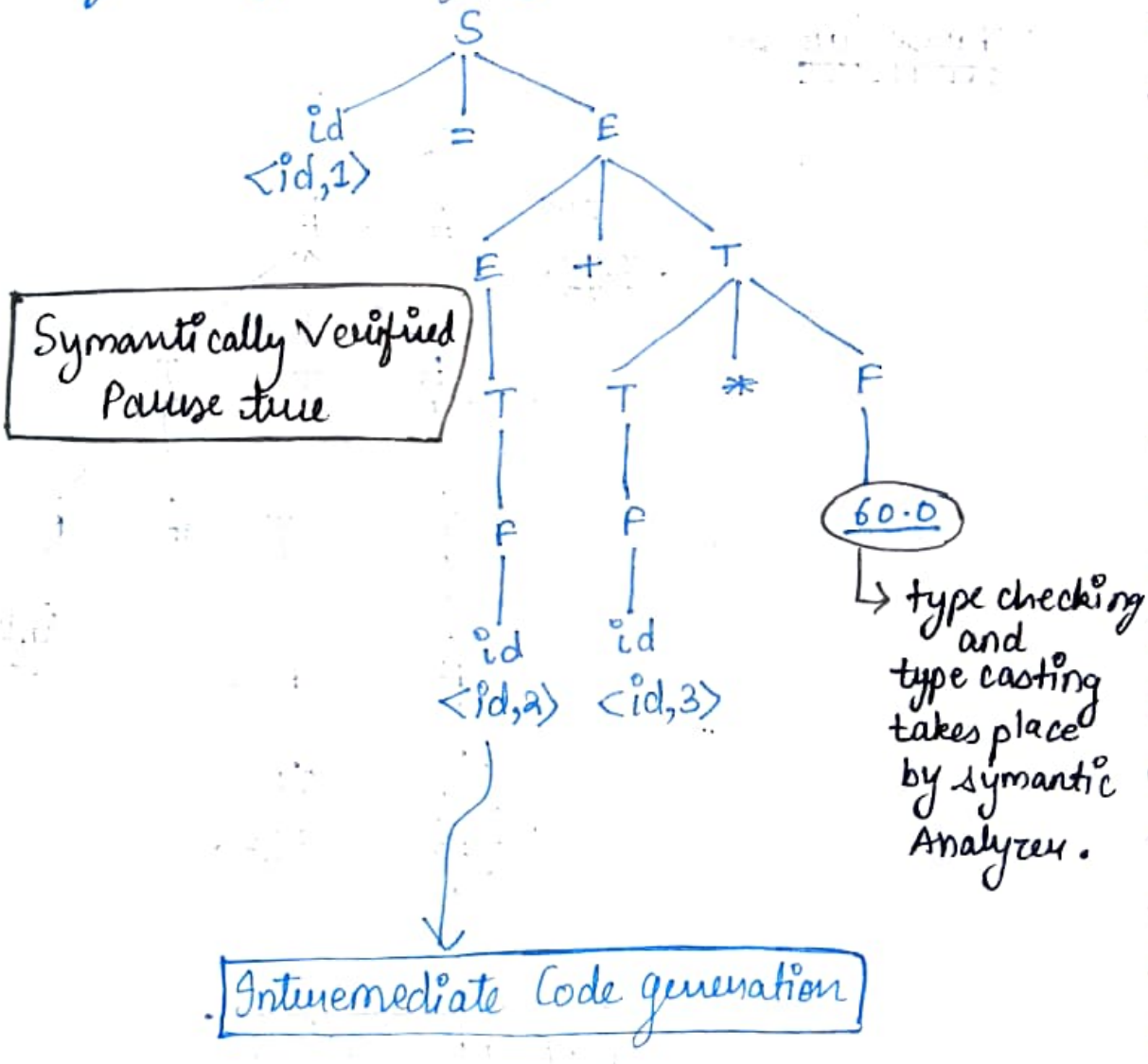
Roles of Symatic Analyzer :-

(I) Type checking → All variables/ operations are homo-type.

(II) undeclared variables

(III) multiple declaration of Variables.

type checking is verified using symbol table.



(4) Intermediate Code generation

Converts Syntactically Verified Parse tree into three address Code.

Three address Codes uses only three Variables in the Code.

Intermidiate Code :-

$t_1 = z * 60.0$
$t_2 = y + t_1$
$x = t_2$

} three address Code.

↓

Code optimization

(5) Code optimization :-

It reduces the size of the Code and optimizes the Code.

$t_1 = z * 60.0$
$x = y + t_1$

} reduced Size Optimised three address code.

This is not compulsory phase of compiler.

↓

Target Code generation

(6) Target Code Generation :-

It converts the three address Code into Assembly language.

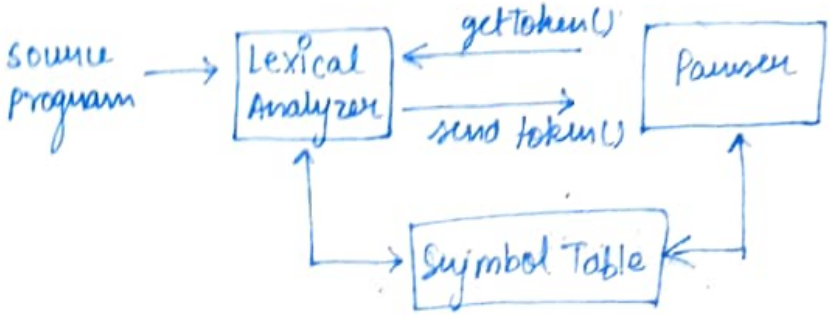
```
MUL  R0, 60.0
ADD  R1, R0
STORE X, R1
```

Lexical Analyzer :-

It divides the given program into ~~meaning~~ ~~wrong~~ meaningful words called tokens.

Tokens are normally identifiers, keywords, operators, constant and special symbol.

L.A eliminates lines of comments from the given program.



All phases of compiler exeute simultaneously. They do not execute in an order.

First phase to execute is syntax analyzer / Parser. It obtains / requests tokens from lexical Analyzer. As it get each token ⇒ it generates Parse tree simultaneously.

L.A eliminates white spaces characters (blank / Tab)

L.A helps in giving error message by providing row no / col no.

Error handler obtains line no. of lexical Analyzer and stores it and Compilation continues, to find further errors in the code.

L.A scans the whole program.

The Syntax Analyzer / Parser informs Error handler about the occurence of the error, and next row no. / col no. is determined by Error handler from Lexical Analyzer.

Tokenization by L.A :-

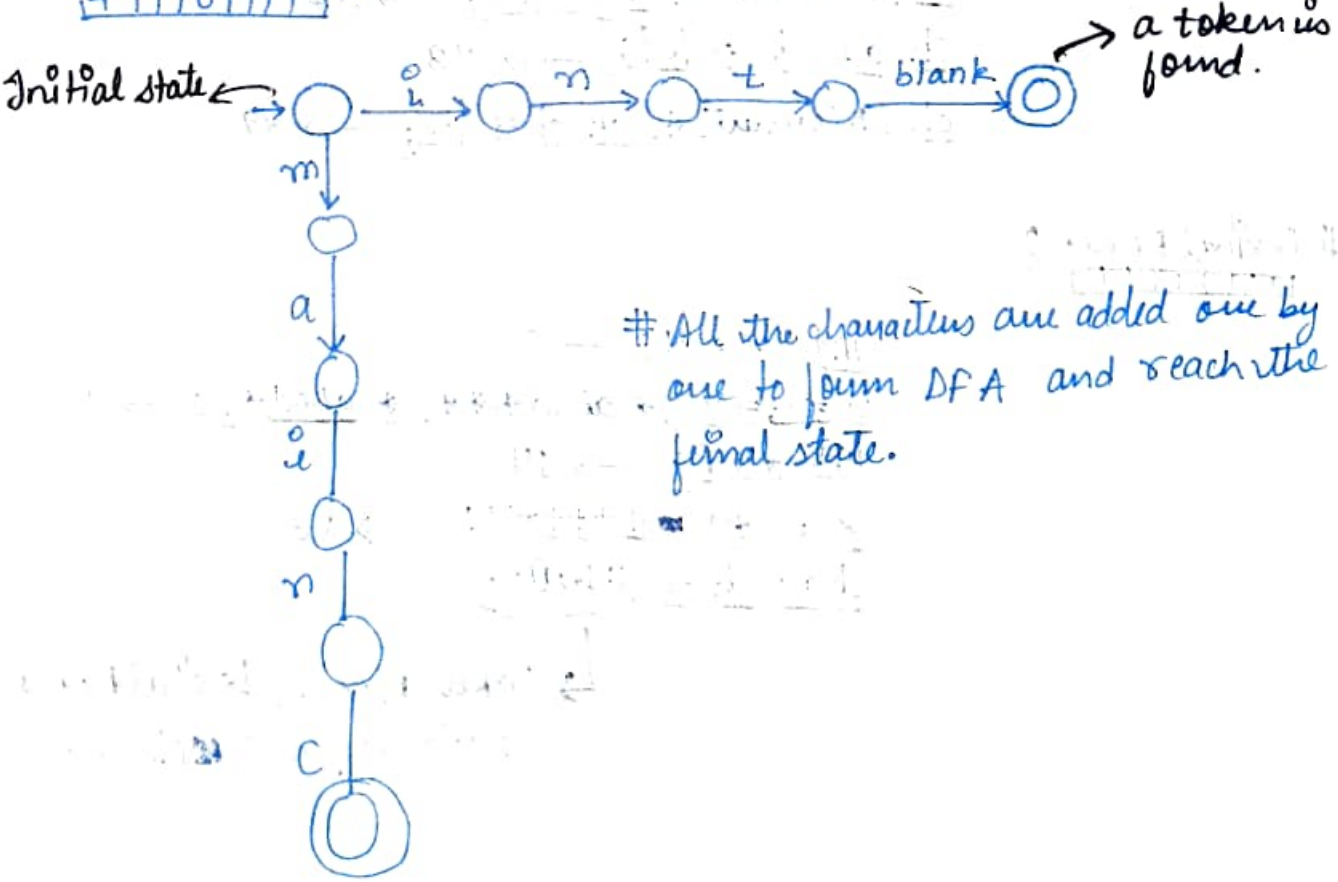
L.A uses DFA to do tokenization.
While doing tokenization L.A always gives importance to longest matching.

Eg :- Program :-

```
int main(){                          → 5
    int i;                           → 8
    for(i=0; i<5; i++){              → 22
        int i = 102;                 → 27
        Printf("\n the Value of i :", i); 34
        i++;                         → 37
    }                                → 38
    return 0;                        → 41
}                                    → 42
```

(Longest matching token)

Use of DFA :-



All the characters are added one by one to form DFA and reach the final state.

Example-2 :-

```
main()(      → 3
}            → 4
  x = a+b*c;     → 12
  int x,a,b,c;   → 21
  y = x+a;       → 27
{                → 28 total tokens.
```

L.A does not identify any semantic errors / It only converts the program into tokens.

L.A converts whole above programs into tokens ⇒ total tokens = 28.

