Compiler Design

(Unit - I)

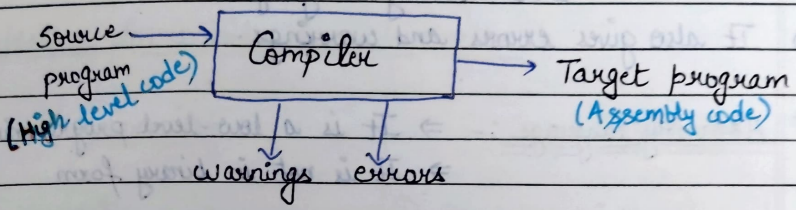
Star

Date

Page No. 1

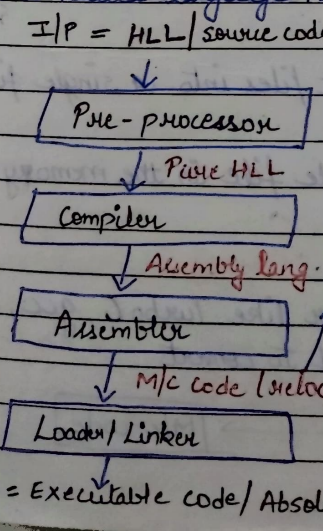
 # Compiler :- It is a software/program that converts a program written in HLL (source language) to a low level lang. (object/Target language).

⇒ It also reports errors present in source programs.



⇒ The main aim of compiler is to convert a high-level lang. (HLL) into a low-level lang. (LLL), then why don't you write your src program into this language (LLL). Reason:- We are not comfortable writing programs in 0's & 1's {Binary language}, rather we can write in English/Normal language.

 # Stages of compiler :- / Language Processing System :- In compiler design, we write the programs in a HLL w/c is easy for us to understand. These programs are fed into a series of tools and OS components to get the desired code that can be used by the m/c. This is called language Processing system.



① Pre-processor :-

⇒ In pre-processing, HLL is converted into pure HLL.

⇒ It removes the preprocessor directives (#include <stdio.h>) and will add the respective files, i.e., File Inclusion.

⇒ It will do macro expansion, operation / operator conversion.

Eg:- a--; —pre-processing→ a=a-1;

② Compiler :- ⇒ compiler during compilation converts pure HLL into assembly language.

⇒ It also gives errors and warnings.

Assembly language :- ⇒ It is a low-level programming language.
⇒ It is not in binary form.

③ Assembler :- ⇒ For every platform (H/W + O.S) we have a assembler

⇒ A assembler for one platform will not work for another platform.

Assembly code —converted into→ Executable M/c code

by assembler ↓ program

Can be loaded at any time & can be run.

④ Loader & Linker :- ⇒ converts relocatable code → absolute m/c code

⇒ A linker links different object files into a single file / executable file, and then

⇒ A loader loads that executable file in the memory and executes it.

Note :- If we talk of any compiler like Turbo C, gcc, etc. all these phases are included to convert

source code → M/c code

Interpreter :- It is also a program that scans the pure high level code line by line and convert it into executable code line by line.

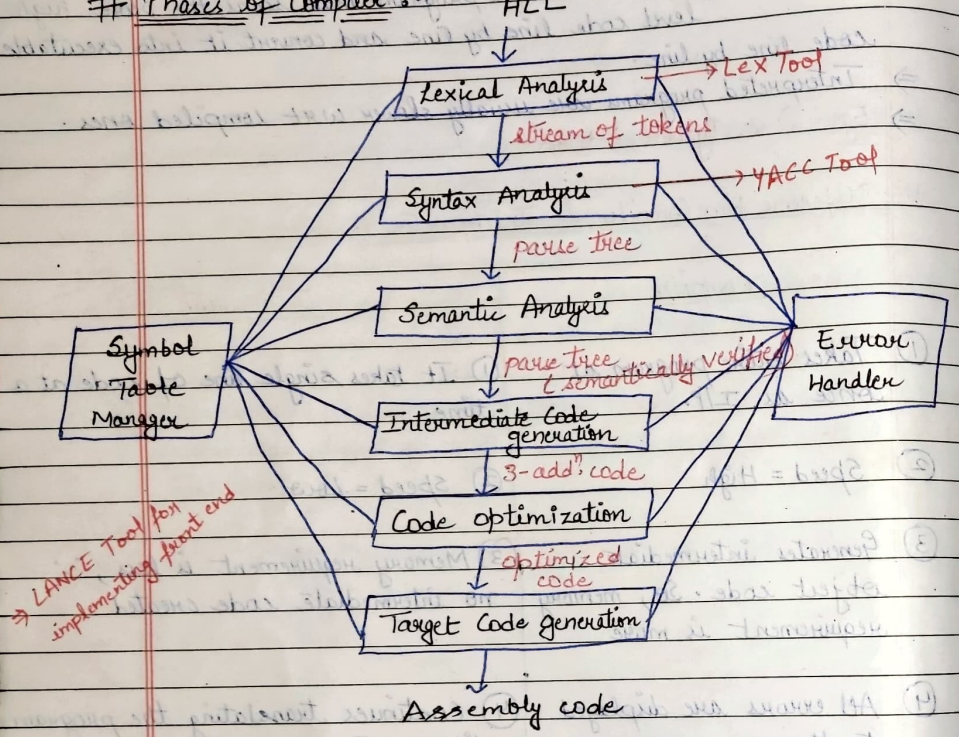
⇒ Interpreted programs are usually slower w.r.t compiled ones.

⇒ Eg:- Perl, Python, Matlab.

Difference b/w Compiler & Interpreter :-

Compiler	Interpreter
① Takes entire program at once as I/P.	① It takes single line of code at a time.
② Speed = High	② Speed = Low.
③ Generates intermediate object code. So, memory requirement is more.	③ Memory requirement is less, ∵ no intermediate code created.
④ All errors are displayed together.	④ Continues translating the program until the 1st error is met, in w/c case it stops.
⑤ Error detection is difficult.	⑤ Error detection is easy.
⑥ Compilers are larger in size.	⑥ Smaller in size.
⑦ C, C++, Scala uses compiler.	⑦ Perl, python, ruby uses an interpreter.

Phases of Compiler :-



Eg:-

$x = a + b * c$; // source code / statements

↓ Lexical analyser → Required I/P is converted into stream of tokens

① Remove whitespaces & comments
② Token pattern consists of $l(l+d)^*$

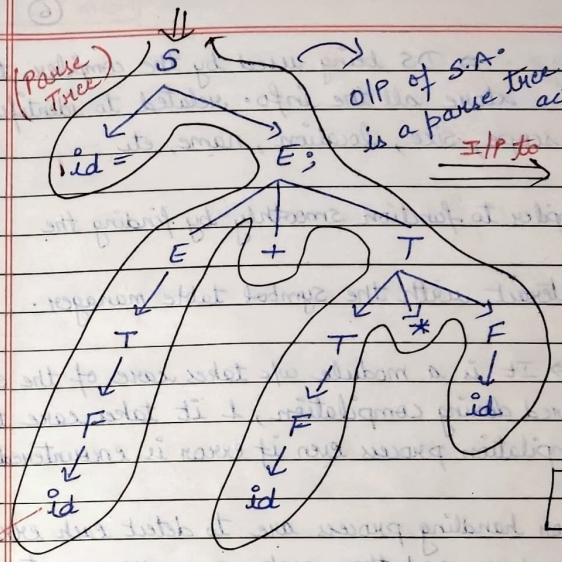
id = id + id * id } stream of tokens {4-add"}

↓ Syntax analyser

context free grammars

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

LHS has to be variable, it cannot be constant, a way name, Ex: n, name



O/P of S.A. is a parse tree w/c act as I/P to → Semantic Analyser → O/P → meaningfully verified parse tree

(Parse tree semantically verified) → I/P to → Intermediate code generation → 3-address code } Most popular ICG

$t_1 = b * c;$
$t_2 = a + t_1;$
$x = t_2;$

↓ Code optimization

$t_1 = b * c;$
$x = t_2 = a + t_1;$

↓ Target Code generation

$R_2 \leftarrow R_1 * R_2$	MUL R_1, R_2	$a \rightarrow R_0$
$R_2 \leftarrow R_0 + R_2$	ADD R_0, R_2	$b \rightarrow R_1$
$X \leftarrow R_2$	MOV R_2, X	$c \rightarrow R_2$

store

⇒ The yield of the parse tree and I/P to the syntax analyser is always same, if not then syntax is wrong.

⇒ 6-phases :-

①	L.A.	Analysis phase	Front End
②	S.A.		
③	Se.A.		
④	ICG		
⑤	CO	Synthesis phase	Back end
⑥	TCG		

Symbol Table Manager :- ⇒ DS being used by the compiler to store all the info. related to identifiers

⇒ Eg :- its types, scope, size, location, name, etc.

⇒ It helps the compiler to function smoothly by finding the identifiers quickly.

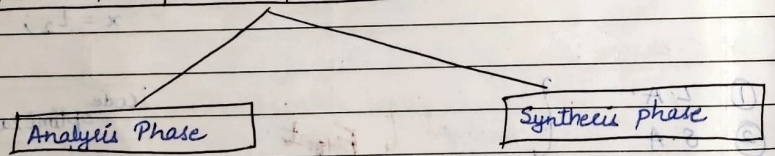
⇒ All the phases interact with the symbol table manager.

Error Handler :- ⇒ It is a module w/c takes care of the error w/c are encountered during compilation, & it takes care to continue the compilation process even if error is encountered.

OR

The task of error handling process are to detect each error, report it to the user and then make some recover strategy and implement them to handle error.

2 parts of compilation process :-



↳ It breaks up the source code/ program into small parts and creates an intermediate representation of the source program.

↳ It takes the intermediate representation of the source program as I/P and creates the desired target code/program.

Lexical Analyzer :- It reads the program and converts it into tokens using a tool called LEX Tool.

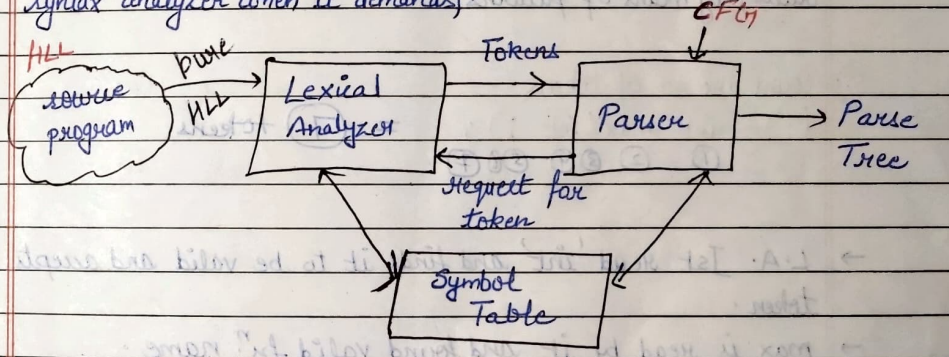
⇒ Tokens are defined by regular expression w/c are understood by the lexical analyzer.

⇒ Removes white spaces, comments, tabs etc.

source program → stream of tokens

⇒ L.A. also called Scanning ⇒ L.A. also helps in macro expansion in the pure HLL code.

Fx^n of L.A. :- ① Reads the I/P program character by character and it produces a stream of tokens (and passes the data to the syntax analyzer when it demands).



② Removing white spaces/ tabs/ newline

③ Remove comments from source program.

④ L.A. generates error and gives the line no. of the error.

Tokens :- A token is a sequence of characters that can be treated as a unit / single logical entity.

⇒ Typical tokens are :- keywords, identifiers, operators, etc.

(for, if, while) (variable name, fx^n name) (+, -, ++, --)

others tokens are constants (literals), special character

separators → , ; etc

Note:- Error msgs include:-
- → Exceeding length
- → unmatched string
- → illegal character

Pattern:- It is a rule describing all those lexemes that can represent a particular token in a source language.

Lexeme:- It is a sequence of characters in the source program that is matched by the pattern for a token.

OR

A sequence of I/P characters that comprises a single token is called a lexeme.

Eg:- "float", "=", "763", ";"

⇒ There are predefined (some) rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of patterns.

Q) Count the no. of tokens:-

(i) `int max(int i);` = (7) tokens
① ② ③ ④ ⑤ ⑥ ⑦

→ L.A. 1st read 'int' and finds it to be valid and accepts as token.
→ max is read by it and found valid fn? name.
→ Lastly, (, int, i,) and ; are all valid tokens.

(ii)
```
int main()
{
    // 2 variables declared below ⟶ ignore them
    int a, b;
    a = 10;
    return 0;
}
```
= (18) tokens

(iii) `printf("Never Give up");` = (5) tokens

(iv) `printf("%d Hello", 4 x);` = (8) tokens

(v)
```
int main()
{
    int a = 10, b = 20;
    printf("sum is %d", a+b);
    return 0;
}
```
= (27) tokens

Tokenization:
① Keyword = int, return
② Identifier = main, a, b, printf
③ Punctuator = (,); { }
④ Operator = , + *
⑤ Constant = 10, 20
⑥ Literal = " "

(vi)
```
int max(x, y)
int x, y;
/* find max of x and y */
{
    return (x > y ? x : y);
}
```
= (25) tokens

(vii)
```
int main)(
{
    x = y + z;
    int x, y, z;
    printf("sum %d %d", x);
(
```
= (26) tokens

(viii)
```
main() {
    a = b (+ +) + (- -) (- -) (+ +) (+ =) =;
    printf("%d %d", a, b);
}
```
= (17) tokens

stop here only, ∵ " " is missing.

(ix)
```
main ()
{
    int a = 10;
    char b = "abc";
    int c = 30;
    char d = "xyz";
in /* comment */ t m = 40.5;
}
```
= (33) tokens

Grammars :- It is basically a set of rules that defines the valid structure of a particular language.

⇒ A grammar consists of 4 components :-

G(V, T, P, S)
- S → start symbol
- P → Productions
- T → Terminals
- V → Variables or non-terminals

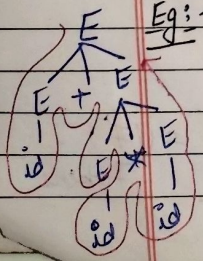
(i) set of terminals ie., that terminate. They are not replaced by any other thing further.

(ii) set of non terminals :- values/variables that are replaced by terminals.

(iii) set of productions, on LHS → non terminal followed by arrows & on RHS we can have T and/or V or combo of both

(iv) Start symbol :- One of the non-terminal is designated as the start symbol from where the production begins.

Eg :- E → E+E / E*E / id



or

P = { E → E+E, E → E*E, E → id } id + id * id

S = E

V = {E}

T = {+, *, id}

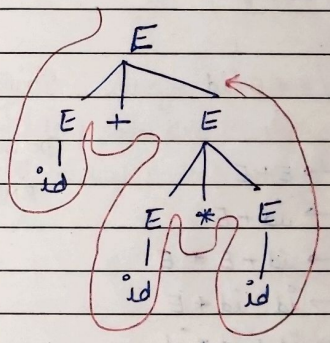
Q. How to derive strings from the grammar?

⇒ Start with the production having the start symbol on the L.H.S.

⇒ Continuously/repeatedly replace all the non-terminals (on the R.H.S.) by the production, for that non-terminal continue until you are only left with terminals in the string.

Note :- All the strings that can be derived from a grammar belong to the language specified by the grammar.

Eg :- E → E+E / E*E / id , Derive :- id + id * id



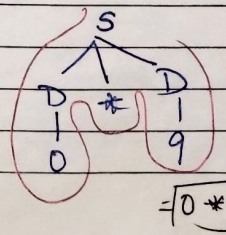
id + id * id

Q. expression with single digits having either * or - in b/w them

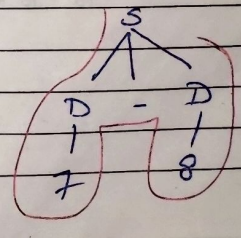
Derive :- ① 0 * 9 ② 7 - 8

S → D*D / D-D / D

D → 0/1/2/3/4/5/6/7/8/9



= 0 * 9



= 7-8

Leftmost Derivation and Rightmost Derivation :-

⇒ Derivation is a sequence of production rules. It is used to get the I/P string through these production rules.

⇒ We have to decide :- ↳ w/c non-terminal to replace.
↳ production rule by w/c the non-terminal will be replaced.

2 options exists

LMD
↳ I/P is scanned & replaced with the production rule from left to right

RMD
↳ I/P is scanned & replaced with the production rule from right to left

Eg :- $E \to E+E/E*E/id$ String :- id + id * id

Derive as LMD ⟹

$E \to E+E$
$\to id+E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

Derive as RMD ⟹

$E \to E+E$
$\to E+E*E$
$\to E+E*id$
$\to E+id*id$
$\to id+id*id$

2nd way :-

Derive as LMD ⟹

$E \to E*E$
$\to E+E*E$
$\to id+E*E$
$\to id+id*E$
$\to id+id*id$

Derive as RMD ⟹

$E \to E*E$
$\to E*id$
$\to E+E*id$
$\to E+id*id$
$\to id+id*id$

Q) $S \to S+S/S-S/a/b/c$ String :- a - b + c

LMD ⟹

$S \to S+S$
$\to S-S+S$
$\to a-S+S$
$\to a-b+S$
$\to a-b+c$

RMD ⟹

$S \to S-S$
$\to S-S+S$
$\to S-S+c$
$\to S-b+c$
$\to a-b+c$

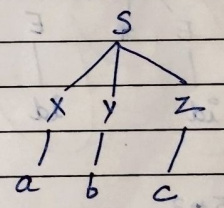
Parse Tree :- It is a typical depiction of how the start symbol of a grammar derives a string in the language.

OR

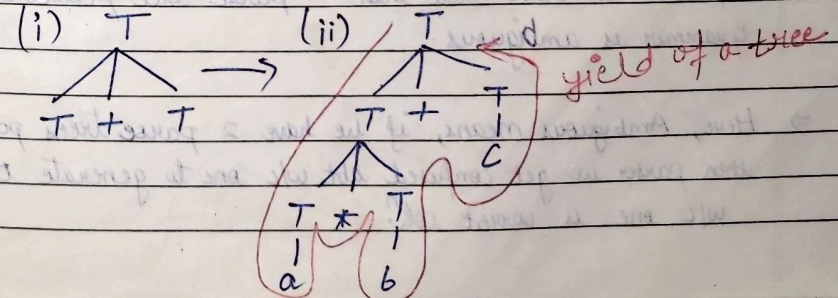
⇒ It is a graphical representation of symbol that can be terminals or non-terminals.

⇒ Properties :- (1) Root is always the start symbol.
(2) All leaf nodes are terminals.
(3) All internal nodes → non-terminals.

Eg :- (1) $S \to XYZ$
$X \to a$
$Y \to b$
$Z \to c/d$



(2) $T \to T+T/T*T/a/b/c$ I/P :- a * b + c



Ambiguous Grammar :- A CFG is said to be ambiguous if there exists more than one derivation trees for the given I/P string.

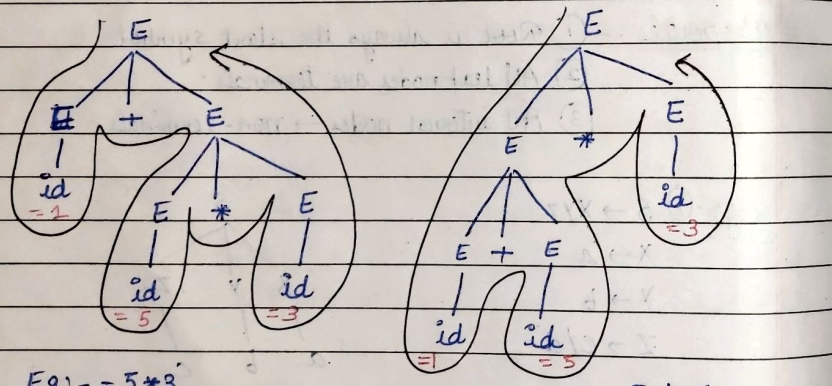
OR

More than one LMD or more than one RMD.

⇒ There is no standard method to check ambiguity. We have to do it by practice / hit & trial method.

⇒ Problem with Ambiguity → precedence of operators is violated / not respected.

Eg:- $E \rightarrow E+E / E*E / id$



Eg:- = 5*3
= 15 + 1 = 16

= 1+5
= 6*3 = 18

⇒ So, since we have more than 1 parse tree possible ∴ Grammar is ambiguous.

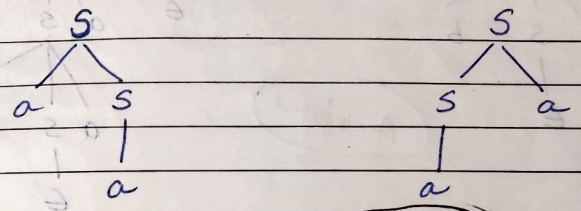
⇒ Here, Ambiguous means, if we have 2 parse trees possible then parser we get confused abt w/c one to generate or w/c one is correct sol^n.

⇒ 1 P.T. = 16 & 2 P.T. = 18

So parsers donot allow ambiguous grammar except operator precedence parser.

