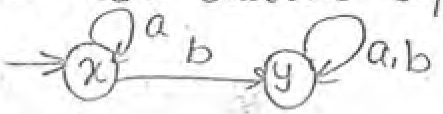
Dated 25.Oct.10

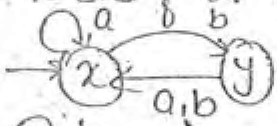
COMPILER

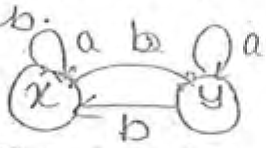
Day 1

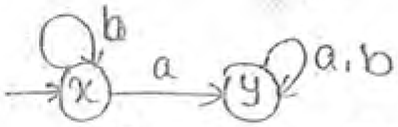
Q How many possible finite automata's are there, where x is always initial state over the alphabet x and y or a and b.

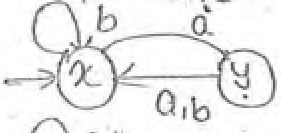
Sol^n

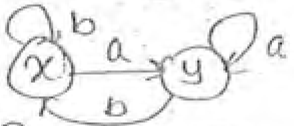
1. 

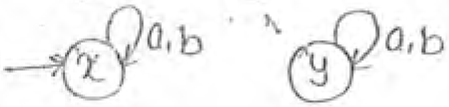
5. 

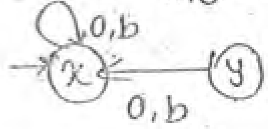
9. 

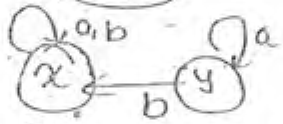
2. 

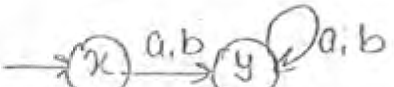
6. 

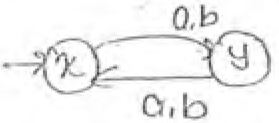
10. 

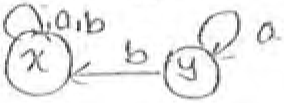
3. 

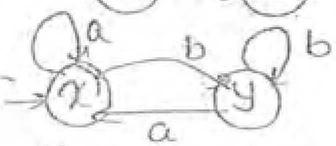
7. 

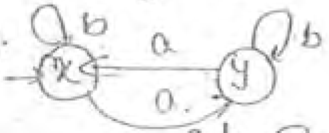
11. 

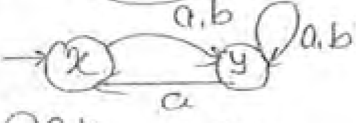
4. 

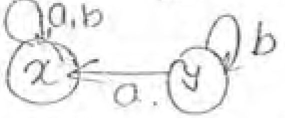
8. 

12. 

13. 

14. 

15. 

16. 

find - On the basis of no of final state

final 2 (x & y) = 16

1 (x (or) y) = 16 + 16 = 32

0 (no final) = 16

64 Ans

→ Consideration of no of possible finite automata's can be done as:-

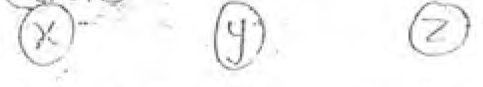
		a	b	
2	x	2	2	→ 4
2	y	2	2	→ 4
4				

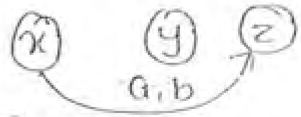
4 × 4 = 16

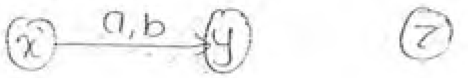
16×4 = 64 Ans

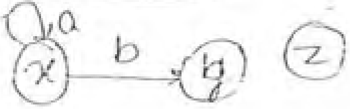
QN. How many possible finite automata's with three states x, y, z over the alphabet a and b. where x is always initial states

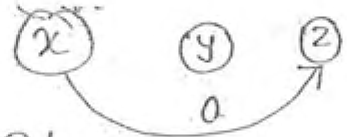
Sol^n

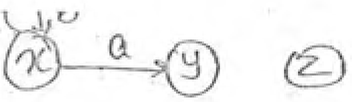
1. 

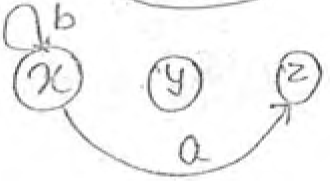
3. 

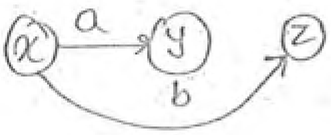
2. 

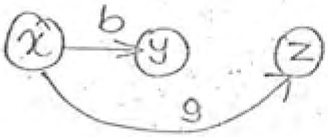
4. 

5. 

6. 

7. 

8. 

9. 

		a	b	
2	x	3	3	⇒ 9
2	y	3	3	⇒ 9
2	z	3	3	⇒ 9
8				

9, 9, 9 → 729

8 → 5832 Ans

final
0 → 1
2 → 3
1 → 3
3 → [illegible]

QN How many possible finite automata's are these states x, y & z over the alphabet a and b.

Soln x is initial → 5832
y is initial → 5832
z is initial → 5832
= 3×5832
= 17496

QN How many possible finite automatas are there with three state x y and z, over the alphabet a, b and c, where x is both initial and final.

	a	b	c
x	3	3	3
y	3	3	3
z	3	3	3

$9^3 \Rightarrow$ 19,683 Ans

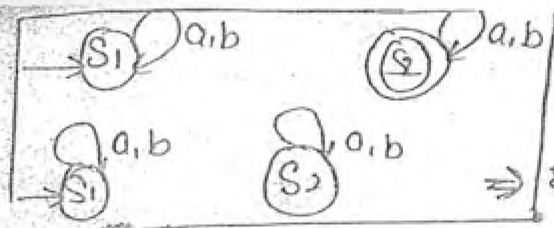
QN How many possible finite automatas are there with state x any, where x is alway initial state over the alphabet a, b that accepts empty language.

Soln

		a	b
2	x	2	2
2	y	2	2
4			

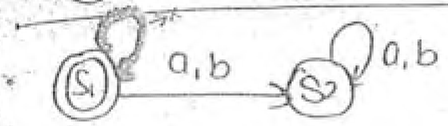
$2^4 \Rightarrow 16$

16×4 = 64



⇒ Empty language. (final state is not reachable)

⇒ Empty language (not any final state)



=) 1, not empty language.

possibilities of finding the automatas. which accepts empty language.

→ No final state = 16 (B as x and B as y)

→ final state are not reachable = 4

20 possibilities

If there are two states then-

2 — 16 (both are final) x

1 → 16 (x is initial) x

16 (4) ✓

0 → 16 (Empty language) ✓

QN How many possible finite automates are there with two states x and y. where x is alway initial state with alphabet a and b, that accepts everything.

Soln 2 → 16 ✓

1 → x → 16 (4) ✓ => 20

y → 16 x

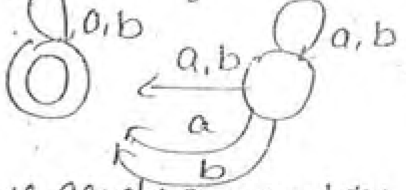
0 → 16 X

Total possibilities = 20

If y is final state ⇒

(y not accepting ε). x

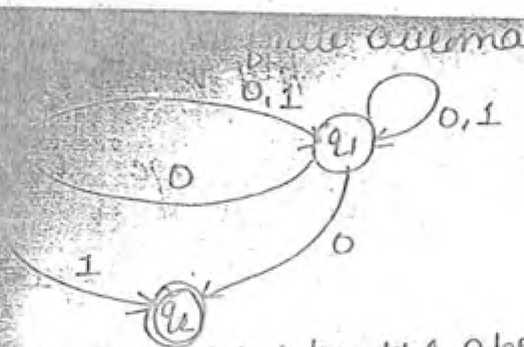
If x is final then-



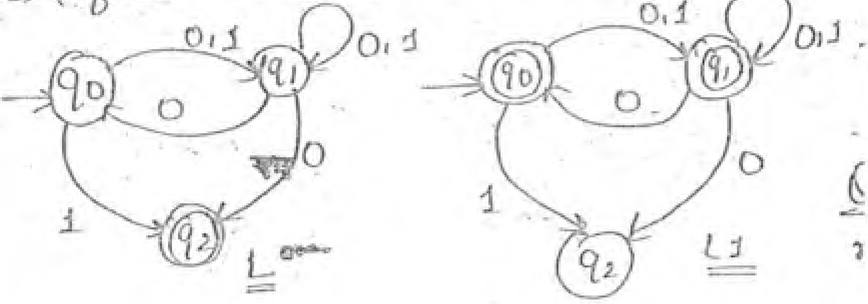
⇒ 4

Note:- If 20 Dfa's are there which are accepting empty language then 20 dfa's should be there which will accepts everything. because of complementation rule.

...nite automata →



...age accepted by the above finite automata and ...ge accepted by the above finite automata by ...al and non final states, then which one of the ... true:-



... 1)* − L

... = (0+1)*

$L_1 \subseteq (0+1)^*$

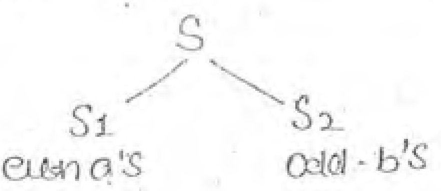
$L_1 = L$

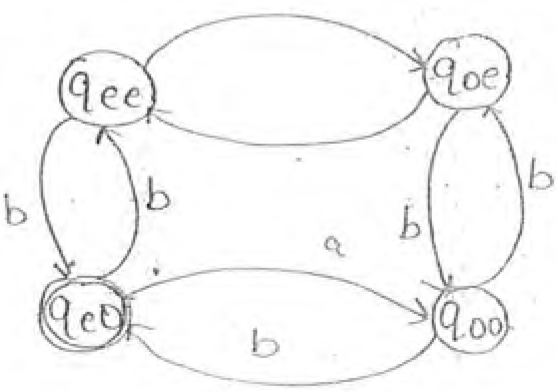
...If it is DFA then ⇒ $(0+1)^* - L$ (Remove L from Σ).

Note:- In the case of DFA we will always get complimented finite automata, but in the case of NFA, we will not always get complimented finite automata (manual checking is only solution).

QN Construct finite automata that accepts all strings of a's and b's where no of a's in given string is even and no of b's in the give string is odd.

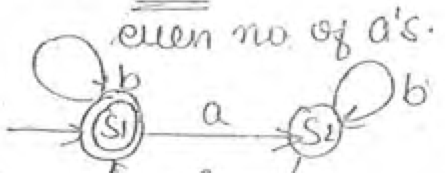
soln



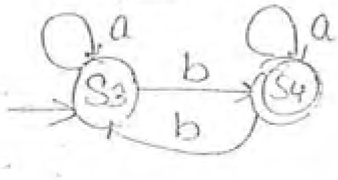


P1

even no of a's.



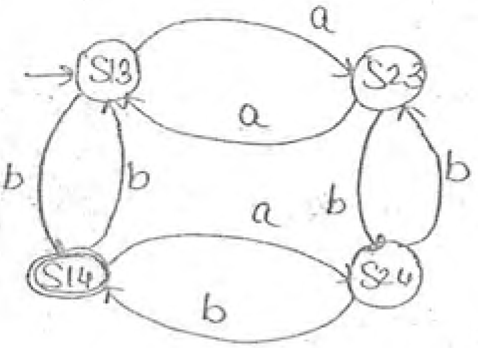
P2 (odd no of b's)



By P1 & P2

$S13 \xrightarrow{a} (S1,a) \cup (S3,a) = S2S3$

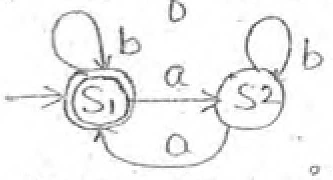
$S13 \xrightarrow{b} (S1,b) \cup (S3,b) = S2S4$



final state ⇒ where both the finals are there.

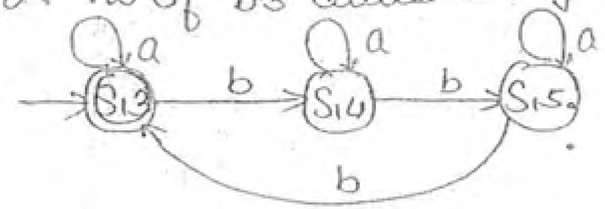
Q14 Construct FA which accepts all strings of a's and b's in which no of a's are divisible by 2 and no of b's are divisible by 3.

P1: no of a's divisible by 2.



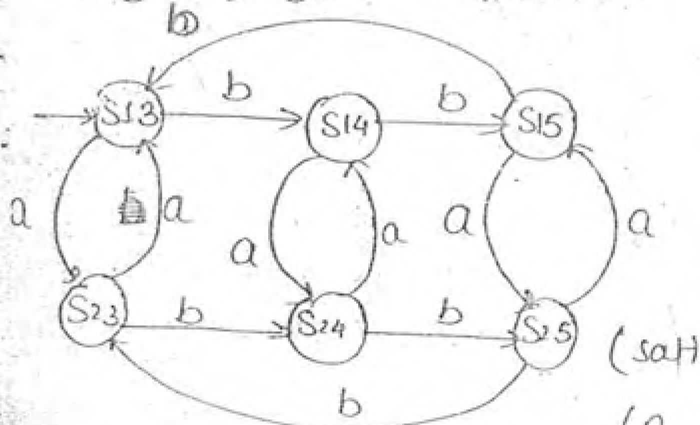
How many state = 2×3 = 6 states.

P2: no of b's divisible by 3.



$S13 \xrightarrow{a} S23$ $S15 \xrightarrow{a} S25$

$S13 \xrightarrow{b} S14$ $S15 \xrightarrow{b} S13$



final states

$P_1 \cap P_2 = S13$

$P_1 \cup P_2 = S13, S14, S15, S23$

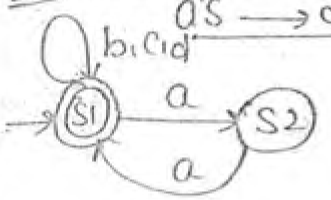
$P_1 - P_2 = S14,, S15$
(satisfying 1st but not second one)

$(P_2 - P_1) = S23, S24, S25$

Q4 Find the minimum no of state required to construct minimal dfa which will accept all strings of a's, b's and c's; d's where the no of a's divisible by 2, no of b's not divisible by 3, no of c's not divisible by 5, no of c's not divisible by 7. How many minimum no of states required.

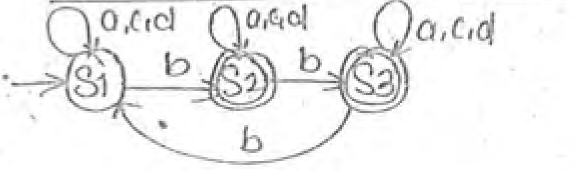
Sol^n P1

a's → divisible → 2



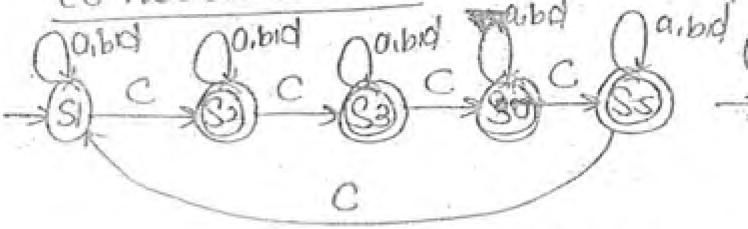
P2

b's → not divisible by 3



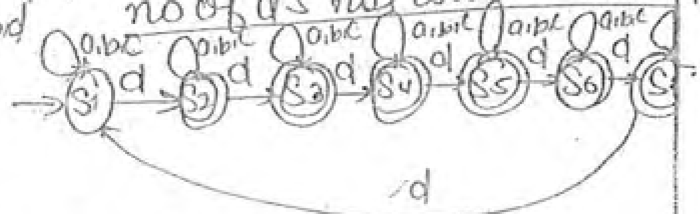
P3

c's not divisible - 5



P4

no of d's not divisible by 7

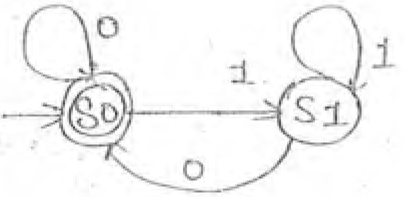


Minimum no of states = 2*3*5*7 = 210 Ans

Q4.2 Construct finite automata that accepts all binary number which are divisible by 2.

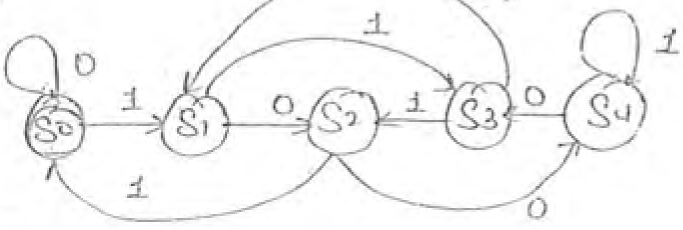
Sol^n - If initial and final states are same then this method can be utilized ⇒

	0	1
→ S0*	S0	S1
S1	S0	S1



⇒ all binary no divisible by 5 - (Remainders = 0,1,2,3,4)

	0	1
→ S0*	S0	S1
S1	S2	S3
S2	S4	S0
S3	S1	S2
S4	S3	S4



→ all ternary no divisible by 7 -

	0	1	2
→S0*	S0	S1	S2
S1	S3	S4	S5
S2	S6	S0	S1
S3	S2	S3	S4
S4	S5	S6	S0
S5	S1	S2	S3
S6	S4	S5	S6

Binary or ternary no metters

always contain 7 states

Note :- The minimum no of state required to construct finite automata that accepts all base m numbers, which are divisible by n, contain n-states.

→ all binary no's divisible by 5 and starting with 0.

	0	1
→ S	S0	D
* S0	S0	S1
S1	S2	S3
S2	S4	S0
S3	S1	S2
S4	S3	S4
D	D	D

⇒ start with 0

divisible by '5'

⇒ start with '1'

⇒ 7 states

→ all binary no divisible by 9 and starts with 1

	0	1
→ S	D	S1
* S0	S0	S1
S1	S2	S3
S2	S4	S5
S3	S6	S7
S4	S8	S0
S5	S10	S2
S6	S3	S4
S7	S5	S6
S8	S7	S8
D	D	D

⇒ starting with 1

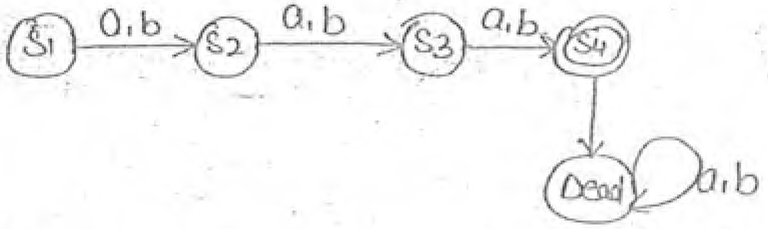
Divisible by 9

⇒ 9+2 = 11 states

⇒ starting with 0

QN Construct a FA that accepts all strings of a's and b's, where the length of the string of exactly 3.

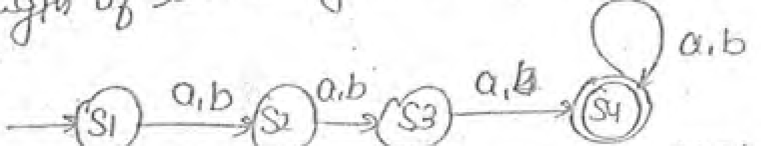
Soln



=) 3+2 = 5 states

Note:- The minimal finite automata that accepts all strings of a's and b's, where the length of string is n exactly contains (n+2) states.

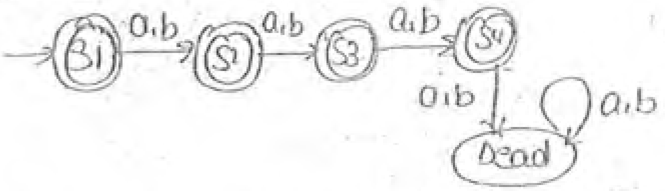
QN Construct minimal FA that accepts all strings of a's and b's where the length of the string is atleast 3.



Note:- The minimal finite automata that accepts all string's of a's and b's, where the length of string atleast n contains (n+1) states.

QN FA, accepts all strings of a's and b's where the length of string is atmost three.

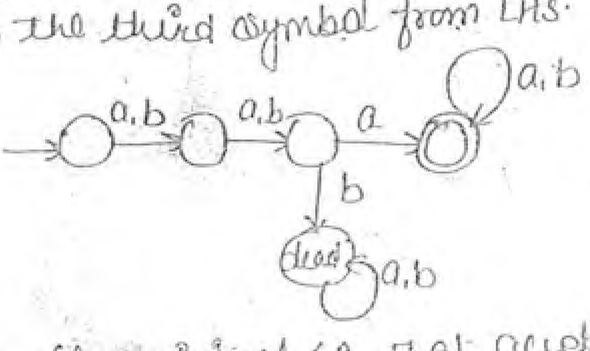
Soln



= 3+2 = 5 states

Note:- Where the length of strings are atmost n. contain n+2 states

QN FA, that accepts all strings of a's and b's, where each string contain 'a' as the third symbol from LHS.



=) 3+2 = 5 states

Note:- The minimal FA that accepts all strings of a's and b's where n'th symbol from L.H.S. contain n+2 states.