Example-3 :-

```
main(){      → 4
  char *c = "strings";   → 10
  float b = 100.74;      → 15
  char d = 'e';          → 20
  int f = 200;           → 25
  in t f = 200;          → 31
  in /*Comment*/ t f = 200;   → 37
  ch ar d = 'e';         → 43
  ch /*Comment*/ ar d = 'e';  → 49
```

Lexical Error :-

```
int x = 4;          → 5
x===y /* abcd***/ *abcd*/;   → 9
int **p;            → 14
x = *p ++++++y;     → 23
char ch = "hello;
```

↳ token Error / Lexical Error.
⇒ string is not closed.

Ambiguous Grammar :-

we need to check if the given grammar is ambiguous or not.

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

Let a string :-

$$w = (id + id*id) \quad \text{---} (1)$$

⇒ write derivations corresponding to the strings ⇒ { LMD, RMD }

Left Most Derivation :-

Variable = {E}

Terminal = {+, *, id}

Considering first Production Rule :- Initially.

∴ $E \rightarrow E+E$ (Replacing Left Most 'E' on Rhs)

$E \rightarrow id+E$ (Replacing left most 'E' on Rhs)

$E \rightarrow id+E*E$

$E \rightarrow id+id*E$

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

$\boxed{E = id+id*id}$ ⇒ eq-(1) is derived.

Right most derivation :-

Considering first Production Rule Initially :-

$E \rightarrow E+E$

$E \rightarrow E+(E*E)$

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

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

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

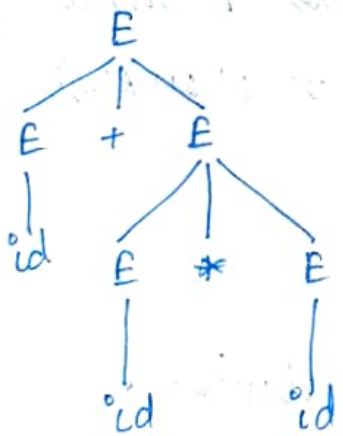
LMD2 :- 2nd Left Most derivation :-

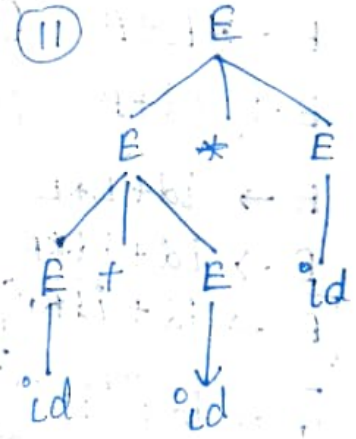
$E \rightarrow E*E$ (using 2nd Production Rule)
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$ eq-① achieved.

RMD2 :- 2nd Right Most derivation :-

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

Derivation Using Parse trees :-

① 

(II) 

Conclusion :-

→ If corresponding to any grammar, any string generated from this grammar has two different Left most derivations or two different Right most derivations or two different structure Parse tree are obtained then such a grammar

is called ambiguous grammar.

Example-2 :- $S \longrightarrow aS|Sa|a$

Ans Assuming the string to be $\boxed{w = "aa"}$

LMD1 :-

using first production Rule

$S \longrightarrow aS$
$\longrightarrow aa$ ✓

LMD2

using second production Rule

$S \longrightarrow Sa$
$\longrightarrow aa$ ✓

RMD1

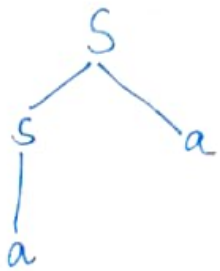
$S \longrightarrow aS$
$\longrightarrow aa$

RMD2

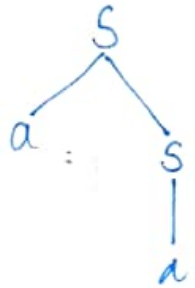
$S \longrightarrow Sa$
$\longrightarrow aa$

Parse Trees

(I)



(II)



the given grammar $S \rightarrow Sa|aS|a$ is also ambiguous.

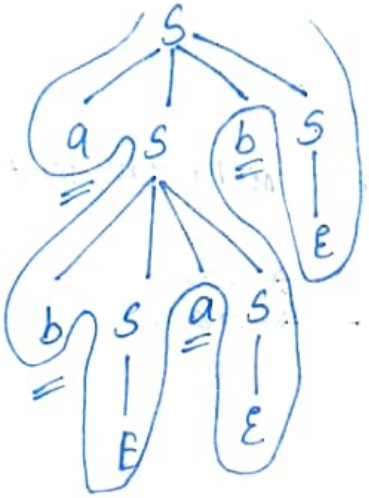
Example - 3

$$S \rightarrow aSbS|bSaS|\epsilon$$

Ans Assuming the string $\Rightarrow w = abab$.

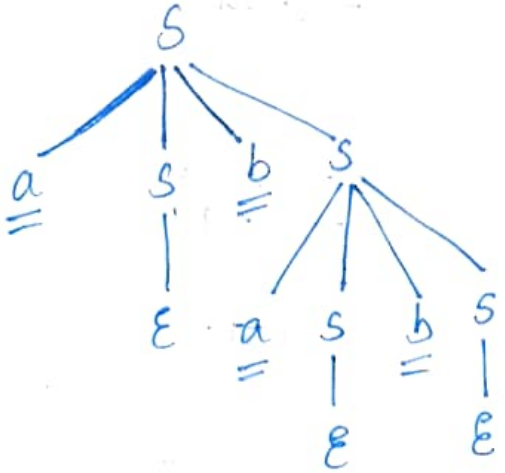
Parse Trees

(I)



$\rightarrow abab$

(II)



$= abab$

The above grammar is also ambiguous due to two different syntax parse tree.

No Algorithm to identify Ambiguous grammar ⇒ purely hit and trial method.

Ambiguous to Unambiguous grammar :-

E → E+id → Left Recursive grammar.

E → id+E → Right Recursive grammar.

If LHS variable is also present in Rhs on left most place. LRG.

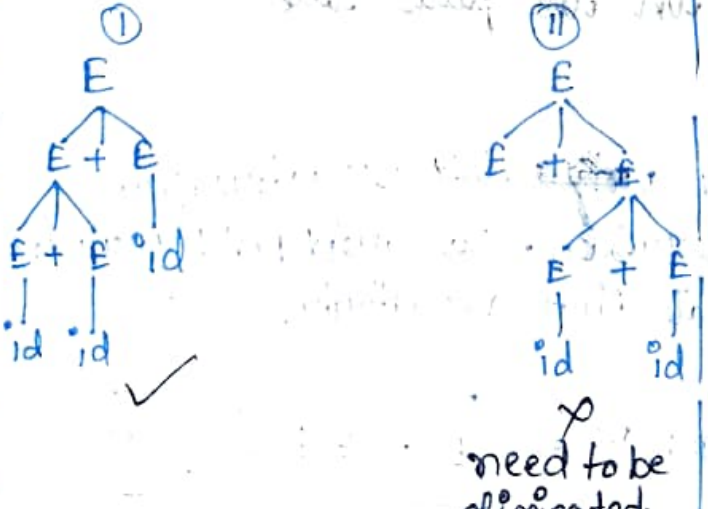
If Rhs variable is also present in Rhs on Right most place, is called Right Recursive grammar.

Considering Ambiguous grammar :-

$$E \longrightarrow E+E \mid E*E \mid id$$

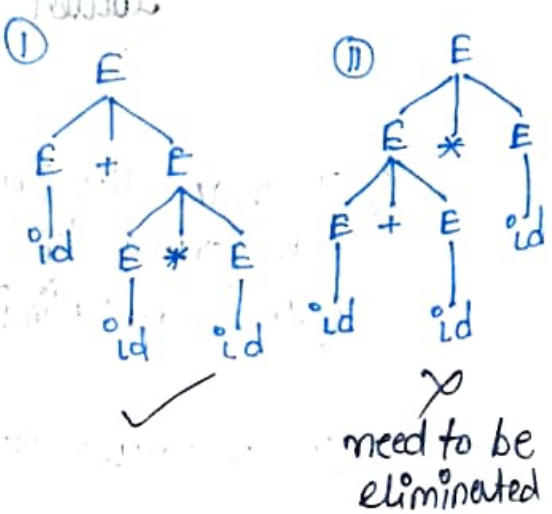
Considering string

$w = id+id+id$



Considering string

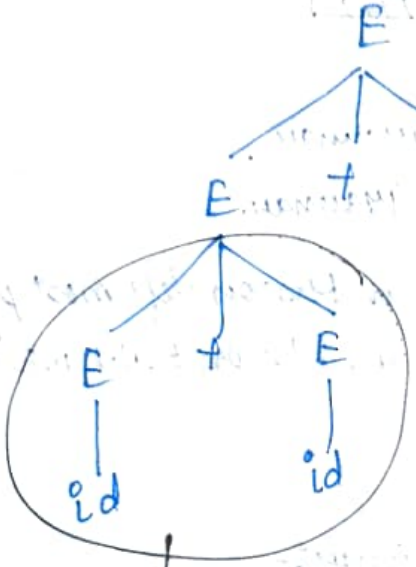
$w = id+id*id$



In first string ⇒ $w = id+id+id$ ⇒ '+' sign has an associativity of left → Right. The string 'w' must be

$$\boxed{w = ((id+id)+id)}$$

corresponding parse tree is the first one.



→ we always calculate the operator at the lowest level of tree first.

→ operators having higher precedence and higher associativity must come lower in the parse tree.

∴ To make the tree a left associative ⇒ we can form a left Recursion. This avoid Right grouping of tree and Eliminates Right associativity.

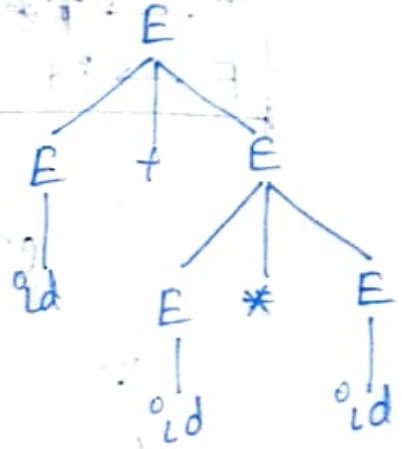
Un Ambiguous grammar ⇒ $\boxed{E \longrightarrow E+id \mid id}$

This follows the associativity Rule.

To make the Tree right associative ⇒ we can form a Right Recursion, so that we may eliminate the left associativity Possibilities and :-

$$E \longrightarrow id + E \mid id$$

In second string ⇒ w = id + (id * id) ⇒ '*' has more precedence than '+' operator then the '*' operator must be in the lower level of Parse tree :-



To convert the grammar to unambiguous ⇒ we need to convert the grammar Production such that all the '+' operators are dealt first and then all the '*' operators are dealt and comes to the lower-level of parse tree.

start variable.

$$E \longrightarrow E + T$$
$$T \longrightarrow T * F$$

→ left Recursion due to left associativity of '+' sign

→ left Recursion due to left associativity of '*' sign.

⇒ we always start with the variable 'E'. All the '+' operators are dealt with first here and then all the 'T' are replaced to obtain the '*' operators and precedence is also handled. we can go from E → T and produce '+' but we can never produce a '*' operator before '+' from T → E.