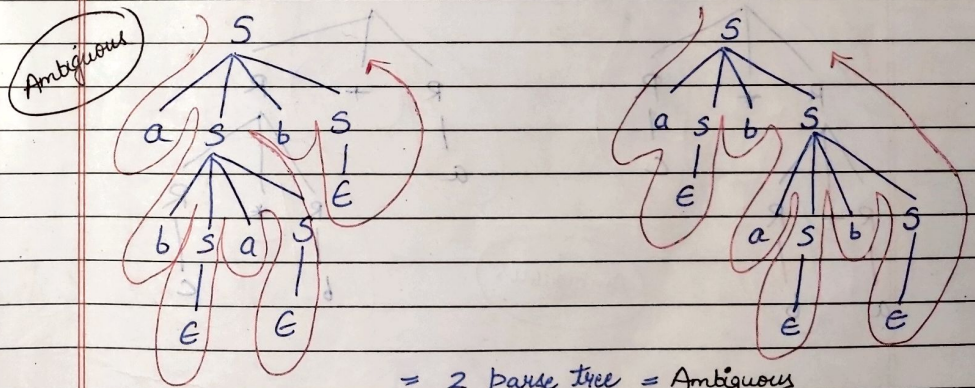
Q) Check whether the grammar is ambiguous or not :-

(i) $S \rightarrow aS/Sa/a$ string = aa



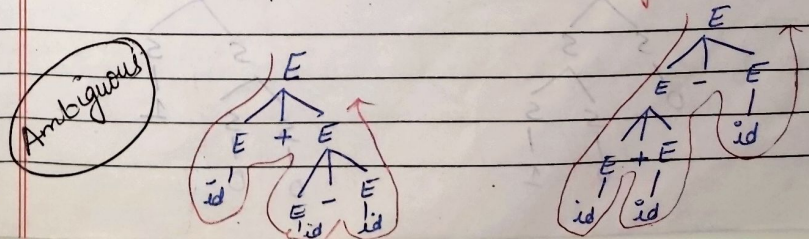
= 2 parse tree = Ambiguous

(ii) $S \rightarrow aSbS / bSaS / \epsilon$ string = abab



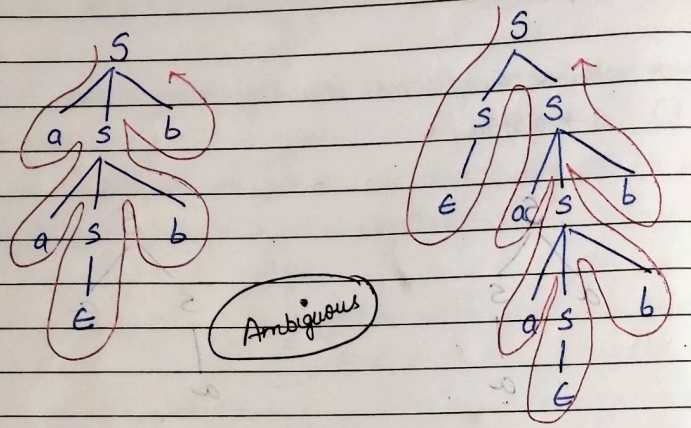
= 2 parse tree = Ambiguous

(iii) $E \rightarrow E+E / E-E / id$ string = id + id - id



(iv) $S \to aSb \mid SS$
$S \to \epsilon$

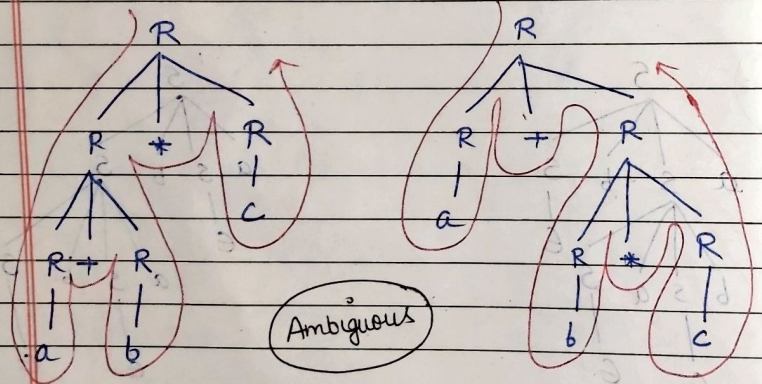
string = aabb



Ambiguous

(v) $R \to R+R \mid R*R \mid R^* \mid a \mid b \mid c$

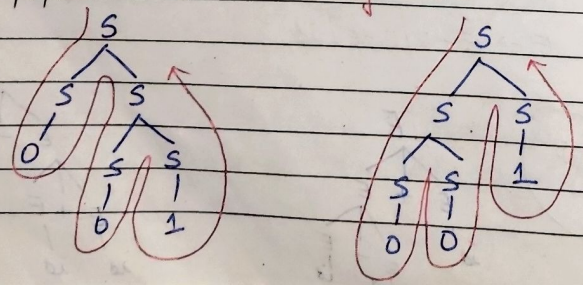
string = a+bc



Ambiguous

(vi) $S \to SS \mid 0 \mid 1$

string = 001

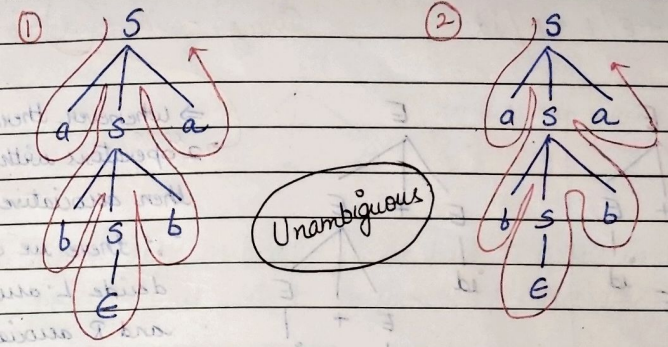


Ambiguous

(vii) $S \to aSa \mid bSb \mid \epsilon$

string = abba, baab

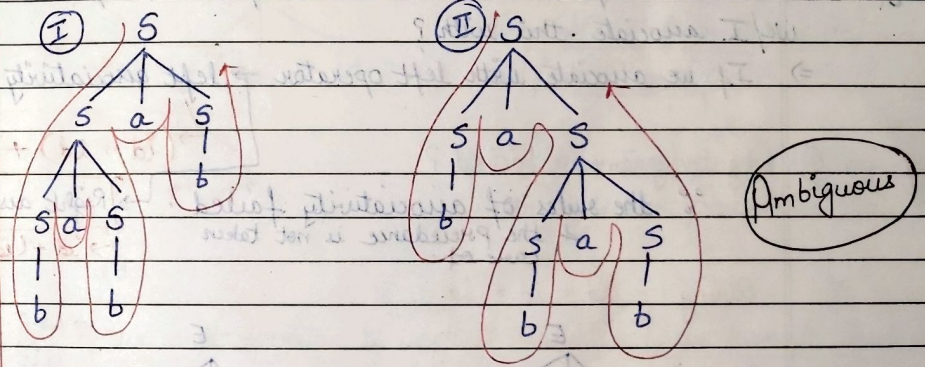
try with this string too.



Unambiguous

(viii) $S \to SaS \mid b$

string = babab



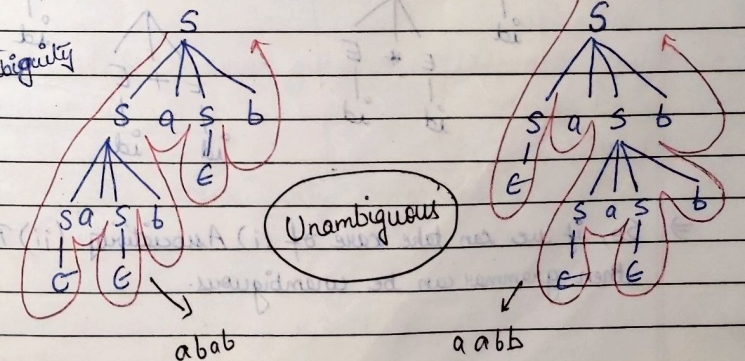
Ambiguous

(ix) $S \to SaSb \mid \epsilon$

string = abab

(x) $S \to AB$
$A \to aA \mid b$
$B \to bB \mid a$
str:- abba

check ambiguity

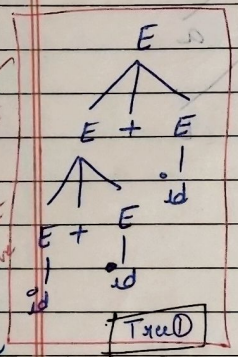


[AG] to [UAG]

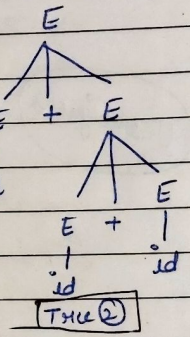
Conversion from Ambiguous to Unambiguous Grammar :-

Eg:- $E \to E+E / E*E / id$

String :- id + id + id



correct ∵ of left associative of + operator



⇒ Whenever there are 2 same operators with operand, then associative fails ∵ there we can't decide L. associative and R. associative of the strings.

⇒ It has 2 operators on either side of it. W/c operator should we/I associate this with?

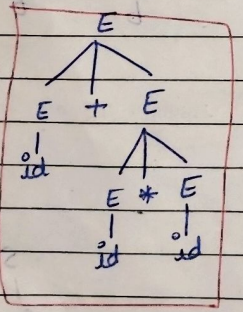
⇒ If we associate with left operator ⇒ left associativity → (id + id) + id

Q) Why the grammar failed?

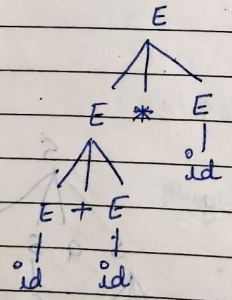
∵ the rules of associativity failed. + the precedance is not taken care of.

→ Right associativity → id + (id + id)

String :- id + id * id



correct ∵ of precedance



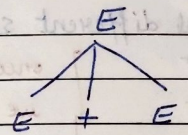
⇒ Now '+' is left associative ∵ Tree① is correct.

⇒ So, if we can take care of (i) Associativity (ii) Precedance then grammar can be unambiguous.

Q) Why are we getting both associativities here?

⇒ ∵ we are defining the Grammar without any order.

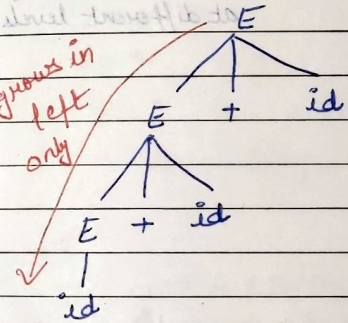
⇒ When we say



so we can grow in any direction.

⇒ To achieve left associativity, we have to grow in left direction only. So, now we will restrict the growth of parse tree.

$E \to E + id / id$



⇒ We are growing only in left direction ∵ Grammar is defined as Left Recursive.

⇒ So, there is no possibility of getting different parse tree.

⇒ So, if we want operator to be left associative, we have to check that whether the grammar is always. Left Recursive

⇒ In short, left recursive grammar is Left most symbol in R.H.S. = L.H.S.

So associativity pblm solved.

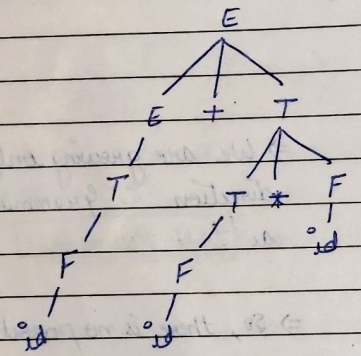
⇒ For precedance pblm :- We should take care that highest precedance operator should be at the least level.

⇒ So we will introduce several different symbols such as :-

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$

} once we have reached T, we cannot generate any +.

⇒ id + id * id derive as,

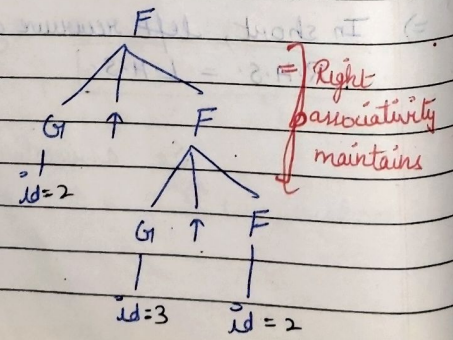


⇒ So, we are taking care of precedance by defining at different levels.

Q) 2 ↑ 3 ↑ 2, convert it into unambiguous grammar.

⇒ $(2 \uparrow (3 \uparrow 2)) = (2^{(3^2)})$ ⇒ Grows right

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to G \uparrow F \mid G$
$G \to id$



} Right associativity maintains

$= (2 \uparrow (3 \uparrow 2))$

Precedance

Q) bExp → bExp OR bExp (boolean expression)
/ bExp AND bExp
/ NOT bExp
/ True
/ False

} Terminals

③ } ② } left associative operator
① → unary operator

$E \to E \text{ OR } F \mid F$
$F \to F \text{ AND } G \mid G$
$G \to \text{NOT } G \mid \text{True} \mid \text{False}$

} unambiguous

Q) R → R+R
/ RR
/ R*
/ a
/ b
/ c

$E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b \mid c$

} unambiguous

Q) Gate 2018

$A \to A \$ B \mid B$
$B \to B \# C \mid C$
$C \to C @ D \mid D$
$D \to d$

Tell associativity and precedance :-

$ > $
# > #
@ > @

⇓ left associative

⇒ Left Recursive grammar

Precedance :- $ < # < @

Q) $E \to E+F / E*F$. Define precedance
$F \to id$

Q) $E \to E*F / F+E / F / F-F / id$

$E \to E+F$
$E \to E*F$
$F \to id$

+ and * → same level ∴ same precedance and both defined as left associative

⇒ suppose if we add $E \to E+F / E*F / F$
$F \to F-id$

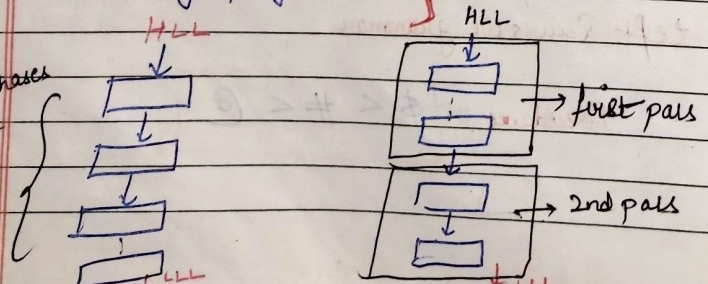
precedance :- * < + < -

Here - is having more precedance than * & +.

Types of Compilers :-

① Single pass compilers :- It is a type of compiler that processes the source code only once.

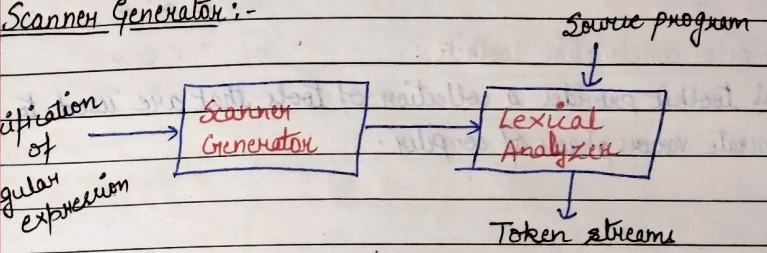
② Multi-pass compiler :- It is a type of compiler that processes the source code multiple times (to convert HLL → low level language) i.e., to convert source code to target/object code.



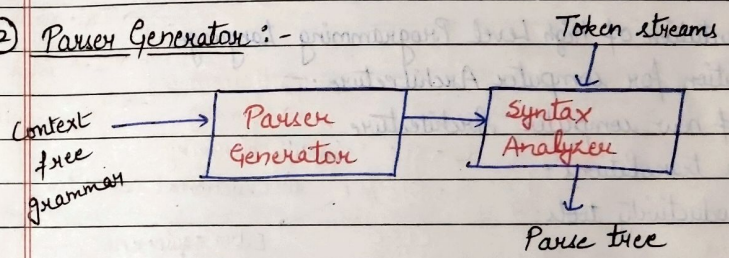
Compiler Construction Tools :-

⇒ These are the specialized s/w tools used to implement various phases of compiler.

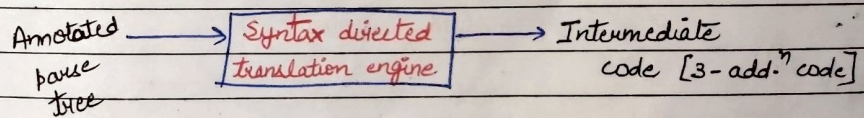
① Scanner Generator :-



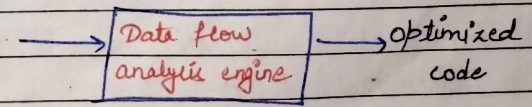
② Parser Generator :-



③ Syntax directed translation engine :-



④ Data flow analysis engine :-



(5) Automatic Code Generator :-

Optimized code → [Automatic code generator] → Machine {for Target Machine}

(6) Compiler construction toolkits :-

⇒ This toolkit provides a collection of tools that are used to generate various phases of compiler.

Application of Compiler Technology :-

1. Implementation of High Level Programming Language.
   - Eg:- C++
   - → Data Abstraction
   - → Inheritance
   - → Polymorphism
2. Optimization for computer Architecture.
   - → Ilelism
   - → Memory
3. Design of new computer Architecture.
   - → SIC {simplified instructional computer}
   - → SIC/XE
   - → CISC
   - → RISC
   - Extra equipment
4. Program translations
5. S/w productivity tools.
   - → Type-checking
   - → Bounds checking
   - → Memory mgmt.
   - → Binary Translation
   - → H/w Synthesis
   - → DB Query Interpreter, Eg:- SQL
   - → Compiled simulation

Gate 2011

Q) The LA for a modern computer language such as java needs the power of w/c one of the following m/c models in a necessary and sufficient sense?

a) Finite state automata
b) Deterministi PDA
c) NDPDA
d) Turning M/C

Lexical Analyzer Implementation :-

x = a + b * c ;

↓

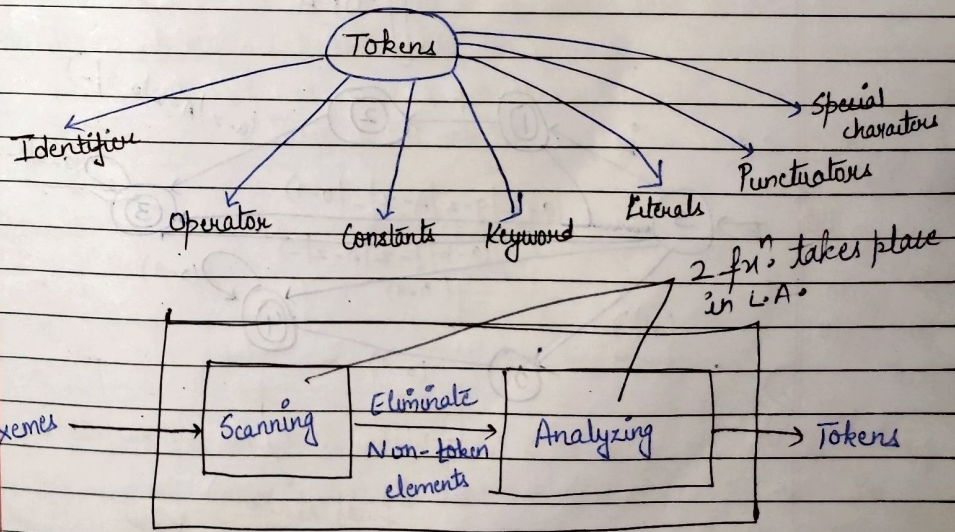
Lexical Analysis → Recognizes tokens using Regexs. Eg:- Regex for identifier :- $l(l+d)^*$

l :- letter
d :- digit
_ :- underscore

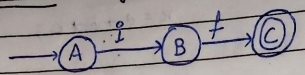
$l(l+d)^* \mid \_(l+d)^*$

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

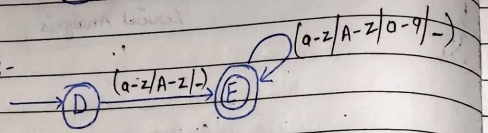
⇒ Scans the Pure HLL code line by line.
⇒ Tokens is produced as O/P and takes Lexemes as I/P.



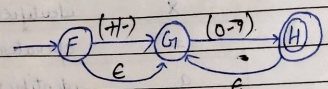
For eg:- C-Tokens :: ① if :-

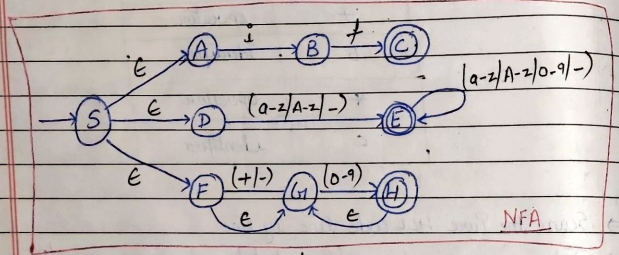


② identifier :-

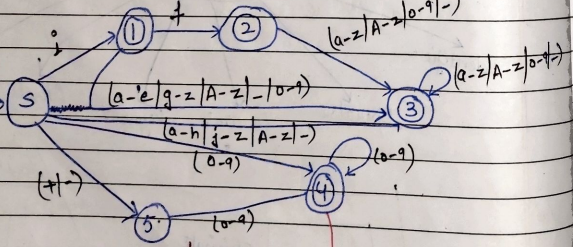


③ integers :-





DFA



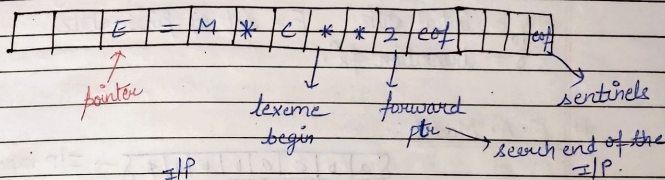
→ Uses DFA for pattern matching

Conclusion :-
final state ① recognizes identifier
final state ② recognizes 'if' keywords

final state ④ recognizes integer

Input Buffering :-

⇒ A buffer contains data that is stored for a short amt. of time, typically in the computer's memory (RAM). The purpose of this is to hold data right before it is used.



I/P

⇒ Scans one character at a time from left to Right.

⇒ LA uses 2 ptr :- bp :- begin ptr
fp :- forward ptr.

⇒ bp and fp set at first character.

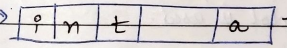
⇒ fp move ahead to find the white space the end of the lexeme.

⇒ After encounter of white space bp and fp are set to next token.

⇒ All process done in secondary storage w/c is costly process ie {scanning process}

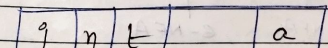
⇒ Buffering Technique :- ① One-buffer scheme ② Two-buffer scheme

Token →

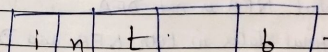


→ length is limited

↓ 2nd-buffer



→ buffer ①



→ buffer ②

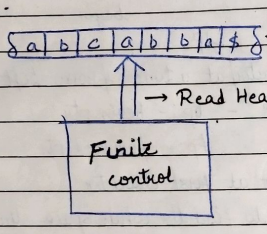
→ improves the speed of buffer

Finite Automata :- It is the simplest m/c w/c is used to recognize patterns.

⇒ It takes the string of symbol as I/P and changes the state accordingly.

⇒ It is a collection of 5-tuple = $(Q, \Sigma, \delta, q_0, F)$, where
Q = set of finite states Σ = set of I/P symbol
q_0 = initial state F = set of final state
δ = transition fn.

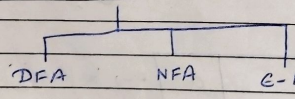
F.A. Model :-



⇒ I/P tape :- It is a linear tape having some no. of cells. Each I/P symbol is placed in each cell.

⇒ Finite ctrl :- The finite ctrl decides the next state on receiving particular I/P from I/P tape. The tape reader reads the cells one by one from left to right, and at a time only one I/P symbol is used.

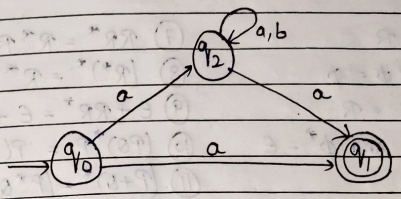
Types of Automata :-



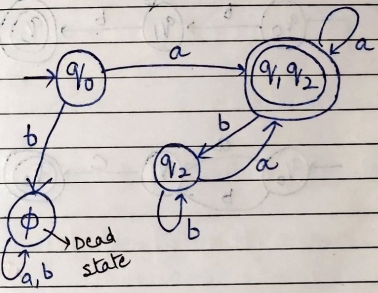
⇒ Every DFA is NFA, but NFA is not DFA.
⇒ There can be multiple final states in both NFA and DFA.
⇒ DFA is used in lexical analyzer in compiler.

Conversion from NFA to DFA / Subset Construction Technique :-

Q)



States	I/P alphabet	
	a	b
$\to q_0$	$q_1 q_2$	$- = \phi$
final state $q_1 q_2$	$q_1 q_2$	q_2
q_2	$q_1 q_2$	q_2



Regular Expression :- The language accepted by finite automata described by simple expression known as R.E.

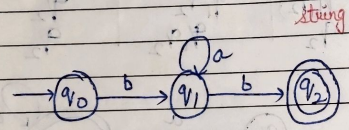
⇒ It is used to describe a sequence of pattern that defines a string.
⇒ It is also used to match character combinations in strings.
⇒ It is used to denote regular languages. ϕ, ϵ, $a \in \Sigma$ are primitive R.E.

Regular Expression Identities :-

1. $\phi + R = R$
2. $\phi R + R\phi = \phi$
3. $\epsilon R = R\epsilon = R$
4. $\epsilon^* = \epsilon$ and $\phi^* = \epsilon$
5. $R + R = R$
6. $R^* R^* = R^*$
7. $RR^* = R^*R$
8. $(R^*)^* = R^*$
9. $\epsilon + RR^* = \epsilon + R^*R = R^*$
10. $(PQ)^* P = P(QP)^*$
11. $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$
12. $(P+Q)R = PR + QR$ and $R(P+Q) = RP + RQ$

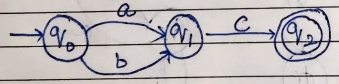
Conversion R.E. to F.A. :-

1. ba^*b



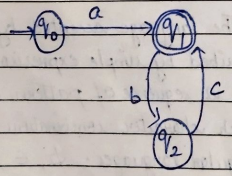
string :- bb, bab, baab, ...