Note:- The minimum FA which accepts all the strings of 0's and b's, where n'th symbol from the right hand side is b contains 2^n states.

stated .Oct.10

lec-2

$L_1 = \{a^n \mid n \geq 1\}$ → Regular

$L_2 = \{a^n b^n \mid n \geq 1\}$ → CFL (Regular + 1 stack)

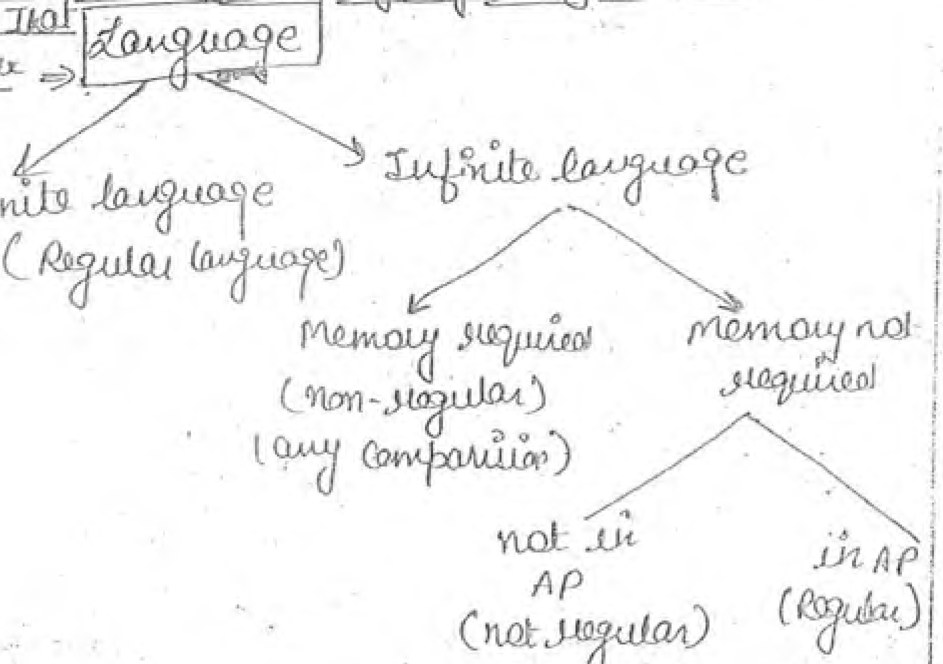
$L_3 = \{a^n b^n c^n \mid n \geq 1\}$ → CSL

$L_4 = \{a^m b^n \mid m \neq n,\ m,n \geq 1\}$ → CFL

$L_5 = \{a^l b^m c^n \mid l \neq m \text{ or } m \neq n\}$ → CFL
$l,m,n \geq 1$

How can we say that what the language is given as:-

(If we want to check that given language is regular or not) ⇒



Examples

$L = \{a^m b^m \mid m \leq 10{,}000\}$

Regular language (just because of finiteness)

$L = \{a^n \mid n \geq 1\}$ (Regular)

Here we are using the concept of generation of series.

$L = \{a^{n!} \mid n \geq 1\}$

Theorem:- For a given problem if you can construct the C pgm, then it can surely be solved by T.M.

non-regular (not in AP)

$L = \{a^{2^n} \mid n \geq 1\}$ = not regular (Not in AP)

④ $L = \{a^{2n} \mid n \geq 1\}$ ⇒ even no of a's

Regular language (in AP)

5 $L = \{a^{n^2} \mid n \geq 1\}$

⇒ not regular (not in AP)

6 $L = \{a^n b^n \mid n \geq 1\}$

⇒ not regular (memory required)

7 $L = \{w w^R \mid n \geq 1\}$

⇒ non regular (memory required)

8 $L = \{a^i b^j \mid \gcd(i,j) = 1\}$

⇒ non regular (memory required).

∮ $\boxed{\text{Regular} \wedge \text{CFL} \Rightarrow \text{CFL}}$ (Intersection of lower and higher language will always go to higher language).

$$(a+b)^* \wedge a^n b^n = a^n b^n$$

∮ $\boxed{\text{CFL} \wedge \text{CFL} = \underset{(\text{CSL})}{\text{not CFL}}}$ → (need not be)

$$\begin{tikzcd}
\underline{a^l b^l} c^m \arrow[dr] & \wedge & a^m \underline{b^n c^n} \arrow[dl] \\
& \underline{\underline{a^l b^l c^l}} \text{ (not CFL)} &
\end{tikzcd}$$

∮ $a^n b^n c^n = \text{CSL}$ ⇒ Compliment of CSL is need not be CSL.

$$\begin{tikzcd}
a^n b^n c^n \arrow[d, Rightarrow] \\
(a^n b^n c^n)' \arrow[d, Rightarrow] \\
\underline{\text{CFL}}
\end{tikzcd}$$

⇒ Compliment of CFL, need not be CFL, it can be CSL. (CFL's are not closed under complimentation).

<u></u>

Note:- ① Intersection → CFL ⇒ need not be CFL or (CSL)

② Compliment of CFL ⇒ need not be CFL or (CSL)

③ Intersection of Regular and CFL is always CFL.

Give the regular expression that derives all strings of a's b's, where each string begin with a and end with b.

$\boxed{R = \{a.(a+b)^*.b\}}$ Ans ⇒ Concatenation order is important
$a+b = b+a$ (order can be changed)

QH Regular expression, where the first and last symbols are different

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

QH Regular expression, that derives all string's of a's and b's, where each string starting and ending symbols are same.

⇒ $a(a+b)^*a + b(a+b)^*b + \lambda + a + b$

QH Give the Regular expression that derives all strings of a's and b's, where all strings contains abb as substring.

⇒ $(a+b)^* abb (a+b)^*$

QH Regular expression, where the length of string is exactly three.

⇒ $(a+b)(a+b)(a+b) = (a+b)^3 \Rightarrow (a+b)^n$

QH Regular expression, where the length of string is atleast 3

⇒ $(a+b)(a+b)(a+b)(a+b)^*$ ⇒ more effecient

⇓ (or)

$(a+b)^*(a+b)^3$ ⇒ more effecient

⇓ (or)

$(a+b)^*(a+b)^3(a+b)^*$ ⇒ more correct but not effecient

QH Regular expression, where the length of string is atmost 3.

$= \lambda + (a+b) + (a+b)(a+b) + (a+b)(a+b)(a+b) = \underline{(a+b+\lambda)^3}$

QH Regular expression, where the length of string is even.

$= ((a+b)^2)^*$

QH. Odd length—

$(a+b)((a+b)^2)^*$ or $((a+b)^2)^*(a+b)$

QH Regular expression, that where each string starts with a and not having two consecutive b's.

$= (a+ab)^+$ (or) $a(a+ba)^*(\lambda+b)$

$(a+ab)^*$

QN Regular expression, where each string does not contain 2 consecutive a's and b's

Soln $= (b+\epsilon)(a+b)^*(a+\epsilon)$ or $(a+\epsilon)(ba)^*(b+\epsilon)$

QN Regular expression, where each string contain exactly 2 a's.

$= b^* a b^* a b^*$

QN Regular expression, where each string contain atmost two a's.

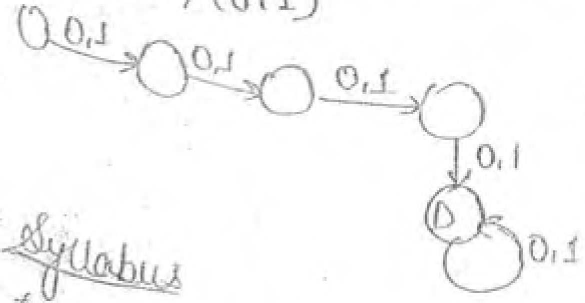
$= b^* + b^* a b^* + b^* a b^* a b^*$

$= b^* (a+\epsilon) b^* (a+\epsilon) b^*$

QN Find the minimal states of dfa that accepts described by the R.E. $= (0+1)(0+1)(0+1)(0+1)$ .... n-times

$(0+1)(0+1)(0+1)$

=



If $n=3$, ⇒ 5 states ⇒ $(3+2)$

If $n=n$ then $\underline{(n+2)}$ states

* Syllabus

1. Lexical analyses
2. Parsing ———————— *
3. Syntax directed Translation — *
4. Intermediate code generation
5. Code Optimization

} ⇒ Imp

| References |

→ Compiler Technique & Aho Ullmen $$ Ravisethi

INTRODUCTION

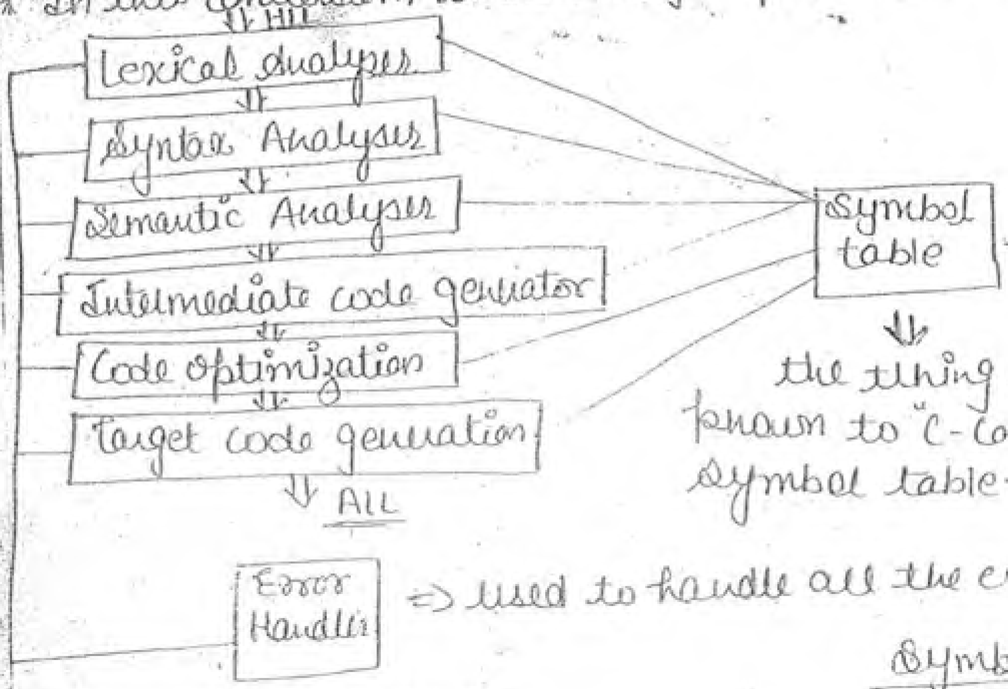
Compiler

⇓ HLL

Assembly level language

* Compiler is a Converter, that can convert High level language into Assembly level language.

* In this conversion, we are using 6-phases, which are as follows:-



* C-Compiler knows everything about C-language.

the thing that are not known to "C-Compiler" is stored in symbol table.

Error Handler ⇒ used to handle all the errors made by user.

Exp:- $x = a+b*60.5$

⇓

Lexical Analyser

⇓

$(id_1, 1) = (id_2, 2) + (id_3, 3) * 60.5$

7 tokens

$id_1 = id_2 + id_3 * 60.5$

⇓

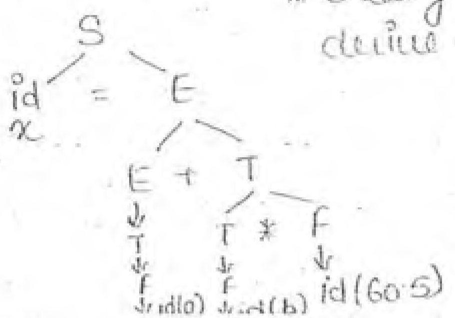
Syntax Analysis

$S \rightarrow id = E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$



Symbol table

1	int	x
2	int	a
3	int	b

x, a, b = identifiers

* C language compiler, uses CFG, to derive all the arithmetic expression.

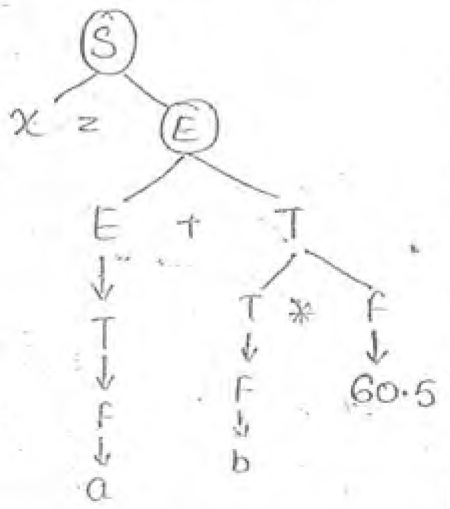
a+b=c = syntax error

⇓

CFG can't generate this

* Type checking will never be take by the syntax ana will done by semantic analyser.

Semantic Analyser (Type checking)



- How can we multiply int to floating point number?
- for this semantic analyser uses the Implicit conversion by float to int (60.5) = 60

* Type mismatch type errors are given by semantic analyser.

---

x = a+b*60 (CFG take care of priorities itself)

Intermediate Operations or code

(t1, t2) = temporary variables

$t_1 = b*60$

$t_2 = a+t_1$

$x = t_2$

⇓

Code Optimization

⇓

$t_1 = b*60$

$x = a+t_1$

⇓

Target Code

⇓

```
MOV b, R1 (b to R1).
MUL R1* 60 (R1 = R1*60)
MOV a, R2 (a to R2)
ADD R1, R2 (R1 = R1+R2)
MOV R1, x (R1 to x)
```

L.A.
Syntax
Semantic
I.C.G. } front end

Code Optimiz. ⇒ Optional phase

Target Code ⇒ Back end

Front end = Depends upon source language

Back end = Depends on processor

* In order to achieve the portability, we seprate the phases of analyser.
for the same source code, we can generate different assembly language

sets, based upon type of processors.

Types of Compiler

① Single pass Compiler

② Multi pass ~~m~~ Compiler

① Single pass Compiler :- all the 6-modules at a time in memory.

Disadvantage :-

① Wastage of space.

Advantage

① No Divide and conquer (less time required).

② Multi pass Compiler :- Front end and back end are placing seprately in memory.

Disadvantage

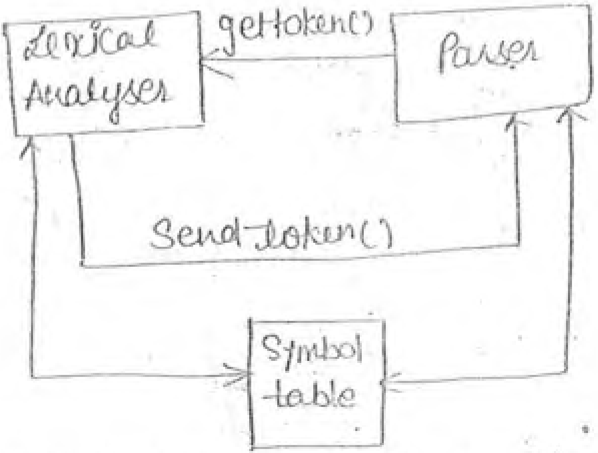
① More time required (Divide and conquer)

Advantage

① Less space required.

Chapter No 1

Lexical Analyser



× Initially the I/P is given to parser. parser calls lexical analyser for tokens

→ Lexical Analyser :- ① It will read the given I/P strings and divides the string into some meaningful groups or words which are called as tokens

② It will remove the comment lines in the given C pgm.

3) It will eliminate white space characters, in the source c
White space charalters → blank(space), tab, newline characte

4) It will help to provide error messages. It scan each and eve
line of source code. The line number is also provided by the
lexical analyser.

QN Find the no. of tokens in the following C pgm:-

```
int max(i,j)
 int i, j;
 \* return max of i & j *\
 {
   return i>j ? i : j;
 }
```

Soln

```
  (1)    (2)   (3)(4)(5)(6)(7)
|int| |max||(|i|,|j|)|
  (8)  (9)(10)(11)(12)
|int||i|,|j|;|
|{|(13)
  |return||i|>|j|?|i|:|j|;|
    (14)  (15)(16)(17)(18)(19)(20)(21)(22)
|}|
(23)
```

= 23 tokens Ans

QN By not seeing the next symbol, we can say that is a token:-

~~a)~~ ++
b) > ⇒ = may be, then $\boxed{>=}$ ⇒ token
 ⇒ user defined variable
c) main ⇒ may be main(or some user defined variable
d) < ⇒ = may be, then $\boxed{<=}$ ⇒ token

QN find the no of tokens,

printf ("Hai x = %d", i);

Soln |printf|(|" Hai x = %d "|,|i|)|;|
  (1) (2) (3) (4)(5)(6)(7)

⇒ Inside " " not need to enter = 7 tokens Ans

QN Find the no of tokens

printf ("i = %d, &i = %x", i, &i);

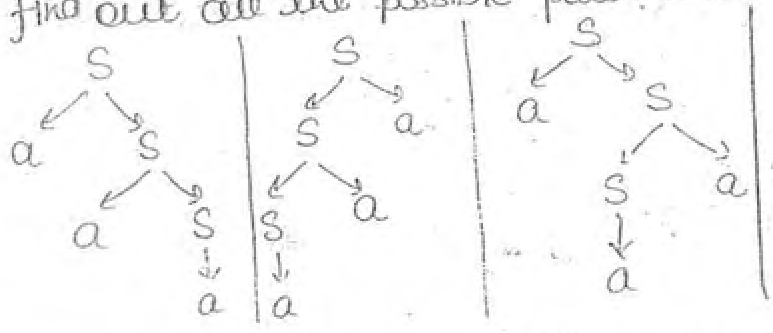
Soln |printf(" i = %d, &i = %x", i, &i); = 10 tokens Ans

∮ Grammars

Exp:- S→ aS|Sa|a

w= aaa

find out all the possible parse trees.



S→aS|sa|a ⇓ ⇒ a^+

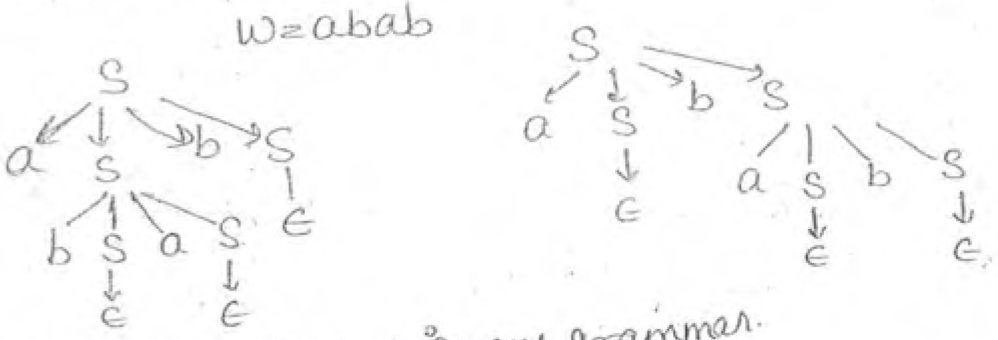
S→ a|as ⇓ ⇒ a^+

The above grammar is ambiguous grammar, because more than 1-parse tree is available to derive~~a~~ string w=aaa

Q) check the following grammar is ambiguous or not.

S→aSbS|bSaS|ε

w=abab

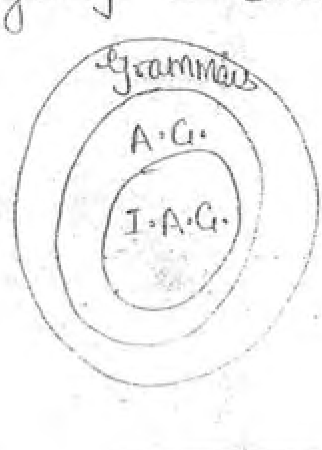


Given grammar is ambiguous grammar.

Note:- ① There is no algorithm to check whether the given grammar is ambiguous grammar or not so it is an undecidable problem.

② There is no algorithm to convert the given ambiguous grammar into unambiguous grammar. It is also undecidable problem.

③ Those ambiguous grammars, from which we can not eliminate ambiguity, is called as inherently ambiguous grammars



Grammar

$\boxed{\text{CFG}}$

Left Recursive Grammar

$A \to Aa \mid a$

$A \to A\alpha \mid \beta$

$\alpha, \beta \in (T+V)^*$

A = single variable

Right recursive Grammar

$A \to aA \mid a$

$A \to \alpha A \mid \beta$

$\alpha, \beta \in (V+T)^*$

A = single variable

Note ① For every Left Most Derivation Tree, Right Most Derivation Tree is also possible.

2) If the given grammar is unambiguous grammar, LMDT, RMDT both are same.

3) Top Down Parser will always keep the LM Recursive grammar into infinite loop

Exp:- $A \to A\alpha \mid \beta$

$W = \cancel{\beta}\alpha^{100}\beta$

$$\boxed{\begin{array}{c} A \\ \downarrow \\ A\alpha \\ \downarrow \\ A\alpha \\ \downarrow \\ A\alpha \\ | \\ | \end{array}} \Rightarrow \text{Infinite loop}$$

Repeated again & again

Exp:- Right Recursion (Top Down Parser)
→ No problem will occur.

$$\boxed{\begin{array}{l} A \to \alpha A \mid \beta \\ W = \alpha^{100}\beta \\ A \\ \downarrow \\ \alpha A \\ \downarrow \\ \alpha A \\ \vdots \\ \alpha^{100}A \Rightarrow \alpha^{100}\beta \\ \downarrow \\ \beta \end{array}}$$

$\boxed{\text{Elimination of Left Recursion}}$

Extended Backneous Nomal form (EBNF)

Exp:- $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id$

Sol^n $\underbrace{E \rightarrow E}+T \mid T$

$\Downarrow$

$T(+T)^0$

$T(+T)^1$

$T(+T+T)^2$

$T(+T+T+T)^3$

|

|

$T+T+T+T+T+T$---

$$\boxed{E \rightarrow T\{+T\}}$$

$\underbrace{T \rightarrow T}*F \mid F$

$\Downarrow$

$F(*F)^0$

$F(*F)^1$

$F(*F*F)^2$

$F(*F*F*F)^3$

|

|

$F*F*F*F*$-----

$$\boxed{T \longrightarrow F\{*F\}}$$

$$\Rightarrow \boxed{\begin{array}{l} E \rightarrow T\{+T\} \\ T \rightarrow F\{*F\} \\ F \rightarrow id \end{array}}$$ Ans

Drawback:- Equivalent grammar is not CFG.

Exp:- $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$
} $\Rightarrow$ Eliminate left recursion:

Recursion should be there but not left recursion.