∴ For string ⇒ $w = id*id*id$ ⇒ we can further modify the grammar as :-

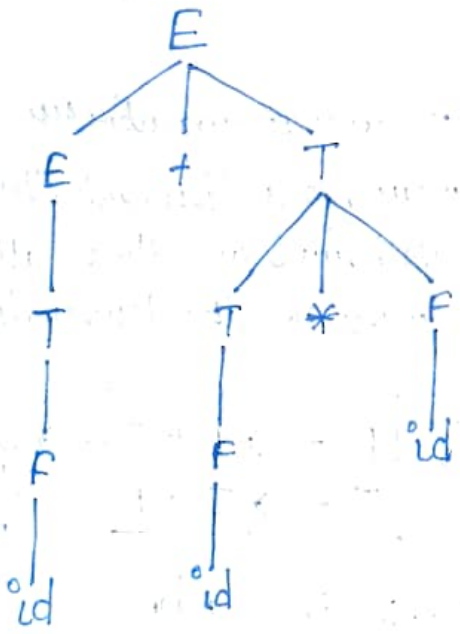
$$E \longrightarrow E+T/T$$
$$T \longrightarrow T*id$$

∴ For string ⇒ $w = id$ ⇒ we can further modify the grammar as :-

$$E \longrightarrow E+T/T$$
$$T \longrightarrow T*F/F$$
$$F \longrightarrow id$$

↝ Un Ambiguous grammar

corresponding Parse tree



only one parse tree is formed following all the associativity and Precedence.

Recursion :-

There are two types of Recursion → Left Recursion, Right Recursion

Left Recursion	Right Recursion
# $A \rightarrow A\alpha / \beta$	$A \rightarrow \alpha A / \beta$
# If Rhs's Left Most Variable is same as Lhs variable.	# If Rhs's Right most Variable is same as Lhs Variable.
# generated Language is :- $[\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha^3, \beta\alpha^4 \ldots]$	# generated language is :- $\{\beta, \alpha\beta, \alpha^2\beta, \alpha^3\beta \ldots\}$
Expression = $\beta\alpha^*$	Regular Exp = $\alpha^*\beta$

Problems :-

① Top-down parser does not allow production Rule with Left Recursion. Left Recursion need to be Removed.

Removing Left Recursion :-

We need to Remove the Left Recursion from the grammar $A \rightarrow A\alpha / \beta$ and produce the same language RE = $\beta\alpha^*$ from new grammar without left Recursion.

∴ $RE = \beta\alpha^*$

Removing left Recursion

$A \longrightarrow A\alpha / \beta$ — (I)

$A \longrightarrow \beta A'$
$A' \longrightarrow \alpha A' / \varepsilon$ — (II)

↳ This grammar generates the same Language $RE = \beta\alpha^*$ without left Recursion but Contain Right Recursion.

Example-1 :- Eliminate left Recursion :-

$A \quad A \quad \alpha$
↗ ↗ →
① $E \longrightarrow E+T / T$ → β

Ans ⟩ Comparision with eq-①

Converting into left Recursion free grammar using Analogous to eq-(II)

$E \longrightarrow TE'$
$E' \longrightarrow +TE' / \varepsilon$ .....

∴ Hence, left Recursion is Removed.

Example-2 :-

(i) $S \longrightarrow S01SOS/01$

$A \quad A \; \alpha \quad \beta$

Ans Let, here Left Recursion Exist ⇒

Converting Analogous to eq-(II)

$S \longrightarrow 01S'$

$S' \longrightarrow \varepsilon | 01SOSS'$

(ii) $L \longrightarrow L,S/S$

$A \quad A \; \alpha \quad \beta$

Ans here, Left Recursion Exists :-

Converting Analogous to eq.-(II)

$L \longrightarrow SL'$

$L' \longrightarrow \varepsilon | ,SL'$

(iii) $A \longrightarrow A\alpha_1 | A\alpha_2 | A\alpha_3 | \ldots\ldots$

Ans we can see it as :-

$A \longrightarrow \beta_1 | \beta_2 | \beta_3 \ldots\ldots$

Converting Analogous to eq-(II)

$A \longrightarrow \beta_1 A' | \beta_2 A' | \beta_3 A' \ldots\ldots$

$A' \longrightarrow \varepsilon | \alpha_1 A' | \alpha_2 A' | \alpha_3 A' \ldots\ldots$

Left Factoring :-

The Process of Converting the Non-deterministic grammar to deterministic grammar is called left factoring.

What is Non-deterministic grammar :-

⇒ Let we have a grammar :-

$$A \longrightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \alpha\beta_4 \longrightarrow \text{Non-deterministic}$$

⇒ If in a grammar ⇒ all the optional production Rules have some common prefix ⇒ then the grammar is called Non-deterministic.

⇒ Here, $\alpha\beta_1, \alpha\beta_2, \alpha\beta_3, \alpha\beta_4$ ⟶ have **Common Prefix = α**

⇒ Let we have a grammar :-

$$\left.\begin{array}{l} A \longrightarrow \alpha\beta_1 \\ B \longrightarrow \alpha\beta_2 \end{array}\right\} \longrightarrow \text{Deterministic}$$

Converting NDCFG to DCFG :-

$$A \longrightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 | \alpha\beta_4$$

then ⇒

$$\left.\begin{array}{l} A \longrightarrow \alpha\beta' \\ \beta' \longrightarrow \beta_1 | \beta_2 | \beta_3 | \beta_4 \end{array}\right\} \text{Non-deterministic grammar.}$$

Examples :- Converting NDCFG to DCFG

(I) $S \rightarrow iEtS/iEtSeS/a$
$E \rightarrow b$

(Ans) For same variable – 'S' ⇒ If we have a common prefix in the production ⇒ ND grammar exist.

Common Prefix = $iEtS$.

Converting :-

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

(II) $S \rightarrow aSSbS/aSaSb/abb/b$

(Ans) # Common Prefix = a

$S \rightarrow aS'/b$ ⟶ Deterministic
$S' \rightarrow SSbS/SaSb/bb$ ⟶ Non-deterministic

Common Prefix = 'S'

$S' \rightarrow SS''/bb$
$S'' \rightarrow SbS/aSb$

Overall grammar :-

$S \rightarrow aS'/b$
$S' \rightarrow SS''/bb$
$S'' \rightarrow SbS/aSb$

(iii) $S \longrightarrow bSSaaS / bSSaSb / bSb / a$

(Ans) # Common Prefix = bS

$S \longrightarrow bSS' / a$

$S' \longrightarrow SaaS / SaSb / b$

Common Prefix = Sa

$S \longrightarrow bSS' / a$

$S' \longrightarrow SaS'' / b$

$S'' \longrightarrow aS / Sb$

Syntax Analyzer | Parser :-

This is 2nd phase of Compiler.

Two Process of generating a string from grammar is and vice versa called Parser.

There are two Types of parser :-

Parser (Syntax Analyzer)

Top down

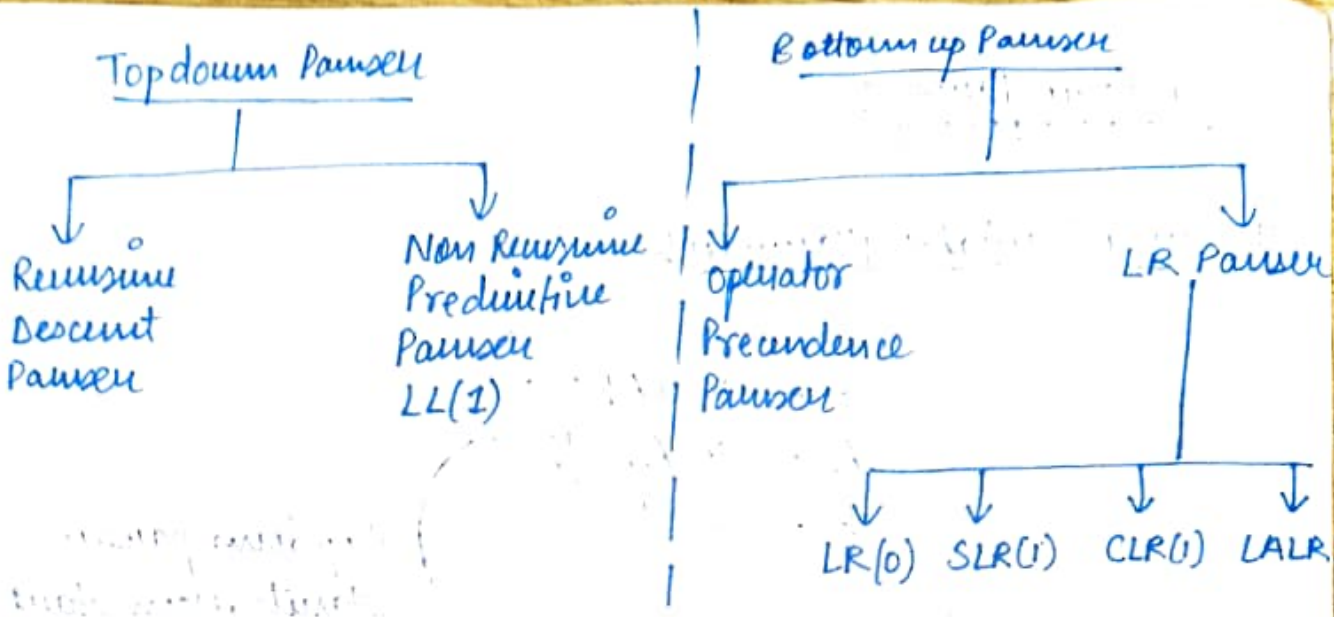
If we Reach start symbol of grammar to the string 'w' it is top down parsing

grammar → string

Bottom up (Shift Reduce parser)

If we Reach start symbol of grammar from string 'w' is bottom up.

grammar ← string



All the parsers have the given grammar → UnAmbigious.

Operator precendence parser also deals with Ambigious grammar only.

If we are using top-down approach to generate a string ⇒ Then the given grammar must be not left Recursive and not Non-deterministic.

LL Parser → L L

L → Start Input taking from Left

L → Derivation used → Leftmost derivation

LR parser →

L → Start from left to Right Reading

L → Derivation used → Right most Derivation in Reverse order due to bottom up.

Top down Pausey :-

given Variable grammar :-

$S \longrightarrow aABe$

$A \longrightarrow Abc/b$

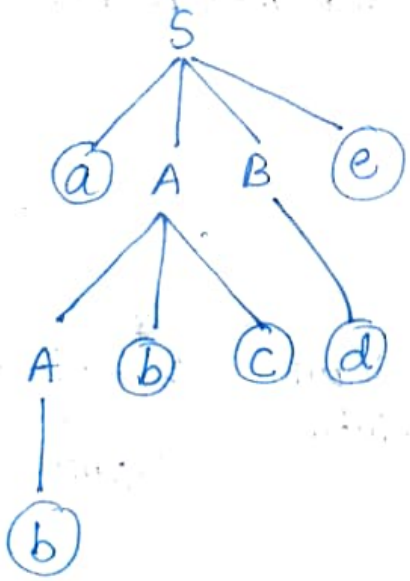
$B \longrightarrow d$

top down pausey starts from start variable 'S' and goes to string.

given string :-

$w = abbcde$

Top-down pausey uses left most derivation :-



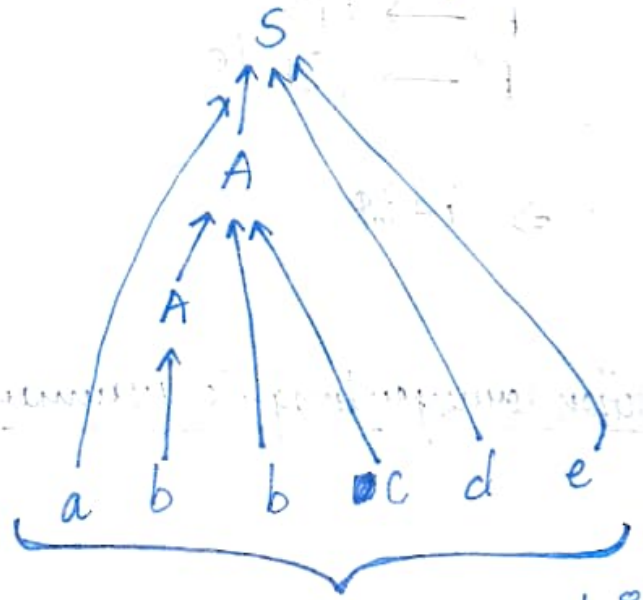
$\therefore$ $w = abbcde$

Bottom up-pausen :-

⇒ given string :- w = abbcde ⇒ grammar ⇒ S → aABe
A → Abc/b
B → d

⇒ In bottom-up pausen, we start with string and goes to start variable of grammar.