2. $(a+b)c$



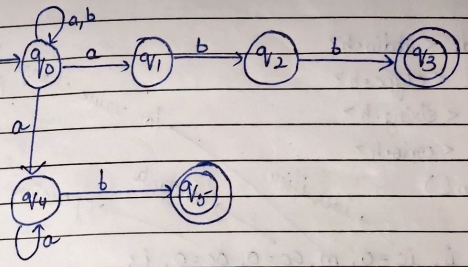
string :- ac, bc

3. $a(bc)^*$

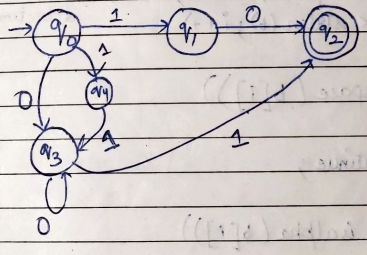


string :- a, abc, abcbc, abcbcbc

4. $(a+b)^*(abb + a^*b)$



5. $10 + (0+11)0^*1$



6. R.E. for string of length exactly 2 over $\Sigma = \{a, b\}$
   $(a+b)(a+b)$

7. R.E. for string of length atmost 2 over $\Sigma = \{a, b\}$
   $(a+b+\epsilon)(a+b+\epsilon)$ / $\epsilon + a + b + (a+b)(a+b)$

8. R.E. for string of even length over $\Sigma = \{a, b\}$ / odd length
   $((a+b)(a+b))^*$ / $((a+b)(a+b))^*(a+b)$

9. Divisible by 3 = $((a+b)(a+b)(a+b))^*$

10. length $\equiv 2 \bmod 3$ = $((a+b)(a+b)(a+b))^*(a+b)(a+b)$

Practical No- 1

Date
Page No. 32

C program for lexical analyzer :-

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <conio.h>
void main()
{
int i, ic=0, m, cc=0, oc=0, j;
char b[30], operators[30], identifiers[30], constants[30];
printf("enter the string : ");
scanf("%[^\n]s", &b);
for(i=0; i<strlen(b); i++)
{
    if(isspace(b[i]))
    {
        continue;
    }
    else if(isalpha(b[i]))
    {
        identifiers[ic] = b[i];
        ic++;
    }
    else if(isdigit(b[i]))
    {
        m = (b[i] - '0');
        i = i+1;
        while(isdigit(b[i]))
        {
            m = m*10 + (b[i] - '0');
            i++;
        }
        i = i-1;
        constants[cc] = m;
        cc++;
    }
    else
    {
        if(b[i] == '*')
        {
            operators[oc] = '*';
            oc++;
        }
        else if(b[i] == '-')
        {
            operators[oc] = '-';
            oc++;
        }
        else if(b[i] == '+')
        {
            operators[oc] = '+';
            oc++;
        }
        else if(b[i] == '=')
        {
            operators[oc] = '=';
            oc++;
        }
    }
}
printf("identifiers : ");
for(j=0; j<ic; j++)
    printf("%c", identifiers[j]);
```

identifier
for value change into ASCII
constants
operator
for space detection
ASCII
90-0 = 57-48 = 9
9*10
48
48

Date
Page No. 33

c=a+90;

```
printf("\n operators : ");
for(j=0; j<oc; j++)
{
    printf("%c", operators[j]);
}

printf("\n constants : ");
for(j=0; j<cc; j++)
{
    printf("%d", constants[j]);
}
```

}

Gate 2018

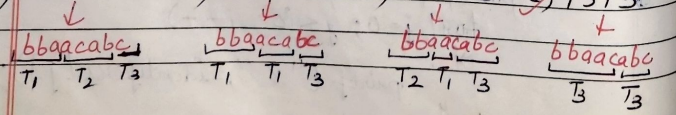
Q) A LA uses the following patterns to recognize 3 tokens T1, T2, & T3 over the alphabet {a, b, c}

T1 : a? (b|c)*a ⟶ (b|c)*a + a(b|c)*a
T2 : b? (a|c)*b ⟶ (a|c)*b + b(a|c)*b
T3 : c? (b|a)*c ⟶ (b|a)*c + c(b|a)*c

Note that 'x?' means 0 or 1 occurrence of the symbol x.
Note also that the analyzer o/p the token that matches the longest possible prefix.

If the string 'bbaacabc' is processed by the analyzer, w/c one of the following is the sequence of tokens its output?

a) T1 T2 T3 b) T1 T1 T3 c) T2 T1 T3 d) T3 T3



Practical No-2

C program to detect tokens :-

```
#include <stdbool.h> → for boolean fxn. → T or F
#include <stdio.h>
#include <string.h> → for I/P as a line of code
#include <stdlib.h>

bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*'
        || ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}')
    return (true);
    return (false);
}

bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' || ch == '/' ||
        ch == '>' || ch == '<' || ch == '=')
    return (true);
    return (false);
}

bool isKeyword(char *str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str, "continue")
        || !strcmp(str, "int") || !strcmp(str, "double") ||
```

```
!strcmp(str, "float") || !strcmp(str, "return") ||
!strcmp(str, "char") || !strcmp(str, "case") ||
!strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") || !strcmp(str, "typedef") ||
!strcmp(str, "switch") || !strcmp(str, "unsigned") ||
!strcmp(str, "void") || !strcmp(str, "static") ||
!strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);
return (false);
}

bool validIdentifier(char* str)
{
    if (str[0] == '0' || str[0] == '1' || str[0] == '2'
|| str[0] == '3' || str[0] == '4' || str[0] == '5' ||
str[0] == '6' || str[0] == '7' || str[0] == '8' ||
str[0] == '9' || isDelimiter(str[0]) == true)
return (false);
return (true);
}

bool isInteger(char * str)
{
    int i, len = strlen(str);
    if (len == 0)
        return (false);
    for(i=0; i<len; i++)
    {
        if (str[i] != '0' && str[i] != '1' &&
str[i] != '2' && str[i] != '3' ... && str[i] != '9'
|| (str[i] == '-' && i > 0))
```



```
        return (false);
    }
return (true);
}

bool isRealNumber(char *str)
{
    int i, len = strlen(str);
    bool hasDecimal = false;
    if (len == 0)
        return (false);
    for(i=0; i<len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
        ... && str[i] != '9' && str[i] != '.' ||
        (str[i] == '-' && i > 0))
        return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

char * subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(sizeof(char) * (right - left
+2));
    for(i = left; i <= right; i++)
        subStr[i - left] = str[i];
        subStr[right - left +1] = '\0';
    return (subStr);
}
```

```
void parse (char* str)
{
    int left = 0, right = 0;
    int len = strlen (str);

    while (right <= len && left <= right)
    {
        if (isDelimiter (str[right]) == false)
            right ++;
        if (isDelimiter (str[right] == true &&
                left == right)
        {
            if (isOperator (str[right]) == true)
            printf ("'%c' is an operator \n",
                    str[right]);
            right ++;
            left = right;
        }
        else if (isDelimiter (str[right] == true &&
                left != right || (right == len &&
left != right))
        {
            char* substr = subString (str, left, right-1);
            if (isKeyword (substr) == true)
                printf ("'%s' is a keyword \n", substr);
            else if (isInteger (substr) == true)
                printf ("'%s' is an integer \n", substr);
            else if (isRealNumber (substr == true)
                printf ("'%s' is a real no \n", substr);
            else if (validIdentifier (substr) == true &&
                    isDelimiter (str[right-1]) == false)
            printf ("'%s' is a valid identifier \n", substr);
```



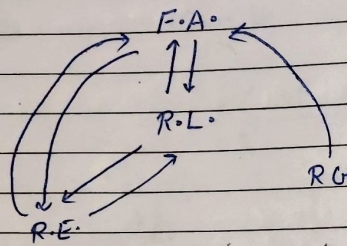
```
            else if (validIdentifier (substr) == false &&
                isDelimiter (str[right-1]) == false)
                printf ("'%s' is not a valid identifier \n", substr);
            left = right;
        }
    }
    return;
}

int main()
{
    char str[100] = "int a = b + 1c;";
    parse (str);
    return 0;
}
```

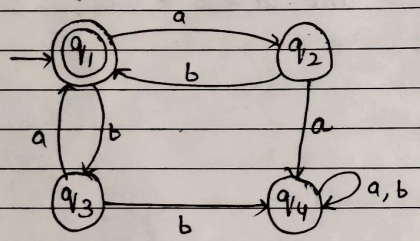
(11) R.E. for string contains atleast 2 a's over $\Sigma = \{a, b\}$

$b^* a b^* a (a+b)^*$

Conversion Finite Automata to R.E. :-



Q1)



null value

$q_1 = \epsilon + bq_2 + aq_3 - (1)$

$q_2 = aq_1 - (2)$

$q_3 = bq_1 - (3)$

$q_4 = aq_2 + bq_3 + (a+b)q_4 - (4)$

re-write

$q_1 = \wedge + q_2 \cdot b + q_3 \cdot a$

Put (2) & (3) in (1)

$q_1 = \wedge + b(aq_1) + a(bq_1)$

$q_1 = \wedge + (ba + ab) q_1$

using Arden's theorem $R = Q + RP$

$\boxed{R = QP^*}$

$q_1 = \wedge + (ab+ba) q_1$

R Q P R

$q_1 = \wedge \cdot (ab+ba)^*$ using $\epsilon \cdot R = R$

$\boxed{q_1 = (ab+ba)^*}$

Similarly, $q_2 = a\, q_1$

$\boxed{q_2 = a(ab+ba)^*}$

$q_3 = b\, q_1$

$\boxed{q_3 = b(ab+ba)^*}$

$q_4 = aa(ab+ba)^* + bb(ab+ba)^* + (a+b) q_4$

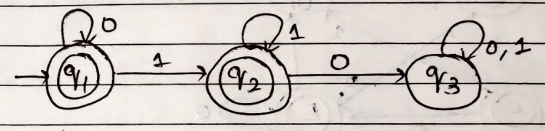
$q_4 = (aa+bb)(ab+ba)^* + (a+b) q_4$

R Q P R

} Arden's theorem

$\boxed{q_4 = (aa+bb)(ab+ba)^* (a+b)^*}$

Q2)



{multiple final states}

$q_1 = 0q_1 + \wedge - (1)$

$q_2 = 1q_1 + 1q_2 + \wedge - (2)$

$q_3 = 0q_2 + (0+1) q_3 - (3)$

$q_1 = \wedge + 0q_1$

R Q P R

} using Arden's

$q_1 = \epsilon \cdot 0^* \Rightarrow \boxed{q_1 = 0^*} \rightarrow (4)$

Put (4) in (2)

$q_2 = 1 q_2 + 1 q_1 + \wedge$

$q_2 = 1 q_2 + 10^* + \wedge$

R P R Q

$\boxed{q_2 = 10^* 1^*} \Rightarrow q_2 = 0^* 1 1^*$

(5)

Now Taking union of eq^n (4) & (5) for final states

h = union of both final states

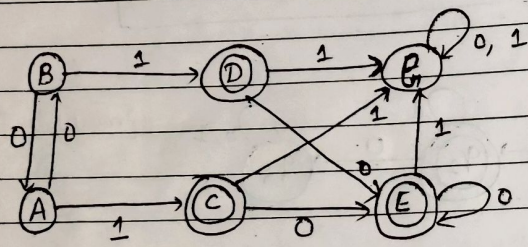
$= 0^* + 0^*11^*$

$= 0^*(\wedge + 11^*)$ $\{\because \wedge + RR^* = R^*\}$

$= 0^*1^*$

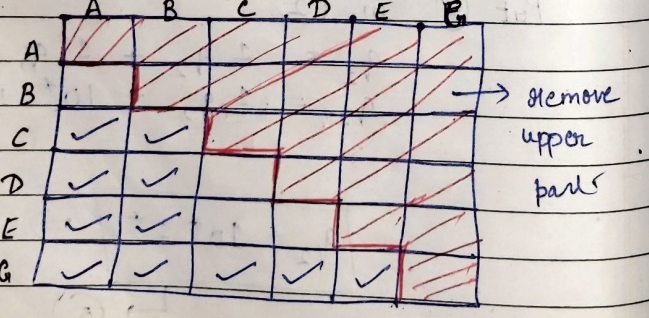
Minimization of DFA :- {Table-filling / Myhill-Nerode Theorem}

Eg:-



Steps:-

1. Draw a table for all pairs of states (P, Q).
2. Mark all pairs where $P \in F$ and $Q \notin F$, F → Final state.
3. If there are any unmarked pairs (P, Q) s.t. $[\delta(P,x), \delta(Q,x)]$ is marked, then mark [P, Q] where 'x' is an I/P symbol. Repeat this until no more markings can be made.
4. Combine all the unmarked pairs and make them a single state in the minimized DFA.



(B, A) → $\delta(B,0) = B$ | $\delta(B,1) = D$
$\delta(A,0) = B$ | $\delta(A,1) = C$ } - check now whether they are marked or not?

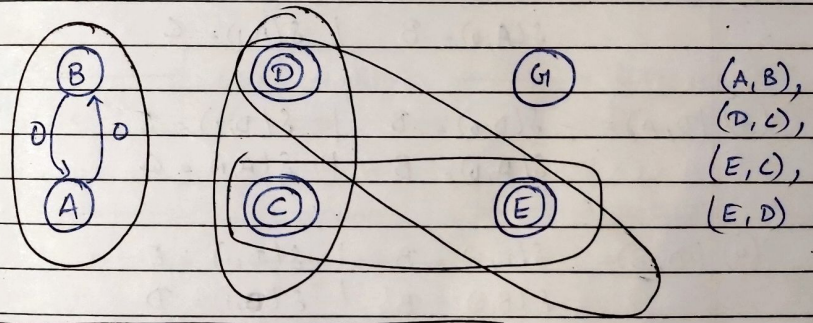
(D, C) → $\delta(D,0) = E$ | $\delta(D,1) = G$
$\delta(C,0) = E$ | $\delta(C,1) = G$

(E, C) → $\delta(E,0) = E$ | $\delta(E,1) = G$
$\delta(C,0) = E$ | $\delta(C,1) = G$

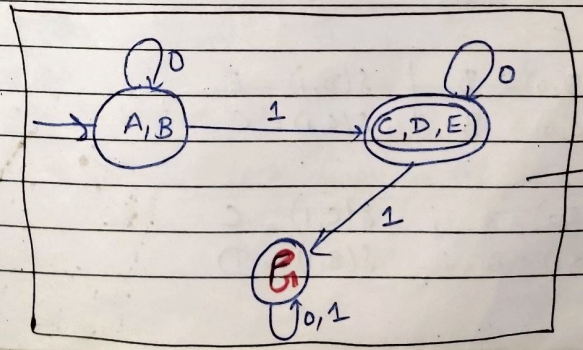
(E, D) → $\delta(E,0) = E$ | $\delta(E,1) = G$
$\delta(D,0) = E$ | $\delta(D,1) = G$

(G, A) → $\delta(G,0) = G$ | $\delta(G,1) = G$
$\delta(A,0) = B$ | $\delta(A,1) = C$

(G, B) → $\delta(G,0) = G$ | $\delta(G,1) = G$
$\delta(B,0) = A$ | $\delta(B,1) = D$

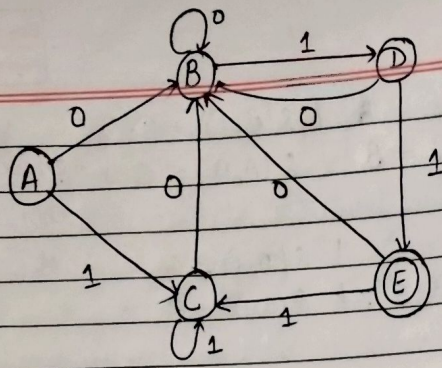


(A, B),
(D, C),
(E, C),
(E, D)



→ final DFA

Q)



	A	B	C	D	E
A					
B	✓				
C		✓			
D	✓	✓	✓		
E	✓	✓	✓	✓	

(1) (B,A) = $\delta(B,0) = B$ | $\delta(B,1) = D$
$\delta(A,1) = B$ | $\delta(A,1) = C$

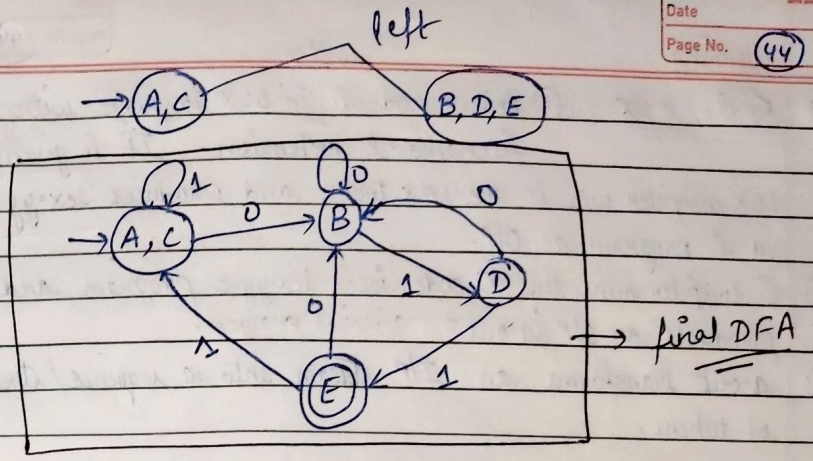
(2) (C,A) = $\delta(C,0) = B$ | $\delta(C,1) = C$
$\delta(A,0) = B$ | $\delta(A,1) = C$

(3) (D,A) = $\delta(D,0) = B$ | $\delta(D,1) = E$
$\delta(A,0) = B$ | $\delta(A,1) = C$

(4) (D,B) = $\delta(D,0) = B$ | $\delta(D,1) = E$
$\delta(B,0) = B$ | $\delta(B,1) = D$

(5) (D,C) = $\delta(D,0) = B$ | $\delta(D,1) = E$
$\delta(C,0) = B$ | $\delta(C,1) = C$

(6) (C,B) = $\delta(C,0) = B$ | $\delta(C,1) = C$
$\delta(B,0) = B$ | $\delta(B,1) = D$

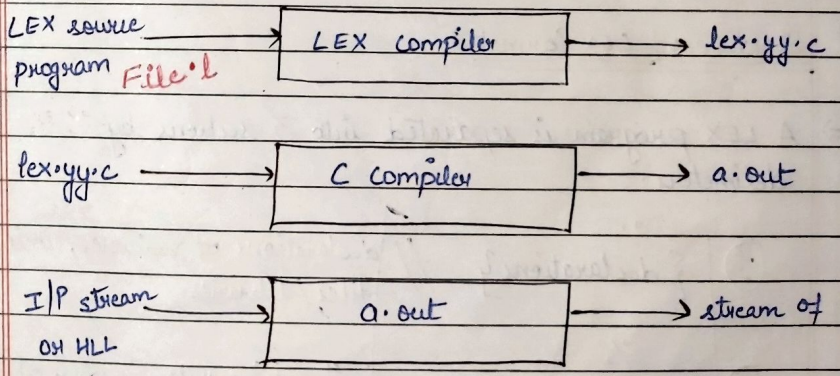


The language specification : LEX Tool :-

⇒ LEX is a tool/computer program w/c generates lexical analyzer. It is used with YACC parser generator.

⇒ LEX written by Mike Lesk and Eric Schmidt and described in 1975.

⇒ LEX compiler :- It will transform the I/P patterns into a transition diagram and generates code in a file called "lex.yy.c"



Block diagram of LEX

Working

Working of LEX :- ① Source code is written in LEX language with filename.l extension. It is given as I/P to LEX compiler w/c is the LEX tool, and produces lex.yy.c a C program as O/P.

② C compiler runs this C code i.e., lex.yy.c program and produces an O/P a.out → lexical analyzer.

③ a.out transforms an I/P stream into a sequence/stream of tokens.

⇒ lex.yy.c → C Program / C language file

⇒ File.l → LEX source program file will always have .l extension.

⇒ a.out → lexical analyzer.

⇒ LEX language is used to program in the LEX tool.

Structure of LEX format :-

⇒ A LEX program is separated into 3 sections by %% delimeters.

I { declarations } // declarations of variables, including files / libraries

%%

II { Translation Rules } // contains rules in form of R.E.

%%

III { Auxiliary fn's }

Eg :-

I % { #include <stdio.h>
int c = 0;
% }
%%

II pattern {action} → Translation rules
%%

III main()
{
}

LEX Rules :-

{Basic} abc → match abc.

[a-z] → match small letter a to z.

[a-z]* → match all the strings in small letters with null or more than 1 character.

[a-z]+ → match all the strings in small letters with 1 or more character.

[A-Z, a-z]+ → match captial & small letters with 1 or more character.

^a → means a should come at start.

a$ → means " " " " end.

[0-9]+ → 1 or more digits.

⇒ LEX fn's :-

yywrap() → called by LEX tool when I/P is exhausted, return 1 if I/P is finished else 0.

yylex() → reads the I/P stream and generate tokens acc. to the R.E. written in rules section.

yytext() → pointer to the I/P string.

Practical No-4

Date
Page No. 46

FLEX program ⇒ Hi → By, otherwise for any I/P show Wrong

(I) section

```
%{
#include <stdio.h>
%}
```

(II)

```
%%
"hi"        {printf("By");}      → Action (will be in C language syntax)
.*          {printf("Wrong");}
%%
```

(III)

```
main()
{
    printf("enter I/P : ");
    yylex();    // Take I/P & generate the Tokens acc. to pattern rule in this section.
}
int yywrap()    // to identify an end of I/P.
{
    return 1;
}
```

Practical No-5

Date
Page No. 47

FLEX program for checking odd or even No :-

```
%{
#include <stdio.h>
/* we can have comments */
int m;
%}
%%
[0-9]+      {                                   → C fn: convert string into integer.
                m = (atoi)(yytext);
                if(m%2==0)
                    printf("even");
                else
                    printf("odd");
            }
.*          {
                printf("Not a no.");
            }
%%
int yywrap()
{
    return 1;
}
int main()
{
    printf("enter the I/P");
    yylex();
    return 0;
}
```

Practical No-6

Date. Page No. 48

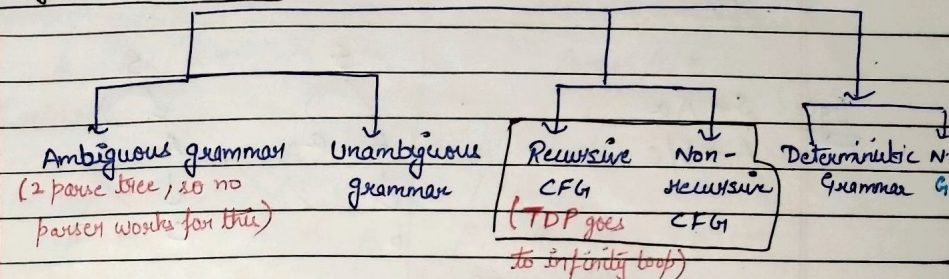
To check operator entered is relational operator or not :-