Sol^n

$E \rightarrow E+T \mid T$ $\quad$ $T \rightarrow T*F \mid F$ $\quad$ $F \rightarrow id$

$E \rightarrow TE'$ $\quad$ $T \rightarrow FT'$

~~$E' \rightarrow +TE$~~ $\quad$ $T' \rightarrow \epsilon \mid *FT'$

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

Ans

Exp:- Eliminate left recursion-

$S \rightarrow aBDh$
$B \rightarrow Bb \mid h$
$D \rightarrow EF$
$E \rightarrow g \mid \epsilon$
$F \rightarrow f \mid \epsilon$

Sol^n

$B \rightarrow Bb \mid h$ $\quad$ $S \rightarrow aBDh$

$B \rightarrow hB'$ $\quad$ $D \rightarrow EF$

$B' \rightarrow \epsilon \mid bB'$ $\quad$ $E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

Ans

minate left recursion

(L) | a

L, S | S

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow \epsilon \mid ,SL'$

Q11. Eliminate left recursion.

$S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid aC$

$B \rightarrow bBC \mid f$

Sol

$S \rightarrow A$

$A \rightarrow aBA' \mid aCA'$

$A' \rightarrow \epsilon \mid dA' \mid eA'$

$B \rightarrow bBC \mid f$

Eliminate left recursion

$\rightarrow Aa \mid b$

$\rightarrow Ac \mid Sd \mid \epsilon$

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid \epsilon \mid bd$

⇓

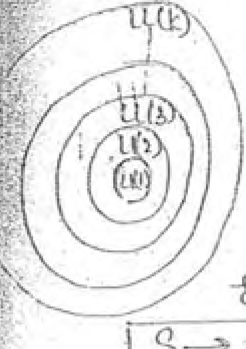
$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow \epsilon \mid cA' \mid adA'$

Left factoring

Parser sees one symbol at a time, from left to right.



$S \rightarrow a\alpha_1 \mid a\alpha_2 \mid a\alpha_3$

$w = a\alpha_3$

⇒ Parser is confused to choose out of these, bcoz all are giving 'a'. This is known as left factoring.

⇓

Elimination

$$\boxed{\begin{array}{l} S \rightarrow a\beta \\ \beta \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \end{array}}$$

Q11 Eliminate left factoring from the following grammar:-

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Sol

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow b$

Q/H Eliminate left factoring -

S → aAd | aB

A → a | ab

B → ccd | ddc

Sol^n S → aS'

S' → Ad | B

A → a A'

A' → ε | b

B → ccd | ddc

Q/H S → abc | abd | aef

Sol^n S → aS'

S' → bc | bd | ef

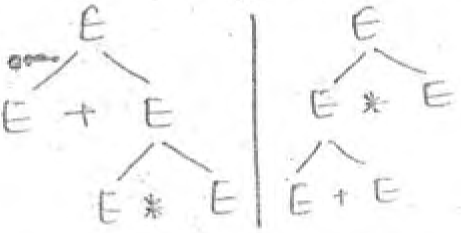
S' → bB' | ef

B' → c | d

⇒ all having the same priorities

φ E → E+E | E*E | E|E | E-E | id

W = a+b*c



ambiguous grammar

E → E+T | T ⇒ lower priority

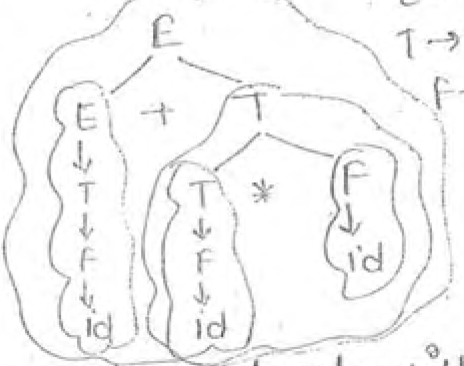
T → T*F | F ⇒ higher priority

F → id

E → E+T | E-T | T

T → T*F | T/F | F

F → id



* In CFG, the thing that will be on lower level, will be evaluated first.

* The operator which have lower priority, make that root.

Q/H Convert the following ambigious into unambiguous grammar.

R → R+R | R·R | R* | a | b

Sol^n R → R+T | T

T → T·F | F

F → (G)* | G

G → a | b

Q/H ambigious → unambigious

Bexpr → Bexp or Bexp | Bexp and Bexp | not Bexp | 0 | 1

Sol^n Bexpr → Bexp or T | T

T → T and F | F

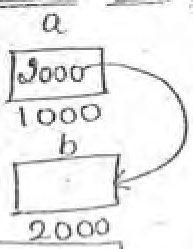
F → not G | G

G → 0 | 1

End

* Some extra and important Points (D.S.)

```
① int* a;
② int b;
③ a = &b
```



a = variable pointer

```
① int (*fp) ()
② int fun();
③ fp = fun;
   (*fp)();
```

fp = functional pointer pointing to function that returns integer

Exp
```
main()
{
  int(*ptr[3])();
  ptr[0] = aaa;
  ptr[1] = bbb;
  ptr[2] = ccc;
  (*ptr[2])();
}
```

```
1000
aaa()
{
  printf("aaa");
}
2000
bbb()
{
  printf("bbb");
}
```

```
3000
ccc()
{
  printf("ccc");
}
```

CHAPTER No. 3
(Parsing)

ptr

5000	5002	5004
1000	2000	3000
0	1	2

O|P = ccc

Qn.1 int (*f)(int, int)

Meaning:- a pointer to a function that takes two integers as I/P and returns O/P as integer.

Qn. What does the following 'C'-statement declare:-

char *(*(*a[N])())()

int(*)() ⇒ pointer to a function that returns integer.

⇒ Array of N pointers, pointing to function, that returns pointer to a function returning pointing to character or character pointer.

Qn What does the following c-statement will do-

void (*abc)(int, void (*def)())

Soln

abc is a pointer to a function, which will return void and which will take two parameter:-

(1) integer parameter

(2) A pointer def to a function, which will return void.

1 (Char*(*)()) (*ptr[N])()

Array of N pointers to functions, that will return pointer to a -nction which will return pointer to a character pointer

* $\boxed{\text{Chapter No 2}}$ *

Parsing

Type of parsers

Parsers

Top down parsers (TDP) | Bottom Up Parsers (BUP)

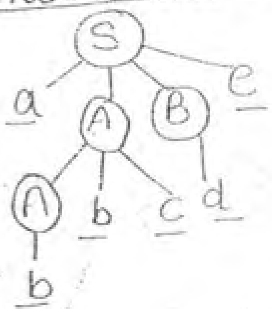
TDP :- In TDP, we will derive the given string by taking the start symbol.

-UP :- In BUP, we will take the string and finally we will get the start symbol.

$\boxed{\text{orking of TDP}}$ ⇒ TDP follow the left most derivation

-xp :- S → aABe
A → Abc/b
B → d

w = abbcde
↑ look ahead symbol



TDP, only sees one symbol at a time.

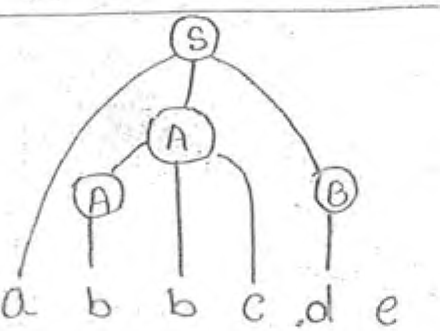
If there are two possibilities, choosing the right one is difficul-

-ote1 :- In TDP we are using $\boxed{\text{Left most derivation.}}$

-ote2 :- The difficulty in TDP, is that if a variable is having more than one possibility, choosing right production.

Working of Bottom-UP parses

Exp:-

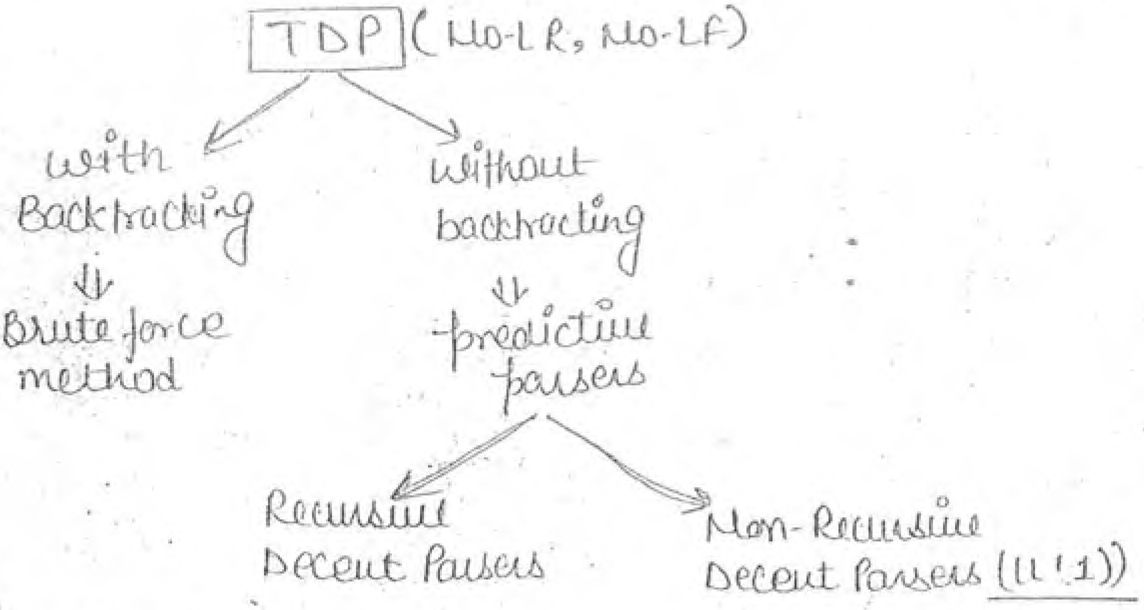


S → aABe
A → Abc/b
B → d
W = abbcde

Note-1:- In Bottom up parsing, we are using reverse of Right most derivation to derive the string.

Note2:- The difficulty in bottom up parsing finding the substring (handle), which will give our required variable, so that we will go to start symbol.

Top Down Parsers



LL(1) Parses

L = left-right (reading the I/P symbol)
L = using left most derivation
taking 1 symbol at a time

L-R(1) Parses

L = left-right (reading the I/P symbol)
R = using right most derivation, taking one symbol at a time

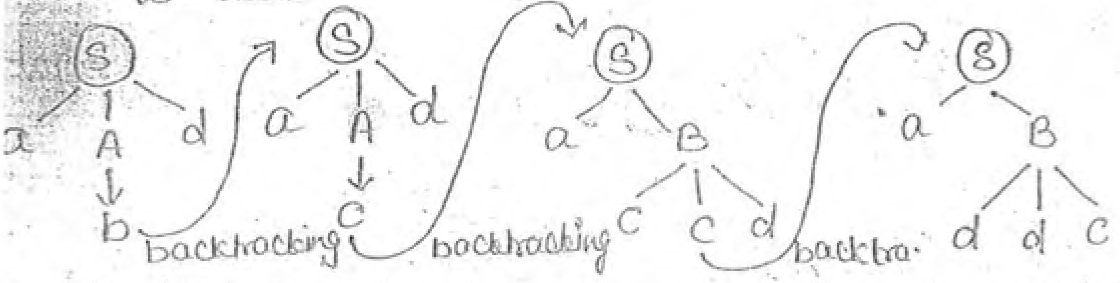
Method

S → aAd | aB

A → b | c

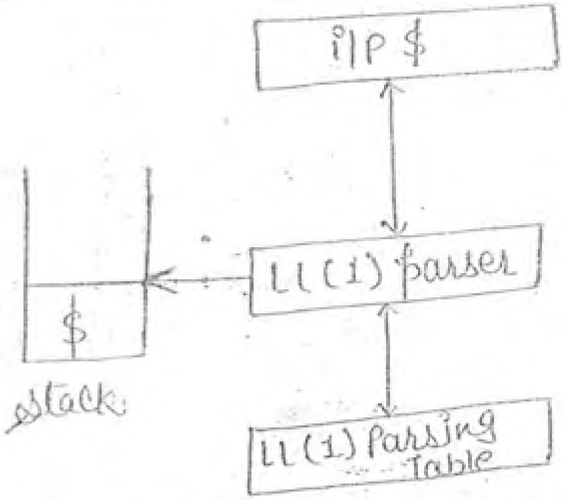
B → ccd | ddc

w = addc



rawbacks:- If will take a lot of time in backtracking. It i at good for debugging.

) Non-Recursive Decent Parsers:-



LL(1) Parsing Algo

If X is top of the stack && 'a' is a lookahead symbol-

1. If((x==a)==$) then, successful completion.

2. If((x==a) ≠ $), pop out from the stack, increment i/p pointe

3. If (X is non terminal) then use LL(1) parsing table entry M[X,

If M[X,a], if x → uvw, replace X by uvw in the reverse order

If M[X,a]= blank space, indicate error.

Exp:—

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$ ⇒ $\boxed{Exp(1)}$

$F \rightarrow id \mid (E)$

$\boxed{\omega = id + id * id\ \$}$

LL(1) Parsing table

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

→ Size of LL(1) Parsing table.

$\boxed{m * (n+1)}$

m = no. of variables in grammar

n = no. of terminals in grammar

* always we are doing reverse, because we want left most derivation.

Removal of left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

⇒

$E \rightarrow TE'$ $T \rightarrow FT'$ $T' \rightarrow \epsilon$

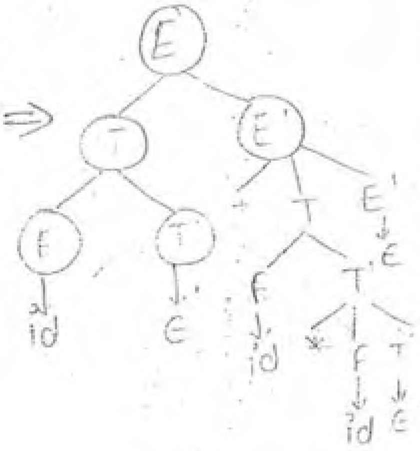
$T \rightarrow FT'$ $F \rightarrow id$ $E' \rightarrow \epsilon$

$F \rightarrow id$ $T' \rightarrow *FT'$

$T' \rightarrow \epsilon$ $F \rightarrow id$

$E' \rightarrow +TE'$.

⇒



Q.N. $\omega = (id + id * id)\ \$$ ⇒ $\boxed{Exp(2)}$

Sol^n

Total productions =

$$\left\{\begin{array}{lll} E \rightarrow TE' & T' \rightarrow \epsilon & T' \rightarrow \epsilon \\ T \rightarrow FT' & E' \rightarrow +TE' & E' \rightarrow \epsilon \\ F \rightarrow (E) & T \rightarrow FT & T' \rightarrow \epsilon \\ E \rightarrow TE' & F \rightarrow id & E' \rightarrow \epsilon \\ T \rightarrow FT' & T' \rightarrow *FT & \\ F \rightarrow id & F \rightarrow id & \end{array}\right\}$$. Ans

Exp 1 stack

$	~~E~~	~~T~~	~~T~~	~~F~~	~~id~~	~~E~~	~~E~~	~~T~~	~~*~~	~~T~~	~~F~~	~~id~~	~~T~~	~~F~~	~~*~~	~~id~~

Exp 2 stack

$	E'	T'	T'	E	)	~~T~~	~~E~~	~~E'~~	~~T~~	~~T'~~	~~F~~	~~id~~	E	~~+T~~	~~F~~	T	~~F~~	~~id~~	T	~~F~~	~~*~~	~~id~~

IN. $S \to (L) | a$

$L \to L , S | S$

$\boxed{W = (a,a,a)}$

$\uparrow\uparrow$

Sol^n $S \to (L) | a$

$L \to SL'$

$L' \to \epsilon \,|\, , SL'$

$\Downarrow$

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

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

Parsing table

$	~~)~~	~~L'~~	~~(~~	~~L'~~	~~S~~	~~a~~	~~L'~~	~~S~~	~~,~~	~~L'~~	S	~~,~~

Construction of LL(1) Parsing table

First(α), gives a set of terminals, that begin in strings derived from α.

Exp:- $\alpha \to abc | def | Ab$

$A \to e$

gives by α' $\Leftarrow$ $\boxed{\begin{matrix} a \\ d \\ e \end{matrix}}$ $\begin{matrix} \Leftarrow abc \\ \Leftarrow def \\ \Leftarrow eb \end{matrix}$

Rule-1:- First(α) = α, $\boxed{\text{if } \alpha \text{ is terminal}}$

Rule-2:- First(ϵ) = ϵ

Rule-3:- $\alpha \to x_1 x_2 x_3$

First(α) = First($x_1 x_2 x_3$)

= First(x_1)

= First(x_1) − ϵ $\cup$ First(x_2, x_3)

only if, $x_1 \to \epsilon$

Exp:- $x_1 \to a | \epsilon$

$x_2 \to b | \epsilon$

$x_3 \to c | \epsilon$

First (x_1, x_2, x_3)

$\Downarrow$

First(x_1)

a $\cup$ First(x_2, x_3)

$\Downarrow$

First(x_2)

$\Downarrow$

b $\cup$ First(x_3)

$\Downarrow$ c, ϵ

N find the First for the following grammar-

$E \to TE'$

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

$T \to FT'$

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

$F \to id | (E)$

Sol^n

E

T

F

IN.

S→

B→

C→

D→

E→

IN.

S→ A

A→ d

B→

C→ f

IN.

S→ A

A→ c

B→ d

Foll

Follo

o th

ule1:-

ule2

Sol^n	First()
E	id, (
E'	ε, +
T	id, (
T'	ε, *
F	id, (

Q1. Find the first for the following grammar:-

S → aBDh   First(S) = {a}
B → cC   First(B) = {c}
C → bC | ε   First(C) = {b, ε}
D → EF   First(D) = {g, f, ε}
E → g | ε   First(E) = {g, ε}
F → f | ε   First(F) = {f, ε}

Q1. Find the first for the following grammar:-

S → ACB | CbB | Ba   First(S) = d, g, h, ε, b, a
A → da | BC   First(A) = d, g, h, ε
B → g | ε   First(B) = {g, ε)
C → h | ε   First(C) = {h, ε}

Q1. Find the first for the following grammar:-

S → AaAb | BbBa   First(S) = c, a, d, b
A → c | ε   First(A) = c, ε
B → d | ε   First(B) = d, ε

Follow(): → Follow(A) ↑ variable.

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

Rule 1:- If A a start symbol, then

$\boxed{Follow(A) = \$}$

Rule 2:- If $X \rightarrow \alpha A \beta$ is in G:-

then, $\boxed{Follow(A) = First(\beta)}$

ule-3: If $x \to \alpha A$ (or) $x \to \alpha A \beta$
$\beta \to \epsilon$

$$\boxed{\text{Follow}(A) = \text{Follow}(x)}$$

Qn. Find first and follow for the following grammar:-

$x \to aABe$
$B \to c|d$
$A \to a$

	Fi()	Fo()
x	a	$
B	c,d	e
A	a	c,d

Qn. Find first and follow for the following grammar:-

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

	First	Follow
E	id, (	$,)
E'	ε, +	$,)
T	id, (	$,), +
T'	ε, *	+, $,)
F	id, (	*, +, $,)

Ans

Qn Find the first and follow of the following grammar-

$S \to aBDh$
$B \to cC$
$C \to bC | \epsilon$
$D \to EF$
$E \to g | \epsilon$
$F \to f | \epsilon$

	First	follow
S	a	$
B	c	g, f, h
C	b, ε	g, f, h
D	g, f, ε	h
E	g, ε	f, h
F	f, ε	h

Ans

Qn. $S \to (L) | a$
$L \to SL'$
$L' \to \epsilon | ,SL'$

	First	Follow
S	(, a	$, ,,)
L	(, a	),
L'	ε, ,	)

A..

~~§ LL1 Table Construction Algo~~

~~or each foroduction, A→α~~
~~repeat, following two steps:-~~

~~) Add A→α, under M[A, b]~~
~~where, b∈ First(α)~~

~~.) If First(α) Contain ∈, then~~
~~add A→α, under M[A,C]~~
~~where, C∈ Follow (A).~~

~~1N Contruct LL(1) parsing table for the following grammar,~~
~~S→(L)|a~~
~~S→~~

§ LL(1) table Conversion algo

For each foroduction, A→α, repeat, following two steps:-

) Add A→α, under M[A, b]
where, b∈ First(α)

) If. first (α) Contain ∈, then
add A→α, under M[A,C]
where, C∈ Follow(A).

N.1 Construct LL(1) parsing table, for the following grammar:-

2N S→(L)|a
L→SL'
L'→∈|, SL'

⟹

S→(L)|a
L→L,S|S
{ actual grammar }

after removing Left Recursion)

	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→∈		L'→,SL'	

- Given grammar is LL(1), because each entry of LL(1) parsing

parsing table contains maximum one entry.