→ bottom up pausen uses Reverse order Right most deviation.



Pausen tree is corresponding to Right most deviation but the working of bottom up pausen is from string to start Variable.

Recursive descent Parser :-

It is top down grammar. Parser.
The given grammar must not be ND and LR, and also not Ambiguous.

given grammar :-

$$E \longrightarrow iE'$$
$$E' \longrightarrow +iE'/\varepsilon$$

string to form ⇒ i+i$

Recursive function corresponding to grammar :-

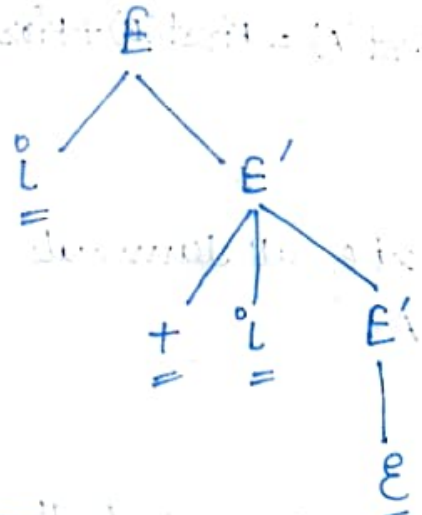
```
# Main(){
        E();
        if (Input==$)
            parsing sucessful;
        }

# E(){
        if (Input==i)
            Input++;
            E'();
        }
```

E'(){

```
if (Input == +){
    Input++;
    if (Input == i){
        Input++;
    }
    E'();
}
Else{
    return;
}
}
```

According to functions :- Parse tree is as followed :-



String formed = i+iε.

First and Follow in Compiler design :-

First :-

⇒ First(A) gives set of terminals that begins in all strings derived from A.

⇒ Rules :-

① If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$

(terminal; α can be variable / terminal)

where, V → Variable
T → Terminal

then ⇒ $\boxed{First(A) = \{a\}}$

② If $A \rightarrow \varepsilon$ then $First(A) = \{\varepsilon\}$

③ If $A \rightarrow BC$ then :-

⇒ $First(A) = First(B)$, if $First(B)$ does not contain ϵ

⇒ $First(A) = First(B) + First(C)$, if $First(B)$ contains ϵ.

Follow :-

⇒ Follow(A) gives set of all terminals that follow immediately to the right of A.

⇒ Rules :-

① If S is start symbol then $Follow(S) = \{\$\}$

② If $A \rightarrow \alpha B\beta$, then $Follow(B) = First(\beta)$, if $First(\beta)$ does not contain ε.

③ If $A \rightarrow \alpha B$, then $Follow(B) = Follow(A)$.

Examples :-

given Context free grammar :- Find First() and Follow() of all Variables present.

$S \longrightarrow aBDh$
$B \longrightarrow cC$
$C \longrightarrow bC \mid \epsilon$
$D \longrightarrow EF$
$E \longrightarrow g \mid \epsilon$
$F \longrightarrow f \mid \epsilon$

Ans

First(S) = {a} According to First Rule - (I)
First(B) = {c} According to First Rule - (I)
First(C) = {b, ε} According to First Rule - (I)
First(E) = {g, ε} " " " " "
First(F) = {f, ε} " " " " "
First(D) = First(E, F) = First(E) ∪ First(F)
= {g, ε, f}
According to First Rule - (III).

Follow(S) = {$}, According to Follow Rule - (I)
Follow(B) = First(D) = {g, f, h}
Follow(C) = Follow(B) = {g, h, f}
Follow(D) = {h}.
Follow(E) = First(F) = {f, h}
Follow(F) = Follow(D) = {h}

follow set cannot Contain any ε Null, First can Contain.

(Q2) grammer :-

$E \longrightarrow TE'$

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

$T \longrightarrow FT'$

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

$F \longrightarrow id | (E)$

(Ans) # Calculating First :-

First(E) = First(T)

:: $E \longrightarrow TE'$

$E \longrightarrow FT'E'$

$E \longrightarrow id | (E)T'E'$ → opening bracket

:: First(F) = {id, (}

:: First(T) = First(F) = {id, (}

:: First(E) = First(T) = {id, (}

First(E') = {ϵ, +}

First(T') = {ϵ, *}

Calculating Follow :-

→ closed bracket

Follow(E) = {$,)}

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

Follow(T) = First(E') = {+, $,)}

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

Follow(F) = First(T') = {*, +, $,)}

[Q39] Given grammar ⇒

$S \longrightarrow Bb/Cd$

$B \longrightarrow aB/\epsilon$

$C \longrightarrow cC/\epsilon$

→ M[E, id]

→ Place the rhs in the stack, pop current top.

[T, id]

[Ans] # Calculating First() :-

First(S) = First(B) ∪ First(C) = {a, b, c, ϵ}

First(B) = {a, ϵ}

First(C) = {c, ϵ}

Calculating Follow() :-

Follow(S) = {$}

Follow(B) = {b}

Follow(C) = {d}

[Q40] $S \rightarrow AaAb/BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

[Ans] # Calculating First() :-

First(A) = {ϵ}

First(B) = {ϵ}

First(S) = First(A) ∪ First(B)
= {a, b}

Calculating Follow() :-

Follow(S) = {$}

Follow(A) = {a, ϵ}

Follow(B) = {b, a}

(Q20) grammer # LL(1) Parser :- Non Recursive Descent parser

It is also called Predictive Parser.

(L)L(1) → Consider 1 symbol at a time.
→ Left most Derivation method
↓
Left to Right traversal

It is a top-down parser.

It uses a stack data structure and LL(1) Parsing Table.

we first need to construct LL(1) Parsing table using LL(1) parsing Table construction Algorithm.

Then we use LL(1) parsing Algorithm and parsing table to make a Parse tree corresponding to the Input.

LL(1) Parsing table construction Algorithm :-

Rules

① Add $A \to \alpha$ under $M[A,a]$ where, $a \in First(\alpha)$

② Add $A \to \alpha$ under $M[A,a]$ where, $Follow(A) \ni a$ if $First(\alpha)$ contain ϵ.

Given Grammar :-

	First()	Follow()
$E \to TE'$	{id, (}	{$,)}
$E' \to \epsilon \mid +TE'$	{ϵ, +}	{$,)}
$T \to FT'$	{id, (}	{+, $,)}
$T' \to \epsilon \mid *FT'$	{ϵ, *}	{+, $,)}
$F \to id \mid (E)$	{id, (}	{+, $,), *}

Count no. of variables in given grammar

⇒ Variables = 5 (V)
⇒ terminal = 5 (T)
⇒ rows = V
⇒ columns = T+1
extra column of $

'=' → M[E, id]
↳ Place the rhs in the stack, pop current top.
[T, id]

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

total cells = (V)(T+1)

∴ For first Production :- E → TE' } Analogy.
A → α }

∴ First(TE') = First(T) = {id, (}

put the Production on M[E, id]
M[E, (]

(II) E' → +TE'

First(+TE') = First(+) = {+} ⇒ M[E', +]

(III) T' → FT'

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

(IV) T' → *FT'

First(*FT') = {*}

(Q2) grammer

(V) $F \rightarrow id / (E)$

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

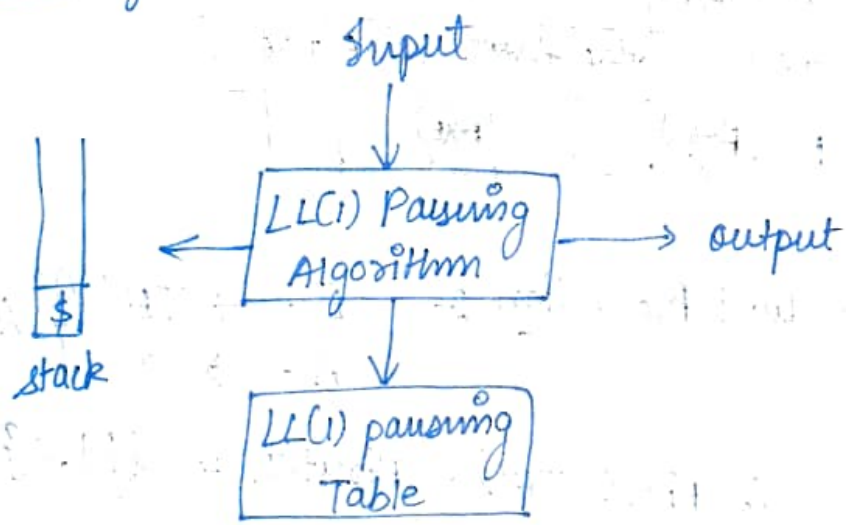
(VI) $E' \rightarrow \epsilon$ (From (II))

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

(VII) $T' \rightarrow \epsilon$ (From (IV))

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

Block diagram of LL(1) Parser :-



---

given input string :-

Stack	Input	production
$, E	id + id $	

By default starting of top down parser

↑ i

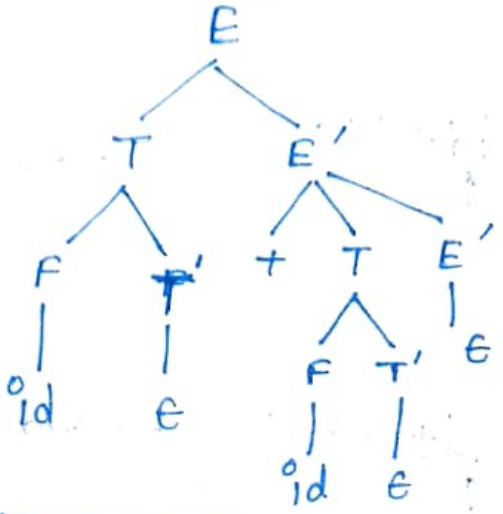
Use the Production placed at M[st.top(), Input[i]]

	Stack	Input	Production
(I)	$ E	id + id $ ↑	E → TE' → M[E, id] → Place the rhs in the stack, pop current top.
(II)	$ E' T	id + id $ ↑	T → FT' → M[T, id]
(III)	$ E' T' F	id + id $ ↑	F → id → M[F, id]
(IV)	$ E' T' id	id + id $ ↑i	Pop()

when top of stack is same as pointer at Input ⇒ Increment the pointer and pop the stack.