```
%
{
    #include <stdio.h>
%
}
%%
">" | "<" | "<=" | ">=" | "==" | "!="
{
    printf("relational operator = %s", yytext);
}
.*
{
    printf("wrong");
}
%%
int yywrap
{
    return 1;
}
main()
{
    printf("enter the I/P: ");
    yylex();
}
```

(Unit - II)

Date Page No. 49

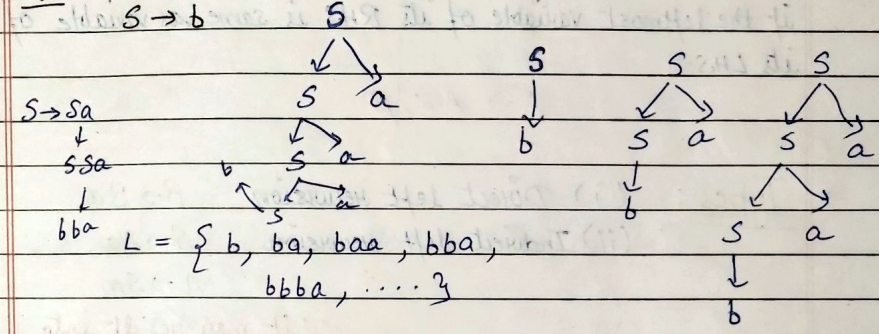
Types of Grammar :-



(I) Recursive CFG: Recursive CFG generates infinite no. of strings.

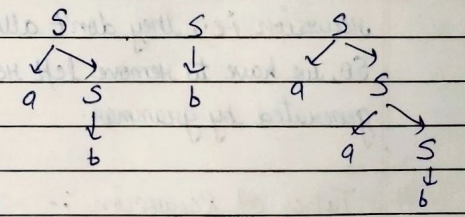
Eg:- (I) $S \to Sa$

$S \to b$



(II) $S \to aS$

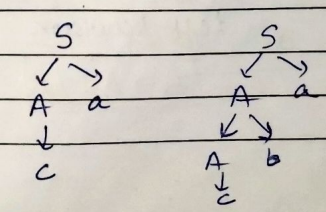
$S \to b$



$L = \{ b, ab, aab, aaab, \ldots \}$

(III) $S \to Aa$

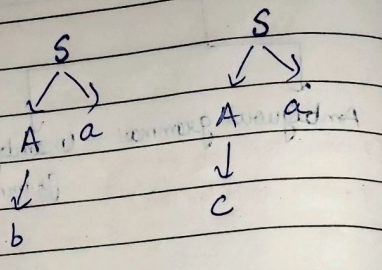
$A \to Ab/c$



$L = \{ ca, cba,$

(2) Non-Recursive:- Non-recursive CFG generates finite no. of strings.
CFG

Eg:- $S \to Aa$
$A \to b/c$



$L = \{ba, ca\}$

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

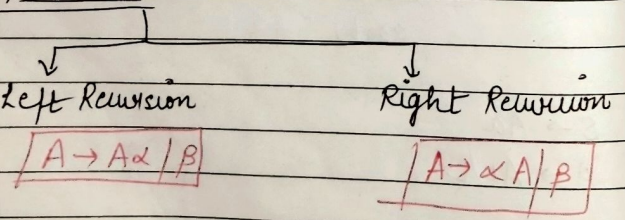
$A \to A\alpha/\beta$

⇒ Types :- (i) Direct left recursion $B \to Ba$
(ii) Indirect left recursion $S \to Aa$
$A \to Sm$

";" it may result into infinite loop.

Why to Remove left recursion ?
⇒ Top down parsers cannot accept the grammar having left recursion i.e., they don't allow left recursive grammar. So, we have to remove left recursion but preserve the language generated by grammar.

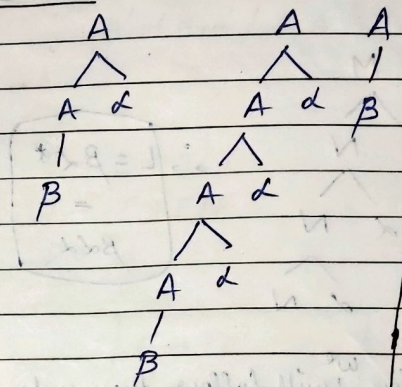
Types of Recursion :-



LR

$A \to A\alpha/\beta$

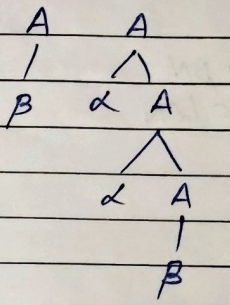
Problem :-



$L = \beta\alpha^*$

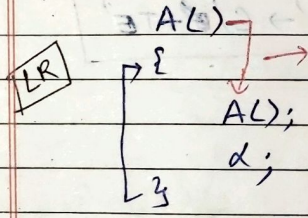
RR

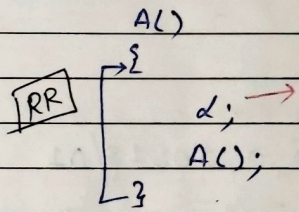
$A \to \alpha A/\beta$



$L = \alpha^*\beta$

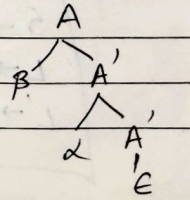
⇒ If we write above problem in form of fn" :-





Conversion from LR to RR :-

Eg:- $A \to A\alpha/\beta \implies L = \beta\alpha^*$



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

A will generate β followed by A'.
Now A' will generate α^*.

∴ $L = \beta\alpha^*$ Remains same.

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

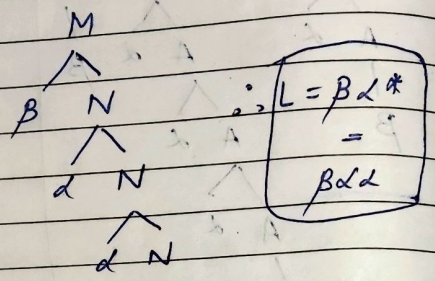
$\Longleftrightarrow$

$A \to \alpha A / \beta$

Right Recursion grammar ≡ Left Recursive grammar

Eg:-

$M \to \beta N$
$N \to \epsilon / \alpha N$



$\therefore L = \beta\alpha^*$
$= \beta\alpha\alpha$

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

Eg:- ① $E \to E+T/T$, remove LR from this grammar.

compare

$A \to A \alpha / \beta$

⇒
$E \to TE'$
$E' \to \epsilon / +TE'$

② $S \to S0S1S/01$

A A α β

⇒
$S \to 01S'$
$S' \to \epsilon / 0S1SS'$

③ $S \to (L)/x$ → No LR
$L \to L, S/S$

A A α β

$S \to (L)/x$
$L \to SL'$
$L' \to \epsilon / ,SL'$

④ $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \cdots$
$/ \beta_1 / \beta_2 / \beta_3 / \cdots$

$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' / \cdots$
$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \epsilon \cdots$

⑤ $E \to E+T/T$
$T \to T*F/F$
$F \to id$

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

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

$\therefore$

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

⑥ $S \to (L)/a$
$L \to L, S/S$

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

⑦ $S \to Aa$
$A \to Sb/c$

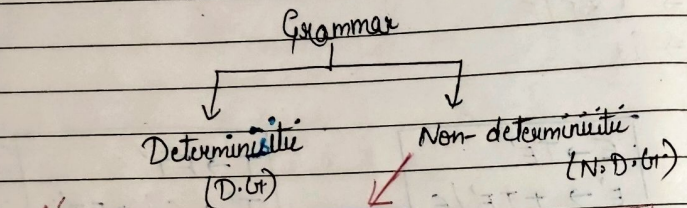
$A \to Aab/c$

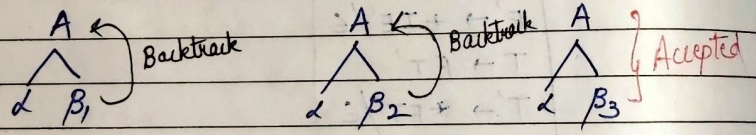
$A \to cA'$
$A' \to abA'/\epsilon$

Left Factoring :-

⇒ Sometimes, it is not clear w/c production to choose to expand a non-terminal ∵ multiple production begins with the same (terminal/non-terminal) / lookahead.

⇒ This type of Grammar is Non-Deterministic or Grammar containing left factoring.



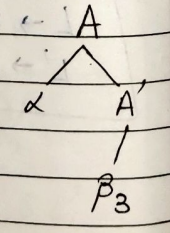


⇒ Backtracking is happening ∵ of the common prefixes, i.e., one or more productions on the RHS are having something common in the prefixes. This is also called common prefix prblm or NDGr.

⇒ So, we convert NDGr into DGr such as :-

A → αA'
A' → β1 | β2 | β3 ...

↓ D.Gr.



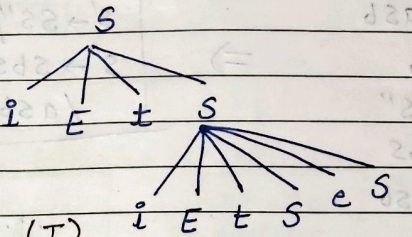
Eg :- (I) S → iEtS / iEtSeS / a
E → b

common prefix

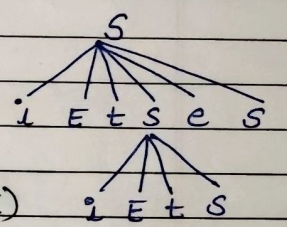
(II) S → iEtSS' / a
S' → eS / ∈
E → b

Check Ambiguity, W = iEtiEtSeS

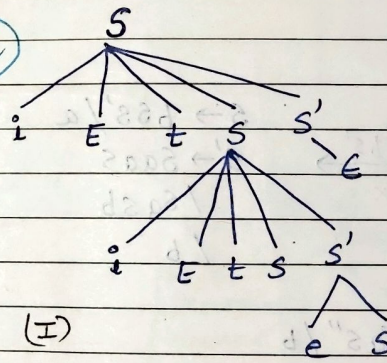
(I)



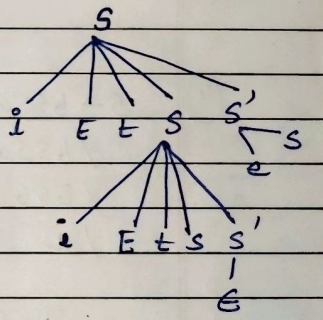
(II)



(I)



(II)



** Note :- Eliminating N.D.Gr or applying left factoring does not remove ambiguity.

(2) $S \to aSSbS$ / $aSaSb$ / abb / b

with 'a' prefix →

$S \to aS' / b$
$S' \to SSbS / SaSb / bb$

with 'S' prefix →

$S' \to SS''$
$S'' \to SbS / aSb$

$S \to aS' / b$
$S' \to SSbS$ / $SaSb$ / bb / $S'' \to SS''$
$S'' \to SbS$ / aSb

$\Rightarrow$

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

D.U.

(3) $A \to aA$
$B \to aB$

} X Not N.D.by.

(4) $S \to bSSaaS$ / $bSSaSb$ / bSb / a

with 'bS' prefix →

$S \to bSS' / a$
$S' \to SaaS$ / $SaSb$ / b

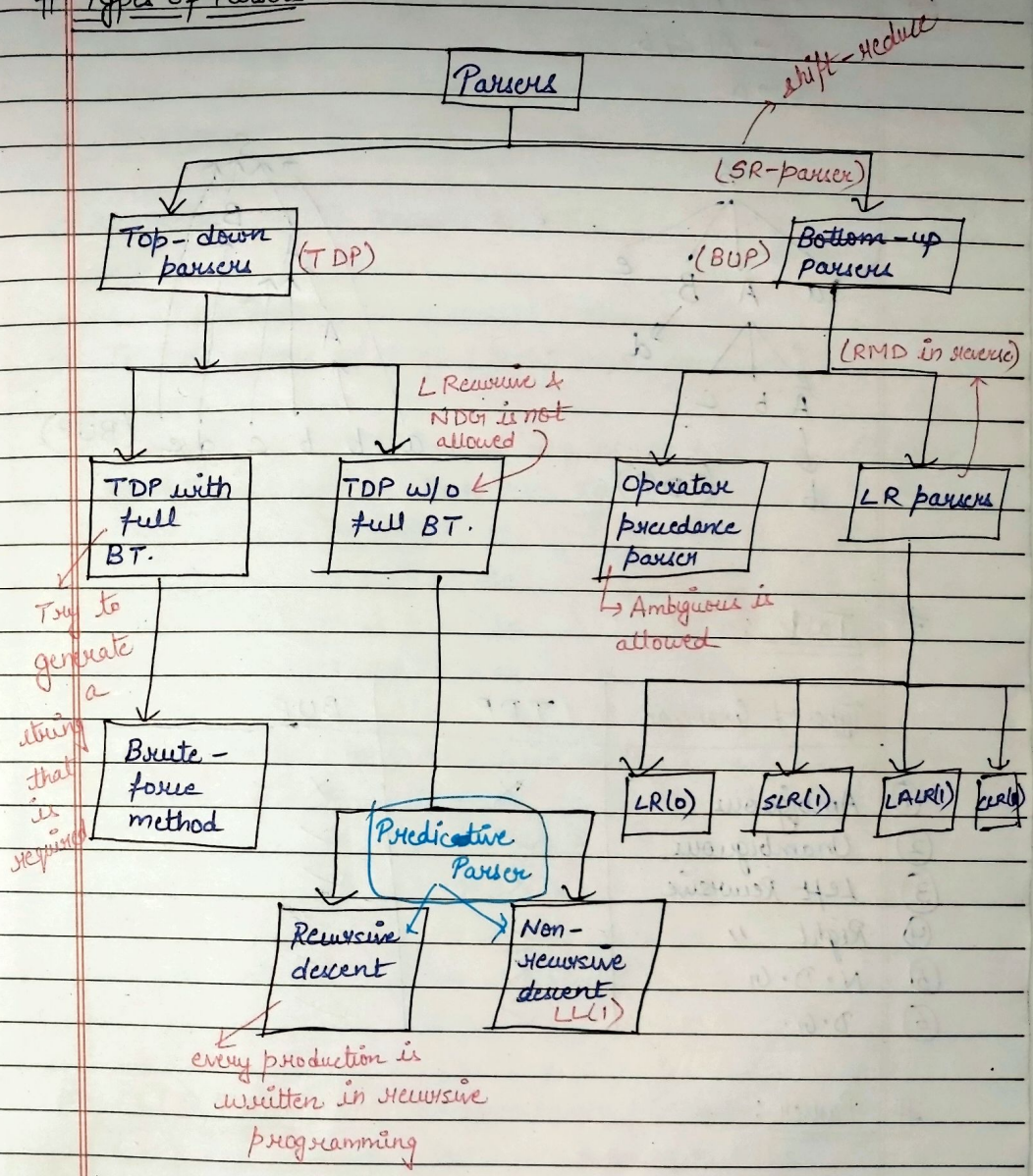
with 'Sa' prefix →

$S' \to SaS'' / b$
$S'' \to aS / Sb$

$S \to bSS' / a$
$S' \to SaS'' / b$
$S'' \to aS / Sb$

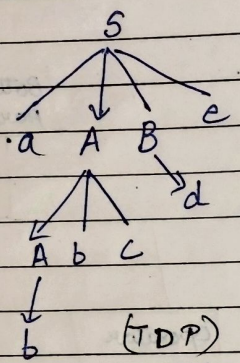
→ D.by

Types of Parsers :-



Eg:- $S \to aABe$
$A \to Abc/b$
$B \to d$

$W = abbcde$



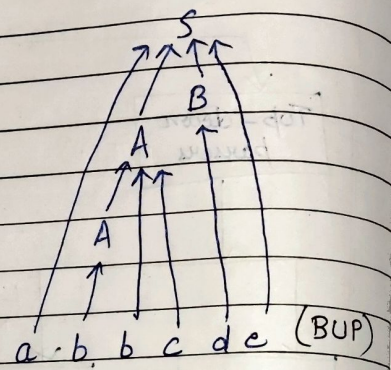


Table :-

	Types of Grammar	TDP	BUP
(1)	Ambiguous	X	X → operator precedence
(2)	Unambiguous	✓	✓
(3)	Left Recursive	X	✓
(4)	Right "	✓	✓
(5)	N.D.G	X	✓
(6)	D.G.	✓	✓

Parser :-

⇒ It is that phase of compiler w/c takes Tokens as I/P and with the help of CFG, converts it into the corresponding parse tree. It is also called Syntax Analyzer.

① Top-down parsing :- It generates parse tree for the I/P string with the help of grammar productions by expanding the non-terminals i.e., it starts from the start symbol and end on the terminals.

⇒ It uses LMD (left most derivation).

② Bottom-up :- It generates the parse tree for the given I/P string with the help of grammar productions by compressing the non-terminals i.e., it starts from the terminals & ends on the start symbol.

⇒ It uses reverse of RMD (Right most derivation).

Important pts :- ⇒ TDP → at every pt. we have to decide what is the next production we should use. Eg:-

$S \to a\textcircled{A}Be$
$\to a\textcircled{A}bc\ Be$
$\to a\ b\ b\ c\textcircled{B}e$
$\to a\ b\ b\ c\ d\ e$

} LMD

⇒ BUP :- In this, the main decision we have to make is when to reduce the given terminal. Eg:-

$S \to a\ A\ B\ e$
$\to a\ A\ d\ e$
$\to a\ A\ b\ c\ d\ e$
$\to a\ b\ b\ c\ d\ e$

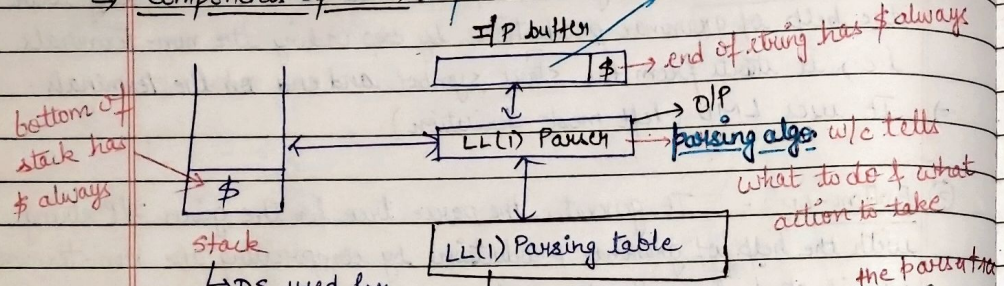
} (RMD in reverse)

LL(1) Parser :- → Top down parser / Predictive Parser

- left to right scanning of I/P
- LMD
- No. of look ahead i.e., how many symbols you need to see while making decision.

A General Parser will have all these components mentioned in LL(1).

⇒ Components of LL(1):-



⇒ $ is used to guess when we should stop.

⇒ Fn" used in LL(1) :- First() & Follow().

① First():- First(A) is a set of terminal(s) that begin in strings derived from A.

Eg:- A → abc / def / ghi

∴ First(A) = {a, d, g}

② Follow():- Follow(A) is a set of terminal(s) that appear immediately to the right of A OR what is the terminal w/c could follow a variable in the process of derivation.

Eg:- S → aAB
A → b
B → c

∴ Follow(A) = {c}

Examples:-

		First()	Follow()
①	S → aABCD	{a}	{$}
	A → b	{b}	{c}
	B → c	{c}	{d}
	C → d	{d}	{e}
	D → e	{e}	{$}

(I) Production will always end with $ such as "S$".

(II) ∈ will never come in Follow() ∵ of (I) reason.

		First()	Follow()
②	S → ABCD	{b, c}	{$}
	A → b/∈	{b, ∈}	{c}
	B → c	{c}	{d}
	C → d	{d}	{e}
	D → e	{e}	{$}

follow(D) = follow(S)

		First()	Follow()
③	S → ABCDE	{a, b, c}	{$}
	A → a/∈	{a, ∈}	{b, c}
	B → b/∈	{b, ∈}	{c}
	C → c	{c}	{d, e} + {$}
	D → d/∈	{d, ∈}	{e, $}
	E → e/∈	{e, ∈}	{$}

		First()	Follow()
④	S → Bb/Cd	{a, b, c, d}	{$}
	B → aB/∈	{a, ∈}	{b}
	C → cC/∈	{c, ∈}	{d}

		First()	Follow()
⑤	E → TE'	{id, (}	{$,)}
	E' → +TE'/∈	{+, ∈}	{$,)}
	T → FT'	{id, (}	{$,), +}
	T' → *FT'/∈	{*, ∈}	{$,), +}
	F → id/(E)	{id, (}	{$,), +, *}

		First()	Follow()
⑥	S → ACB/CbB/Ba	{d, g, ∈, h, b, a}	{$}
	A → da/BC	{d, g, ∈, h}	{h, g, $}
	B → g/∈	{g, ∈}	{$, a, h, g}
	C → h/∈	{h, ∈}	{g, b, $, h}

		First()	Follow()
(7)	S → aABb	{a}	{$}
	A → c/ε	{c, ε}	{d, b}
	B → d/ε	{d, ε}	{b}

		First()	Follow()
(8)	S → aBDh	{a}	{$}
	B → cC	{c}	{g, f, h}
	C → bC/ε	{b, ε}	{g, f, h}
	D → EF	{g, f, ε}	{h}
	E → g/ε	{g, ε}	{f, h}
	F → f/ε	{f, ε}	{h}

⇒ LL(1) Parsing Table :- ← We require First() and Follow() for construction of Parsing Table.

		First()	Follow()
(1)	E → TE'	{id, (}	{$,)}
Since two productions exists for 1 variable so we need 2 f.t's..	E' → +TE'/ε	{+, ε}	{$,)}
	T → FT'	{id, (}	{+, $,)}
	T' → *FT'/ε	{*, ε}	{+, $,)}
	F → id/(E)	{id, (}	{+, *, $,)}

TDP require Ist symbol ⇒ Now construct parsing table in order to make parser tree.

Terminals → / Variables ↓	id	+	*	(	)	$ (end of the string)
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)		

⇒ ε-production are always placed in follow of corresponding variable.

⇒ For every grammar, we can't construct parsing table, as for single terminal we can get multiple entries so w/c make it difficult to choose relvant production. [TDP]

		First()	Follow()
(2)	S → AaAb/BbBa	{a, b}	{$}
	A → ε	{ε}	{a, b}
	B → ε	{ε}	{b, a}

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

⇒ If the grammar is Left Recursive / N.D.G → then we cannot construct LL(1) Parsing table.

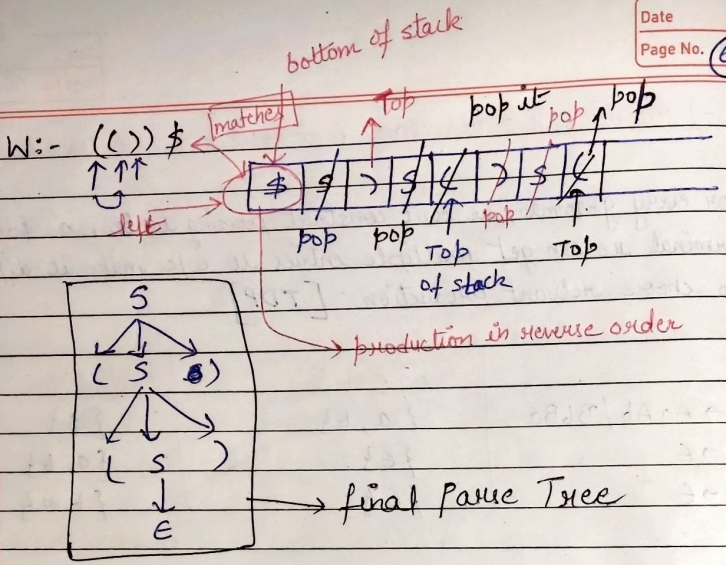
⇒ We have to eliminate L.Recursive → R.Recursive and N.D.G → D.G by left factoring, but still there is no guarantee for LL(1) parsing.

Construction of Parse Tree from LL(1) Parser :-

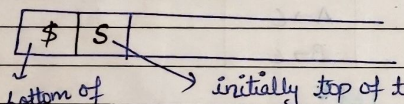
Eg:

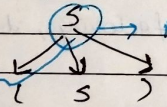
	First()	Follow()
S → (S)/ε	{(, ε}	{$,)}

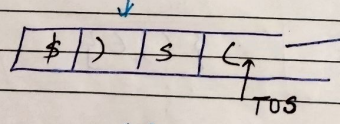
	(	)	$
S	S → (S)	S → ε	S → ε



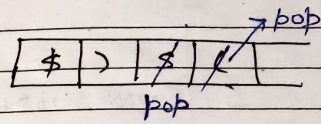
Steps :- W:- (( )) $

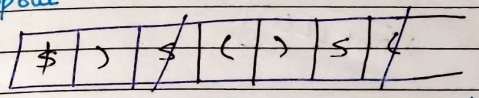
① 

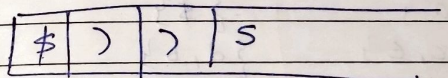
② S on seeing '(', we use the production S → (S) (open brace, insert) so tree looks like 

③ Now, replace S with RHS such that leftmost symbol appears on top, i.e., Left symbol matches with TOS so pop it out / that symbol ∴ 

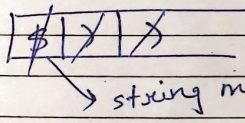
④ Now, TOS is open brace (') and I/P is also open brace i.e., Match occurs so we generated whatever we want.

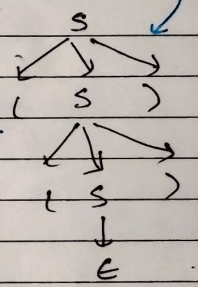
⑤ So pop off and increment lookahead (( )) $ 

⑥ Now pop out '(' and ptr is incremented to ')'. Now 

S on close brace is ϵ that means pop it off. 

⑦ Now again incremented again there is a match close brace and ⑧ Now, S on close brace is ϵ ∴ Tree becomes :- pop it off.

⑨ Now pop S ⑧ Now increment to '$' so string match ∴ it is acceptable string. 



Check whether the grammar is LL(1)?

Eg:- A → α1 b | α2 m | α3 n
α1 → aBb
α2 → aCm
α3 → aD
B → ϵ
C → ϵ
D → ϵ

	a	b	m	$
A	A → α1 b A → α2 m			
α1				
α2				
α3				
B				
C				
D				

X Not LL(1)

② $S \to aSbS / bSaS / \epsilon$

	First()	Follow()
S	{a, b, ϵ}	{$, b, a}

	a	b	$
S	S→aSbS, S→ϵ	S→bSaS, S→ϵ	S→ϵ

✗ Not LL(1)

③

	First()	Follow()
S→aABb	{a}	{$}
A→c/ϵ	{c, ϵ}	{d, b}
B→d/ϵ	{d, ϵ}	{b}

	a	b	c	d	$
S	S→aABb				
A		A→ϵ	A→c	A→ϵ	
B		B→ϵ		B→d	

✓ LL(1)

④

	First()	Follow()
S→A/a	{a}	{$}
A→a	{a}	{$}

	a	$
S	S→A, S→a	

✗ Not LL(1)

⑤

	First()	Follow()
S→aB/ϵ	{a, ϵ}	{$, a}
B→bC/ϵ	{b, ϵ}	{$}
C→cS/ϵ	{c, ϵ}	{$}

	a	b	c	$
S	S→aB			S→ϵ
B		B→bC		B→ϵ
C			C→cS	C→ϵ

✓ LL(1)

⑥

	First()	Follow()
S→A	{a, b, c, d}	{$}
A→Bb/Cd	{a, b, c, d}	{$}
B→aB/ϵ	{a, ϵ}	{b}
C→cC/ϵ	{c, ϵ}	{d}

	a	b	c	d	$
S	S→A	S→A	S→A	S→A	
A	A→Bb	A→ϵ	A→Cd	A→ϵ	
B	B→aB	B→ϵ			
C			C→cC	C→ϵ	

✓ LL(1)

Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

→ A RDP is a TDP built from a set of mutually recursive procedures where each such procedure implements one of the non-terminals of the grammar.

Recursive Descent Parser:- It is a TDP that constructs the parse tree from top and the I/P is read from L to R.

⇒ A procedure is associated with each non-terminal of the grammar.

⇒ This technique recursively parses the I/P to make a parse tree, w/c may or maynot require BT.

⇒ A form of recursive descent parser that doesnot require any BT is called **Predictive parsing.**

Eg:- $E \to iE'$ → for every variable, we have to write recursive fn. one procedure
$E' \to +iE' / \epsilon$ so here 2 fn. will be written for E and E'
$W = i + i \$$