2M Construct LL(1) parsing table - (or) given grammar is LL1 or not

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

Eliminate left recursion -

$E \to TE' \mid T$

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

$T \to F.T' \mid F$

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

$F \to id \mid (E)$

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

QN Check the following grammar LL(1) is not -

$S \to A$

$A \to aB \mid Ad$

$B \to b$

$C \to g$

⇓

$S \to A$

$A \to aBA'$

$A' \to \epsilon \mid dA'$

$B \to b$

$C \to g$ ⇒ LL(1) grammar

	a	b	d	g	$
S					
A					
A'			A'→dA'		A'→ε
B					
C					

QN Check the following Grammar is LL(1) or not :-

$S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

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

Q.N.
S→
A→
B→
⇓
LL
Note
→ α
gramm
Note
A→α
1M. Ch
S→E
→ Q
4M S
M. Ch
S→
A→
B→

Q.N. Find the grammar is LL(1) or not-

$S \to AaAb \mid BbBa$

$A \to b$

$B \to a$

$\Downarrow$

LL(1)

	a	b	$
S	$S \to AaAb$	$S \to BbBa$	
A		$A \to b$	
B	$B \to a$		

Note 1:-

$\to \alpha_1 \mid \alpha_2 \mid \alpha_3$ ⇒ If this is the given form of grammar, then that grammar is said to be LL(1) only if-

$$\boxed{\begin{array}{l} \text{First}(\alpha_1) \wedge \text{First}(\alpha_2) = \phi \\ \text{First}(\alpha_1) \wedge \text{First}(\alpha_3) = \phi \\ \text{First}(\alpha_2) \wedge \text{First}(\alpha_3) = \phi \end{array}} \Rightarrow \text{pairwise mutually disjoint}$$

Note 2 If the grammar is in the form of -

$A \to \alpha_1 \mid \alpha_2 \mid \epsilon$, Grammar is called LL(1) only if-

$$\boxed{\begin{array}{l} \text{First}(\alpha_1) \wedge \text{First}(\alpha_2) = \phi \\ \text{First}(\alpha_1) \wedge \text{follow}(A) = \phi \\ \text{First}(\alpha_2) \wedge \text{follow}(A) = \phi \end{array}} \Rightarrow \text{pairwise mutually disjoint}$$

Q.N. Check the following grammar is LL(1) or not - (GATE)

$S \to E \mid a$

$E \to a$

Sol $\underline{S}$ ⇒ first(E) ∧ first(a)

$a \wedge a = a$ <u>not (LL1)</u> Ans

Q.N. Check the following grammar is LL1 or not-

$S \to aABb$

$A \to a \mid \epsilon$

$B \to d \mid c$

$\underline{A}$

first(a) ∧ follow(A)

(a, ~~c~~) ∧

$\underline{B}$

first(d) ∧ follow(B)

Given Grammar is LL(1) grammar. Ans

Qn. $S \rightarrow aSA \mid \epsilon$

$A \rightarrow c \mid \epsilon$

Soln $\underline{S}$ $\quad\quad$ $\underline{A}$

first(A) ∧ follow(A) $\quad\quad$ first(A) ∧ follow(A)

$a \wedge c, \$ = \phi$ $\quad\quad$ $c \wedge \$, c = c$

This grammar is not LL(1) grammar. Ay

Qn. Check the following grammar is LL(1) or not,

$S \rightarrow AB$

$A \rightarrow a \mid \epsilon$

$B \rightarrow b \mid \epsilon$

Soln $\underline{A}$ $\quad\quad$ $\underline{B}$

first(A) ∧ follow(A) $\quad\quad$ first(b) ∧ follow(b)

$a \wedge b, \$ = \phi$ $\quad\quad$ $b \wedge \$ = \phi$

This grammar is LL(1) grammar. Ay

Qn. Given grammar is LL(1) or not-

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

Soln $\underline{S}$

first((L)) ∧ first(a) $\quad\quad$ first(L,S) ∧ first(S)

$(\wedge a = \phi$ $\quad\quad$ $(, a$

first [illegible]

$\Downarrow$

[illegible]

Note:- Any left recursive grammar is not LL(1).

Note-2:- Any grammar which contain left factoring is not LL(1)

Qn. Construct LL(1) parsing table for the following grammar

program → begin ① d semi X end

X → d semi X ② | SY ③

Y → Semi S Y ④ | ϵ ⑤

	Semi	begin	d	end	S	$
gram		①				
X			②		③	
Y	④			⑤		

⇒ LL(1) grammar

Q. Find first and follow for the following grammar-

$E \to aA \mid (E)$

$A \to +E \mid *E \mid \epsilon$

Sol

	first	follow
E	a, (	$,)
A	+, *, ε	$,)

Q. Which one of the following is true:-

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

$F \to F-F \mid id$

1) * has higher precedance than +.

2)

3) *, - has same precedance

4) + has higher precedance.

φ

Recursive Decent Parsers

exp:-

$E \to E+T \mid T$

$T \to T*F \mid F$ ⟹ Eliminate left Recursion

$F \to id$

$E \to TE'$

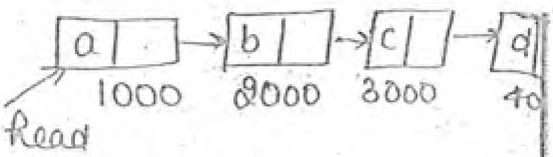
$E' \to \epsilon \mid +TE'$

$T \to FT'$

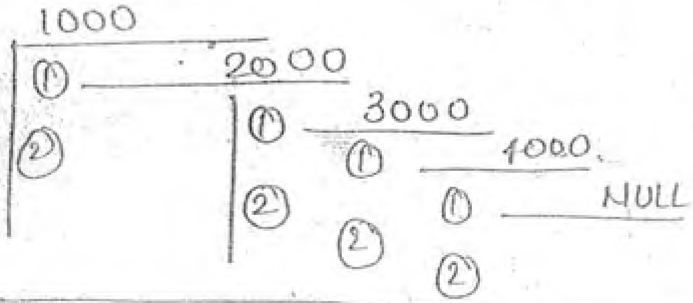
$T' \to \epsilon \mid *FT'$

$F \to id$

```
E()        | E'()       | T()     | T'()
{          | {          | {       | {
  E()      |   E'()     |   T()   |   T'()
}          | }          | }       | }
```

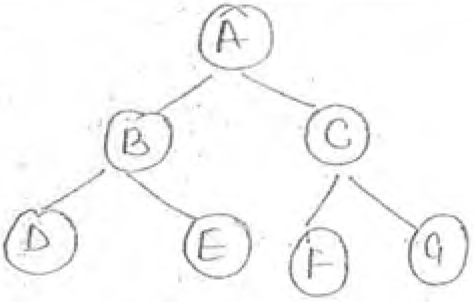


```
Return(head)
{
  if(head==NULL)
    return;
  else
  {
    ① Reverse(head→next)
    ② printf(Head→data);
  }
}
```

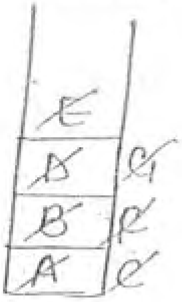


Non Recursive Pgm → Stack at the place of recursion

Preorder without Recursion



Stack



A, B, D, E, C, F, G

Note:- In recursive decent parser, we will write the recursive pgm. for every variable, such that if any variable contai more than one possibility, it will chose the correct productio.

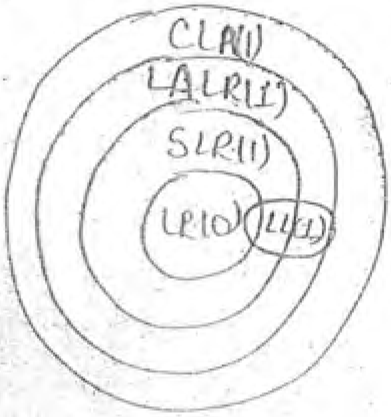
Bottom-Up Parser

Bottom-up Parser (Shift-Reduce Parsers).

- Operator Precedance Parser ⇐ Operator grammar {ambiguous, unambiguous}
- LR-Parser ⇓ applied only on unambiguous grammar

LR-Parses

LR(0)	SLR(1)	CLR(1)	LAR(1)
	Simple LR-parses	Canonical LR-parses	Look Ahead- LR Parses
	less cost	more cost	medium cost
inally do Parsing (Less powerful)	less power	powerful	Medium

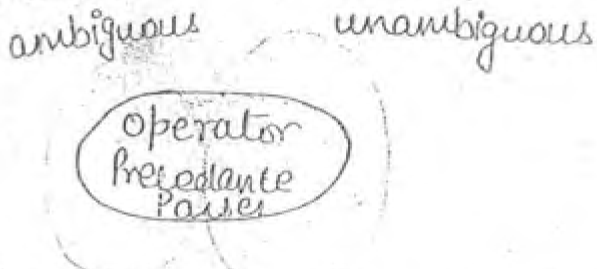


* Bottom-up Parses are much powerful than ~~top~~ down parsers, but designing is more complex.

$\boxed{LL(k) \subseteq LR(k)}$

Operator Precedance Grammar (Parsers)

Operator Grammar:-



Grammar is said to be a operator grammar, if it satifies following two conditions-

No null productions

No two variables side by side on R.H.S. of production

Exp:- $E+E \mid E*E \mid id$ ✓

Exp:- $E \rightarrow \boxed{AB}$
$A \rightarrow a$ ✗
$B \rightarrow b$

Exp:- $E \rightarrow A+B$
$A \rightarrow a$
$B \rightarrow b \mid \textcircled{\epsilon} \rightarrow$ ✗

Q Check the following grammar is operator grammar or not

$S \rightarrow \boxed{SAS} | a$

$A \rightarrow bSb | b$

This grammar is not operator grammar.

Conversion into Operator grammar

H $S \rightarrow SbSbS | SbS | a$ $\Rightarrow$ $S \rightarrow SbSbS | SbS | a$

$A \rightarrow bSb | b$ remove A, useless production

H $P \rightarrow SR | S$

$R \rightarrow bSR | bS$

$S \rightarrow WbS | W$

$W \rightarrow L*W | L$

$L \rightarrow id$

$\Rightarrow$

$P \rightarrow SbP | SbS | S$

$R \rightarrow bP | bS$

$S \rightarrow WbS | W$

$W \rightarrow L*W | L$

$L \rightarrow id$

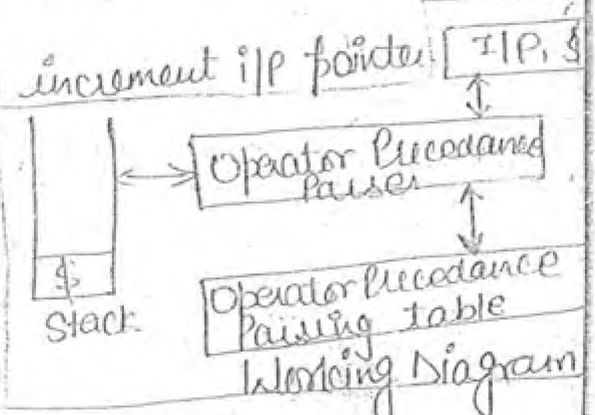
$$\begin{tikzcd} & P \arrow[dl] \arrow[dr] & \\ S & & R \end{tikzcd}$$

Q Operator Precedence Parser (Parsing algorithm):-

, If a is the top of the stack, then b is the look ahead sym then-

1) If a<b or a=b, then shift b and increment i/p pointer.

2) If a>b, repeat
   {
      pop out from the stack
   } until
   (top < recently popped out)



Working Diagram

3) If a=b=\$, then successful completion.

Ex:- Parsing table

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

① w = id + id * id \$

\$	~~id~~	~~+~~	~~id~~	~~*~~	~~id~~

id id id * + Acc

② w = id * id * id \$

\$	id	*	id	*	id

id id * id * Acc

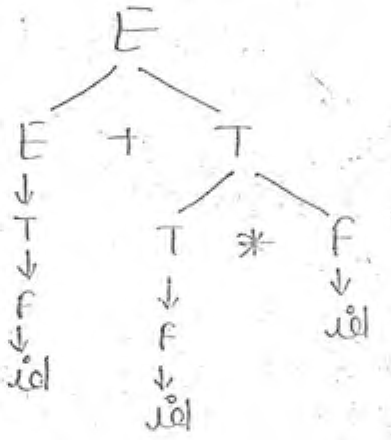
). E → E+T | T

T → T*F | F

F → id

$\$ \lessdot \boxed{id}^{①} \gtrdot \overset{⑤}{(+)} \lessdot \boxed{id}^{②} \gtrdot \overset{④}{(*)} \lessdot \boxed{id}^{③} \gtrdot \$$

$$\boxed{\begin{array}{c} + < * \\ + > + \\ * > * \end{array}} = \underline{5\ \text{handle}}$$



$	~~id~~	~~*~~	~~id~~	~~*~~	~~id~~

id → ① handle   * → 4 handle

id → ② handle   + → 5 handle

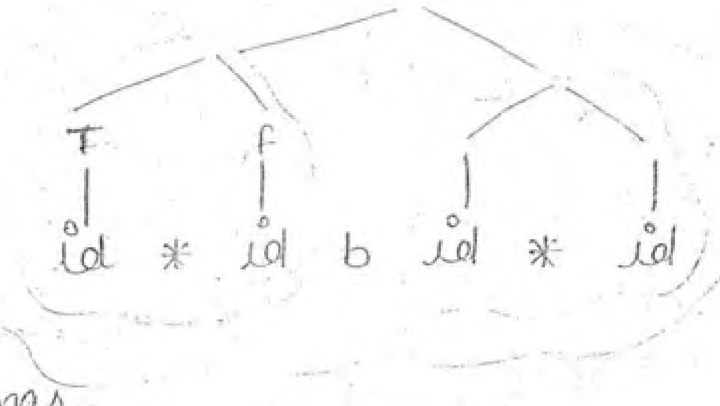
id → ③ handle

Q. Define the operator precedance parsing table for, w=

$\boxed{w = id * id\ b\ id * id}$

Operators ⇒ *, b and $\boxed{* > b}$ $\boxed{b < b}$

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



Q. Consider the grammar:-

E → E+n | E*n | n. for the I/P string,

$\boxed{w = n+n*m}$, the handels are:-

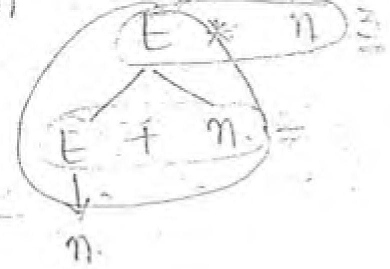
) n, E+n and E+n*n

) n, E+n and E+E*n

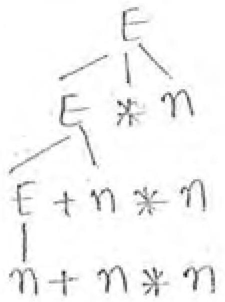
) n, n+n and n+n*n

) n, E+n and E*n

Parse tree



1st handle = n



L-R Parsing Algo *

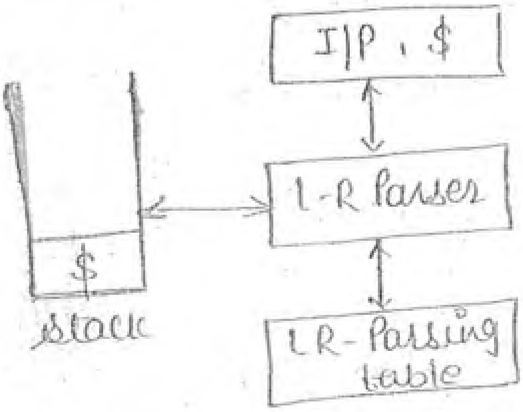
If S- state on the top of the stack and a- lookahead symbol, then-

1. If action[S,a]= Si, then shift a and i and increment the i/p pointer.

· If action[S,a]= Rj, and if Rj is, $\alpha \rightarrow \beta$, then pop $2 * |\beta|$ sym from the stack and replaced by α.

If S_{m-1} is the state below α, then push Goto$[S_{m-1}, \alpha]$.

3. If action [S,a]= acc, then successful completion.



Note:- For all LR-parsers, parsing algorithm is same, parsing tables are different.

Exp:- S → AA ①
A → aA ② | b ③

$\boxed{W = aabb\$}$

LR-(0) Parsing table

	a	b	$	S	A
	action			Goto	
0	S3	S4		1	2
1			acc		
2	S3	S4			5
3	S3	S4			6.
4	r3	r3	r3		
5	r1	r1	r1		
6.	r2	r2	r2		

Rows = State No

Coloums = Action | Goto

5

$	0	~~a~~	~~3~~	~~a~~	~~3~~	~~b~~	~~4~~

~~A~~ ~~2~~ ~~A~~ ~~6~~ ~~A~~ ~~6~~
S 1 ~~b~~ ~~4~~
~~A~~ ~~5~~

a a b b $
↑ ↑ ↑ ↑ ↑
a a a a a

⇒ <u>acc</u> = <u>accepted</u>

IP: $\boxed{W = abab\$}$
↑ ↑ ↑ ↑ ↑

$	0	~~a~~	~~3~~	~~b~~	~~4~~	~~b~~	~~4~~

~~A~~ ~~2~~ ~~A~~ ~~6~~
~~a~~ ~~3~~ A ~~6~~
A ~~5~~
S 1

= acc = <u>accepted</u>

r3
A → b
|b| = 1*2 = 2
r2
A → |aA|
= 2*2 = 4
r3 A → b
|b| = 1*2 = 2

r1
S → AA
= 2*2
= 4

Problem with Shift → Reduce ⇒ <u>Action part</u>

↳ because these entries are only on action part

Another parsers other than LR(0), has some minimization in reduction entries, because they can predict the next symbol.

<u>Conflict</u>:– Shift and reduce together Sj/ri

Eg S → AA
A → aA|b
⇓ Augmented Grammar (added to know successful completion).

S' → S ⇒ S·S
S → AA
A → aA|b

Eg <u>Item</u>
S → ·xyz (Item)
S → x·yz
S → xy·z
S → xyz· → (<u>final production</u>
<u>Complete production</u>
<u>reduced production</u>)

Closure of an item

⇓

applied only on item

<u>Given Grammar</u>

<u>Exp</u>:- S → AA
A → aA|b

Now, $\boxed{S' \to \cdot S \cdot}$ ⇒ item (closure) (i) $\boxed{S \to A \cdot A}$ ⇒ closure (S)

① S' → ·S

② S → ·AA

③ A → ·aA
·b

S → A·A

A → ·aA
·b

ф Finding closure of a variable

Closure (I) =

① Add item I to closure(I)

2) If $\alpha \rightarrow \beta \cdot Aa$ is I and $A \rightarrow BC$ in G, then add $A \rightarrow \cdot BC$ to closure of

3) repeat previous two steps. for every newly added production

Exp:- $S \rightarrow AA$
$A \rightarrow aA|b$

① $S \rightarrow \cdot S$

② $S \rightarrow \cdot AA$

③ $A \rightarrow \cdot aA$
$\cdot b$

ф Finding Goto (I, x) :-

Find the Goto of (I, x) by following method :-

Write the same item (I) as it is, by moving (·) after x.

1) Apply closure function on the result of step 1.

ф LR(0) Parsing table construction

Step-1 Find the augmented grammar.

Step-2 Find the closure of augmented production

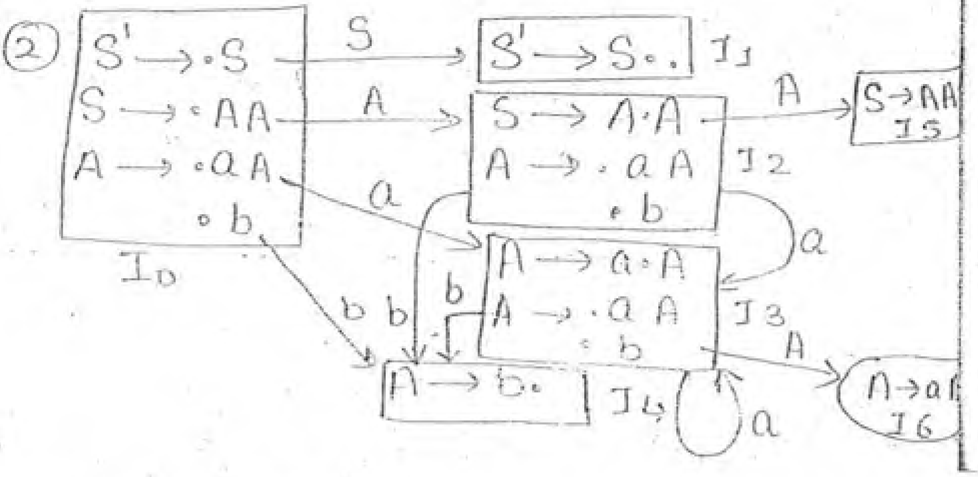
Step-3 Using closure of augmented production construct the dfa.

Step-4 Reduce dfa into table.

Exp:- Construct LR(0) Parsing table, for the following grammar:-

$S \rightarrow AA$ ①
$A \rightarrow aA$ ② $| b$ ③

Sol: ①
$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA|b$



Construction of table

M	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			acc		
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

* If in a I_j more than one production is completed, it will become Reduce-Reduce conflict

* Shift-Reduce conflict = Reduce-Shift conflict

* No Shift-Shift conflict
⇓
– because it is a d.f.a.

⇒ given grammar is LR(0) grammar, because no entry of this table contain more than one value AS

note:– S-S conflict is not possible, because given diagram is dfa.