(V)	$ E' T'	+ id $ ↑	T' → ϵ → M[T', +] → do not Push ϵ
(VI)	$ E'	+ id $	E' → +TE' → M[E', +]
(VII)	$ E' T +	+ id $	Pop()
(VIII)	$ E' T	id $	T → FT' → M[T, id]
(IX)	$ E' T' F	id $	F → id → M[F, id]
(X)	$ E' T' id	id $	Pop
(XI)	$ E' T'	$	T' → ϵ → M[T', $]
(XII)	$ E'	$	E' → ϵ → M[E', $]
(XIII)	$	$	Pop(); Accept

Forming parse tree using the Table formed. Just follow all the steps and keep forming the Parse Tree.



How to check given Grammar is LL(1) or not :-

⇒ We need to find if the Grammar is LL(1) or LL(1) Parsing table can be produced for the grammar or not.

① If G does not Contain ε :- Apply for Each Production for multiple optional

$$A \rightarrow \alpha_1 | \alpha_2 | \alpha_3$$

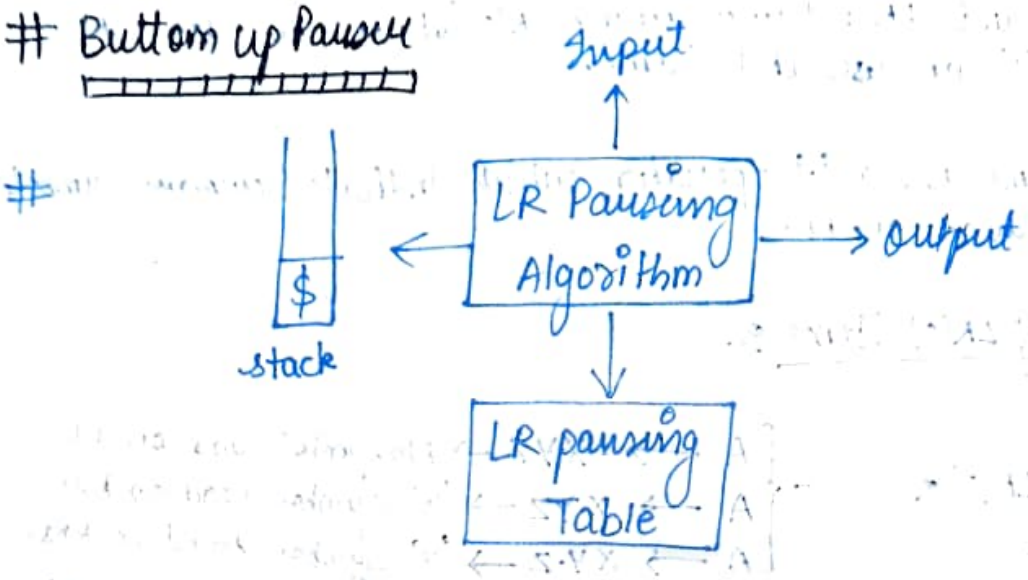
$$\left.\begin{aligned} First(\alpha_1) \cap First(\alpha_2) &= \phi \\ First(\alpha_2) \cap First(\alpha_3) &= \phi \\ First(\alpha_3) \cap First(\alpha_1) &= \phi \end{aligned}\right\} \text{given grammar is LL(1)}$$

② If G does contain ε :- Apply for each production for multiple optional.

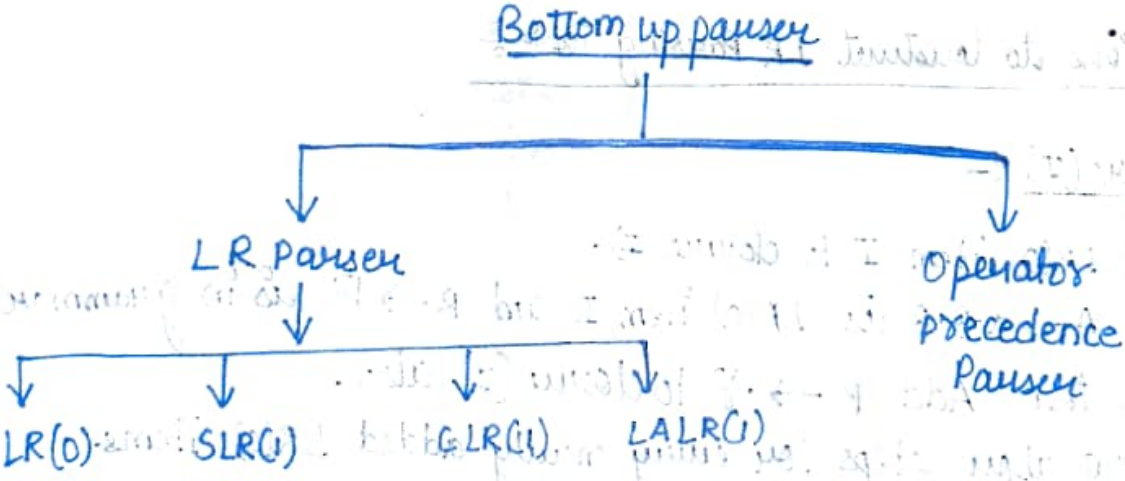
$$A \rightarrow \alpha_1 | \alpha_2 | \epsilon$$

$$\left.\begin{aligned} First(\alpha_1) \cap First(\alpha_2) &= \phi \\ First(\alpha) \cap Follow(A) &= \phi \\ First(\alpha) \cap Follow(A) &= \phi \end{aligned}\right\} \text{given grammar is LL(1)}$$

Bottom up Parser



Types of Parser :-



grammar given is always unambiguous in LR Parser.
grammar given can be ambiguous / unambiguous in operator precedence parser.

Two functions to Construct LR Parsing Table :-

i) closure (I) :-

For LR(0) and SLR(1) Parsers we use LR(0) Items and for CLR(1) and LALR(1) we use LR(1) Items.

LR(0) Items uses a '.' operator which Indicate how many symbols we have read on Rhs.

Working of LR(0) items :-

Non-final LR(0) Items ←
- $A \longrightarrow .XYZ$ → No symbol read on Rhs
- $A \longrightarrow X.YZ$ → 'X' symbol read on Rhs.
- $A \longrightarrow XY.Z$ → 'Y' symbol read on Rhs.

final LR(0) items ←
- $A \longrightarrow XYZ.$ → 'XYZ' symbol read on Rhs. No A can be replaced with 'XYZ'.

Functions to Construct LR Parsing Table :-

(i) Closure(I) :-

1) Add LR(0) item I to closure(I).
2) If $A \rightarrow \alpha.B.\beta$ is LR(0) item I and $B \rightarrow \gamma$ is in grammar G, then Add $B \rightarrow .\gamma$ to closure(I) also.
3) Repeat above steps for every newly added LR(0) items.

∴ given grammar

$S \longrightarrow AA$

$A \longrightarrow aA/b$

∴ Closure(I) = closure($S \rightarrow .AA$) = $S \longrightarrow .AA$

As the closure of Production always Include the Production itself with '.' at the beginning, According to Rule-1

As there is a variable 'A' after '.' in I ⇒ we need to Include all the Production 'A' in closure(I) :-

closure($S \rightarrow .AA$) =
- $S \longrightarrow .AA$
- $A \longrightarrow .aA$
- $A \longrightarrow .b$

Add '.' at the starting of every production.
we also need to find closure of newly added productions like :-

$A \rightarrow .aA \Rightarrow$ closure $(A \rightarrow .aA) = A \rightarrow .aA$

$A \rightarrow .b \Rightarrow$ closure $(A \rightarrow .b) = A \rightarrow .b$

→ **Goto (I, X) :-** I → LR(0) item and X → symbol after '.'.

1) Add LR(0) items I by moving dot after X.
2) Apply closure to the result in step-1.

given grammar ⇒

$S \rightarrow AA$
$A \rightarrow aA/b$

Goto $(S \rightarrow .AA, A)$

Placing '.' after 'A' and finding closure ⇒

closure $(S \rightarrow A.A) = \{S \rightarrow A.A,\ A \rightarrow .aA,\ A \rightarrow .b\}$

LR(0) Parser :-

∴ given grammar :-

$S \rightarrow AA$
$A \rightarrow aA/b$

Create a Parse tree for bottom up parsing corresponding to the grammar :-

LR(0) Parsing Table Construction Algorithm :-

(1) Find Augmented grammar :- Create a variable 'S' which derives the start Variable of grammar and add this extra production in the grammar. $\boxed{S' \rightarrow S}$

(2) I_0 = closure (Augmented LR(0) items). LR(0) augmented item is $S' \to S$ Production.

(3) Apply closure and goto function and find all collection of LR(0) item using Finite Automata (DFA).

(4) Reduce DFA = LR(0) parsing table.

Constructing table :-

given grammar :-

$S' \to S$ → Augmented LR(0) item

$S \to AA$

$S \to aA/b$

Converting $S' \to S$ to LR(0) item $S' \to \cdot S$ and finding its Closure

closure $(S' \to \cdot S) = \left\{ \begin{array}{l} S' \to \cdot S \\ S \to \cdot AA \\ A \to \cdot aA \\ A \to \cdot b \end{array} \right\}$ I_0 state → contains 4 LR(0) items

Considering first LR(0) item :- $S' \to \cdot S \Longrightarrow$ goto() $\Rightarrow S' \to S \cdot$

↓ closure()

$S' \to S \cdot$

I_1 state

Considering first LR(0) item $\Rightarrow S \to \cdot AA \Rightarrow$ goto() $\Rightarrow S \to A \cdot A$

closure $(S \to A \cdot A) = \left\{ \begin{array}{l} S \to A \cdot A \\ A \to \cdot aA \\ A \to \cdot b \end{array} \right\}$ I_2 state