```
(1) E() {
(2)   if (l == 'i')
(3)   {
(4)     match ('i');
(5)     E'();
(6)   }
(7) }
```

lookahead :- ptr to a string ie., 1st character of string.

(II)

```
E'()
{
    if (l == '+')
    {
        match ('+');
        match ('i');
        E'();
    }
    else
        return;
}
```

(III)

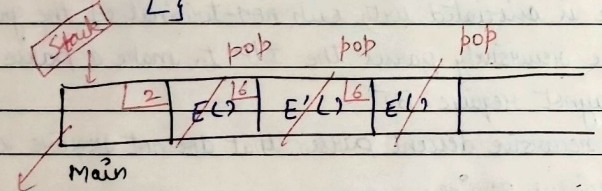
```
match (char t)
{
    if (l == t)
        l = getchar();
    else
        printf ("error");
}
```

l is going to the next character.

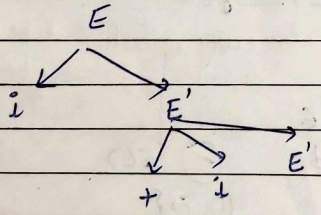
(IV)

```
main()
{
    E();
    if (l == '$')
        printf ("success");
}
```

Eg: i + i $



success & pop main



↳ Another example on pg no 121

Operator Precedance Parser :-

⇒ Operator Grammar :- A grammar that is used to define mathematical operations / fn's with some restriction on the grammar.

⇒ It has (2) properties :-

1) no RHS of any production has a ∈.
2) no two non-terminals are adjacent on RHS.

Eg: (1) E → E+E/E*E/id → valid operator grammar. (OG)

(2) E → EAE/id, A → +/* → X not valid (OG) → convert it into OG → After conversion

(3) S → SAS/a, A → bSb/b — Expand → S → SbSbS/SbS/a, A → bSb/b X → not to add (Final OG)

Operator Relation Table :- [Rules] :-

(1) a ⋗ b = It means 'a' has higher precedance than the terminal 'b' OR 'a' takes precedance over 'b'.

(2) a ⋖ b = It means 'a' yields precedance to 'b' OR 'a' has lower precedance than terminal 'b'.

(3) a ≐ b = It means 'a' has same precedance as 'b'.

⇒ We will construct operator relation table for a given ambiguous grammar. Eg:- E → E+E/E*E/id

It is a

⇒ BUP that interprets an OGr.
⇒ This parser is only used for Operators.
⇒ Ambiguous grammar are not allowed in any parser except (in this) operator precedance parser.

Highest precedance

left associativity

least precedance

	id	+	*	$
id	–	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	–

left associativity

⇒ Two id's will never be compared ∵ they will never come side by side.
⇒ Identifier will be given highest precedance as compared to any other operator, and $ has least precedance.
⇒ Using this operator precedance table, parser will parse the I/P.

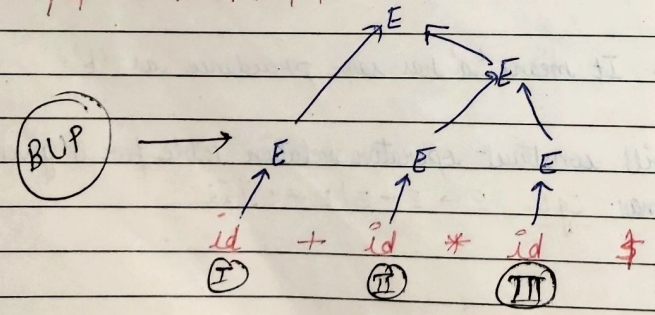
(V)

W:- id + id * id $ match/compare with $

Stack

$	id	+	id	*	id

pop pop pop pop pop

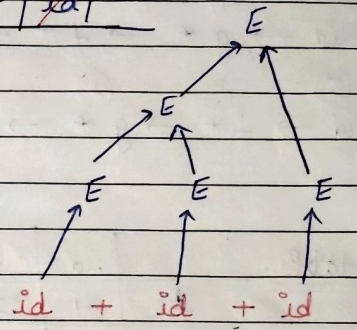


(II)

W:- id + id + id $

pop pop pop pop pop

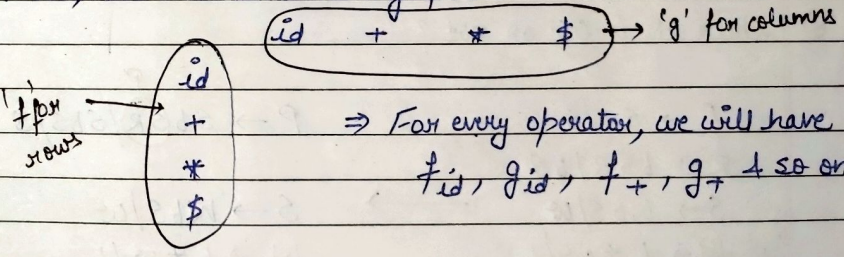
$	id	+	id	+	id



NOTE:- If we have 'n' operators, then total combination in operator relation table will be = $O(n^2)$

⇒ Due to above disadvantage, and in order to reduce/less the size of the table, we will reduce the complexity using another method namely, operator fn. table.

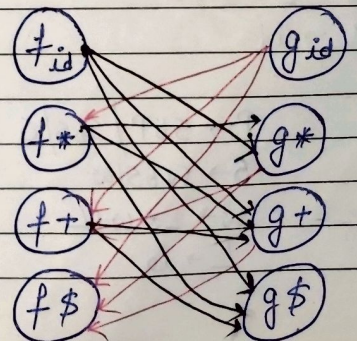
⇒ Now, we will construct a graph:-



⇒ For every operator, we will have f_{id}, g_{id}, f_+, g_+ & so on...

$f \rightarrow g$ {>}

$f \leftarrow g$ {<}



⇒ To make fⁿ? table, we have to find out for every node that what is the length of the longest path starting from that node.

Longest path:
$f\,id \xrightarrow{1} g\,* \xrightarrow{2} f\,+ \xrightarrow{3} g\,+ \xrightarrow{4} f\,\$$
$g\,id \xrightarrow{5} f\,* \xrightarrow{6} g\,* \xrightarrow{7} f\,+ \xrightarrow{8} g\,+ \xrightarrow{9} f\,\$$ (10)

Fⁿ? table :-

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

∴ Fⁿ? table size = $O(2n)$ → Advantage

⇒ Disadv :- Even though we have blank entries in relation table we get non-blank entries in fⁿ? table. So error detecting capability of fⁿ? table < error detecting capability of operator relation table.

Q) Check it is OG or not?

P → SR/S
R → bSR/bS
S → WbS/w
W → L*w/L
L → id

→

P → SbSR/SbS/S
S → WbS/w
W → L*w/L
L → id

P → SbP/SbS/S
B → WbS/w
W → L*w/L
L → id

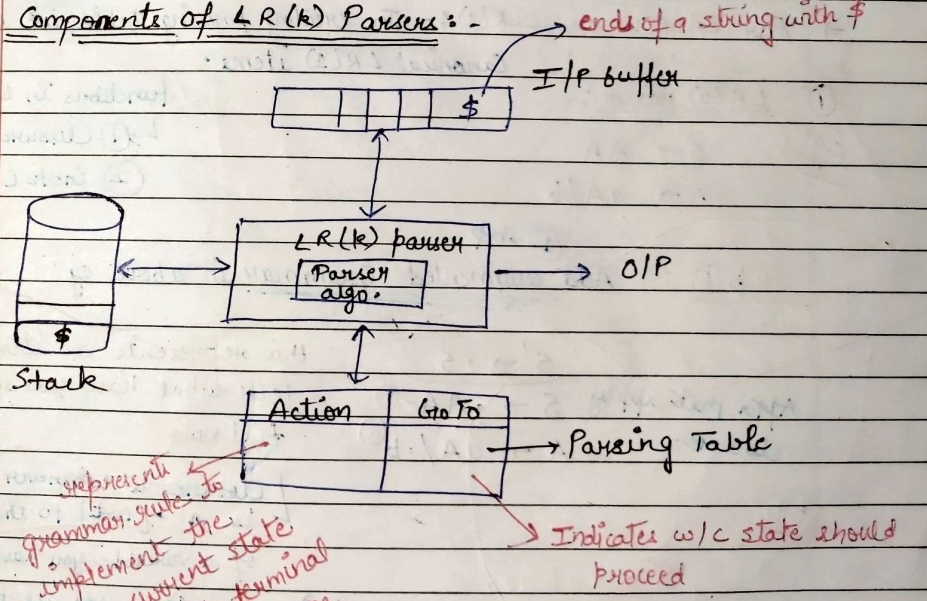
since w is R. Recursive & has high priority

	id	*	b	$
id	—	>	>	>
*	<	<	>	>
b	<	<	<	>
$	<	<	<	—

⇒ LR(0) Parsing Table Construction Algorithm :-
① Find Augmented Grammar.
② I_0 = closure [Augmented LR(0) item]
③ Apply closure and Goto fⁿ? and find all collection of LR(0) item using DFA.
④ Reduce DFA = LR(0) Parsing table.

LR parsers :- → BDP
→ L to R scanning of I/P
→ RMD in reverse
LR(k) :- → the no. of lookahead

Components of LR(k) Parsers :-



Types of LR(k) Parsers :-

LR(0) — SLR(1) (Simple) — LALR(1) (Look Ahead) — CLR(1) (Canonical)

Least powerful → Most powerful

⇒ For all the 4 parsers, LR parsing algorithm is same but construction of parsing table varies.

⇒ For LR(0) and SLR(1) :- In creating parsing table, we use Canonical LR(0) items.

⇒ For LALR(1) and CLR(1) :- In creating parsing table, we use Canonical LR(1) items.

① LR(0) parser :-

Eg:- $S \to AA$
$A \to aA/b$

functions in LR parsers
↳ ① Closure()
② Goto()

① step

Step ① :- Add augmented Grammar in above eg:

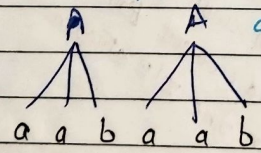
$S' \to .S$ → this represents as item, which tells what have you seen till now.

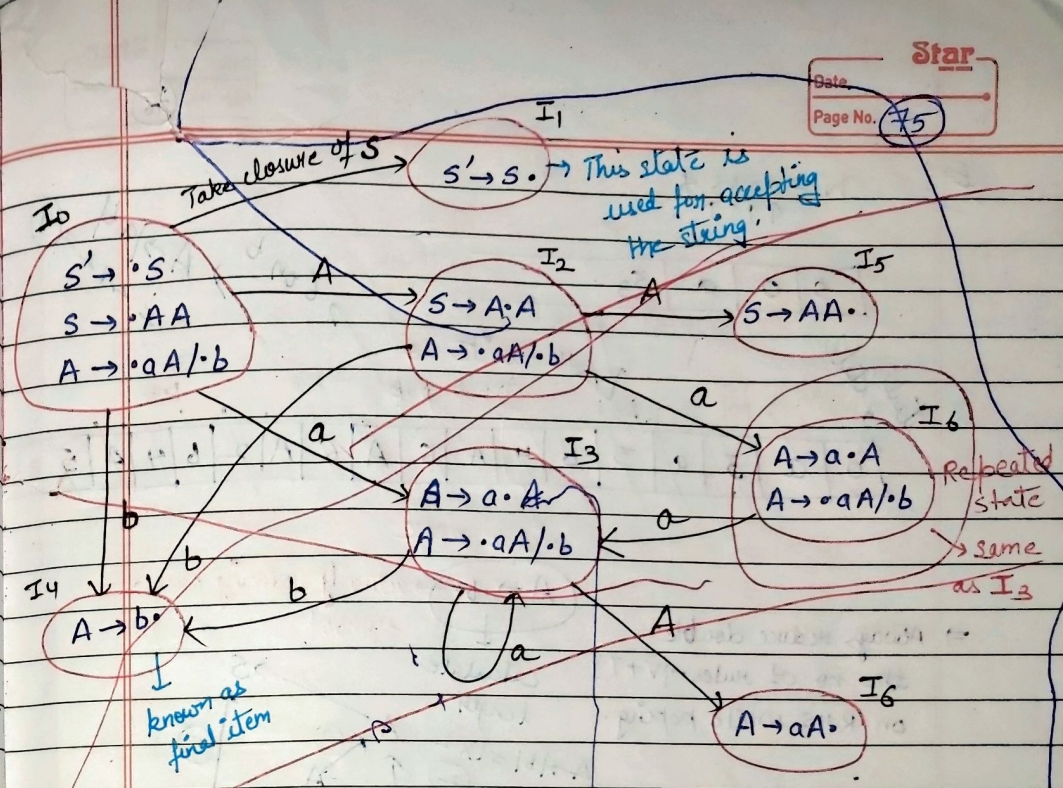
After that apply closure. $S \to .AA$ ① , $A \to .aA/.b$ ② ③

[Closure is whenever there is a •[Dot] to the left of a variable you have to add all the productions of that variable with •[Dot] in the beginning.

Eg:- $S \to aabaab$

Helps in reduction while creating parse tree





⇒ LR(0) Parsing Table :-

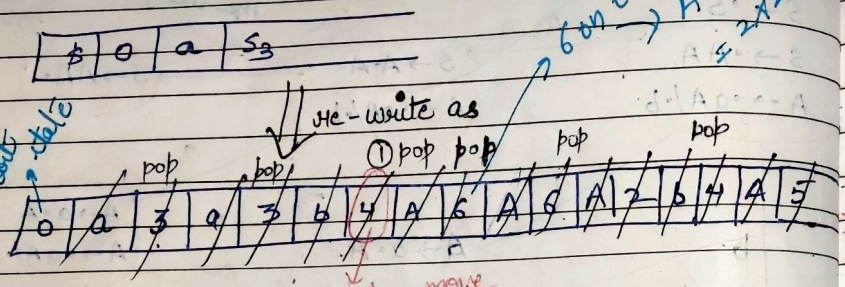
States (I)	Action (Terminals)			Goto (Variables)	
	a	b	$	A	S
I0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

shift the variables & terminals

starting / augmented grammar known as Accept.

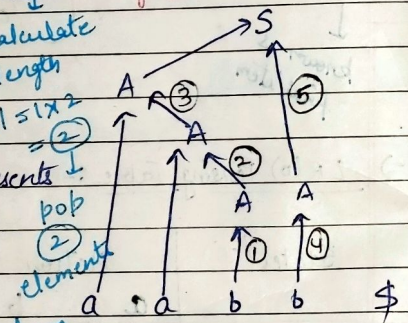
reduce to final items

Eg: $W = aabb\$$



⇒ Always reduce double the no. of rules → {V+T} on RHS while poping.

⇒ Blank entries in table represents error.



Q

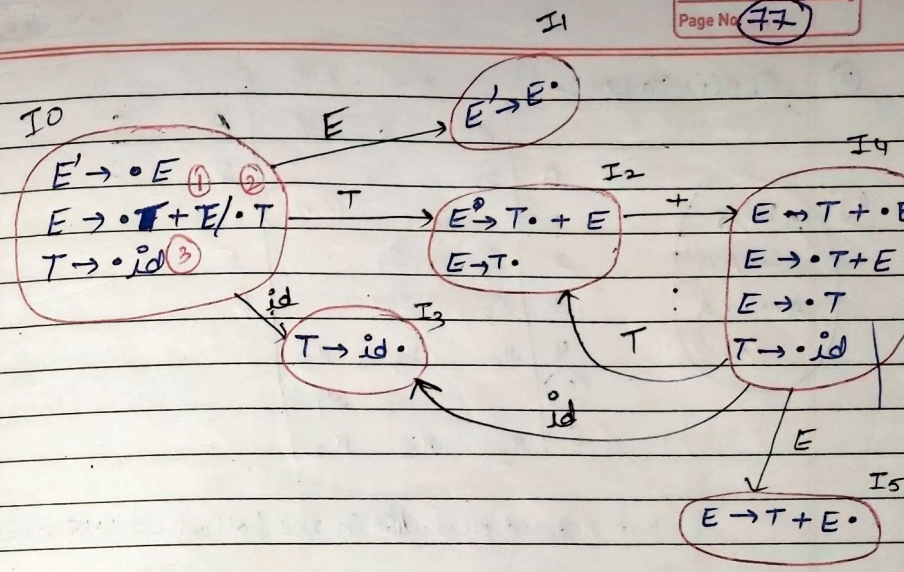
$E \to T + E / T$
$T \to id$

Shift = Push
Reduce = Pop

(1)

$E \to T + E / T$
$T \to id$

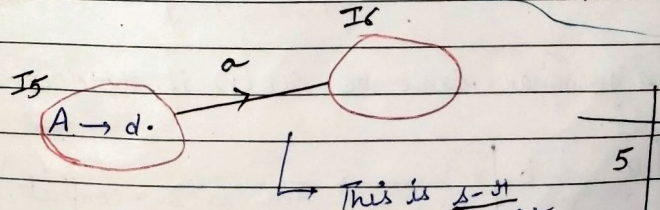
$E' \to \cdot E$
$E \to \cdot T + E / \cdot T$
$T \to \cdot id$

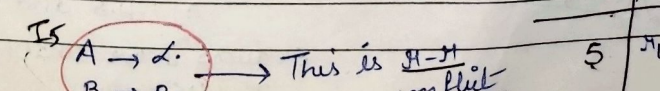


⇒ LR(0) Parsing Table :-

States (I)	Action (Terminals)			Goto (Variables)	
	id	+	$	E	T
0	S_3			1	2
1			Accept		
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		

S-R conflict





	a	b	$
5	S_6/R_1	R_1	R_1

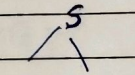
5	R_1/R_2	R_1/R_2	R_1/R_2

② SLR(1) Parser :-

	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	R_3	R_3	R_3		
5			R_1		
6	R_2	R_2	R_2		

Eg:- $S' \to .S$; $S \to .AA$ ①; $A \to .aA$ ②; $/.b$ ③

x put reduced production in the follow of LHS variable.

R_1  $\to$ follow of (S) = $

R_2 $A \to a, b, \$$ (follow)

R_3 follow(A) $\to$ a, b, $

⇒ SLR(1) is more powerful than LR(0) parser, as it can detect errors earlier.

Imp. pts :-

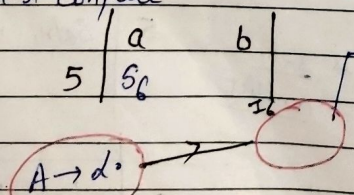
① If there is more than one production in the final item then grammar can't be LR(0).

② If there is R-R & S-R conflict then grammar can't be LR(0).

③ Clearly, a grammar can't be SLR(1) if there is S-R & R-R conflict.

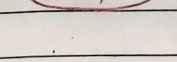
Eg :-

	a	b
5	S_6	



This is S-R conflict, then check follow(A) = {a, ...} of L.H.S and put reduced move under follow of LHS only.

If $A \to \alpha.$

If $A \to \alpha.$  This is R-R conflict, then check

follow(A) ∩ follow(B) = ∅

Practice Questions :-

① $S \to dA/aB$ ①②; $A \to bA/c$ ③④; $B \to bB/c$ ⑤⑥

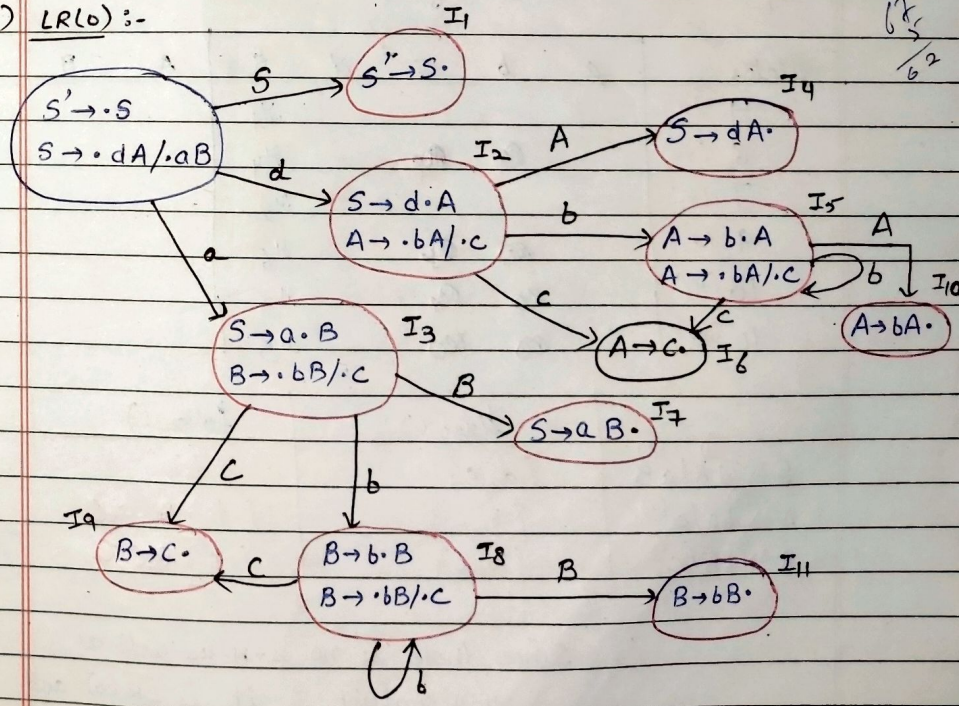
Is this (i) LL(1)
(ii) LR(0)
(iii) SLR(1)

(i) LL(1) :-

	d	a	b	c	$
S	S → dA	S → aB			
A			A → bA	A → c	
B			B → bB	B → c	

Since there is no conflict, ∴ it is LL(1) grammar.

(ii) LR(0) :-



States	Action a	b	c	d	$	Goto A	B	S
0	S_3			S_2				1
1					Accept			
2		S_5	S_6			4		
3		S_8	S_9				7	
4	H_1	H_1	H_1	H_1	H_1			
5		S_5	S_6			10		
6	H_4	H_4	H_4	H_4	H_4			
7	H_2	H_2	H_2	H_2	H_2			
8		S_8	S_9				11	
9	H_6	H_6	H_6	H_6	H_6			
10	H_3	H_3	H_3	H_3	H_3			
11	H_5	H_5	H_5	H_5	H_5			

⇒ If we use SLR(1) parsing table :-

States	a	b	c	d	$	A	B	$
4 (H_1)					H_1			
6 (H_4)					H_4			
7 (H_2)					H_2			
9 (H_6)					H_6			
10 (H_3)					H_3			
11 (H_5)					H_5			

	first()	follow()
$S \to$ ① dA / ② aB	{d, a}	{$}
$A \to$ ③ bA / ④ c	{b, c}	{$}
$B \to$ ⑤ bB / ⑥ c	{b, c}	{$}

∴ Since there is no S-H as well as H-H conflict ∴ it is LR(0) and SLR(1) parser grammar.

(2) Ambiguous grammar

$S \to A/a$
$A \to a$

(i) ✗ LL(1) (ii) ✗ LR(0) (iii) ✗ SLR(1)

(i) LL(1) :-

	first()	follow()
$S \to A/a$	{a}	{$}
$A \to a$	{a}	{$}

	a	$
S	$S \to A$, $S \to a$	
A	$A \to a$	

→ ✗ LL(1) grammar

(ii) LR(0) :-

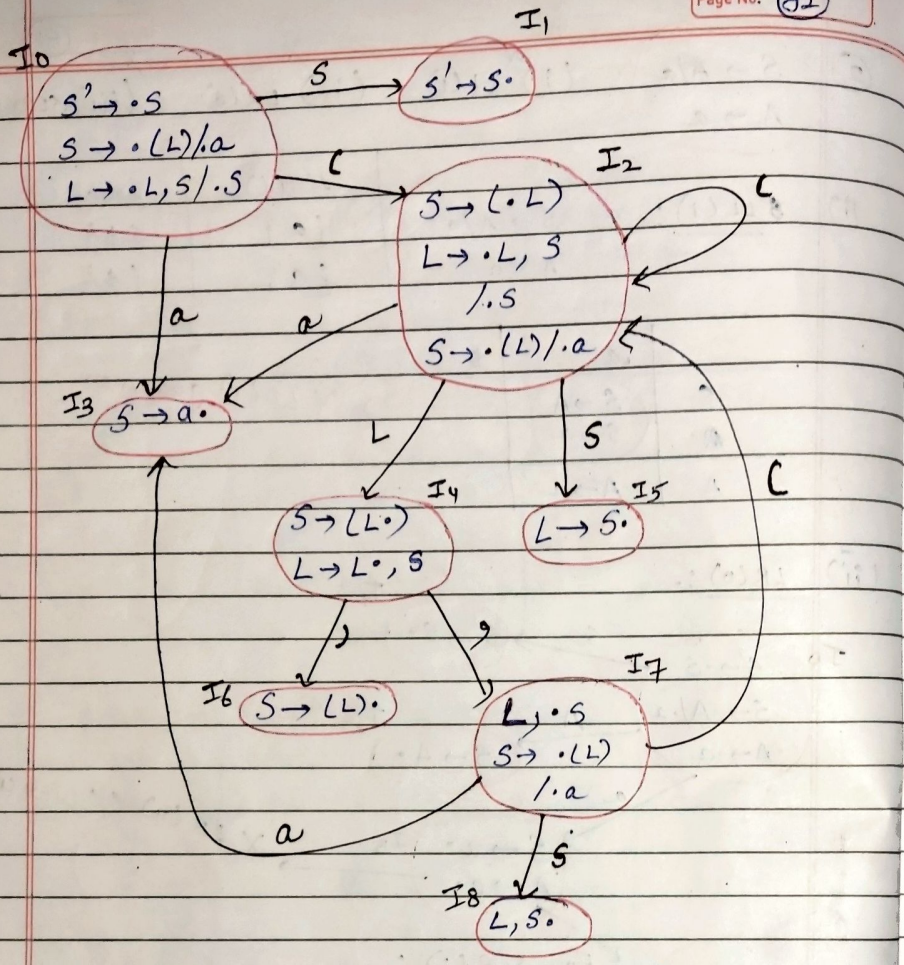
I_0: $S' \to \cdot S$, $S \to \cdot A / \cdot a$, $A \to \cdot a$
S → I_1: $S' \to S \cdot$
A → I_2: $S \to A \cdot$
a → I_3: $S \to a \cdot$, $A \to a \cdot$

✗ LR(0) ∵ H-H conflict

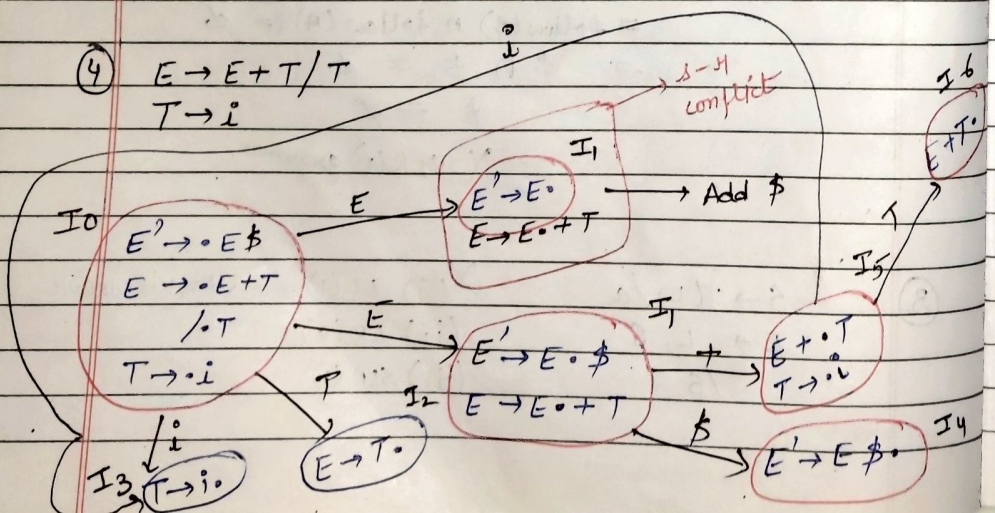
For SLR(1) :-
⇒ follow(S) ∩ follow(A) = ∅
⇒ $ ∩ $ ≠ ∅
$ ≠ ∅
✗ SLR(1) grammar

(3) $S \to (L)/a$
$L \to L, S$
$/S$

✗ (i) LL(1) → left Recursive
✓ (ii) LR(0)
✓ (iii) SLR(1)

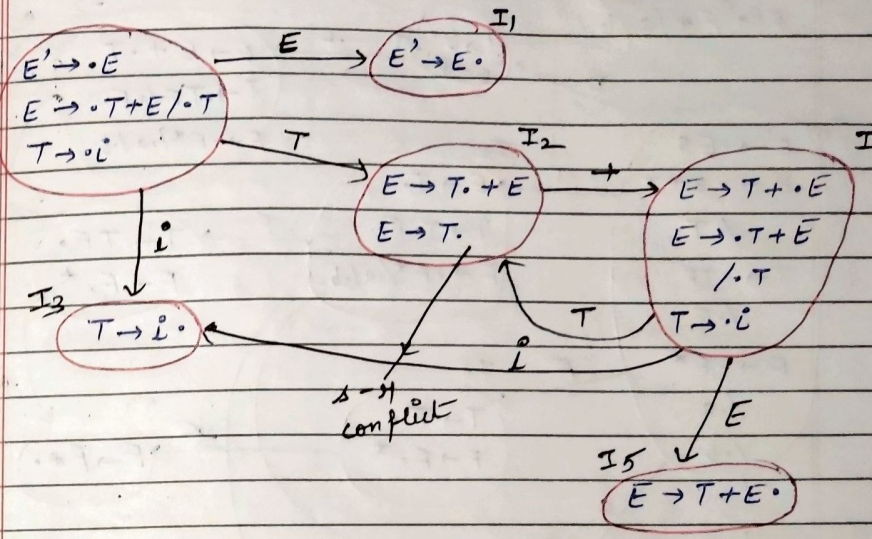


(4) $E \to E+T/T$
$T \to i$



(5) $E \to T+E/T$ (1) (2)
$T \to i$ (3)

(i) LL(1) (ii) LR(0) (iii) SLR(1)



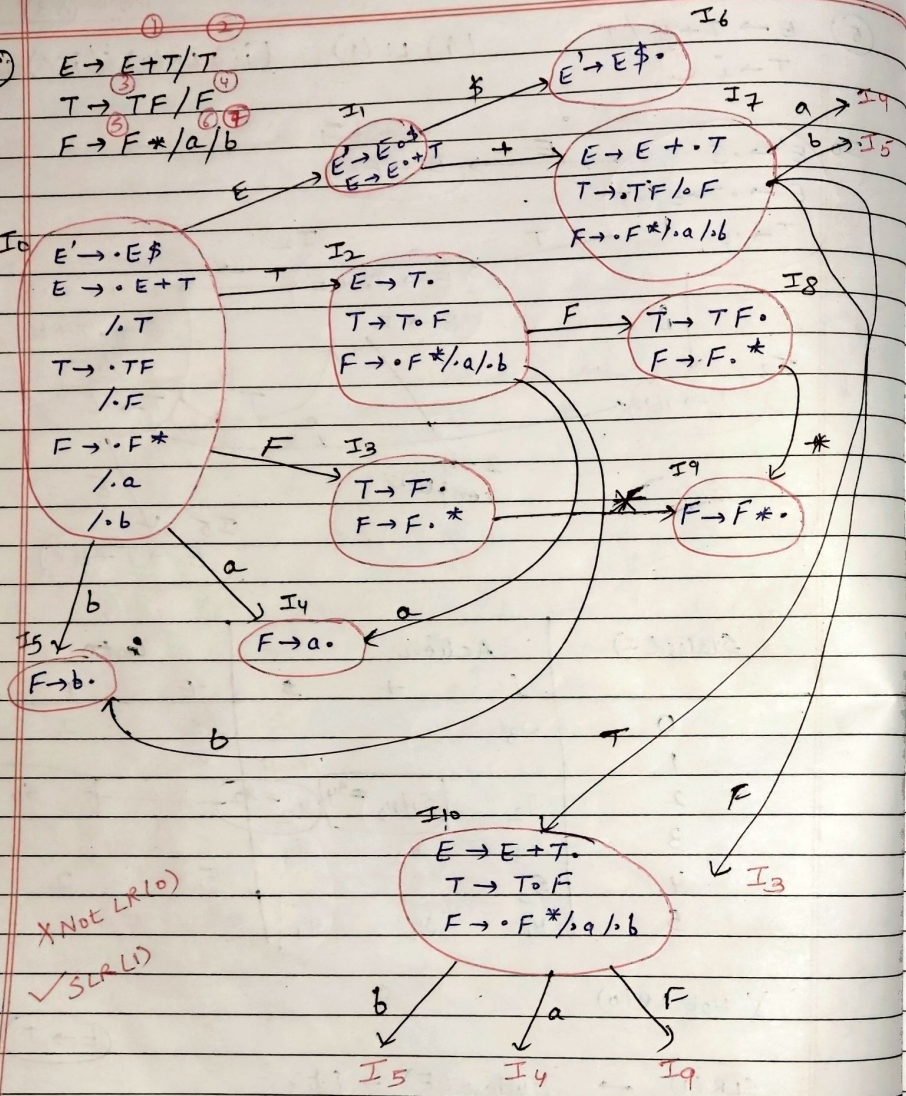
State (I)	Action			Goto	
	i	+	$	E	T
0	S_3			1	2
1			Accept		
2	R_2	$S_4/R_2 = S_4$	$R_2 = R_2$	S_4 R_2	
3	R_3	R_3	R_3		
4	S_3			5	2
5	R_1	R_1	R_1		

✗ Not LR(0)

$E \to T$

SLR(1) → Follow(E) = {$}
Follow(T) = {+, $}

(6) $E \to E+T/T$ ① ②
$T \to TF/F$ ③ ④
$F \to F*/a/b$ ⑤ ⑥ ⑦



✗ Not LR(0)

✓ SLR(1)

For state I_2

	a	b	+	*	$
2	S_4	S_5	R_2		R_2
3	R_4	R_4	R_4	S_8	R_4
7	R_3	R_3	R_3	S_8	R_3
9	S_4	S_5	R_1		R_1

follow(E) = {+, $}
follow(T) = {+, $, a, b}
follow(F) = {*, $, a, b, +}

CLR(1) and LALR(1) Parser :-

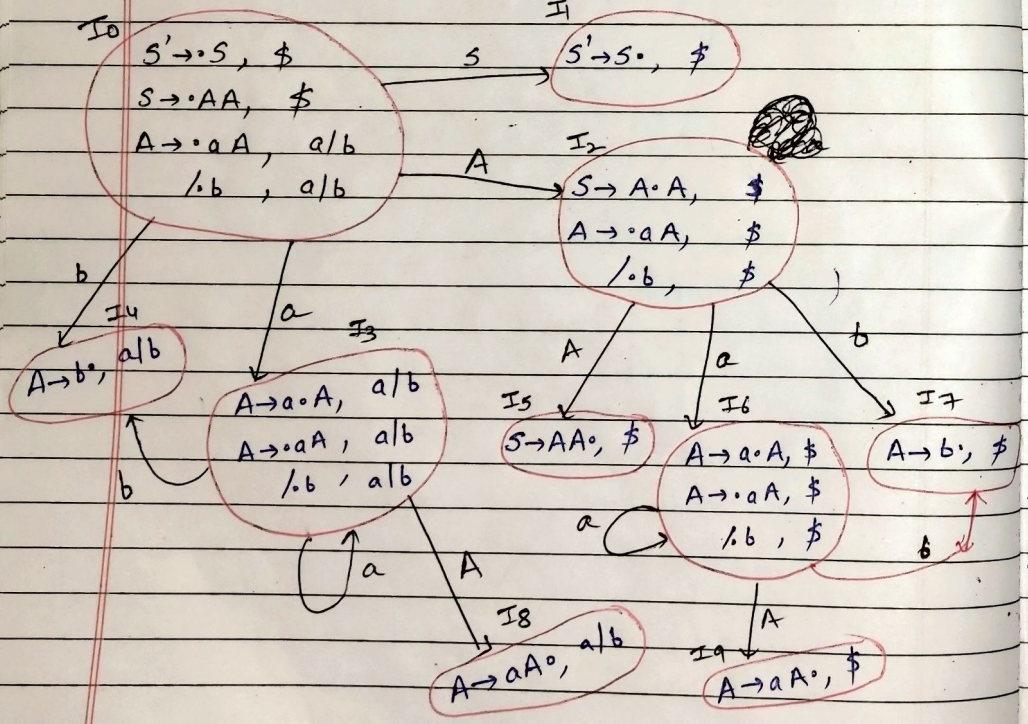
→ It works on LR(1) item = LR(0) item + lookahead

→ Whenever there is final item we can place it in lookahead symbol rather than placing it in the whole production.

→ Lookahead for Augumented grammar is always $.

Eg:- $S \to AA$
$A \to aA / b$

$S' \to \cdot S, (\$)$
$S \to \cdot AA, \$$ ← first of this
$A \to \cdot aA, a/b$ ← first of A
$\cdot b, a/b$



⇒ CLR(1) Parsing table :-

	a	b	$	S	A
0	S_3	S_4		2	1
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		

⇒ On merging common states { $I_3, I_6 = I_{36}$, $I_4, I_7 = I_{47}$, $I_8, I_9 = I_{89}$ } we can reduce the size of CLR(1) parsing table.

⇒ No. of states in (i) LR(0) = SLR(1) = LALR(1) {after merging}
(ii) CLR(1) ≥ (i)

Conversion CLR(1) to LALR(1) parsing table :-

	a	b	$	A	S
0	S_3	S_4		2	1
1					
26	S_6	S_7		59	67 / 18 / 4
3	S_3	S_4		8	
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

∴ No. of states = 7

States	Action a	Action b	Action $	Goto A	Goto S
0	S_{36}	S_{47}		2	1
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
4	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

∴ No. of states = 7

Conflicts in CLR(1) & LALR(1):-

LR(1) item	S-R conflicts	R-R conflicts
	I_9	I_9 / I_{11}
	$A \to \alpha \cdot$, a/$ $B \to \beta \cdot a\gamma$, c/d	$A \to \alpha \cdot$, a $B \to \beta \cdot$, b / $A \to \alpha \cdot$, b $B \to \beta \cdot$, a

can be solved

(I)	a	c	d	$
9	R_1	S_1	S_1	R_1

on merging

(I)	a	b
9	R_1	R_2
11	R_2	R_1

$A \to \alpha \cdot$, a/b
 $B \to \beta \cdot$, b/a

After merging R-R conflict occurs in LALR(1), but no conflict in CLR(1)

Imp. pts:-

(1) If the grammar is not CLR(1), then definitely it will not be LALR(1) but vice-versa not true.

(2) If there is no S-R conflict in CLR(1) then there will no S-R conflict in LALR(1).

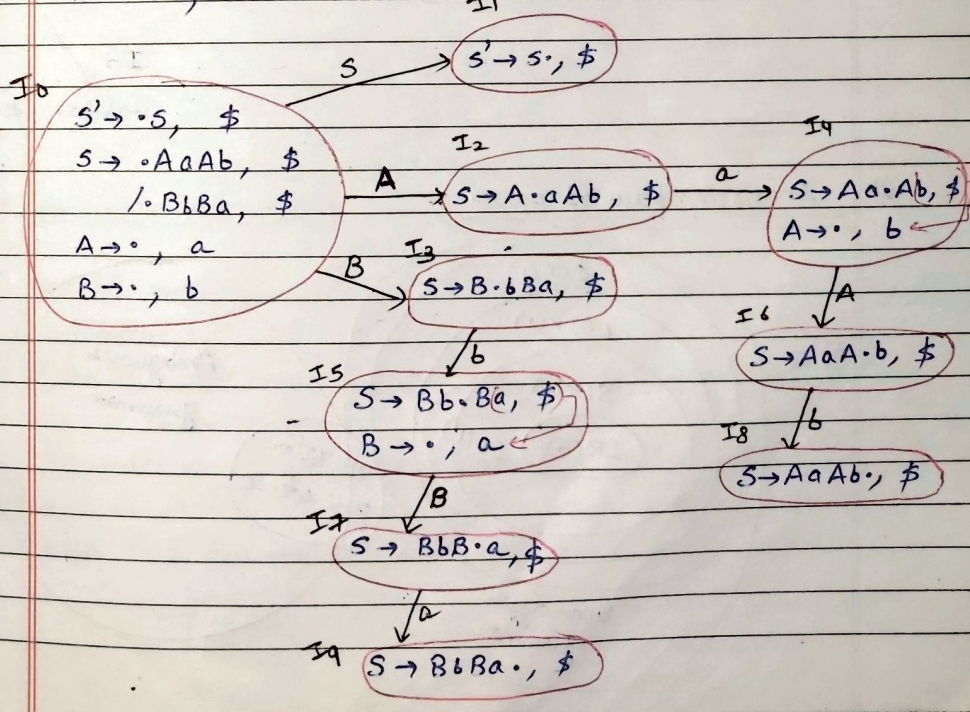
(3) If there is no R-R conflict in CLR(1) then there could be R-R conflict in LALR(1).

Eg:- (1) $S \to AaAb / BbBa$, $A \to \epsilon$, $B \to \epsilon$

(i) LL(1) (ii) LR(0) (iii) SLR(1) (iv) CLR(1) (v) LALR(1)

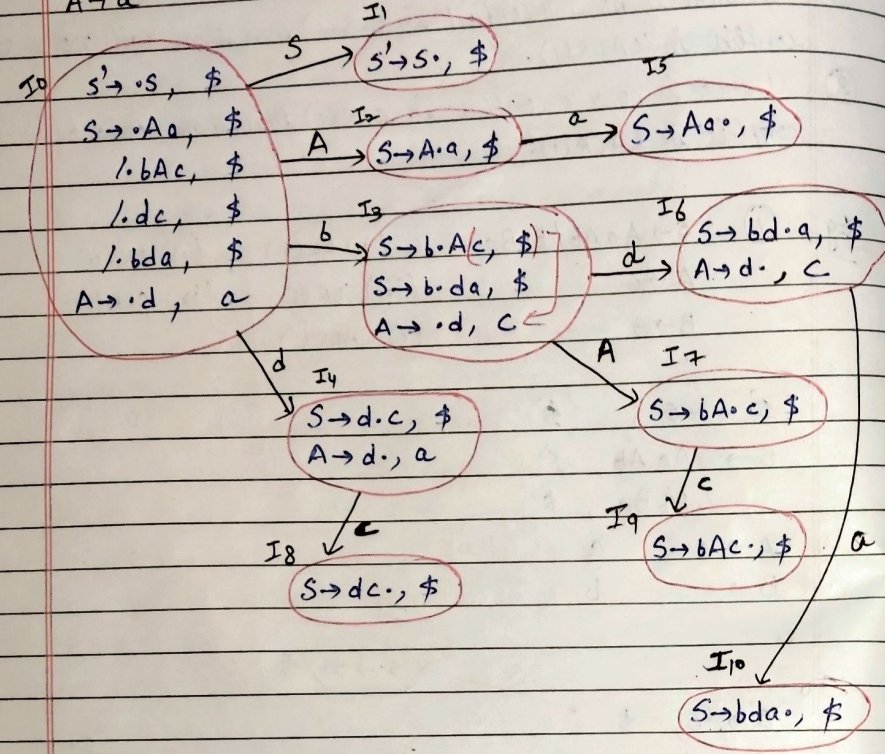
$S' \to \cdot S$, ($)
$S \to \cdot AaAb$, $
$/ \cdot BbBa$, $
$A \to \cdot$, a
$B \to \cdot$, b

(first, first, first)

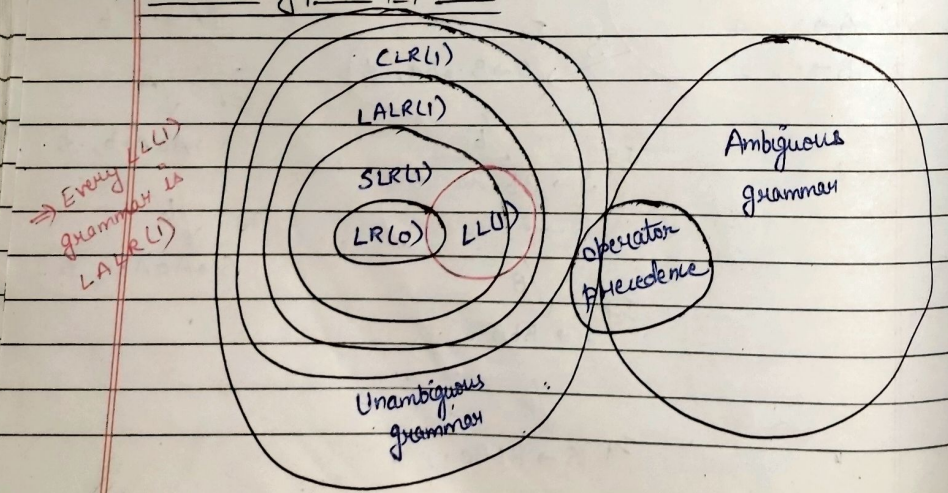


② $S \to Aa / bAc / dc / bda$
$A \to d$

(i) LL(1) (ii) LR(0) (iii) SLR(1)
(iv) CLR(1) (v) LALR(1)