QN. Check the following grammar is LR(0) or not–

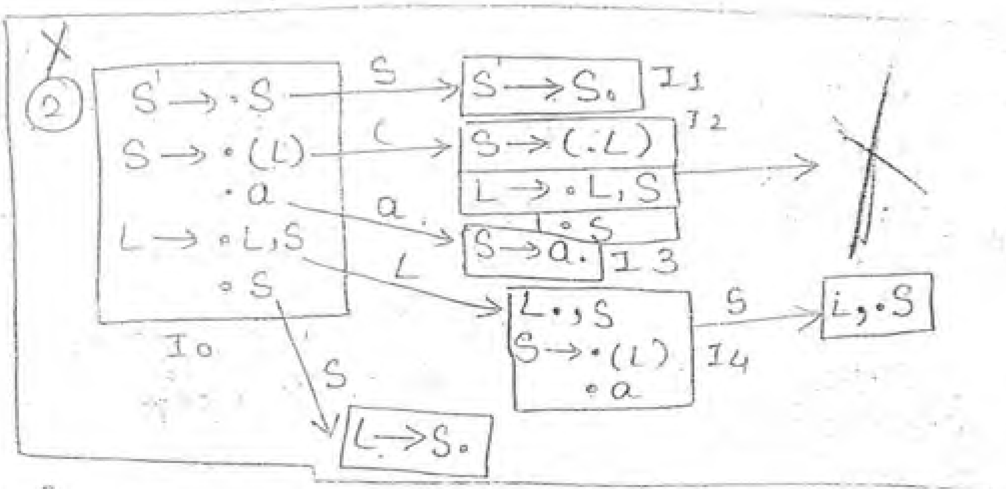
S → (L) | a

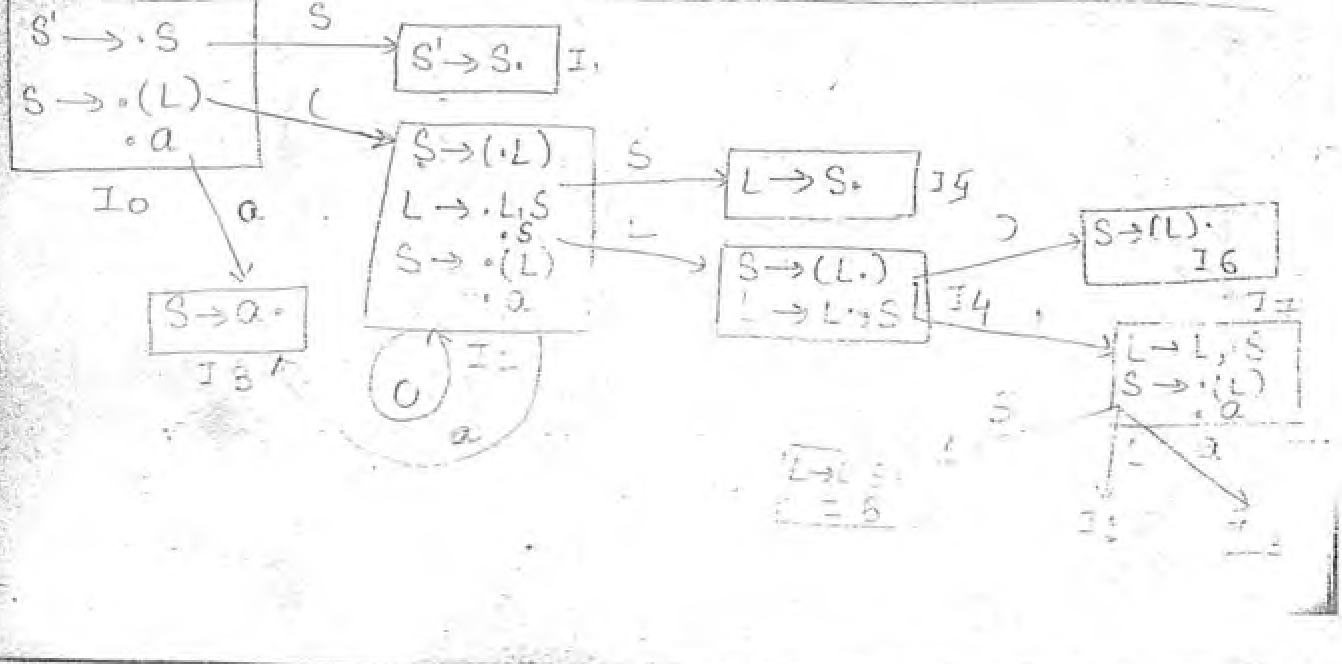
L → L, S | S

(1) S' → S

S → (L) | a

L → L, S | S





Parsing table

M	Action					Goto		
	a	(	)	,	$	S	A	L
0	S3	S2				1		
1					acc			
2	S3	S2				5		4
3	r2	r2	r2	r2	r2			
4				S6	S7			
5	r4	r4	r4	r4	r4			
6	r1	r1	r1	r1	r1			
7	S3	S2				8		
8	r3	r3	r3	r3	r3			

⇒ LR(0)

QN. Check the following grammar is LR or not:-

S → dA | aB

A → bA | C

B → bB | C

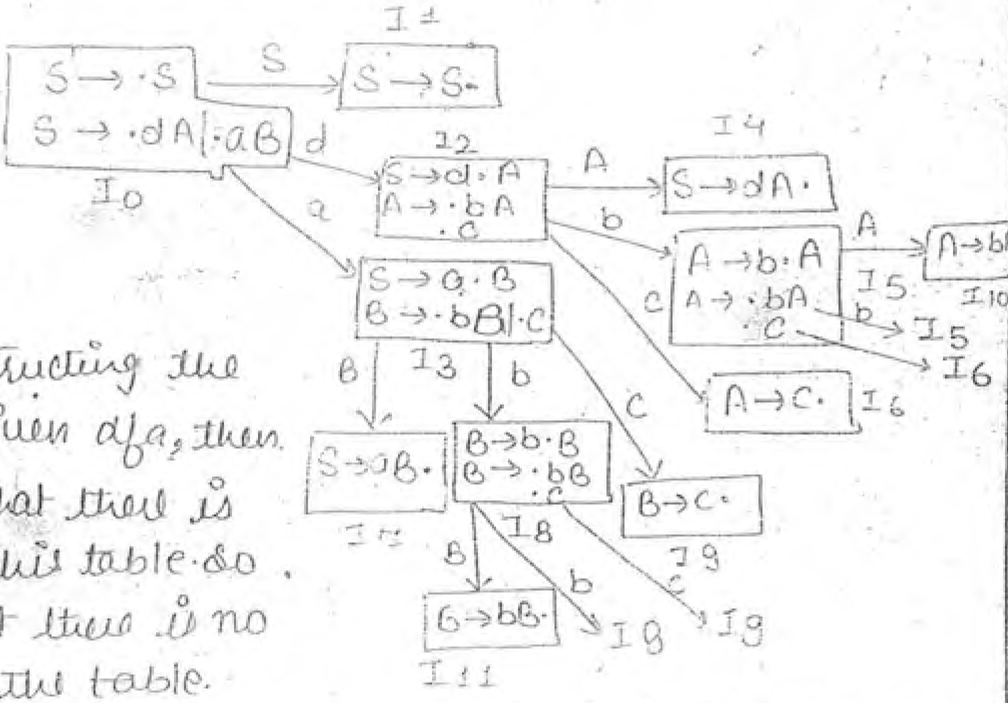
Solⁿ S → S

S → dA | aB ⇒

A → bA | C

B → bB | C

⇓



If we are constructing the table, for the given dfa, then we will find, that there is no conflict in this table. So, it is clear, that there is no any conflict in the table.

⇓

States = 0 - 11

Action = d, b, C, a, $

Goto → S, A, B, $

⇒ There is no conflict in this given Grammar so it is LR(0) Grammar.

Q4 Check the following grammar is LR(0) or not:-

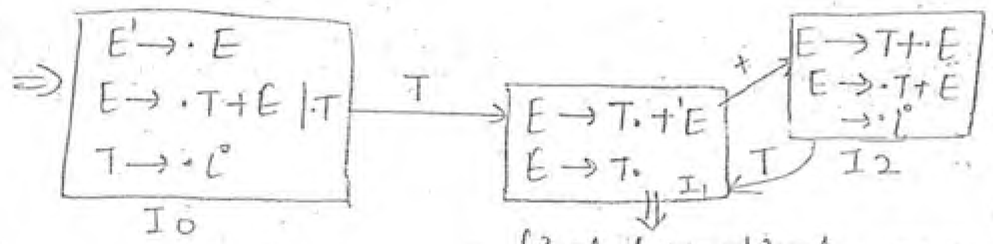
E → T+E | T

T → i

Soln E' → E

E → T+E | T

T → i ⇒



final & Nonfinal ⇒ SR conflict

Not LR(0) Ans

Exp:- (S → A·B, A → b·) ⇒ final and non final but still conflict $\overset{\wedge}{\text{not}}$

bcoz S → A·B (not action part) ⇓ Goto part

S → b· (purely action part)

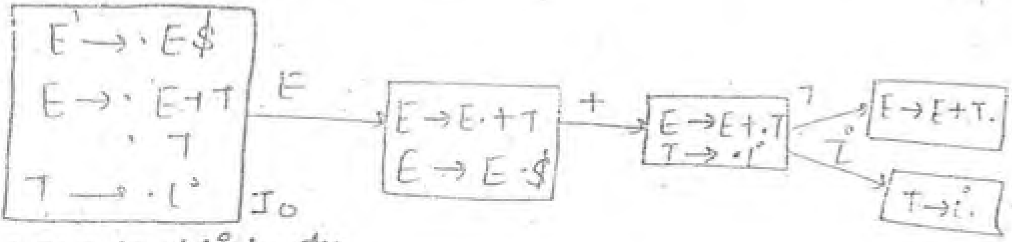
Q4 E → E+T | T

T → i

Soln E' → E$

E → E+T | T ⇒

T → i



It is LR(0) → no conflict Ay

Q4 following grammar is LR(0) or not:-

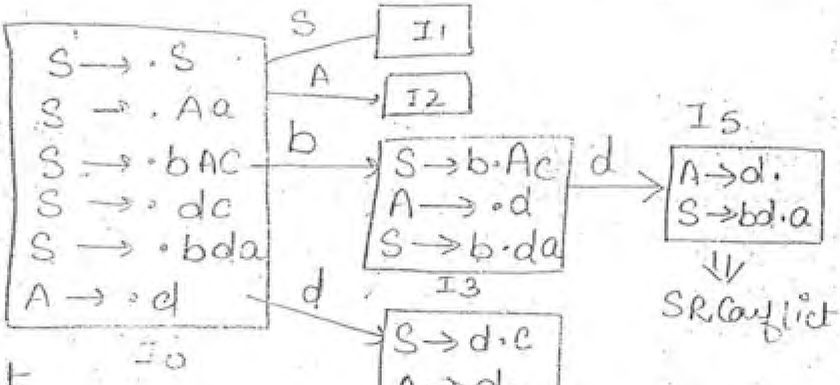
S → Aa | bAc | dc | bda

A → d

Soln S' → S

S → Aa | bAc | dc | bda ⇒

A → d



This grammar is not LR(0) because two states are not acceptable

⇓ SR conflict

⇓ SR conflict

Q4 Check the following grammar is SLR(1) or not:-

Soln S → Aa | bAC | dc | bda

A → d

given grammar is not SLR(1) because, the conflicts which are there in LR(0) are not eliminated by SLR(1).

QN Check the following grammar is SLR(1) or not-

E→ E+T|T

T→TF|F

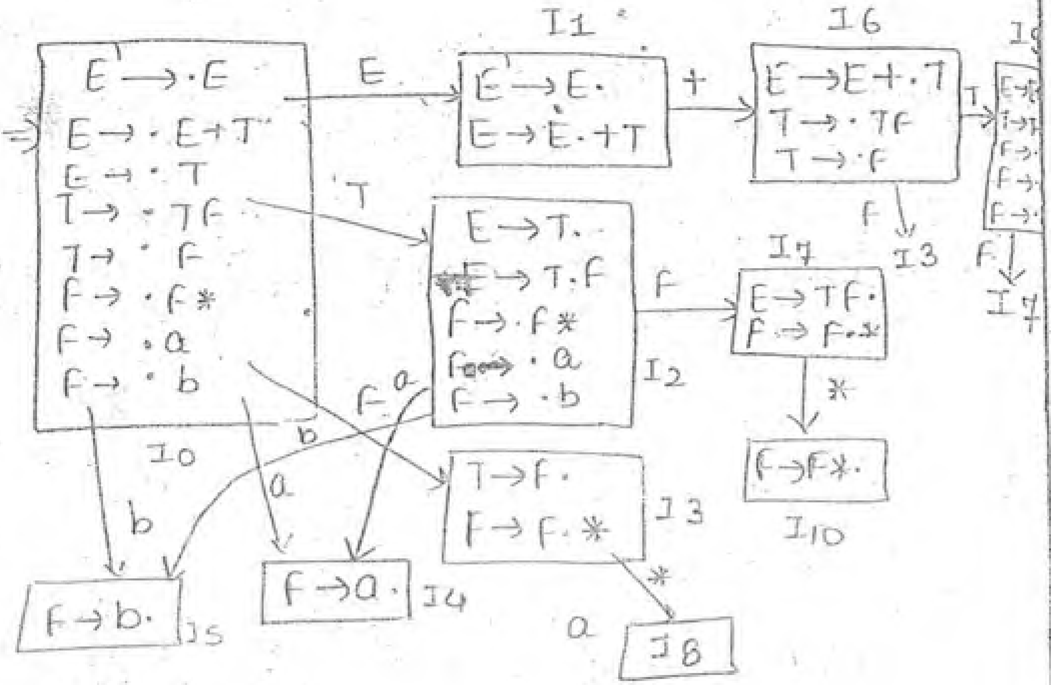
F→ F*|a|b

Soln E'→ E

E→ E+T|T

T→ TF|F

F→ F*|a|b.



	a	b	*		$
2	r2/S4	r2/S5	r2	r2	r2
3	r4	r4	r4/S8	r4	r4
9	r1/S4	r1/S5	r1	r1	r1
7	r3	r3	r3/S10	r3	r3

⇒ not LR(0), just bcoz of conflicts
⇓

Now removing conflicts, apply SLR(1). Find follow of the complete entries and intersection with shift entries If we got φ = no conflict and LR(1)

SLR(1) table

	a	b	*	+	$
2	S4	S5	r2	r2	r2
3	r	r	S	r	r
9	S	S		r	r
7	r	r	S	r	r

I_2 = R = + $
S = a, b
φ

I3, R = + $ a, b
S = *
φ

I7 + $, a, b
*
φ

I9, +, $
a, b
?

= SLR(1) ⇒ all the conflicts of LR(0) are removed here.

Q* Check the following grammar is SLR(1) or not:-

$S \to AaAb \mid BbBa$

$A \to \epsilon$

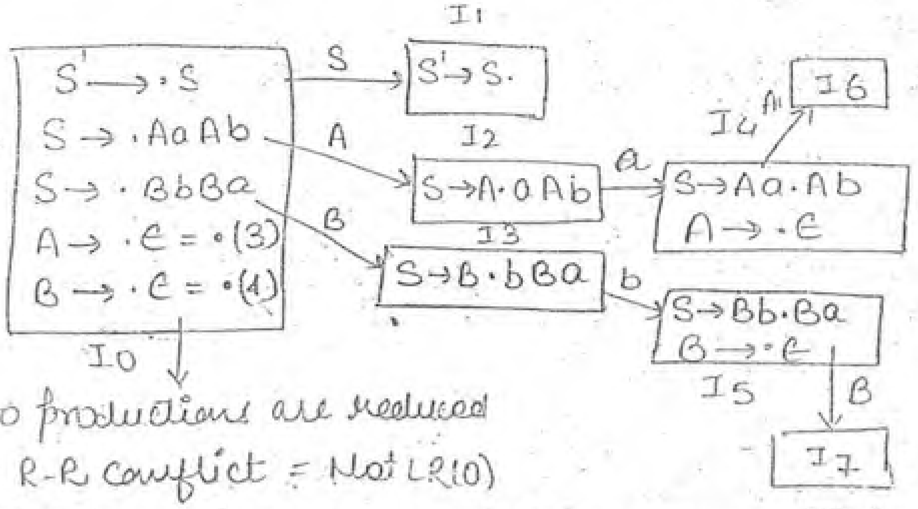
$B \to \epsilon$

soln $S' \to S$

$S \to AaAb \mid BbBa \Rightarrow$

$A \to \epsilon$

$B \to \epsilon$



Two productions are reduced so R-R conflict = Not LR(0)

given grammar is not LR(0) because LR(0) is faulty state which will contain R-R conflict

→ Check for SLR(1)

follow(A) = b, a

follow(B) = a, b

	a	b
0	r3/r4	r3/r4

⇒ not SLR(1)

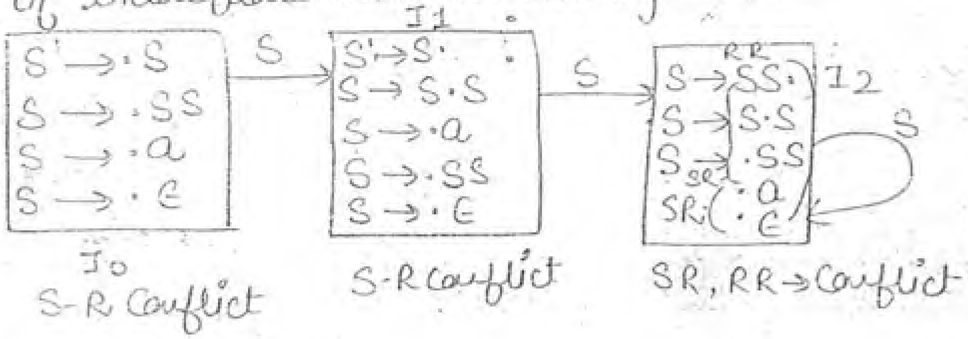
Q Consider the following grammar-

$S \to SS \mid a \mid \epsilon$

find the number of inadequate state in LR(1) grammar

soln $S' \to S$

$S \to SS \mid a \mid \epsilon$



	a	b
2	r1/r3	r1/r3
2	r1/S3	r1
2	r3/S3	r3

4-SR, 1-RR → (3) → I.S.

CLR(1) Grammar:-

Conflicts in LR(0)

$A \to \alpha.$, $B \to .a\beta$ = SR; $I_i \xrightarrow{a} I_5$

$A \to \alpha.$, $B \to \beta.$ = RR; I_j

I_i

	a	\$
i	r1/S5	r1

I_j

	a	\$
j	r1/r2	r1/r2

LR(0) → item

LR(0) SLR(1)

⇓

LR(1) item

⇓

LR(0) item + 1 - look ahead symbol.

LR(1) item

CLR(1) LALR(1)

Qn. $S \to AA$

$A \to aA \mid b$

⇓

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

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

① $S' \to \cdot S, , \$$

② $S \to \cdot A A, \$$

③ $A \to \cdot aA, a|b$

$\cdot b, a|b$

Q. Find the closure of LR(1) item

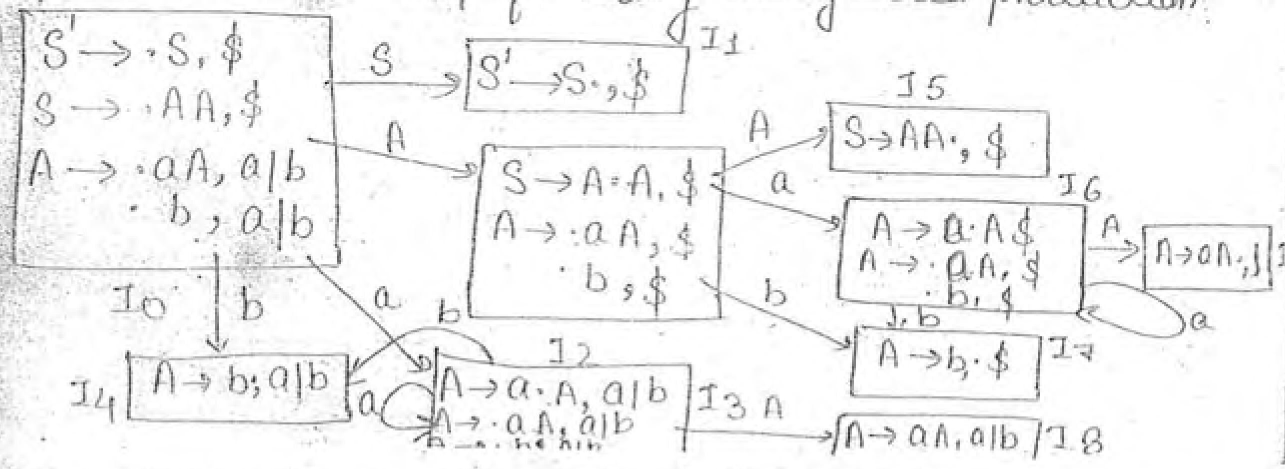
Closure of Item (I) =

1) Add the same LR(1) item. ($S' \to \cdot S, \$$)

2) If $A \to \alpha \cdot B(\beta, \$)$ is LR(1) item I, and $B \to C$ is in G. then add $B \to \cdot C$, first(β, \$) to closure of LR(1) item I.

3) Repeat the second step for every newly added production.