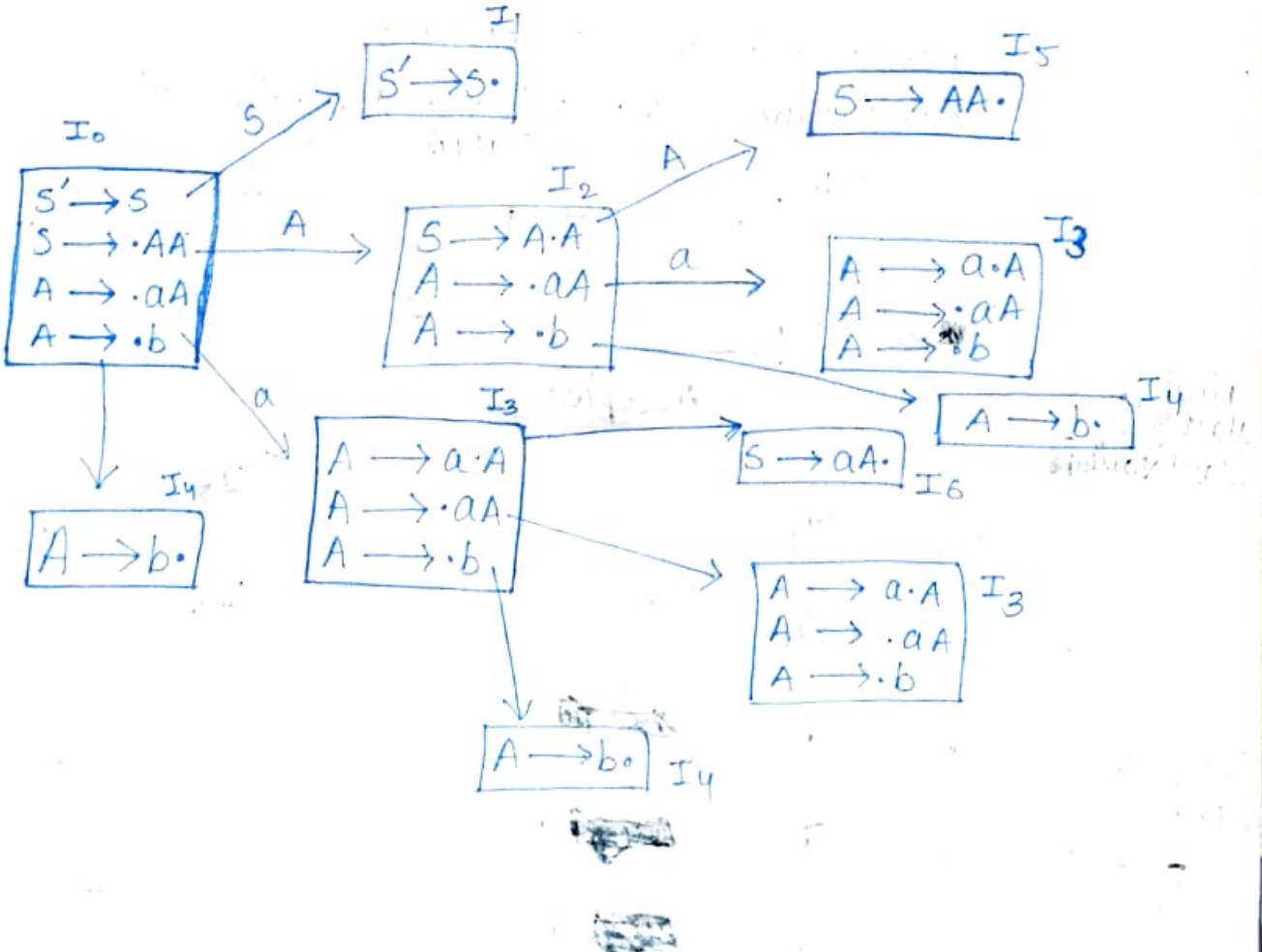
Considering $A \rightarrow \cdot aA \Rightarrow$ goto $\Rightarrow A \rightarrow a \cdot A$

$$\text{Closure}(A \rightarrow a \cdot A) = \begin{bmatrix} A \rightarrow a \cdot A \\ A \rightarrow \cdot aA \\ A \rightarrow \cdot b \end{bmatrix} I_3 \text{ state}$$

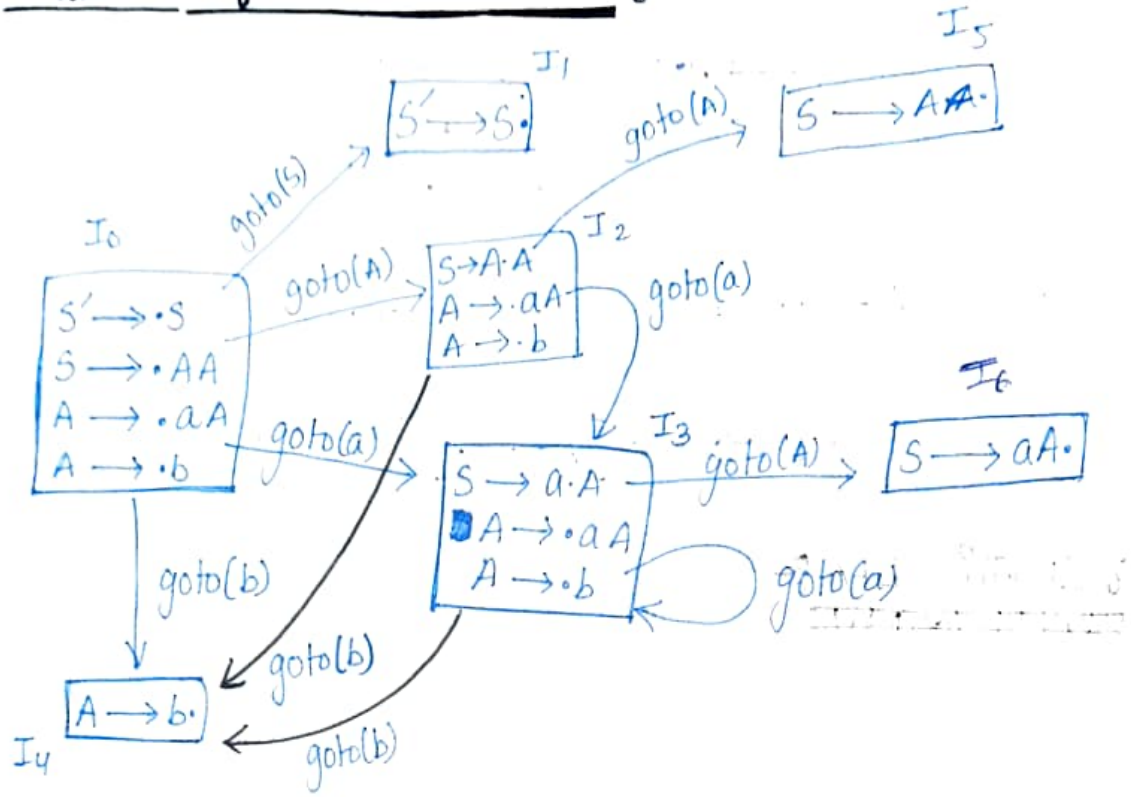
Considering $A \rightarrow \cdot b \Rightarrow$ goto() $\Rightarrow A \rightarrow b \cdot$

$$\text{Closure}(A \rightarrow b \cdot) = \boxed{(A \rightarrow b \cdot)} \; I_4 \text{ state.}$$

Constructing DFA :-



Reduce DFA → fourm minimum state ∴-



LR(0) Parsing Table ∴-

(I) S → AA → r_1 → production → 1
(II) A → aA → r_2 → production - 2
(III) A → b → r_3 → production - 3

No. of Rows in table = No. of states
No. of Columns in table = terminals in action part + Variables in goto

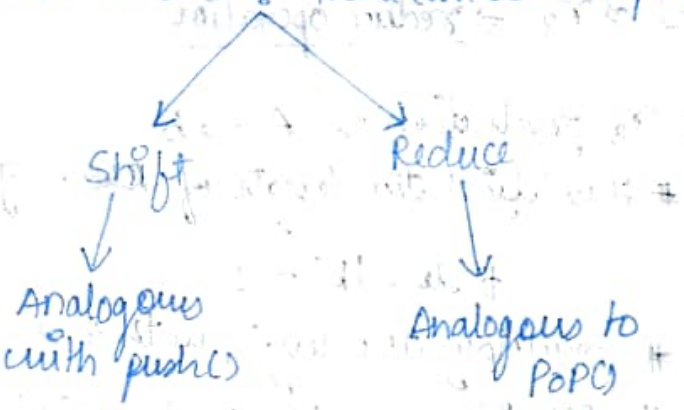
	Actions			goto	
	a	b	$	S	A
0	S_3	S_4		I_1	I_2
1 ← final state for start variable			Accepted		
2	S_3	S_4			I_5
3	S_3	S_4			I_6
4 ← for final state	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

If after the construction of LR(0) table ⇒ If there are no conflicts or there, there are atmost 1 entry in each cell then the table formed is a valid LR(0) Parsing table.

If corresponding to any grammar ⇒ An valid LR(0) Parsing Table is formed then the grammar is LR(0) grammar.

Let Input string:- $S = 'aabb\$'$

stack has a state number = 0 initially
top of the stack is always state number.
Scan the given string from left to Right
Check of $M[st.top()][S[i]] = ?$ there can be two operations

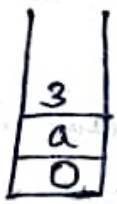


∴ If $M[st.top][S[i]] = S_3$ → Shift state three

$M[0][a] = S_3$

↳ Push current $S[i] = a$ into stack and push the state = 3, as well

Increment the pointer



$S = 'aabb\$'$

i

Now M[3][a] = S_3 ⇒ Push S[i] then state = 3
Increment i

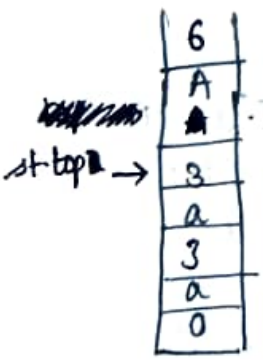
$S = aa\underset{\uparrow}{b}bb\$$

Now M[3][b] = S_4 ⇒ Push S[i] then state = 4
Increment i

$S = aa\underset{\uparrow}{b}b\$$

Now M[4][b] ⇒ r_3 ⇒ reduce operation

r_3 production ⇒ A → b
then find the length of Rhs in grammar
len |b) = 1
multiply the length with 2 ⇒ len = len*2.
pop 'len' number of elements from the stack.
~~# push the lhs back the stack~~.
then check M[~~state~~ lhs][st.top] = M[A][3] = 6
this is the state no. also ~~[illegible]~~.
push the lhs in the stack
then state number.
This actually means that the element/symbol in string which we read in reduce operation the ~~[illegible]~~ previous of this current symbol is reduced to lhs using r_3.



A

$S = aa\underset{\uparrow}{b}b\$$

Now M[6][b] = r_2 :-

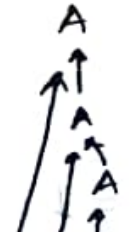
r_2 production ⇒ A → |aA|

len = 2
len = len×2 = 4
pop 4 elements of top.
~~push~~
state = ~~M[ ][ ]~~
M [top] [lhs] ⇒ M [3] [A] = 6
push (lhs)
push (state)

S = aabb$ → we use previously reduced A and 'a' in string to reduce it to A = lhs.
↑
i

Now M[6][b] = r_2 :-

r_2 production ⇒ A → [aA]
len = 2
len = len×2 = 4
pop 4 elements from top.
state M [top] [lhs] = M [0] [A] = 2
push (lhs)
push (state)



S = a a b b $
↑
i

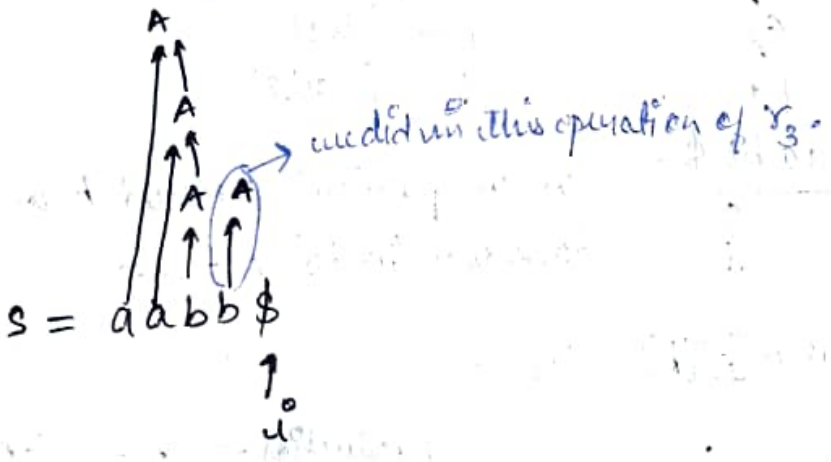
Now M[2][b] = S_4 ⇒ push S[i]=b state = 4
⇒ Increment pointer.