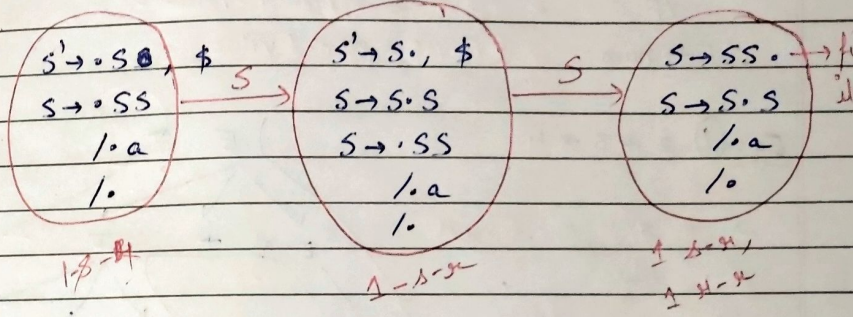
Increasing power of parser :-



Q) Given no of states in (i) n_1 = SLR(1), (ii) n_2 = LALR(1) (iii) n_3 = CLR(1), what is the relationship b/w n_1, n_2 & n_3?

$$n_1 = n_2 \leq n_3$$

Q) Find the no of SR and RR conflict in DFA with LR(0) items
$S \to SS / a / \epsilon$.



$= 3$ S-R 1 R-R

Advantages of Ambiguous and unambiguous grammar :-

① No. of production are less in AG. They are shorter and easy to represent.
② Production are natural & meaningful i.e., natural to read and easier to understand.
③ The rules and associativity is not fixed so it is flexible.
④ $E \to id$ in AG can directly derive in just single step but in UAG it is deriving in $E \to T \to F \to id$ in ③ steps.

Eg:-

$E \to E + E$ / $E * E$ / id

AG to UG

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

Syntax directed Translation (SDT):- (Annotated)

Grammar + Semantic rules = SDT

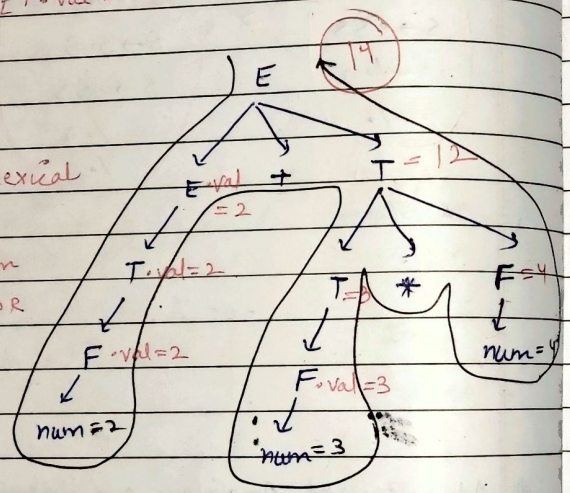
⇒ SDT for evaluation of expression:-

$E \to E+T$	{ E.value = E.value + T.value }
/T	{ E.value = T.value }
$T \to T*F$	{ T.value = T.value * F.value }
/F	{ T.value = F.value }
$F \to num$	{ F.val = num.lvalue }

Eg: (1) 2 + 3 * 4

→ 2, 3, 4 are lexical value

→ Traverse from T to D & L to R



Appl:-

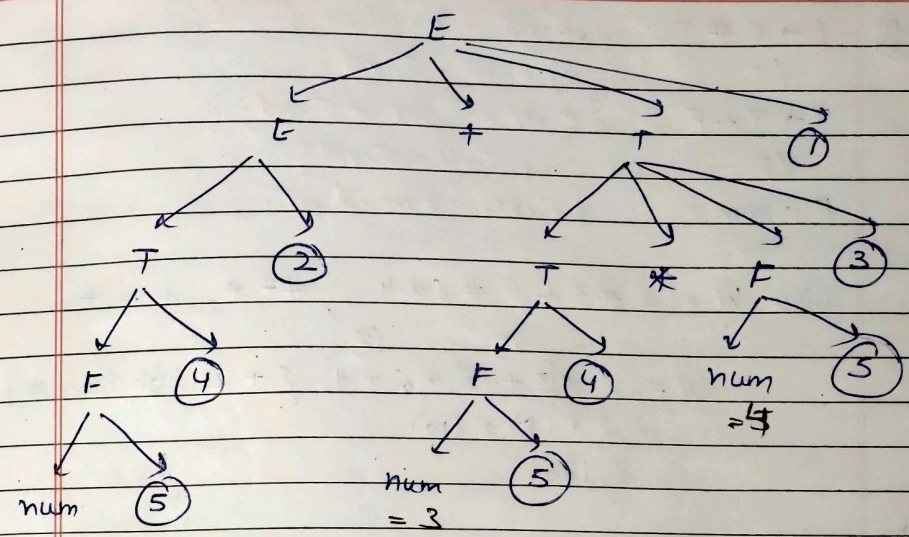
(2)

$E \to E+T$	{ printf("+"); }	(1)
/T	{ }	(2)
$T \to T*F$	{ printf("*"); }	(3)
/F	{ }	(4)
$F \to num$	{ printf(num.(val)); }	(5)

Grammar for postfix expression

⇒ convert infix to postfix expression

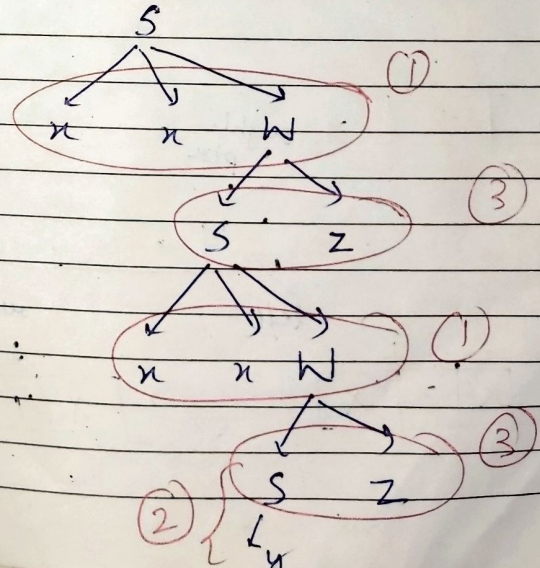
2 + 3 * 4



TDP O/P = 2, 3, 4, *, +

** Gate (3)

$S \to xxW$	{ printf(1); }	String:- xxxxyzz
/y	{ printf(2); }	
$W \to Sz$	{ printf(3); }	



O/P = 2, 3, 1, 3, 1

Q) $E \to E \# T$ { E.val = E.val * T.val; }
/T { E.val = T.val }
$T \to T \& F$ { T.val = T.val + F.val; }
/F { T.val = F.val }
$F \to num$ { F.val = num.lval; }

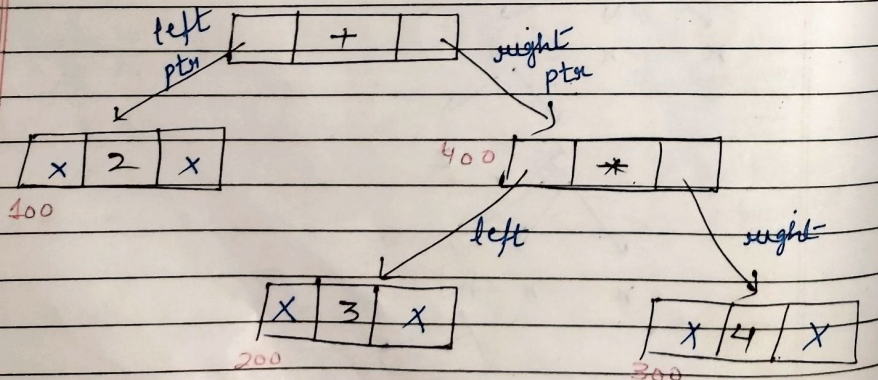
Given W = 2 #3 & 5 #6 & 4, # = *, & = +

W = 2 * (3 + 5) * (6 + 4) { + is high than * }
① ②
= 2 * (8) * (10)
= 160

Std to build Syntax Tree :-

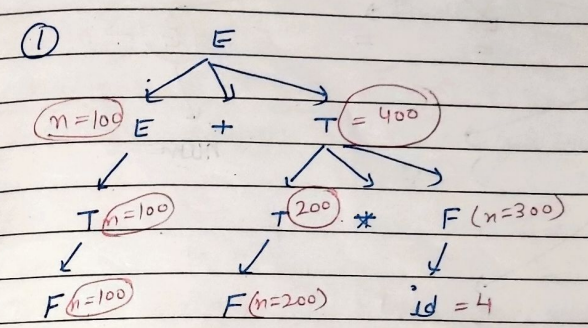
Eg:- $E \to E_1 + T$ { E_1.nptr = mknode (E_1.nptr, '+', T.nptr); }
/T { E.nptr = T.nptr; }
$T \to T_1 * F$ { T.nptr = mknode (T_1.nptr, '*', F.nptr); }
/F { T.nptr = F.nptr; }
$F \to id$ { F.nptr = mknode (null, id name, null); }

2 + 3 * 4 = I/P



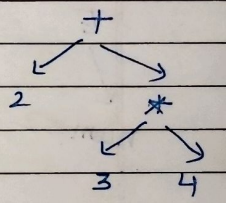
X = Null
100, 200, 300, 400 = starting address

⇒ Types of Syntax tree :-



Concrete syntax tree

② Abstract Syntax tree
↳ not show details of variables

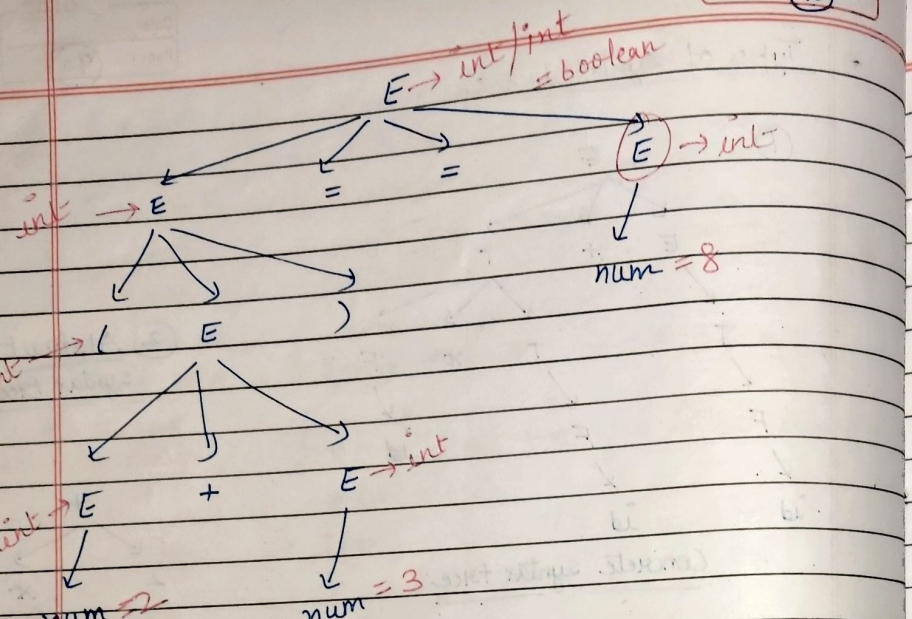


Std for type checking :-

$E \to E_1 + E_2$ { if ((E1.type == E2.type) && (E1.type = int)) then ① }
$/E_1 == E_2$ { if ((E1.type == E2.type) && (E1.type = int/boolean)) then ② }
$/(E_1)$ { E.type = E1.type; }
/num { E.type = int; }
/True { E.type = boolean; }
/False { E.type = boolean; }

① E.type = int else error;
② E.type = boolean else error;

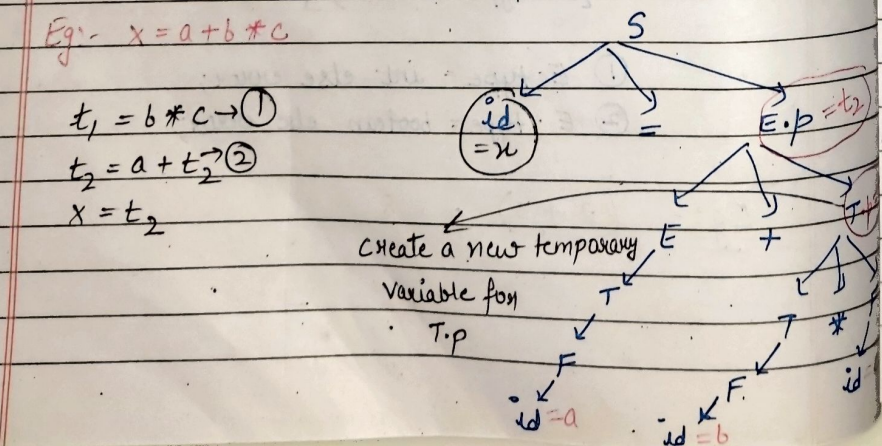
Eg:- (2 + 3) == 8



Std to generate 3-address code :-

S → id = E { gen (id.name = E.place); }
E → E1 + T { E.place = newTemp(); gen (E.place = E1.place + T.place); }
/T { E.place = T.place }
T → T * F { T.place = newtemp(); gen (T.place = T1.place * F.place); }
/F { T.place = F.place }
F → id { F.place = id.name; }
↳ attribute for place holder

Eg:- x = a + b * c



Types of SDT :- → S-attributed SDT
→ L-attributed SDT

⇒ classification of attributes :-

(i) Synthesized attribute :- If the attribute (node) takes value from its children.

Eg:- A → BCD, A.s = {B.s, C.s, D.s}

(ii) Inherited attribute :- if the node takes value from its parent or sibling.

Eg:- A → BCD, Ci = Ai, Ci = Bi, etc....

Difference b/w S-SDT & L-SDT :-

S-SDT	L-SDT
① Uses only synthesized attributes.	① Uses both synthesized and inherited attributes. Each inherited attribute is restricted to inherit either from parent or left sibling only. Eg:- A → XYZ { YS = AS, YS = XS, YS = ZS }
② Semantic actions are placed at right end of production. Eg:- A → BCC{} Also called as postfix SDT.	② Semantic actions are placed anywhere on RHS. Eg:- A → {}BC /D {}E /FG{}
③ Attributes are evaluated during Bottom-up parser.	③ Attributes are evaluated by traversing the parse tree depth first, L to R.

Not S-SDT

Q) $A \to LM \{ L.i = f(A.i); M.i = f(L.s); A.s = f(m.s); \}$
$A \to QR \{ R.i = f(A.i); Q.i = f(R.i); A.s = f(Q.s); \}$
NOT L-SDT

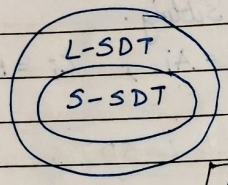
a) S-SDT b) L-SDT c) Both d) None ✓

Q) $A \to BC \{ B.s = A.s \}$

a) S-SDT b) L-SDT ✓ c) Both d) None

Q) W/c is more powerful?
⇒ L-SDT as it allows both L-SDT & S-SDT.

Power:-



Unit-4

Symbol Table :- It is an important data structure created and maintained by compilers in order to store information about the occurrences of various entities such as variables names, fn^n names, objects, classes, interfaces, etc.

⇒ The info. is collected by the analysis phase of a compiler and used by the synthesis phase to generate target code.

Usage of Symbol Table by all phases of a compiler:-

	Phase	Usage
(1)	Lexical Analysis	Creates new entries for each new identifiers.
(2)	Syntax Analysis	Adds info. regarding attributes like type, scope, dimension, line of reference, and line of use.

	Phase	Usage
(3)	Semantic Analysis	Uses the available info. to check for semantics and is updated.
(4)	ICG	Info. in symbol table helps to add temporary variables info.
(5)	CO	Info. in symbol table used in M/c-dependent optimization by considering add^n and aliased variables info.
(6)	TCG	Generates the code by using the add^n. info. of identifiers.

⇒ Symbol Table Entries :-
Each entry in the symbol table is associated with attributes that support the compiler in different phases. These attributes are:-
(i) Name (ii) Size (iii) Dimension (iv) Type (v) Line of declaration
(vi) Line of usage (vii) Address
→ All the attributes are not of fixed size

(i) Name	(2)(ii)	(iii)	(iv)	v	(vi)	(vii)
RAVI	char	4	1	-	-	-
AGE	int	2	0	-	-	-

↳ Maintain linked list if variable declared/used more than (1) place.

⇒ Limitation of fixing the size :-
(1) If choosen small, it cannot store more variables.
(2) If choosen large, a lot of space is wasted.
→ Size of symbol Table is dynamic in order to allow the Tes in size at compile time.

⇒ **Operations on the Symbol Table :-** it depends on whether the language is block-structured or non-block structured.

⇒ **Block structured vs Non-Block structured :-**

Block structured	Non-Block structured
↳ Here the variables may be re-declared and its scope is within that block.	↳ contains only 1 instance of the variable declaration and its scope is through out the program.
↳ The operations are :- a) Insert b) Set c) Reset d) Lookup	↳ The operations are :- a) Insert b) Lookup.
↳ Eg :- { int i; } scope { int i; } scope	↳ Eg :- { int i; = = } scope

Gate 2017

Q) Access time of the symbol table will be logarithmic if it is implemented by a :-

a) Linear list $O(n)$
b) Search tree $O(\lg_k n)$
c) Hash table $O(1)$
d) None

Various Implementation on Symbol Table :-

Implementation	Insertion time	Look-up time	Disadvantage
① Linear list :-			⇒ Lookup time directly proportional to table size ⇒ Every insertion operation preceeded with lookup operation.
a) Ordered list ↳ Array	$O(n)$	$O(\lg_k n)$	
↳ Linked List	$O(n)$	$O(n)$	
b) Un-ordered list	$O(n)$	$O(n)$	
② Self-organising list	$O(1)$	$O(n)$	⇒ Poor performance when less frequent items are searched.
③ Search Tree	$O(\lg_k n)$	$O(\lg_k n)$	⇒ We have to always keep it balanced.
④ Hash Table	$O(1)$	$O(1)$	⇒ When there are too many collisions, the time complexity Tes to $O(n)$.

Code Optimization :- It is a program transformation technique, w/c tries to improve the code by making it consume less resources i.e., CPU time and memory and delivers high speed.

Rules for CO :-

1. The O/P code must not, in any way, change the meaning of the program.
2. Optimization should ↑es the speed of the program and if possible, the program should demand less number of resources.
3. CO should itself be fast and should not delay the overall compiling process. ∴ the compilation time must be kept reasonable.

⇒ CO Process:-

1. At the beginning, users can change/rearrange the code or use better algorithm to write the code.
2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
3. While producing the target m/c code, the compiler can make use of memory hierarchy and CPU registers.

⇒ When to Optimize?

→ CO is often performed at the end of the development stage since it reduces readability and adds code that is used to ↑es the performance.

⇒ Why to Optimize?

1. Reduce the space consumed and ↑es the speed of compilation.
2. Manually analyzing datasets involves a lot of time, hence we make use of s/w. Usually, manually performing the optimization

is also tedious and is better done using a code optimizer.

3. An optimized code often promotes re-usability.

⇒ Types of CO :-

1. M/c Independent Optimization :- This CO phase attempts to improve the intermediate code to get a better target code as the O/P. The part of the intermediate code w/c is transformed here does not involve any CPU registers or absolute memory locations.

2. M/c Dependent Optimization :- This CO is done after the target code has been generated and when the code is transformed acc. to the target m/c architecture. It involves CPU registers and may have absolute memory references rather than relative references. This CO put efforts to take maximum advantage of Memory hierarchy.

⇒ CO is done in the following different ways:-

IV

1. Compile Time Evaluation :-
   (i) A = 2 * (22.0/7.0) * r
   (ii) x = 12.4, y = x/2.3

2. Variable Propagation :-

Before optimization	After optimization
c = a * b	c = a * b
x = a	x = a
till	till
d = x * b + 4	d = a * b + 4

(III)

(3) Constant Propagation :-

→ If the value of a variable is constant, then replace the variable with the constant. The variable may not always be a constant.

Eg :- ① int k = 2;
if (k) go to L3;
⇓
go to L3 { ∵ k=2, the cond? is true }

Eg ② pi = 3.14
r = 10
Areq = pi * r * r
circle

Note: Value will be substituted at compile time and save the time at run time.

(4) Constant Folding :-

→ Consider an expression a = b op c and the values b and c are constants, then the value of a can be computed at compile time.

Eg :-
```
#define k 5
x = 2 * k
y = k + 5
```
Compile time O/P :-
```
x = 10
y = 10
```

⇒ In constant propagation :- the variable is substituted with its assigned constant
In constant folding :- the variables whose values can be computed at compile time are considered and computed.

(II) (5) Copy propagation :-

a = b+e
b = d+e
c = d+e

→ Extension of constant propagation.
→ After a is assigned to x, use a to replace x till a is assigned again to another variable or value or expression.
→ It helps in reducing the compile time as it reduces copying.
→ In order to eliminate the common subexpression from the statement, we must use a new variable to hold that value.

Copy propagation

Eg :-

Before	After
c = a * b	c = a * b
x = a	delete → x = a
till	till
d = x * b + 4	d = a * b + 4

(I) (6) Common sub-expression elimination :-

a * b and x * b are common sub expression

Eg :-
```
a = x+y+z;
q = p+q;
b = x+y+q;
⇓
t = x+y;
a = t+z;
q = p+q;
b = t+q;
```

(7) Dead code elimination :-

→ Copy propagation often leads to making assignment statements into dead code.
→ A variable is said to be dead if it is never used after its last definition.
→ In order to find the dead variables, a data flow analysis should be done.

Eg :-
```
c = a * b
x = a
till
d = a * b + 4
```
After elimination
```
c = a * b
till
d = a * b + 4
```

(8) Unreachable code elimination :-

→ Firstly control flow graph should be constructed.
→ The block w/c does not have an incoming edge is an unreachable code block.
→ After constant propagation and constant folding, the unreachable branches can be eliminated.

Eg:-