Solⁿ



The given grammar is CLR(1), because no state contain conflicts

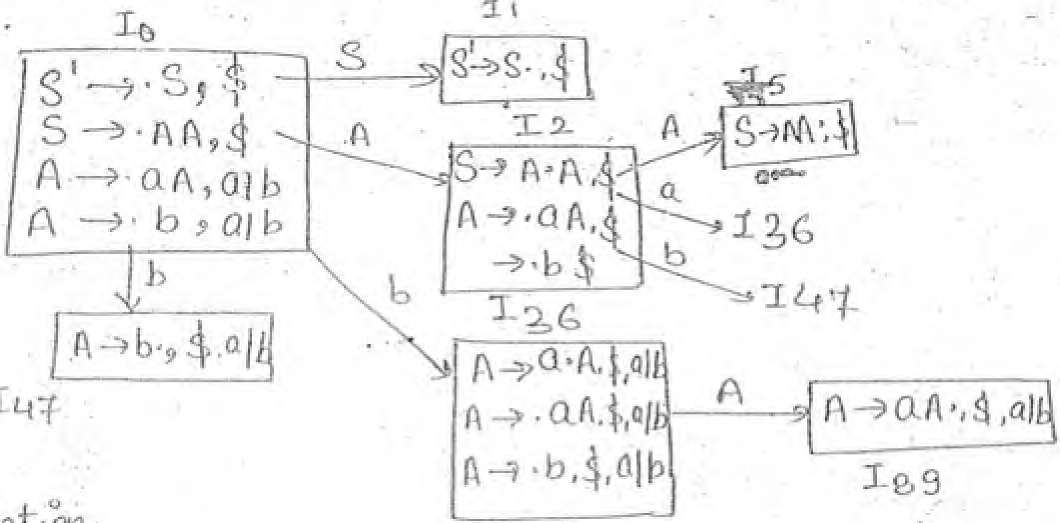
Relation b/w the states of LR(0), SLR(1), CLR(1)-

LR(0)	CLR(1)	SLR(1)	LALR(1)
⇓	⇓	⇓	⇓
n_1	n_3	n_2	n_4

$\boxed{n_1 = n_2 \le n_3}$ $\boxed{n_1 = n_2 = n_4 \le n_3}$

LASLR(1) Parsers

It will combine the two states, which are differe only by lookahead symbol.



minimization

Q1. Check the following grammar is CLR(1) or not -

S → AaAb | BbBa

A → ∈

B → ∈

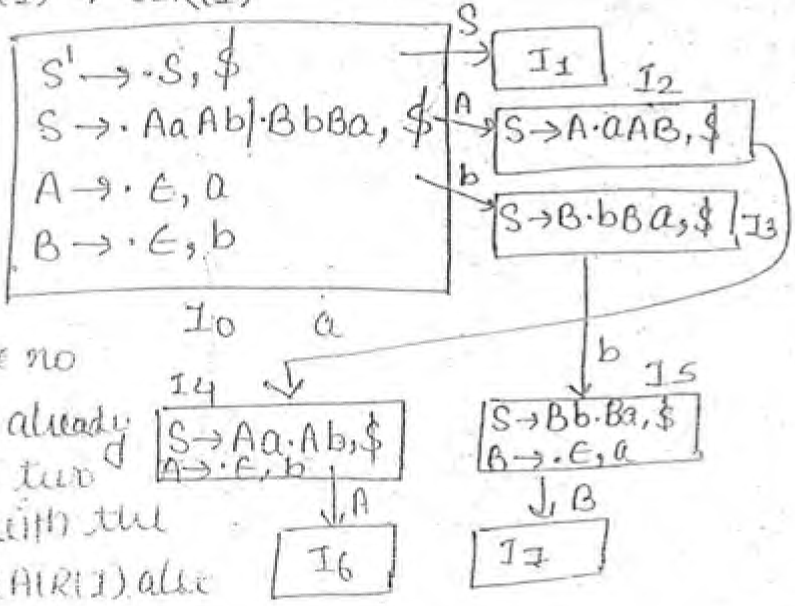
Sol: LL(1) → LALR(1) → CLR(1)

S' → S

S → AaAb | BbBa ⇒

A → ∈

B → ∈

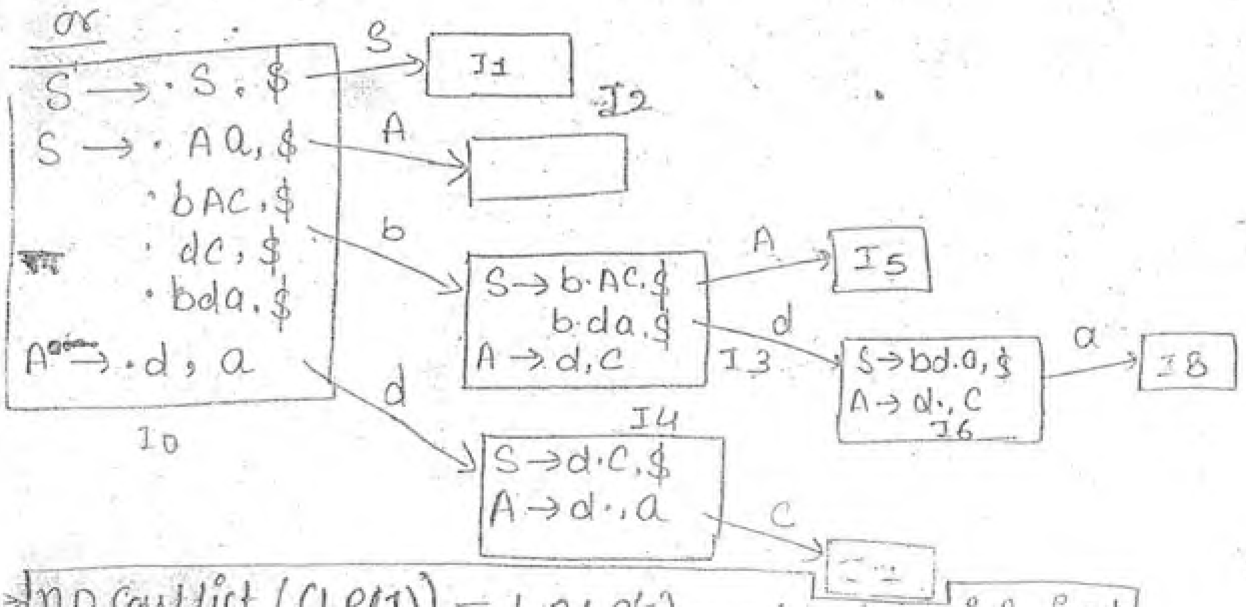


The given grammar is CLR(1) grammar because no conflicts. The given dfa is already minimized, because no two states are differ only with the look ahead symbol (ALR(1) also

Q. Check the following grammar is CLR(1) or not→
S→AA/bAC|dc| bda
A→d

Soln S'→S
→AQ|bAC|dc|bda ⇒ S'→·S,$
S→·AQ|·bAC|·dc|·bda, $
A→d A→·d,d

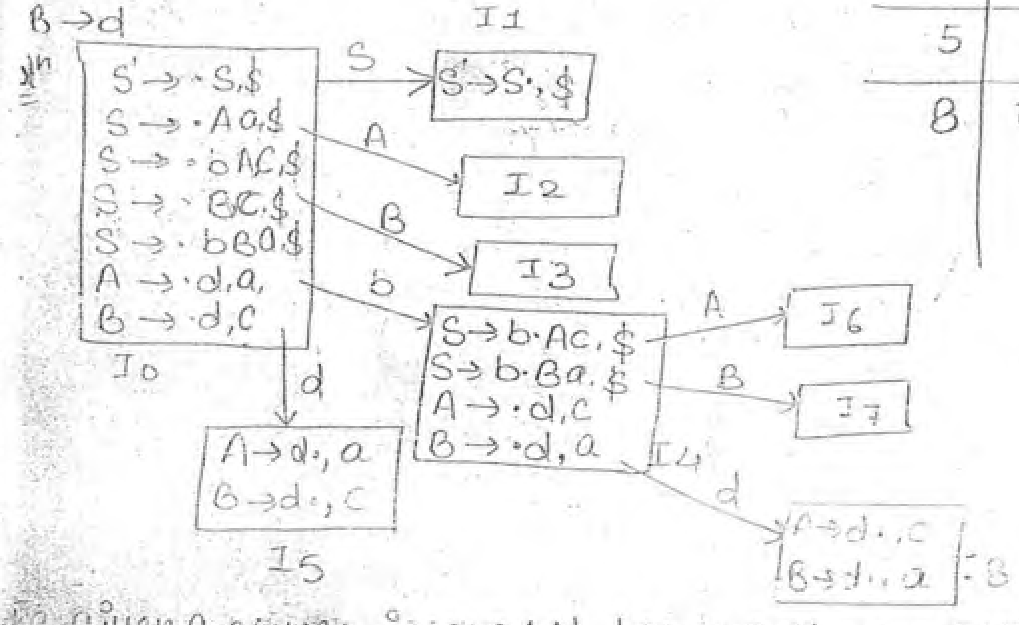
or



→ no conflict (CLR(1)) = LALR(1) = already minimized.

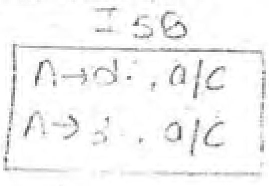
Q. Check the following grammar is LALR(1) or not -
S→Aa|bAC|BC|bBa
A→d
B→d

Soln



	a	c
5	r5	r6
8	r6	r5

LALR(1)



	a	c
58	r5, r6	r5, r6

The given grammar is CLR(1), because of no conflicts The given grammar is not LALR(1), because after combining states 5 and 8 there is conflict

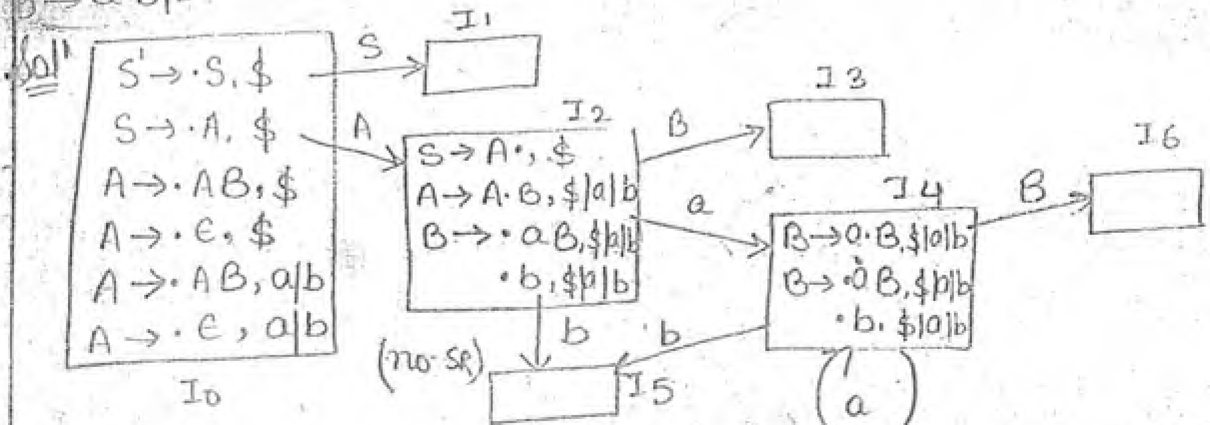
Q: Check the following grammer is LALR(1) or not-

S → A

A → AB | ∈

B → a B | b

Sol



given grammar is CLR(1). It is LALR(1) also, because it is already minimized.

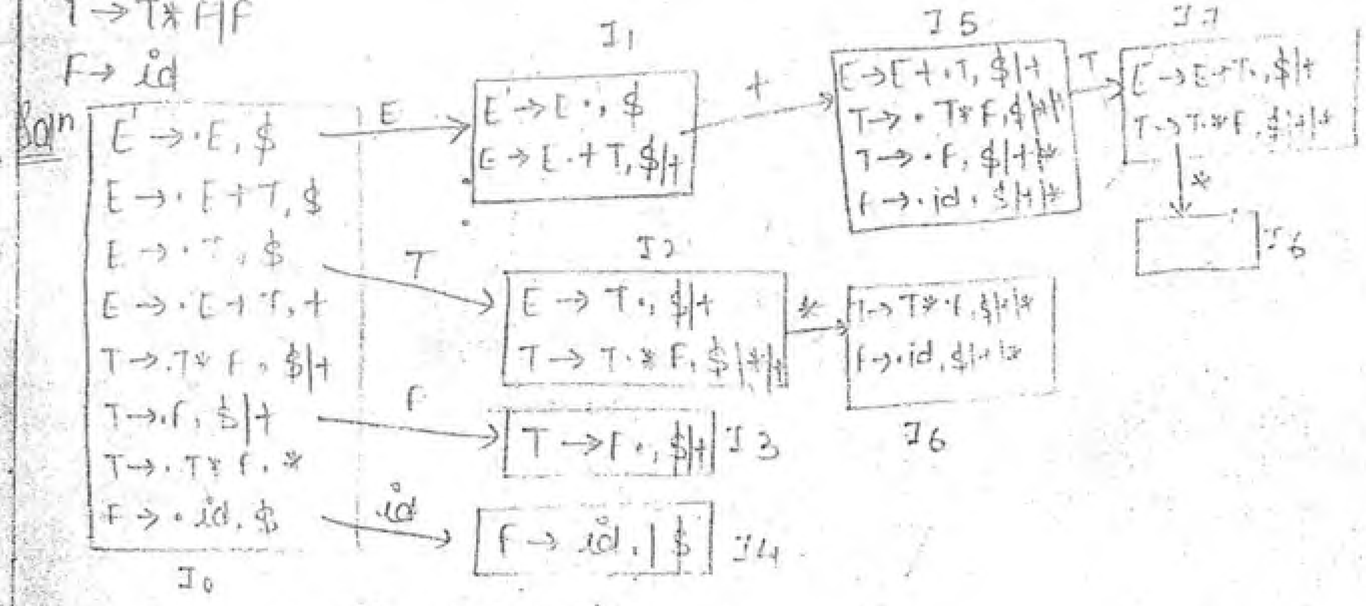
Q: Check the following grammar is LALR(1) or not-

E → E+T | T

T → T* F | F

F → id

Soln



It is CLR(1). It is LALR(1) also

GATE QUESTIONS

Q. Consider SLR(1) and LR(1) table for the given CFG:-
Which of the following is true.

a) GO TO of both the tables may be different

Shift entries are identical in both the tables

Reduce entries in both the tables may be different

None of above.

1.2 Let, SLR(1) parsing table will take n_1 rows, LALR(1) will take n_2 rows, relation b/w n_1 and n_2 :-

$\boxed{n_1 == n_2}$ Ans

1.3 Consider the following CFG-

S → CC

C → cC/d

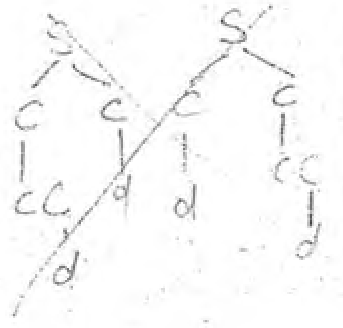
on which one of the following is true:-

LL(1)

SLR(1) not LL(1)

LALR(1) not SLR(1)

CLR(1) not LALR(1).



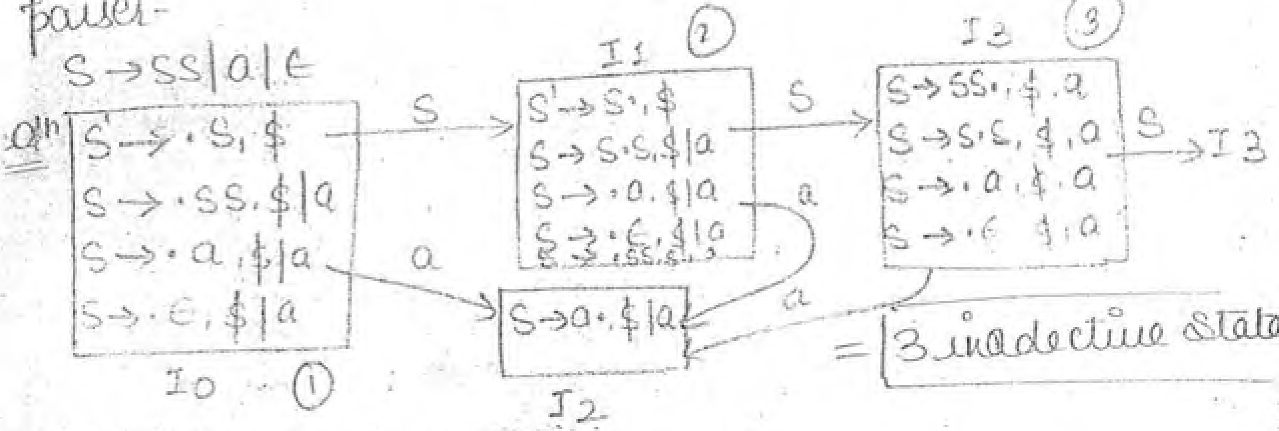
$\boxed{\text{LL(1)} \rightarrow \text{surely LALR(1)}}$

⇓

Have to check for LL(1)

1.4 Find the number of inadequate states while constructing CL(1) parser-

S → SS|a|ϵ



= $\boxed{\text{3 inadective states}}$

Ans

1.5 Consider the following grammar:-

A → AA|(A)|ϵ is not suitable for Operator precedance parser bcoz

ambiguous

left recursion

right recursion

none of the above (operator Grammar)

1.6 Consider the following grammar-

S → (S)|a

LR(1) → n_1

LR(1) → n_2

LR(1) → n_3

$\boxed{n_1 = n_3 \le n_2}$

a) n_1

b) n_1

c) n

d) n_1

Solⁿ

S

S·

S-

** Not

suff

** Not

pre

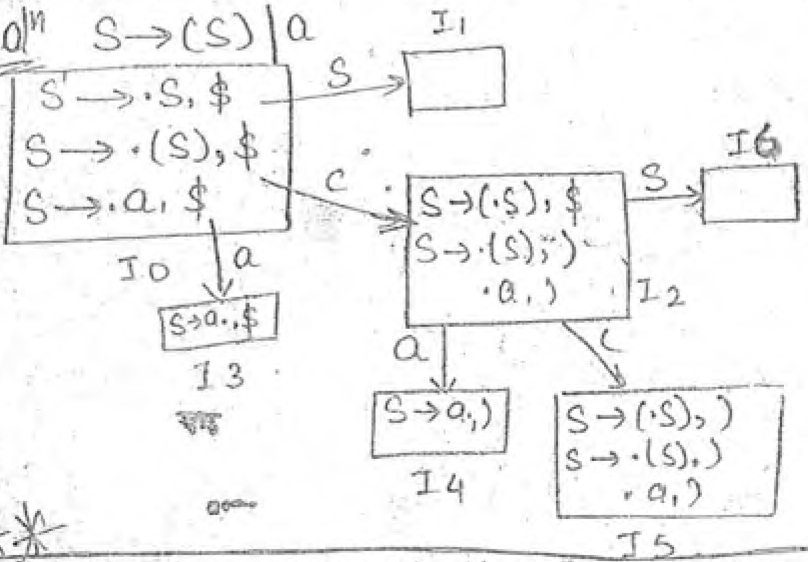
a) $n_1 < n_2 < n_3$

b) $n_1 = n_3 < n_2$

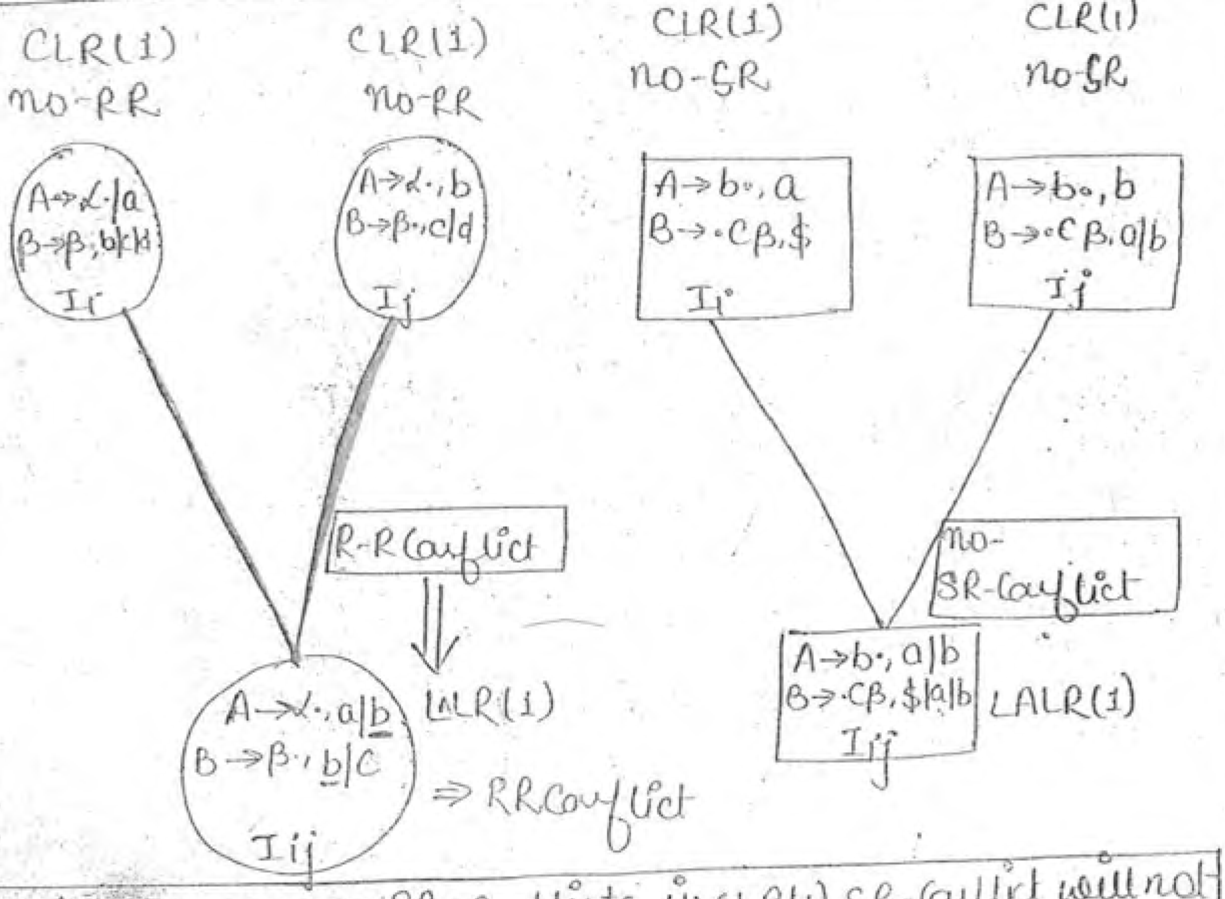
c) $n_1 = n_2 = n_3$

d) $n_1 \geq n_3 \geq n_2$

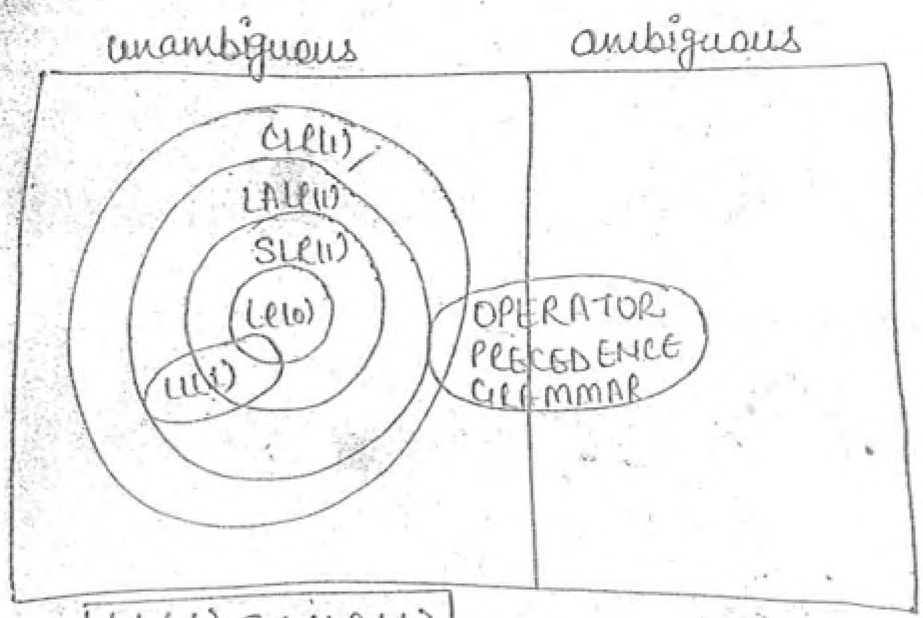
Sol^n $S \rightarrow (S) \mid a$



**
Note1:- Even though, there are no R-R conflict in CLR(1), still RR conflict may present in LALR(1).



*
Note 2:- If there are no SR-Conflicts in CLR(1), SR-Conflict will not present in LALR(1) also.