S = aabb$
↑

Now M[4][$] = r_3 ⇒

production r_3 = A → b

len = 1
len = len × 2 = 2
pop 2 elements
state[2][A] = 5
push[A]
push(state)



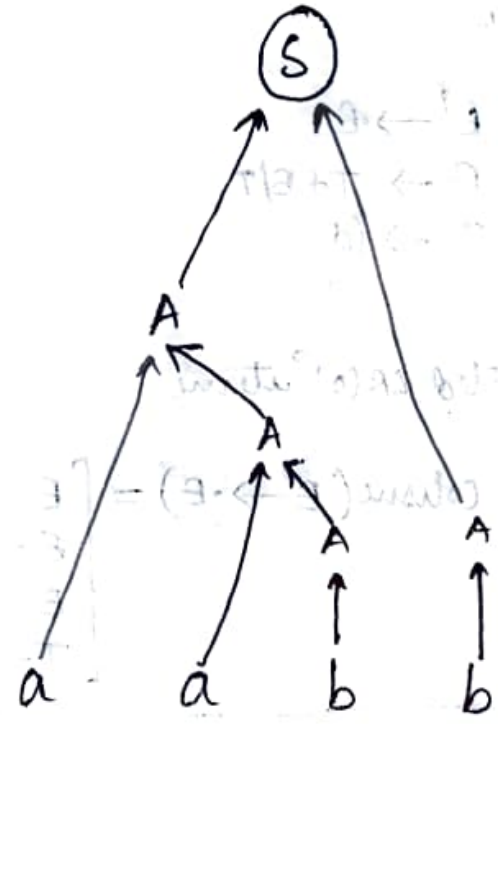
Now M[5][$] = r_1

production r_1 = S → A A

len of Rhs = 2
len = len × 2 = 4
pop top 4 element
lhs = S
state = M[0][S] = 1
push(lhs)
push(state)

Now M[1][$] = Accept

Corresponding parse tree



SLR(1) Parser

given grammer :- $E \to T+E/T$
$T \to id$

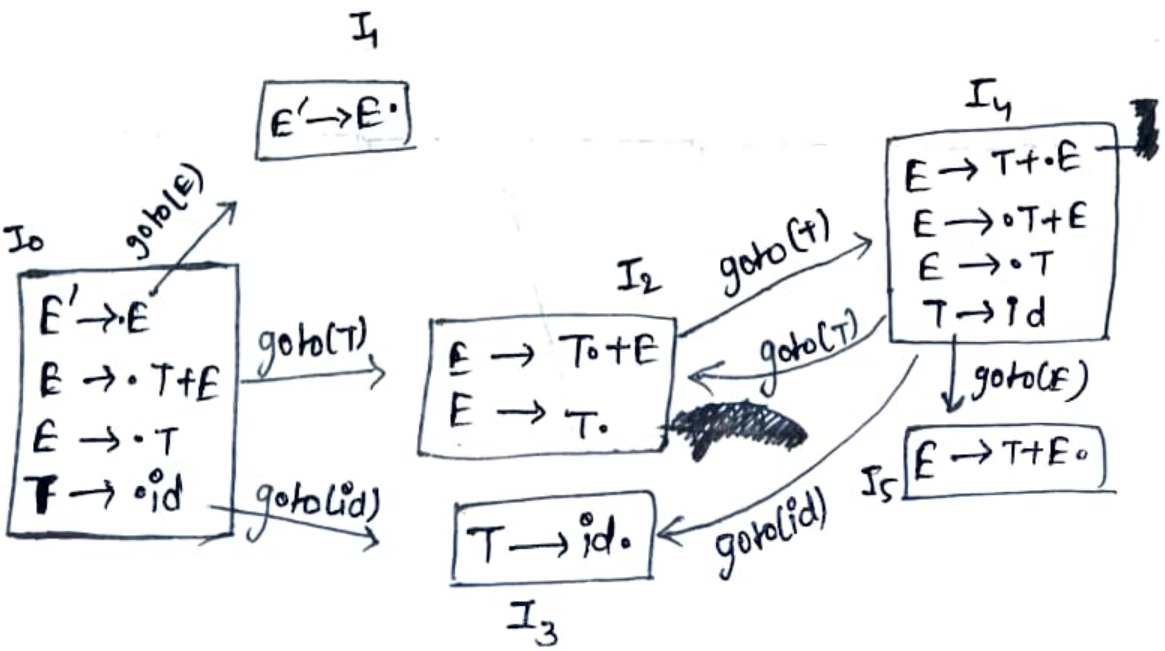
① Add Augmented grammar

$E' \to .E$
$E \to T+E/T$
$T \to id$

② Find closure of Augmented LR(0) items

$$\text{Closure}(E' \to .E) = \begin{bmatrix} E' \to .E \\ E \to .T+E \\ E \to .T \\ T \to .id \end{bmatrix} I_0$$

DFA



SLR Parsing table :-

No. of states = 5
No of V+T = 3T+2V

$E \rightarrow T+E$ (I)
$E \rightarrow T$ (II)
$T \rightarrow id$ (III)

	Action			GoTo	
	id	+	$	E	T
0	S_3			1	2
1			Acc		
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

A conflict has be detected so the table is not LR(0) table and the grammar is not LR(0) grammar.

To construct SLR table ⇒ when we reach the end / final state ⇒ '.' at end ⇒ then we calculate the follow(V) of variable present at the lhs of production.

then r_3 is filled in those element e which

$e \in$ follow(lhs)

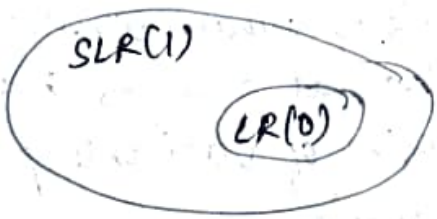
one cell having shift/reduce both is called S-R Conflict.
one cell having reduce/reduce both is called R-R Conflict.

⇒ (I) E → T+E
(II) E → T
(III) T → id

Follow(E) = [$]
Follow [E] = [$]
Follow [T] = [redacted] [$, +]

	Action			goto	
	id	+	$	E	T
0	S_3			1	2
1			Acc		
2		S_4	γ_2		
3		γ_3	γ_3		
4	S_3			5	2
5			γ_1		

This is SLR(1) parsing table



∴ LR(0) is a subset of SLR(1)

Any state with either S-R conflict or R-R conflict is called inadequate state. A state with two final productions using γ_i and γ_j reductions or a state with shift and reduce both happening simultaneously.

Parsing table for SLR is formed using same set of rules/Algorithm as LR(0)

CLR(1) Parser :-

⇒ In CLR(1) parser and LALR(1) Parser we use LR(1) items.

→ LR(1) items = LR(0) items + Lookahead symbols

given grammar :-

$S \longrightarrow AA$
$A \longrightarrow aA/b$

Augmented grammar :-

$S' \longrightarrow S$
$S \longrightarrow AA$
$A \longrightarrow aA/b$

∴ LR(1) of augmented Production ⇒ $S' \longrightarrow \cdot S, \$$

Here "," is not terminal

$S' \longrightarrow \cdot S$: LR(0)
$\$$: Lookahead symbol

Find Closure of LR(1)

closure ($S' \longrightarrow \cdot S, \$$) =

To calculate the lookahead of any production ⇒ check the production variable due to which current production is added then calculate First of the suffix of the variable

$S' \longrightarrow \cdot S, \$$

$S \longrightarrow \cdot AA, \text{First}(\$)$
$\$$

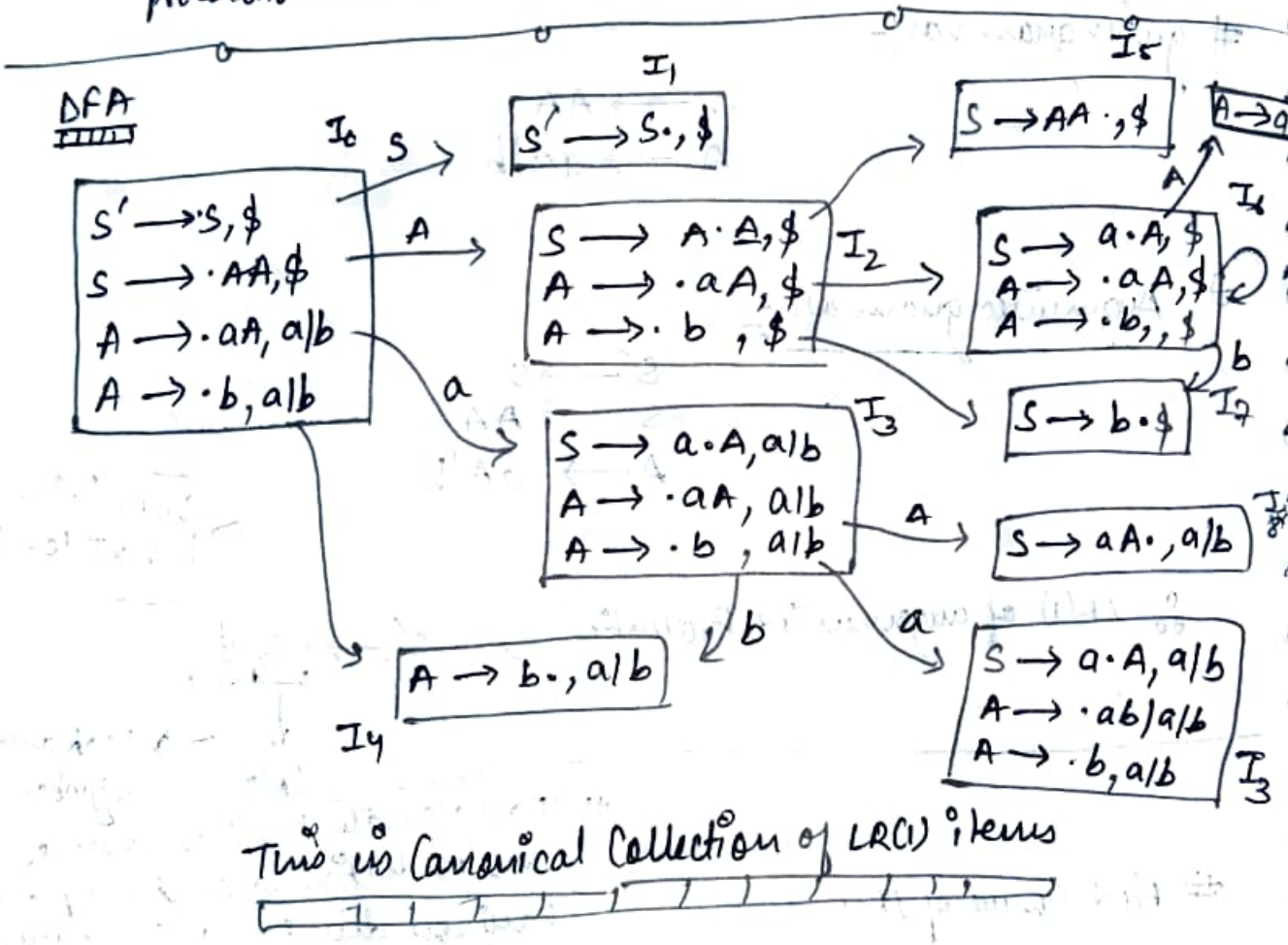
added because of variable A.
lookahead = first(A)
= a.