```
#include<iostream.h>
using namespace std;

int main()
{
    int num;
    num=10;
    cout << "Hi";
    return 0;
    cout<<num;
}
```

(unreachable code)

After elimination

```
int main()
{
    int num;
    num=10;
    cout<<"Hi";
    return 0;
}
```

(9) Function Inlining :-

→ Here, a fn^n call is replaced by the body of the fn^n itself.

→ This saves a lot of time in copying all the parameters, storing the return address, etc.

(10) Function Cloning :-

→ Here, a specialized codes for a fn^n are created for different calling parameters. Eg:- fn^n overloading.

(11) Induction Variable & Strength Reduction :-

→ An induction variable is used in the loop for the following kind of assignment i = i + constant. It is a kind of loop optimization technique.

→ Strength reduction means replacing the high strength operator with a low strength.

Eg:-

```
a = a * 16
a = a << 4
```

} Multiplication with power of 2 can be replaced by shift operator w/c is less expensive

Loop Optimization Techniques :-

(1) Code motion or frequency reduction :-

→ The evaluation frequency of expression is reduced.

→ The loop invariant statements are brought out of the loop.

Eg:-

```
a = 200;
while(a>0)
{
    b = x+y;
    if(a%b==0)
    pf("%d", a);
}
```

→ run continuously with same value, so take out of loop.

↓

It does not change the loop.