$\boxed{LL(1) \subsetneq LR(1)}$

$\boxed{LL(k) \subseteq LR(k)}$

$\boxed{LL(1) \subseteq LALR(1)}$

$S \to a|ab$: it is LALR, not LL(1)

Note-1

① Bottom up parsers are more complex to design as compared to T.D.P

② Bottom up parser is accepting more no of grammars comparing with the top down parser

③ $\boxed{\text{Size of Bottom Up Parser table} = 2 * \text{top down Parser table size}}$

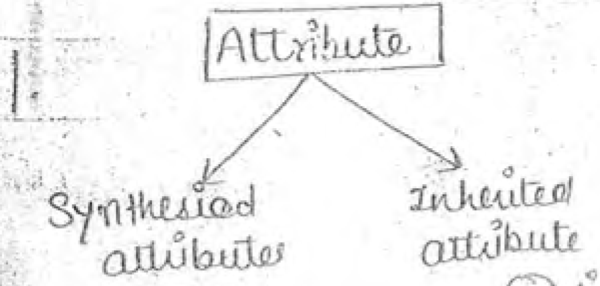
Dated
9.Dec.10

Chapter No.3

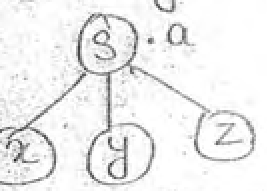
SYNTAX DIRECTED TRANSLATION (SDT)

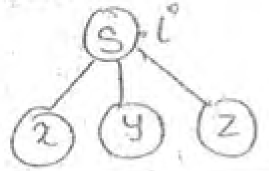
$$\boxed{\begin{array}{c} G \\ + \\ \text{Semantic rules} \\ \text{(or)} \\ \text{Translation rules} \end{array}} \Longrightarrow \text{SDT}$$

Syntax tree
or
Parse tree



$S \rightarrow xyz$





$\boxed{y.i = f(x.i \mid S.i \mid z.i)}$

$\boxed{S.a = f(x.a \mid y.a \mid z.a)}$

Applications of syntax directed translations

* Converting the given infix expression to postfix expression
* evaluating the given infix expression.
* Binary to decimal conversion.
* Creating syntax tree
* Creating directed acyclic graph
* To generate intermediate code
* Storing the data into symbol table

Construct Syntax Directed Translation (SDT) to convert the given the infix expression to postfix expression

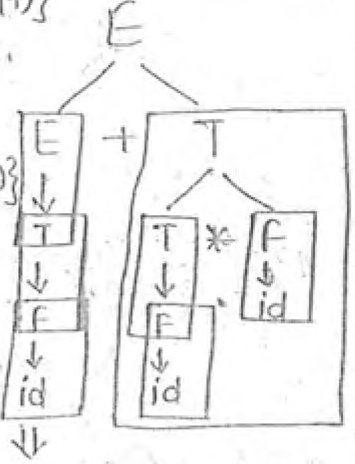
I/P: 2+3*4

O/P: 234*+

Grammar:— E → E+T{printf(+)}
          |T{–}

T ⟶ T*F{printf(*)}
     |F{–}

F ⟶ id{printf(id)}



⇓

F→id{printf(id)}

ON Construct SDT to find out no of reductions to evaluate the giv infix expression:-

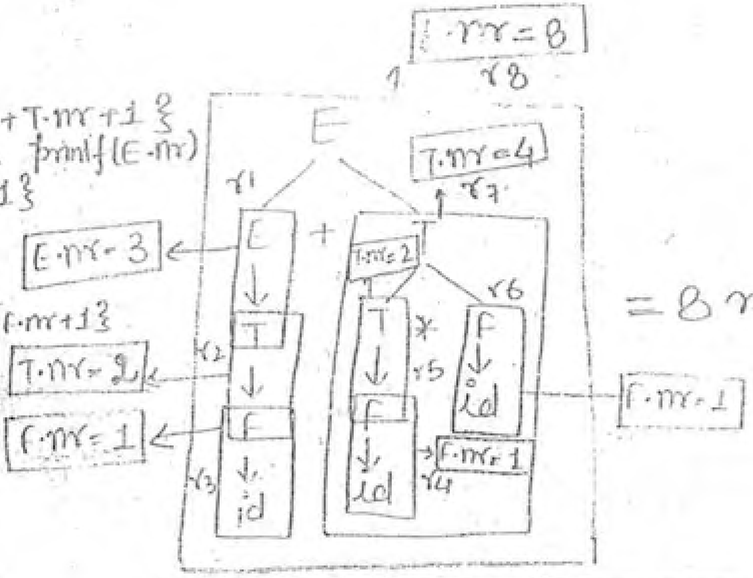
i/p = 2+3*4

O/P = 8

E→E+T {E.nr=E.nr+T.nr+1} printf(E.nr)
   |T {E.nr=T.nr+1}

T ⟶ T*F{T.nr=T.nr+F.nr+1}
   |F {T.nr=F.nr+1}

F→id {F.nr=1}



= 8 reductions

attribute = nr = synthesized attribute

ON Construct SDT to evaluate the given infix expression-

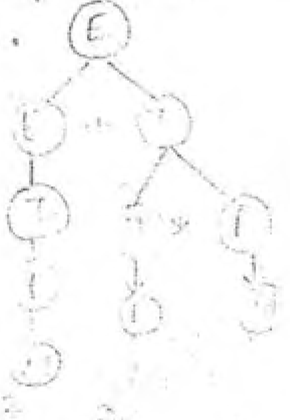
I/P = 2+3*4

O/P = 14

E → E₁+T } printf (E₁.val+T.val)}
   |T {E.val = T.val}

T → T₁*F {T.val = T₁.val * F.val}
   |F {T.val = F.val}

F → [illegible]



1M. Construct SDT to find the bits in the given binary number.

Soln I/P: 101011 | 1011.111

O/P : 6 | O/P = 7

S → L | L.L {printf (L₁.nb+L₂.nb}

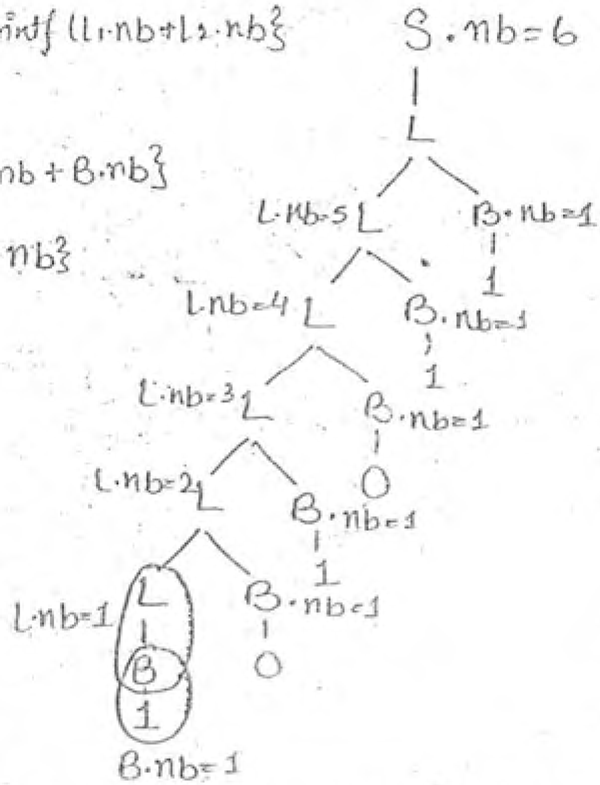
↓ {printf(L.nb)}

L → L B {L.nb=L.nb + B.nb}

| B {L.nb= B.nb}

B → 0 {B.nb= 1}

| 1 {B.nb= 1}



2M. Construct a SDT to convert given decimal no into binary number:-

I/P: 101.101

O/P: 5.625

Soln S → L / L.L ⇒ {printf(L₁.DV + $\frac{L_2.DV}{2^{L_2.DV}}$)}

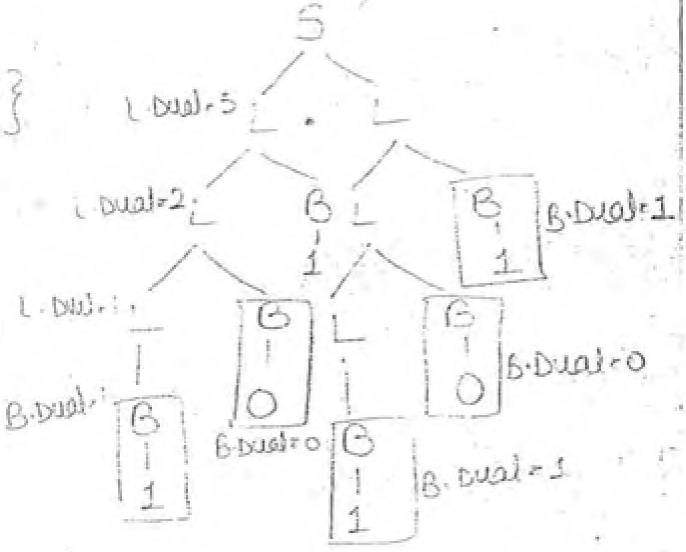
↓ printf{(L.DV)}

L → L B {L.nb = L.nb + B.nb ; L.DV = 2* L.DV + B.DV}

| B {L.DV = B.DV ; L.nb = B.nb}

B → 0 {B.nb = 1 ; B.Dual = 0}

| 1 {B.Dual = 1 ; B.nb = 1}



2M. Construct SDT to convert the given infix expression into prefix expression:-

I/P:- 2*3+4

O/P :- + * 2 3 4

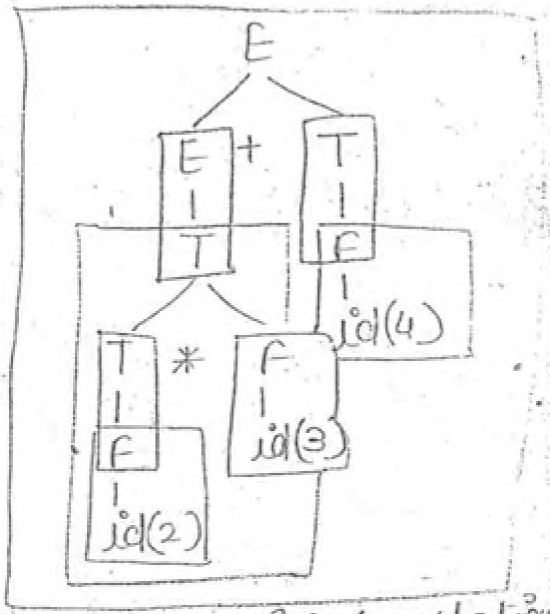
Sol^n $E \rightarrow E + T \Rightarrow \{printf(+)\}\ E+T$

$\quad |\, T$

$T \rightarrow T * F \Rightarrow \{printf(*)\}\ T*F$

$\quad |\, F$

$F \rightarrow id \Rightarrow \{printf(id)\}$



Qn. [GATE] Consider the grammar with the following translation rules:- $E as the start symbol-

$E \rightarrow E_1 \# T\ \{E.val = E_1.val * T.val]$

$\quad |\, T \quad \{E.val = T.val\}$

$T \rightarrow T_1 \$ F\ \{T.val = T_1.val + F.val$

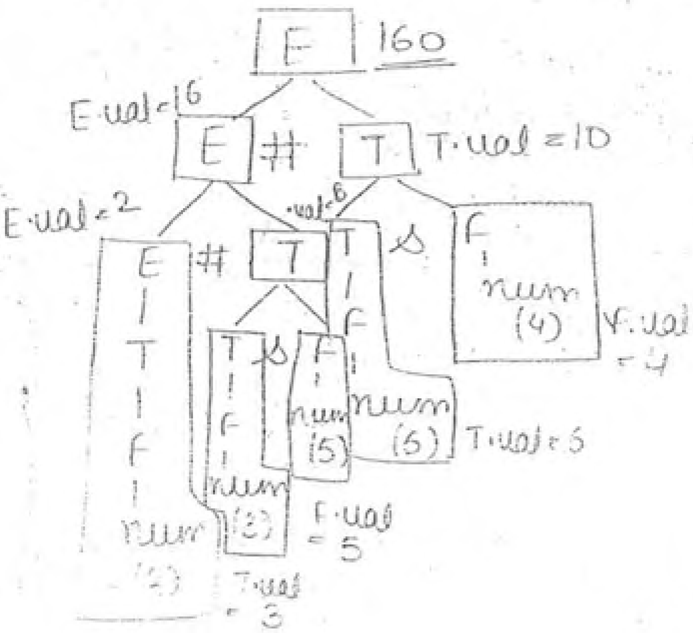
$\quad |\, F\ \{T.val = F.val\}$

$F \rightarrow num\ \{F.val = num\}$

Compute the E.val for the root of the parse tree for the expression-

$\boxed{2 \# 3 \$ 5 \# 6 \$ 4}$

Sol^n



$\boxed{E.val = 160}$ Ans

4.2 E → E#T {E.val = E1.val * T.val} |
|T {E.val = T.val}

T → T\$F {——} T.val = T.val − F.val
|F {T.val = F.val}

F → num {F.val = num}

(4.a) If the expression 8#12\$4#16\$12#4\$2 is evaulauated to 12, which one of following is correct-

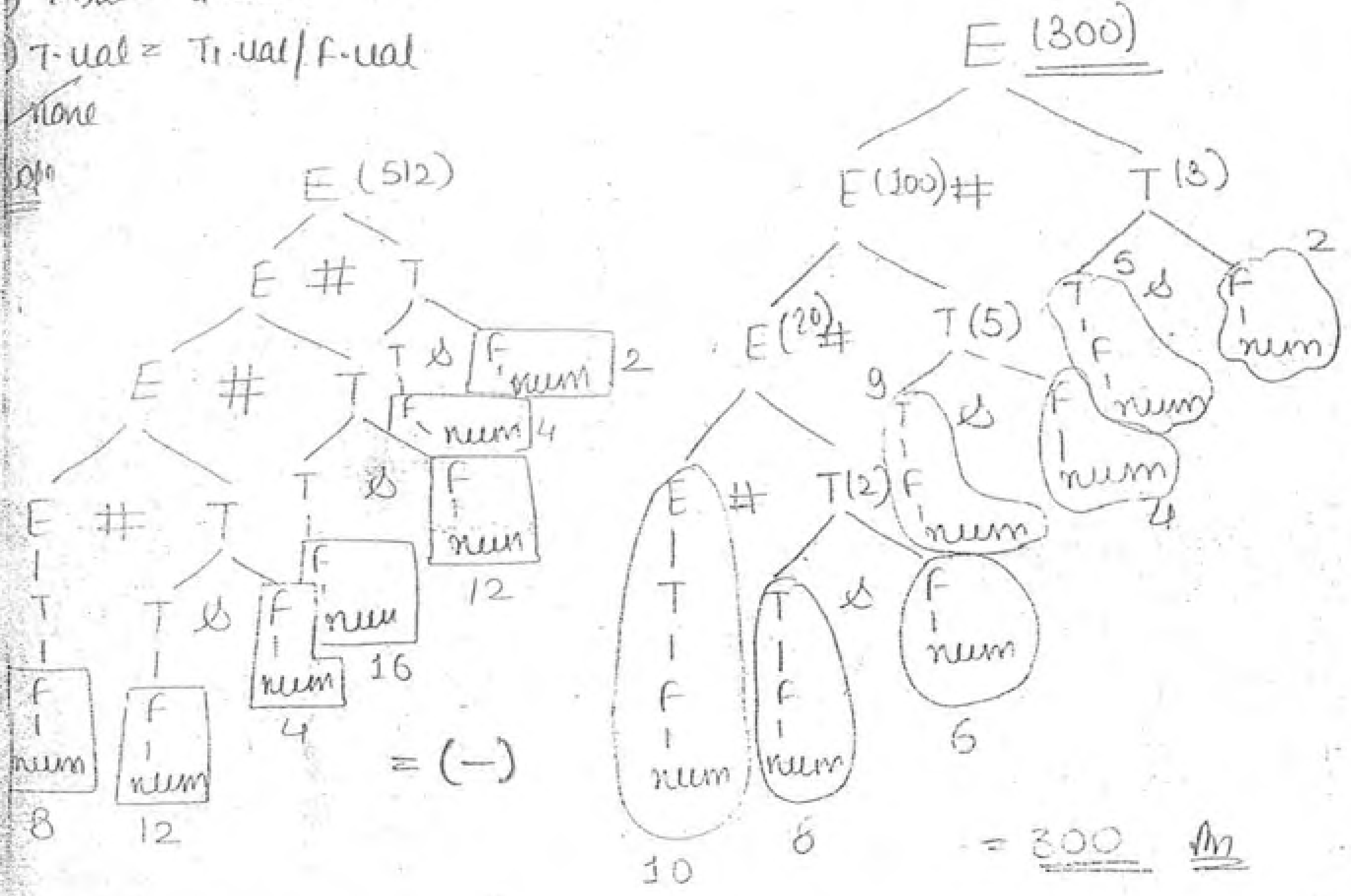
) T.val = T1.val * F.val

) T.val = T1.val + F.val

) T.val = T1.val / F.val

✓ None

QN(b) Compute 10#8\$6#9\$4#5\$2



QN If the given grammar is-
S → TR
R → + T {printf(+)} R | ε
T → num {printf(num)}

If the I/P is- 9+5+2, what will be the O/P-

a) 9+5+2

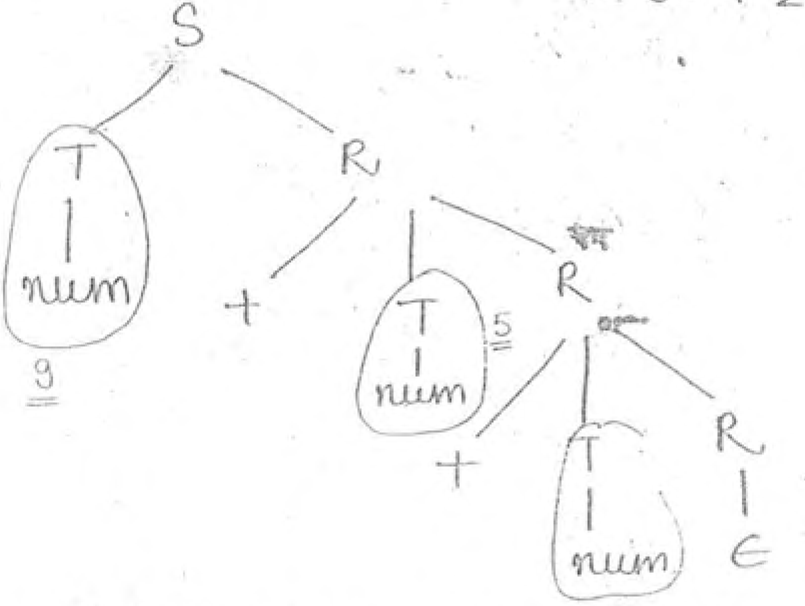
b) ✓ 95+2+

c) 952++

d) ++952

⇒ 95+2+

Soln



Construct the SDT to store type info into symbol table.