$A \longrightarrow \cdot aA, a$
$A \longrightarrow \cdot b, a$

I_0

$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot AA, \$$
$A \rightarrow \cdot aA, a/b$
$A \rightarrow \cdot b, a/b$

The lookahead remains same due to goto ()
The lookahead may change when we are finding closure of new production added in closure.



This is Canonical Collection of LR(1) items

CLR Parsing table :-

$S \rightarrow AA \rightarrow (I)$
$S \rightarrow aA \rightarrow (II)$
$S \rightarrow b \rightarrow (III)$

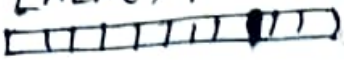
	Actions			Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

This is CLR Parsing Table

here all filling Rule remain same ⇒ only Reduce entry changes.

only when final state is reached ⇒ check from look ahead symbols ⇒ fill the reduce entry of production only at those symbols which appear in look ahead.

LALR(1) Parsers :-



It is a minimized form of CLR(1) Parsers.

In the DFA) canonical Collection of LR(1) items in CLR(1) DFA ⇒ we have to minimed.

↳ Identify the states with same LR(0) items but different lookahead symbols.

↳ In last Example ⇒ I_3 and I_6 must be merged. As they have same LR(0) items

↳ $S \rightarrow a \cdot A$
$A \rightarrow \cdot aA$
$A \rightarrow \cdot b$

and with different look ahead symbols $, a/b.

Merge state = 3 and state = 6 in the CLR(1) parsing table to to form a state new that is (36)

	a	b	$	S	A
36	S_{36}	S_{47}			8,9

Merge state state = 4 and state = 7 in the CLR(1) parsing table to form a state new that 47. that have same LR(0) items ⇒ $A \rightarrow b \cdot , a/b$ and $A \rightarrow b \cdot , \$$ and different lookahead.

	a	b	$	S	A
47	r_3	r_3	r_3		

Merging state = 8, and state = 9 → $A \to aA.,\ a/b$; $A \to aA.,\ \$$

	a	b	$	S	A
89	r_2	r_2	r_2		

LALR Table of Parsing :-

		Actions			Goto	
		a	b	$	S	A
0	0	S_3	S_4		1	2
1	1			Accept		
2	2	S_6	S_7			5
3	36	S_{36}	S_{47}			89
4	47	r_3	r_3	r_3		
5	5			r_1		
	36	S_{36}	S_{47}			89
	47	r_3	r_3	r_3		
6	89	r_2	r_2	r_2		
	89	r_2	r_2	r_2		

If no Cell in the LALR(1) table has multiple entries and All entry are single the the table is a valid LALR(1) Parsing table and no Conflict.

If A valid LALR(1) parsing table can be found then the given grammar is LALR(1) grammar.

Making CLR(1) table and collection of LR(1) items is must before LALR(1)

If there are no common states in CLR(1) canonical collection of LR(1) items and no merging is possible then $\boxed{CLR(1) == LALR(1)}$ same table

LALR(1) is a subset of CLR(1)

No two reductions can be come in same cells.

two shifts can exist in one cells and be merged →

$S_3 + S_6 \rightarrow S_{36}$

$r_3 + r_3 \neq r_{36} \rightarrow r_3/r_3$ → Conflict

If r-r conflict exist ⇒ then the LALR(1) parsing table cannot be formed.

Operator precedence parser :-

⇒ It is a bottom up parser.

⇒ It can handle both ambiguous and unambiguous grammar.

⇒ Grammar used here is operator grammar.

operator grammar :-

⇒ A grammar G does not contain :-

1) Null Production. $(A \rightarrow \epsilon)$ ✗

2) 2-Adjacent Variables on Rhs of production is also not allowed.

$A \rightarrow BC$ ✗

↓ two variables

$A \rightarrow BCD$ ✗

Example of operator grammar

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

Converting Non-operator grammar to operator grammar :-

given Non-operator grammar :-

$S \rightarrow SAS/a$ ← Violation

$A \rightarrow BSb/b$ ← Putting

Putting $A \rightarrow bSb$ and $A \rightarrow b$ in $S \rightarrow SAS/a$

$S \rightarrow SbSbS/SbS/a$

$A \rightarrow bSb/b$

given Non-operator grammar :-

P → SR|S — (1)
R → bSR|bS
S → wbS|w
W → L*W|L
L → id

putting R = bSR in P → SR|S

R = bS in P → SR|S

P → SbSR|bSbS|S
R → bSR|bS
S → wbs|w
W → L*W|L
L → id

putting P in the place of SR from eq (1)

in | P → SbSR|SbS|S
R → bSR|bS

P → SbP|SbS|S
R → bP|bS
S → wbs|w
W → L*W|L
L → id

→ Operator grammar

⇒ If J cannot reach any production from starting variable then the Production is useless/redudant and can be removed.

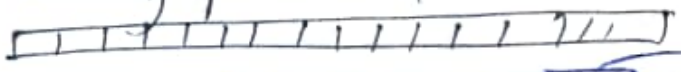
Given Grammar :-

operator grammar
Ambiguous grammar

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

∴ Terminal = (+, *, id) → identifier
→ operator

Creating Operation Relation Table



Right operators

left associativity

left operators

M	id	*	+	$
id	—	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	—

No two identifiers can be compared as they do not Exist Adjacent to Each other.

identifier has greater precedence than operator.

given string ⇒ id + id * id $

we maintain a stack ⇒ stk = | $ |

Initially $ is inserted.

s = id + id * id $
    ↑
    i

check M[st.top()][s[i]] = M[$][id]

If precedence of s[i] is greater then push s[i]
If precendence of st.top() is greater then pop the stack.

precedence of id > precedence of $ ⇒ push

| $ | id |

Increment pointer.

s = id + id * id $
      ↑
      i

check M[id][+]

⇒ precedence [id] > precedence [+]
pop()

→ Every pop is ~~reduce~~ corresponding to a reduction.

E
↑
id + id * id $
↑
i

reduction on pop.

s = id + id * id $
    ↓
    i

| $ |

do not increment pointer

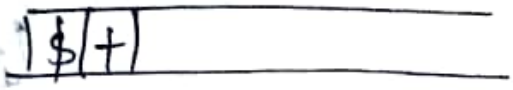
check M[$][+] ⇒ + > $ ⇒ push

S = id + id * id $
    ↑
    i

check M[+][id] ⇒ id > + ⇒ push.

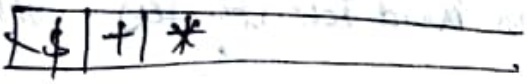
S = id + id * id $
        ↑
        i

check M[id][*] ⇒ id > * ⇒ pop.



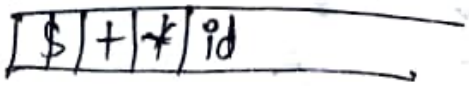
E E
↑ ↑
S = id + id * id $
        ↑
        i

check M[+][*] ⇒ * > + ⇒ push



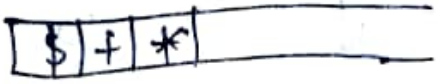
S = id + id * id $
        ↑
        i

check M[*][id] ⇒ push id



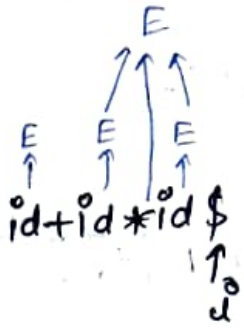
S = id + id * id $
            ↑
            i

check M[id][$] ⇒ pop ⇒ id > $

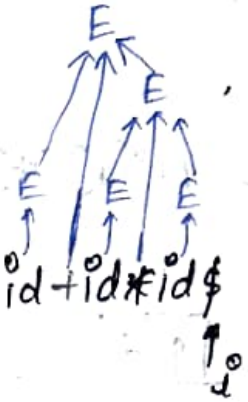


E E E
↑ ↑ ↑
S = id + id * id $
            ↑
            i

check M[*][$] → * > $ → pop.



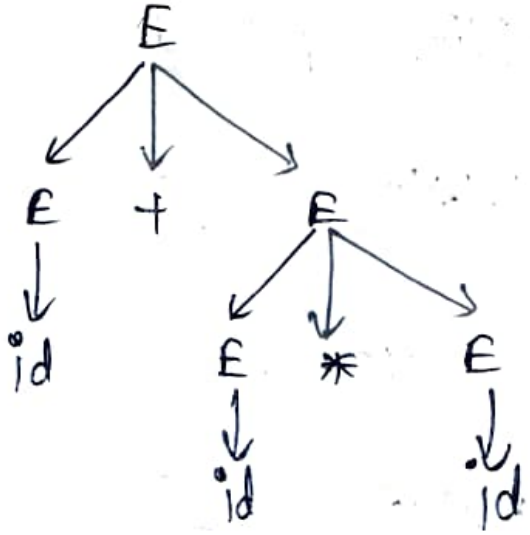
check M[+][$] → + > $ → pop.



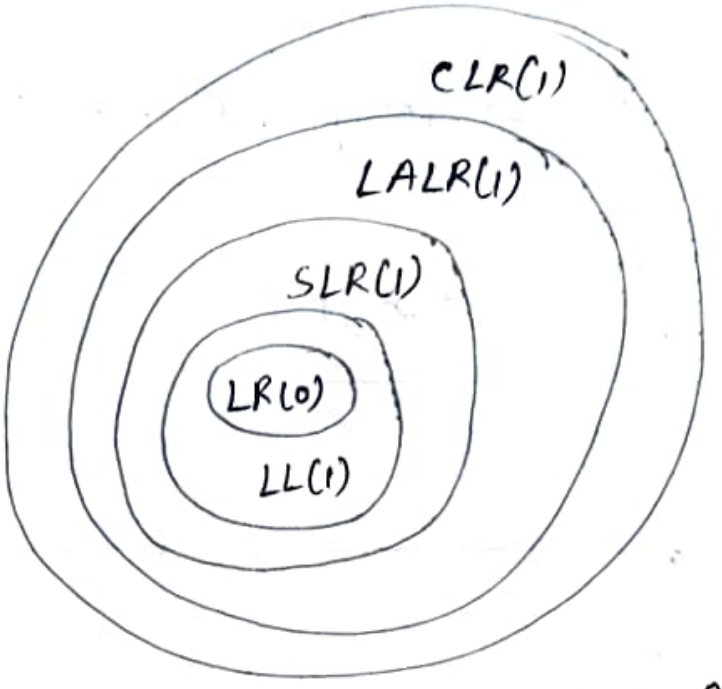
⇒ when S[i] = St.top = $ ⇒ Accept

operator precedence parser can also generate a unique parser tree corresponding to some ambiguous grammar with operators and identifiers with operator precedence table.

Parse tree :-



Relationship of Pausers :-



All pausers which deal with Unambiguous grammar only.

CLR(1) pauser is also known as LR(1) Pausers.

If a grammar is LR(0) then that grammar G will be SLR(1), LALR(1), CLR(1) also.

If a grammar G is SLR(1) then it may or may not be LR(0). Need to be checked.

If a grammar is LL(1) → top down pausers then it definitely will be LALR(1) and CLR(1). Rest it may not or may be SLR(1) or LR(0).

If we have total states in (I) LR(0) is n_1
(II) SLR(1) → n_2
(III) LALR(1) → n_3
(IV) CLR(1) → n_4