```
a = 200;
b = x+y;
while(a>0)
{
    if(a%b==0)
    pf("%d", a);
}
```

Further optimization

(2) Loop Jamming :- / Loop Fusion :-

→ Two or more loops are combined in a single loop. It helps in reducing the compile time.

Eg:-

Before Jamming

```
for(int k=0; k<10; k++)
{
    x = k*2;
}
for(int k=0; k<10; k++)
{
    y = k+3;
}
```

After Jamming

```
for(int k=0; k<10; k++)
{
    x = k*2;
    y = k+3;
}
```

③ Loop Unrolling :-

⇒ It helps in optimizing the execution time of the program by reducing the iterations.

⇒ It Tes the program's speed by eliminating the loop control and test instructions.

Eg:-

```
for (int i=0; i<2; i++)
{
    printf ("Hello");
}
```

Before

```
printf ("Hello");
printf ("Hello");
```

After

Where to apply optimization?

① Source program
② Intermediate Code
③ Target code
④ Local optimization
⑤ Regional optimization
⑥ Global optimization
⑦ Interprocedural optimization.

Advantages of CO :-

① Improved performance
② Reduction in code size
③ Increased portability
④ Reduced power consumption
⑤ Improved maintainability

Disadvantage of CO :-

① Increased compilation time.
② Increased complexity.
③ Potential for introducing bugs.
④ Difficulty in assessing the effectiveness.

Peephole Optimization :-

→ It is a type of code optimization performed on a small part of the code.

→ The small set of inst^{n} or small part of code on w/c peephole optimization is performed is known as Peephole / Window.

⇒ Objectives of Peephole optimization :-

① To improve performance.
② To reduce memory footprint.
③ To reduce code size.

⇒ Peephole optimization techniques :-

① Redundant load and store elimination :- In this, redundancy is eliminated.

Eg:-

Initial code	Optimised code
y = x + 5;	y = x + 5;
i = y;	w = y * 3;
z = i;	
w = z * 3;	

② Constant folding :- The code that can be simplified by the user itself.

Eg:-

Initial	After
x = 2 * 3;	x = 6;

③ Strength reduction :- The operators that consume higher execution time are replaced by the operators consuming less execution time.

Eg:-

Before	After
i) y = x * 2;	y = x + x; OR y = x << 1;
ii) y = x / 2;	y = x >> 1;

④ Null sequences / Simplify Algebric Expressions :-

→ Useless operations are deleted.

Eg:-
```
a := a + 0;
a := a * 1;
a := a / 1;
a := a - 0;
```

⑤ Combine operations :- Several operations are replaced by a single equivalent operation.

⑥ Deadcode elimination :- A part of the code w/c can never be executed, eliminating it will improve processing time and reduces set of inst^{n}.

Eg:-
```
int dead (void)
{
    int a = 10;
    int b = 20;
    int c;
    c = a * 10;
    return c;
    b = 30;
    b = b * 10;
    return 0;
}
```
After → deadcode eliminated

Summary of CO :-

CO

M/c or platform dependent Techniques	M/c or platform independent techniques
↳ Peephole optimization	↳ Loop optimization
↳ Inst^{n}. level Ilelism } pipelining	↳ Constant Folding
↳ Data level Ilelism } pipelining	↳ constant propagation
↳ Cache optimization	↳ common subexpression elimination
↳ Redundant resources	

Gate Q) Given the code :-
```
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        if(i%2)
        {
            x += (4*j + 5*i);
            y += (7 + 4*j);
        }
    }
}
```

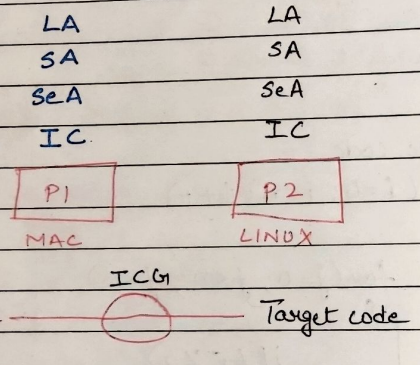
a) Code contains loop invariant computation.
b) " " common sub-expression elimination.
c) " " scope of strength reduction.
d) " " " " deadcode elimination.

W/c of the above statement is false?

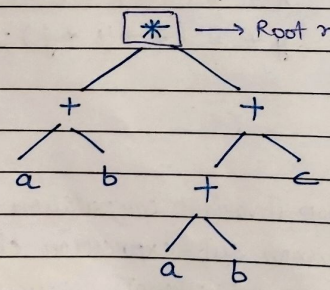
Intermediate Code Generation (ICG) :- There are following ways to represent/generate IC :-

→ M/c independent
→ Abstract syntax Tree → Intermediate representation of source code as a tree structure.
→ Direct Acyclic graph (DAG)
→ Postfix
→ 3-address code

① M/c independent :-

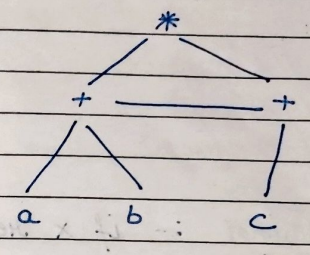


② Abstract syntax Tree :- (a+b) * (a+b+c)



→ A DAG for an expression identifies the common sub expression w/c occurs more than once of the expression.

③ DAG :- (a+b) * (a+b+c)



④ Postfix :- (a+b) * (a+b+c)

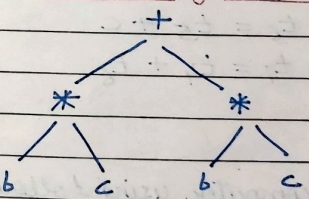
a b + a b c + + *

⑤ 3-add^n code :- (a+b) * (a+b+c)

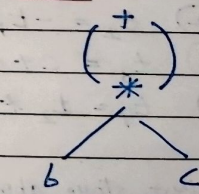
$t_1 = a+b$
$t_2 = a+b$
$t_3 = t_2 + c$
$t_4 = t_1 * t_3$

Q) (b*c) + (b*c)

① Abstract syntax Tree :-



② DAG :-



③ Postfix :-

b c * b c * +

④ 3-add^n code :-

$t_1 = b * c$
$t_2 = b * c$
$t_3 = t_1 + t_2$

67
22/5

3-address code :- Statements in 3-add^{n} code

① Assignment :-
↳ x = y op z // Binary
↳ x = op z // Unary
↳ x = y // Assignment

② Jump :- → Conditional :- if x rel op y goto L
↳ Unconditional goto L

③ Array assignment :- x = y[i] x = y[0]
x[i] = y x[0] = y

④ Pointer address assignment :- ~~address~~ y = &x
x = *y

Various representations of 3-add^{n} code :-

Eg :- -(a+b) * (c+d) + (a+b+c)

① $t_1 = a+b$
② $t_2 = -t_1$
③ $t_3 = c+d$
④ $t_4 = t_2 * t_3$
⑤ $t_5 = a+b$
⑥ $t_6 = t_5 + c$
⑦ $t_7 = t_4 + t_6$

⇒ Now, store above statements in computer using following format :-

Quadruples

	Opr	op1	op2	result
①	+	a	b	t_1
②	-	t_1	(Null)	t_2
③	+	c	d	t_3
④	*	t_2	t_3	t_4
⑤	+	a	b	t_5
⑥	+	t_5	c	t_6
⑦	+	t_4	t_6	t_7

Triple

	Opr	op1	op2
①	+	a	b
②	-	①	
③	+	c	d
④	*	②	③
⑤	+	a	b
⑥	+	⑤	c
⑦	+	④	⑥

Indirect triples

Inst^{n}.	
100	①
101	②
102	③
103	④
104	⑤
105	⑥
106	⑦

	Quadruples	Triple	Indirect triples
Adv :-	Statements can be moved around.	Space is not wasted.	Statements can be moved.
Disadv :-	Too much of space is wasted.	Statements cannot be moved.	Two-way Memory access.

Q) -(a*b) + (c*d+e)

① $t_1 = a*b$
② $t_2 = -t_1$
③ $t_3 = c*d$
④ $t_4 = t_3 + e$
⑤ $t_5 = t_2 + t_4$

Advantage of ICG :-
↳ Easier to implement
↳ Facilitates CO
↳ Platform independence
↳ Code reuse
↳ Easier debugging

Disadv. of ICG :-
↳ Increased compilation time
↳ Additional memory usage
↳ Increased complexity
↳ Reduced performance

Errors in Lexical Analysis & their Recovery :-

1. Long identifiers. [create table std_detail_car_prev_ ...].
2. Numerical literals that are too long. [lies within range such as for int = 65535 on 32-bit]
3. Spelling mistakes.
4. Ill-formed numeric literals. [int A = "$123";]
5. I/P characters that are not in the source language.

⇒ Error Recovery :-

1. Delete :- Unknown characters are deleted. It is also known as Panic Mode Recovery.
   Eg :- "Charr" correct as "Char".
2. Insert :- An extra or missing character is inserted to form a meaningful token.
   Eg :- "Cha" —('r' add)→ "Char"
3. Transpose :- Based on certain rules we can transpose 2 characters.
   Eg :- "Whiel ⟶ "While"
4. Replace :- Based on replacing one character by another.
   Eg :- "Charr" ⟶ "Char".

⇒ Syntactic Phase Errors :-

1. Errors in structure
2. Missing operator
3. Misspelled keywords
4. Unbalanced parenthesis

⇒ Error Recovery for syntactic phase error :-

1. Panic mode recovery :- In this successive characters from the I/P are removed one at a time until a valid set of tokens is found.
   → It is easy to implement and guarantees not to go into an infinite loop.
   → Disadv :- A considerable amt of I/P is skipped w/o checking it for additional errors.
2. Statement mode recovery :- In this, when a parser encounters an error, it perform the necessary action/correction on the remaining I/P so that the rest of the I/P statement allows the parser to parse ahead.
   → The correction includes :- deletion of extra ;
   replacing the , with ;
   inserting missing ;
   → Disadv :- it finds it difficult to handle situations where the actual error occurred before pointing of detection.
3. Error production :- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
   → Disadv :- it is difficult to maintain.
4. Global correction :- The parser examines the whole program and tries to find out the closet match for it w/c is error-free.
   → Due to high time & space complexity, practically it is not implemented.

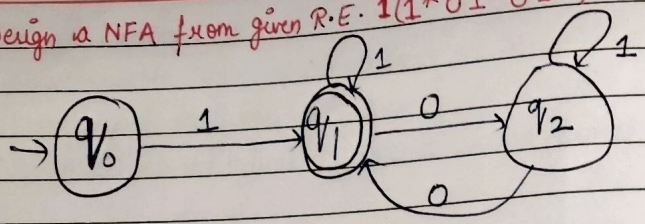
⇒ Semantic Errors :-

1. Incompatible type of operands.
2. Undeclared variables.
3. Not matching of actual arguments with a formal one.

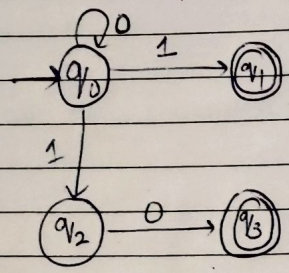
⇒ Error recovery for semantic errors :-

1. Undeclared identifier :- To recover from this a symbol table entry for the corresponding identifier is made.
2. Data types incompatible :- Automatic type conversion is done by the compiler.

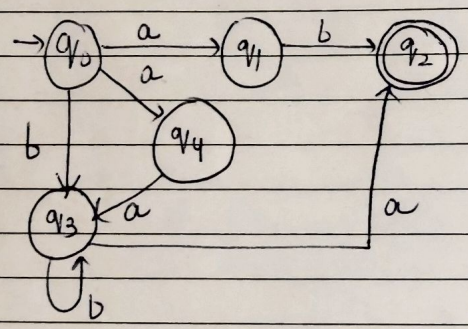
Q) Design a NFA from given R.E. $1(1^*01^*01^*)^*$



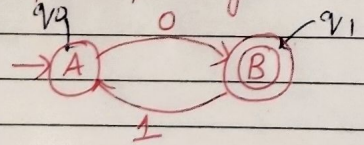
Q) Construct the FA for R.E: $0^*1 + 10$.



Q) Design a FA from the given R.E: $[ab + (b + aa)b^*a]$



Q) Find RE for the following DFA :-



Ans = $0(10)^*$

$q_0 = \wedge + q_1 \cdot 1$ — (1)

$q_1 = q_0 \cdot 0$ — (2)

Put (2) in (1)

$q_0 = \wedge + q_0 \cdot 0 \cdot 1$

$= (01)^* q_0$

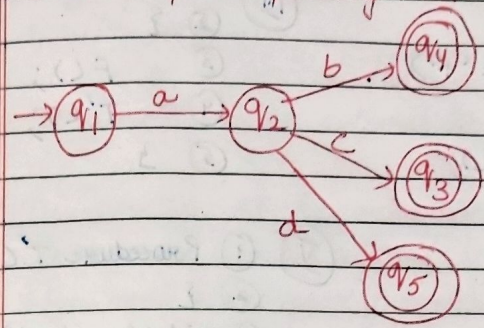
$\Rightarrow (01)^* 0$

$q_0 = \wedge + 10q_0$

$= (10)^*$

$q_1 = 0(10)^*$

Q) Find R.E. for the following DFA :- Ans = $a(b + c + d)$



Q) LL(1) Grammar :-

	First()	Follow()	
a) $S \to aSA/\epsilon$	$\{a, \epsilon\}$	$\{c, \$\}$	
$A \to c/\epsilon$	$\{c, \epsilon\}$	$\{c, \$\}$	✗ NOT
b) $S \to aAa/\epsilon$	$\{a, \epsilon\}$	$\{a, \$\}$	✗ NOT
$A \to abS/\epsilon$	$\{a, \epsilon\}$	$\{a\}$	
c) $S \to iEtSS'/a$	$\{i, a\}$	$\{e, \$\}$	
$S' \to eS/\epsilon$	$\{e, \epsilon\}$	$\{e, \$\}$	✗ NOT
$E \to b$	$\{b\}$	$\{t\}$	

Q) Write down the algo. using Recursive procedures to implement the following grammar :-

$E \to TE'$

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

$T \to FT'$

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

$F \to id/(E)$

(I)

(1) Procedure E()

(2) {

(3) T();

(4) E'();

(5) }

(II)
```
Procedure E'()
{
    if (l == '+')
    then
    advance();
    T();
    E'();
}
```

(III)
```
Procedure T()
{
    F();
    T'();
}
```

(IV)
```
Procedure T'()
{
    if (l == '*')
    then
    advance();
    F();
    T'();
}
```

(VI)
```
Procedure F()
{
    if (l == 'id')
    then
    advance();
    else if (l == '(')
    then
    advance();
    E();
    if (l == ')')
    advance();
    else
        printf ("error");
    else
        printf ("error");
}
```

Advantages of Recursive Descent Parser :-

1. It mirrors grammar, making it easy to understand and maintain.
2. It efficiently predicts parsing paths.
3. Programmers have full control and can tailor parsers to specific languages.
4. It can be optimized for specific languages.
5. It doesn't rely on external tools or generators.

Disadvantages of Recursive Descent Parser :-

1. It is most effective with LL(1) grammars, requiring adjustments for complex grammars.
2. Handling left recursion, left factoring, and some constructs can be challenging.
3. Custom error handling is needed, making error recovery complex.
4. Efficiency may be lower for deeply nested or complex syntax.
5. Development is more manual and time consuming compared to parser generators.

TDP with Full Backtracking [Brute Force Method] :-

⇒ Whenever a non-terminal is expanding first time, then go with the first alternative and compare with the I/P string.

⇒ If it does not matches, go for the second alternative and compare with I/P string.

⇒ If still, it does not matches go with the 3rd alternative and continue with each and every alternative, and so on.

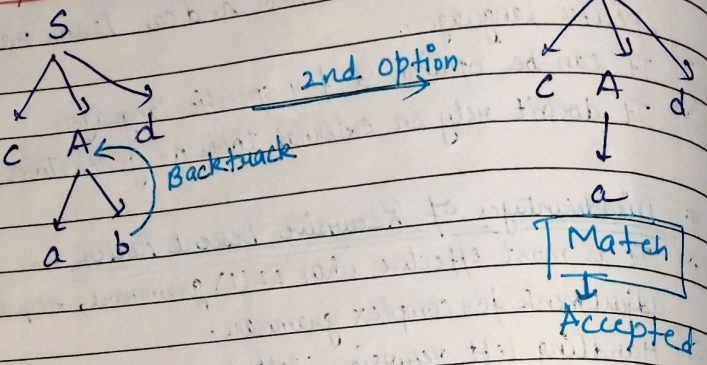
⇒ If the matching occurs for at least one alternative, then the parsing is successful, otherwise parsing fails.

Eg:- : $S \rightarrow cAd$ $\quad W_1 = cad$

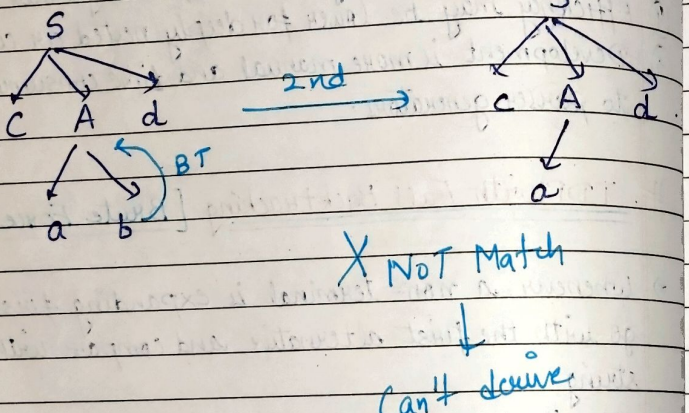
$A \rightarrow ab/a$

$W_2 = cada$

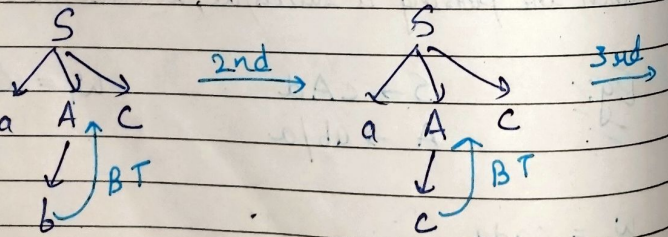
For W_1 = cad

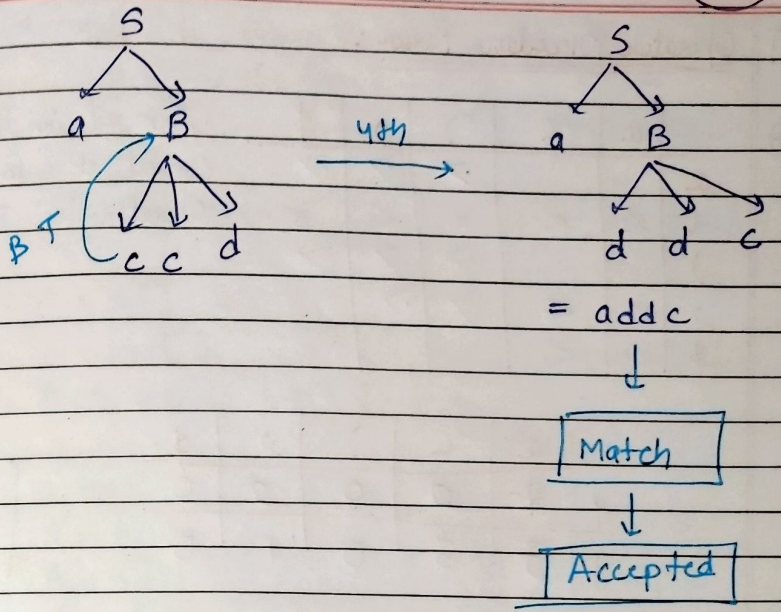


For W_2 = cada



Q) S → aAc / aB W = addc
A → b / c
B → ccd / ddc





Important pts :-

(1) TDP may be constructed for both left factor and non-left factor grammar.
(2) If the grammar is N.D.G. then we use brute technique and if the grammar is D.G., then we go with the predictive parser.

Drawbacks of Brute force Technique :-

(1) Brute force requires lot of BT → $O(2^n)$.
(2) It is very costly & reduces the performance of parser.
(3) Debugging is very difficult.

Operator Precedance Parser :-

Q) Given relation table. Convert it into Operator fun^{n} table.

	a	(	)	?	$
a			⋗	⋗	⋗
(	⋖	⋖	=	⋖	
)			⋗	⋗	⋗
?	⋖	⋖	⋗	⋗	
$	⋖	⋖			

	a	(	)	?	$
f	2	0	2	2	0
g	3	3	0	1	0

Q) Convert AG into UAG :-

R → R+R
/R·R
/R*
/a
/b

⇒

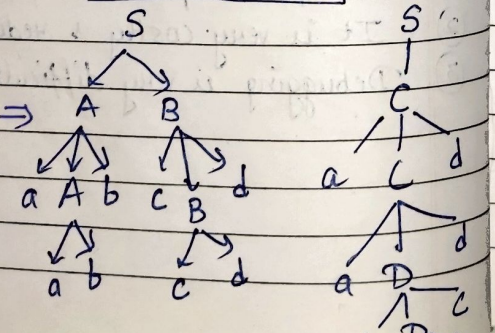
E → E+T/T
T → T·F/F
F → F*/G
G → a/b

⇒ E → E+T/T
T → T·F/F
F → F*/a/b

Q) Check the ambiguity :-

S → AB/C
A → aAb/ab
B → cBd/cd
C → aCd/aDd
D → bDc/bc

W = aabbccdd



Q) Eliminate left factoring :-

S → a/ab/abc/abcd

a → common prefix

S → aS'
S' → b/bc/bcd/ϵ

b → common prefix

S → aS'
S' → bA/ϵ
A → c/cd/ϵ

c → common prefix

S → aS'
S' → bA/ϵ
A → cB/ϵ
B → d/ϵ

Q) (i) Find first() and follow() :-

	first()	follow()
S → aXZh	{a}	{$}
X → cY	{c}	{g, f, h}
Y → bY/ϵ	{b, ϵ}	{g, f, h}
Z → EF	{g, f, ϵ}	{h}
E → g/ϵ	{g, ϵ}	{f, h}
F → f/ϵ	{f, ϵ}	{h}

Q) (ii)

S → X
X → aY/Xd
Y → b
C → t

left Recursion →

first()		follow()
{a}	S → X	{$}
{a}	X → aYX'	{$}
{d, ϵ}	X' → dX'/ϵ	{$}
{b}	Y → b	{$, d}
{t}	C → t	{$}

N.A.

	first()	follow()
Q) $S \to (F) / g$	{(, g}	{$,), ;}
$F \to SF'$	{(, g}	{)}
$F' \to ;SF' / \epsilon$	{;, ϵ}	{)}

	first()	follow()
Q) $S \to XCY / CbY / Ya$	{d, g, h, ϵ, b, a}	{$}
$X \to da / YC$	{d, g, ϵ, h}	{h, g, $}
$Y \to g / \epsilon$	{g, ϵ}	{$, a, h, g}
$C \to h / \epsilon$	{h, ϵ}	{g, $, b, h}

Gate 2024

Q) Consider the following syntax-directed definition (SDD):

$S \to DHTU$ {S.val = D.val + H.val + T.val + U.val;}

$D \to$ "M" D_1 {D.val = 5 + D_1.val;}

$D \to \epsilon$ {D.val = −5;}

$H \to$ "L" H_1 {H.val = 5 * 10 + H_1.val;}

$H \to \epsilon$ {H.val = −10;}

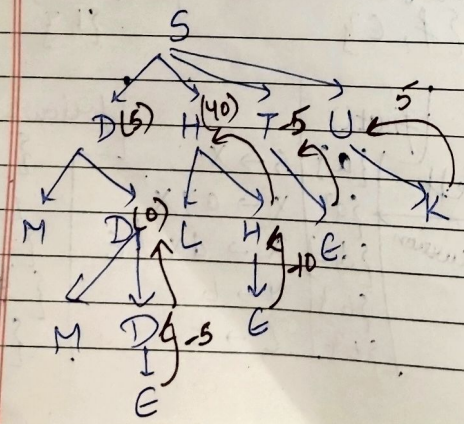
$T \to$ "C" T_1 {T.val = 5 * 100 + T_1.val;}

$T \to \epsilon$ {T.val = −5;}

$U \to$ "K" {U.val = 5;}

Given "MMLK" as the I/P, w/c one of the following options is the correct value computed by the SDD? [S.val]?

a) 45 b) 50 c) 55 d) 65



⇒ 5 + 40 − 5 + 5

⇒ 45

Gate 2024

Q) Consider the following augmented grammar, w/c is to be parsed with a SLR parser. The set of terminals is {a, b, c, d, #, @}

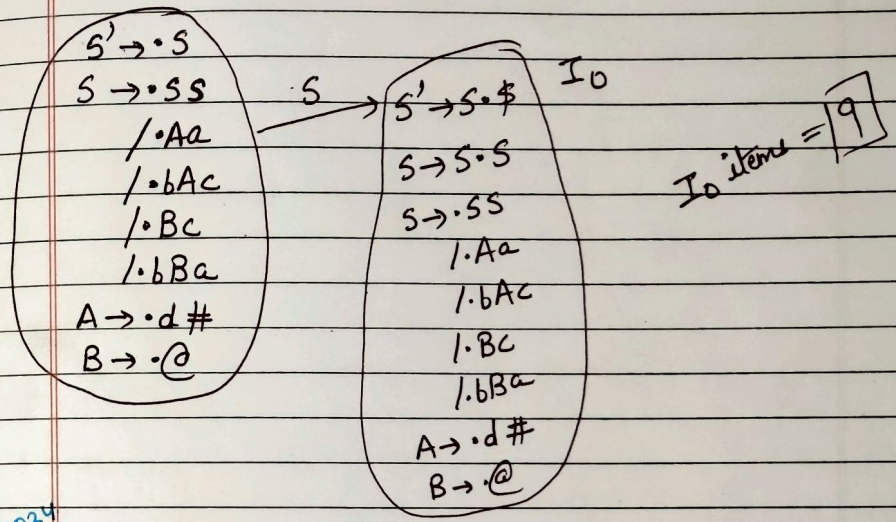
$S' \to S$

$S \to SS / Aa / bAc / Bc / bBa$

$A \to d\#$

$B \to @$

Let I_0 = Closure ({S' → ·S}). The no. of items in the set Goto (I_0, S) is 9.



Q) Gate 2024

(P)	L.A	(1)	AST
(Q)	S.A.	(2)	Token
(R)	I.C.G.	(3)	P.T.
(S)	C.O.	(4)	Constant folding

P → 2, Q → 3, R → 1, S → 4