I/P: int x,y,z;

O/P:

V.name	V.type
x	int
y	int
z	int

Soln D → D₁, id {D.type = D₁.type} {addtype(id, D₁.type)}

| T id {D.type = T.type} {addtype(id, T.type)}

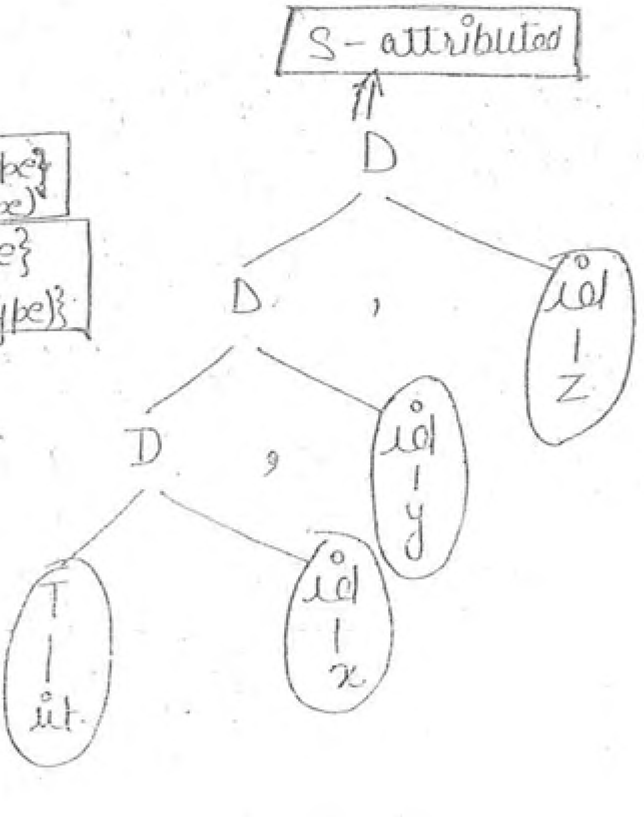
T → int {t.type = int}

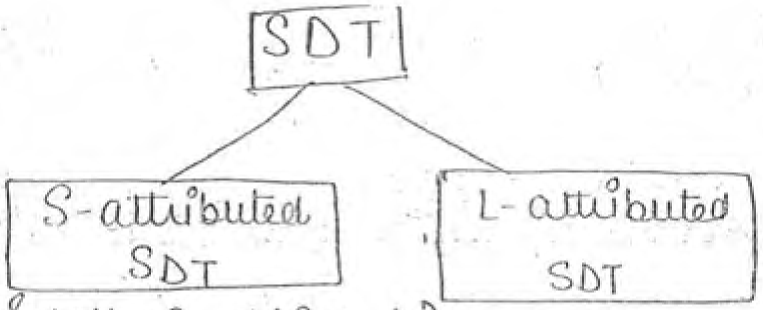
| float {t.type = float}

| char {t.type = char}

id → a|b|c

|z





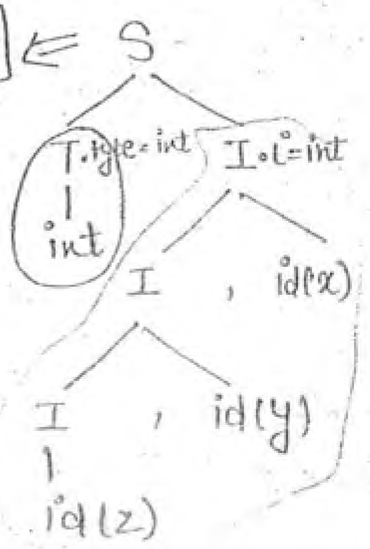
Synthuized (Postorder) attribute

Left to right

) Problems Using Inherited attributes (Preorder evaluation)

QH S → T I {I.i = T.type}

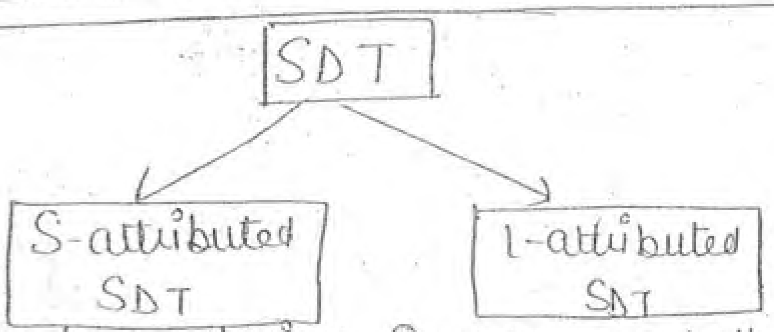
L-attributed ⇐



T → int {T.type = int}
  | char
  | float

I → I₁, id {A I₁.i = I.i}
{add type(id, I₁.type)}.
  id → {addtype(id, I₁.type)}

: If one required value is on L.H.S and other value is on R.H.S., no nethod other than inherited attribute.



we will use only synthesizer ttibute

① we can use both synthesizer and inherited attribute But in the case of inherited attribute, only, values from root or the left siblings can be taken not from right sibling

Semantic rules will be placed fter end of the production

② We can place the semantic rule, anywhere at the right hand side of production evaluation is ③ The order of evaluation - Preorder.

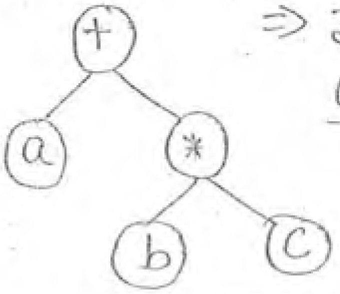
attributed definitions are also S-attributed definition

$\boxed{S\text{-attribute} \subseteq L\text{-attribute}}$

QH. Construct SDT to construct syntax tree for the given infix expression:-

I/P: a+b*c

O/P:



⇒ In the syntax tree leaf nodes are Operands and intermediate nodes are operator.

⇒ makenode(lp, id, rp) = Creating a node

⇓

will return the location, where the node is created.

Grammar

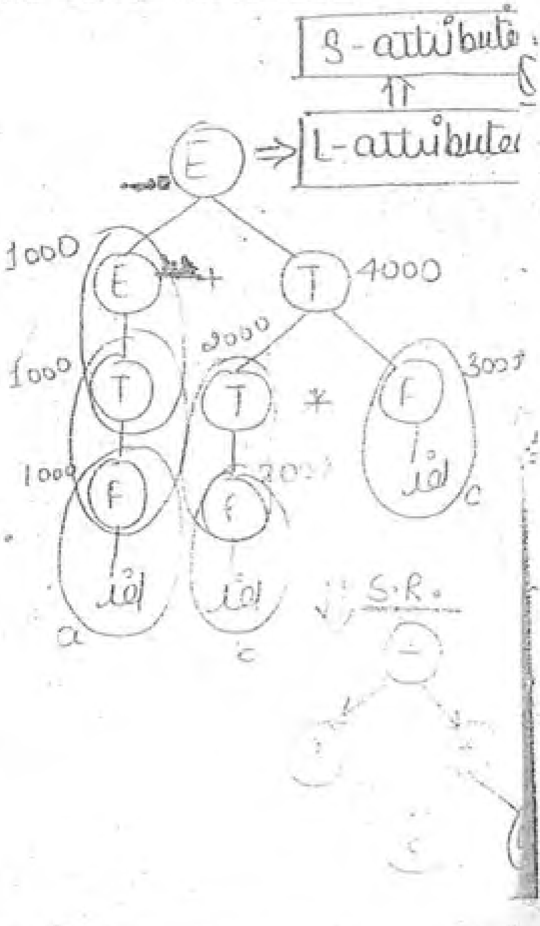
E → E₁ + T {mknode(E₁.ptr, +, T.ptr)}

| T {E.ptr = T.ptr}

T → T₁ * F {T.ptr = mknode(T₁.ptr, *, F.ptr)}

| F {T.ptr = F.ptr}

F → id {F.ptr = mknode(N, id, N)}



1 Give a grammar to reverse the given infix expression-

I/P: (a+b) * (c+d)

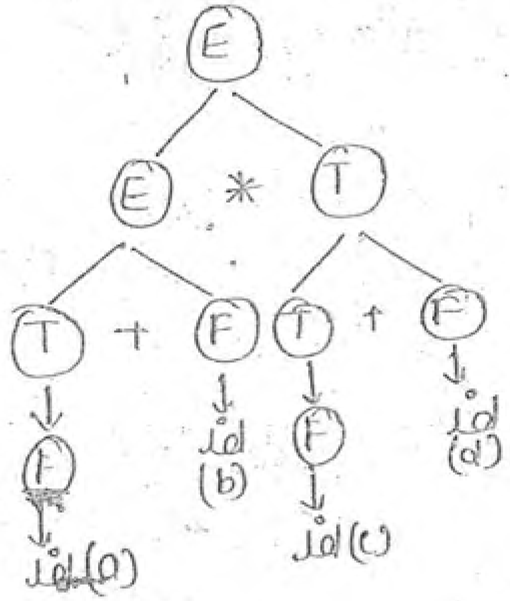
O/P: (d+c) * (b+a)

Sol

Solⁿ E → E*E | (T)

T → T+F | F

F → id



Semantic rules

E → E*E

→ (T)

→ T+F

| F

F → id {F.val = id}

IV. Construct SDT to generate Three address code (Intermediate code) for the given infix expression:-

i/p: x = a+b*c → newtemp() = Create temporary variable

o/p: t1 = b*c

t2 = a+t1 → gen(t=b*c)

x = t2

Solⁿ S → id = E {gen(id = E.val)}

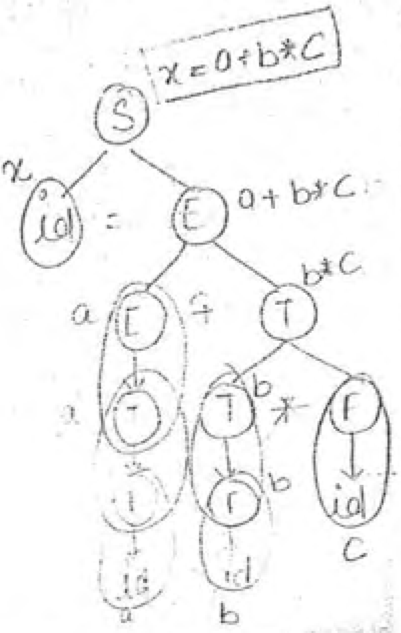
E → E+T {Q = newtemp()
gen(Q = E1.val + T.val)
E.val = Q}

| T {E.val = T.val}

T → T1*F {P = newtemp();
gen(P = T1.val * F.val
T.val = P}

| F {T.val = F.val}

F → id {F.val = id}



QH GATE

Consider the SDT shown below:-

$S \rightarrow id = E \{ gen(id.place = E.place) \}$

$E \rightarrow E_1 + E_2 \{ t = newtemp(); \; gen(t = E_1.place + E_2.place) \; E.place = t; \}$

$E \rightarrow id \{ E.place = id \}$

Here gen is a function that generates the O/P code and new temp(); is a function that returns the name of new temporary variable at every call. Assume that ti are the new temporal variable name, generated by new temp for the statement $\boxed{x = y + z}$, of the three address code generated by the above SDT is-

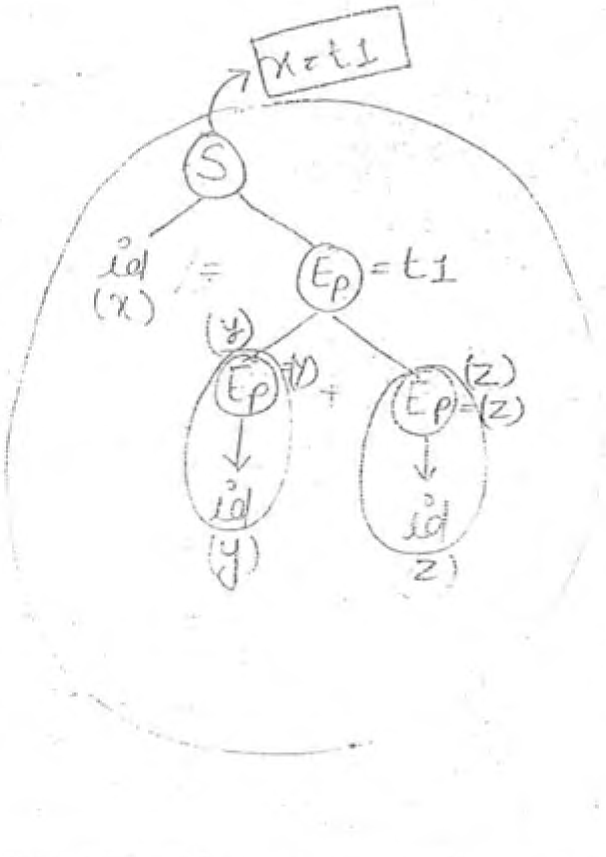
a) $x = y + z$

b) $t_1 = y, \; t_2 = t_1 + z$
$x = t_2$

c) ✓ $t_1 = y + z$
$x = t$

d) $t_1 = y. \; t_2 = z$
$t_3 = t_1 + t_2, \; x = t_3.$

Soln $t_1 = y + z$
$\boxed{x = t_1}$



1. Consider the following SDT:-

E→ number {E.val = number}

| E+E {E.val = E1.val + E2.val}

| E*E {E.val = E1.val * E2.val}

YACC

⇓

$\boxed{\text{Yet Another Compiler Compiler}}$

Soln I/P: 3*2+1

YACC (Give more priority to shift (Push), rather than reduce (pop)

*	+	

(3, (2, 1, +) *)

QH a) The above grammar and semantic rule is given to YACC tool for parsing and evaluating arithmetic expressions, which one of the following is true, about the action of YACC for given grammar-

2

i) It detects recursion and eliminate

ii) It detects reduce-reduce conflicts and resolves

iii) It detect shift-reduce conflict and resolve the conflict and resolves in favour of shift over reduce.

iv) resolves favour of reduce over shift

b) Assume the conflict in QH(a), what will be the precedance and associativity for the expression- $\boxed{3*2+1}$

i) equal precedance and left associative, evaluated to 7.

ii) Equal precedance and right associativity, evaluated to 9.

YACC tool = LALR(1) parser generator

φ Parser → no multiple value entry

⇓

LL(1) or LR(1) ⇒ LL(1). Because in LR(1) there is YACC tool.

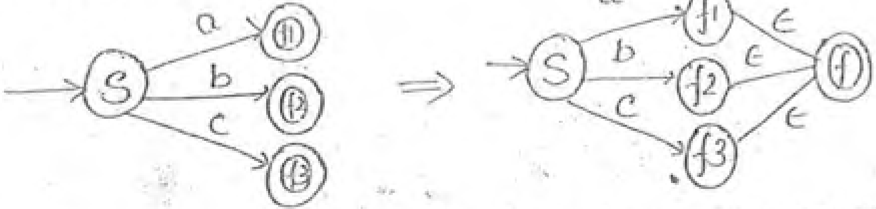
Dated 28.Dec.10

FA→RE

Steps

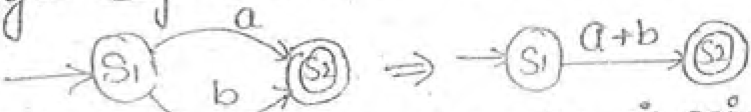
Step-1 If more than 1 final state is there, make it as single final by adding ε-transition

Exp:-



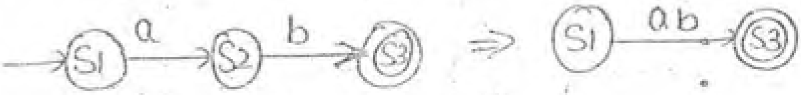
* A NFA with more than one final state can be converted into equivalent NFA with single final state, but it is not possible in case of DFA.

Step-2 If more than 1 edge going in same direction make it as single edge and label with union with symbol.

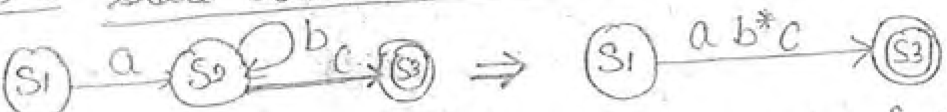


Step-3 If more than one edge is going in same direction one after another, making it as single edge, with the label of concatenation of symbols.

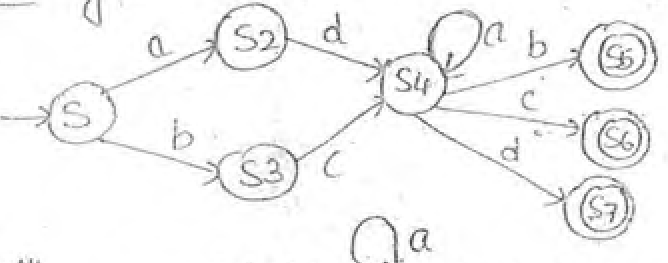
Exp:-



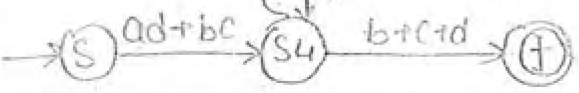
Step-4 State Elimination method



QN.1 Give the equivalent the regular expression:-

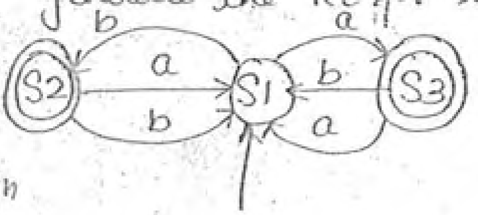


Soln

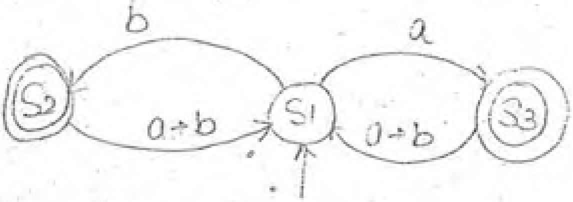


$= \boxed{(ad+bc)\, a^* \,(b+c+d).}$ Ans

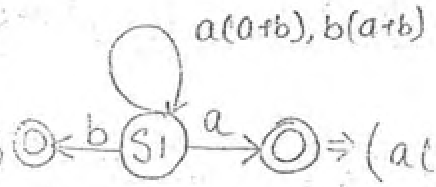
QN Generate the RE for the following automata :-



Sol



⇒



$\Rightarrow (a(a+b)+b(a+b))^* (a+b).$

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

$= ((a+b)^2)^* (a+b)$

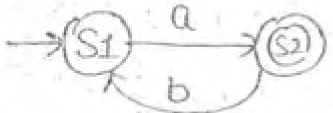
⇓

Even length string followed by a or b

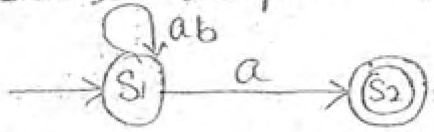
⇓

Odd length string Ans

QN Give the RE for the following finite automata :-

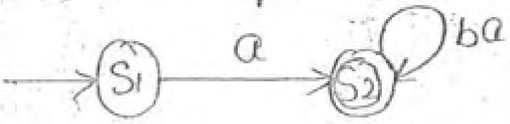


Maintain the loop at S1 :-



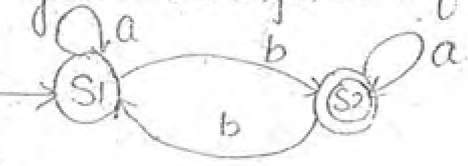
$\Rightarrow (ab)^* a$

Maintain the loop at S2 :-

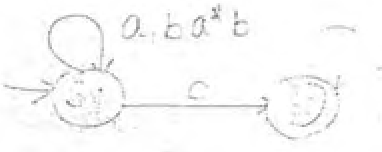


$\Rightarrow a(ba)^*$

QN Give the RE for the following finite automata :-



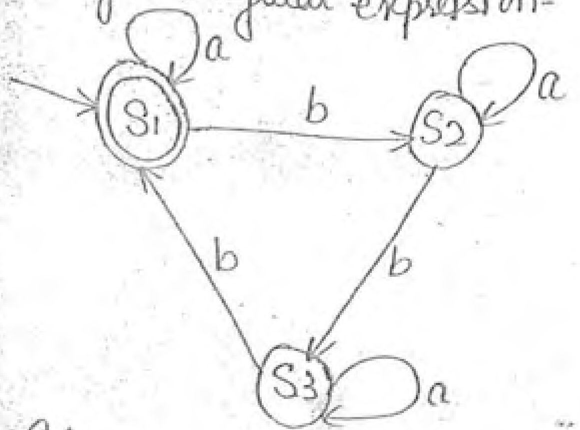
Sol



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

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

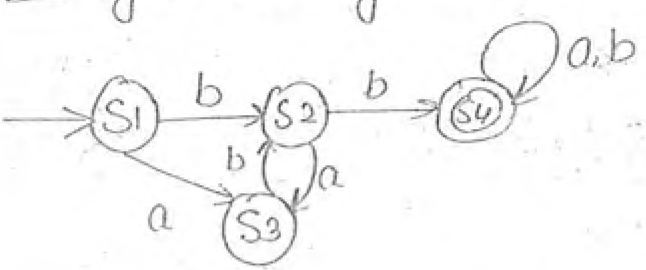
QN Give regular expression-

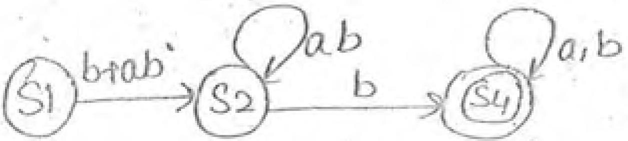


Soln

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

QN Give the regular expression:

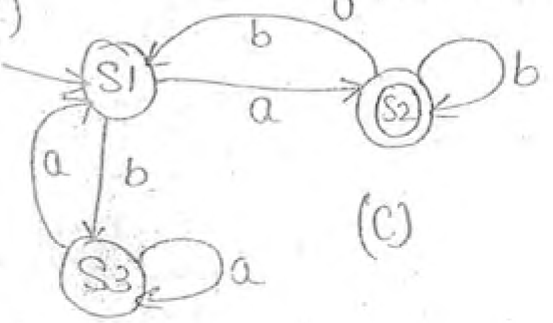




$$= (b+ab)(ab)^* \, b\,(a+b)^*$$

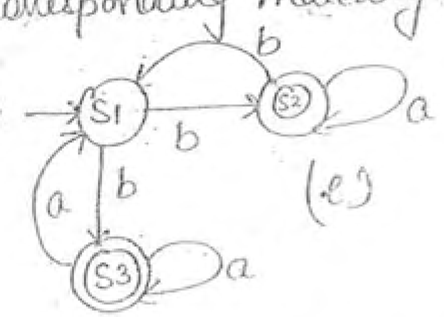
QN Match each of the NFA, with corresponding matching option-

1)



(C)

(5)



(e)

a) $(aa^*b + ba^*b)^* \, ba^*$

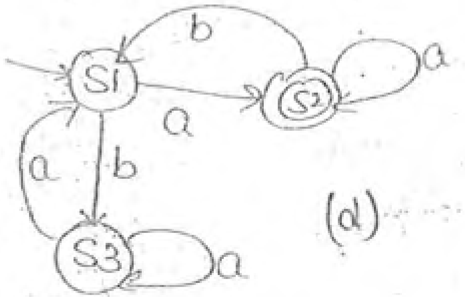
b) $(aa^*a + aa^*b)^* \, aa^*$

c) $(ba^*a + ab^*b)^* \, ab^*$

d) $(ba^*a + aa^*b)^* \, aa^*$

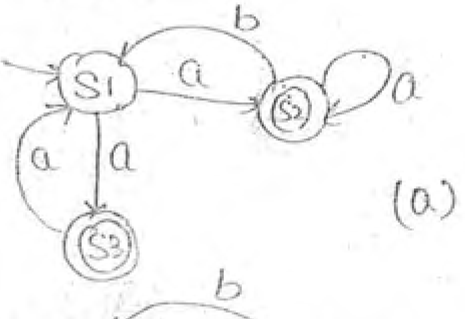
e) $(ba^*a + ba^*b)^* \, ba^*$

2)



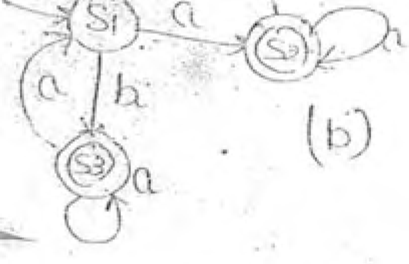
(d)

3)



(a)

4)



(b)

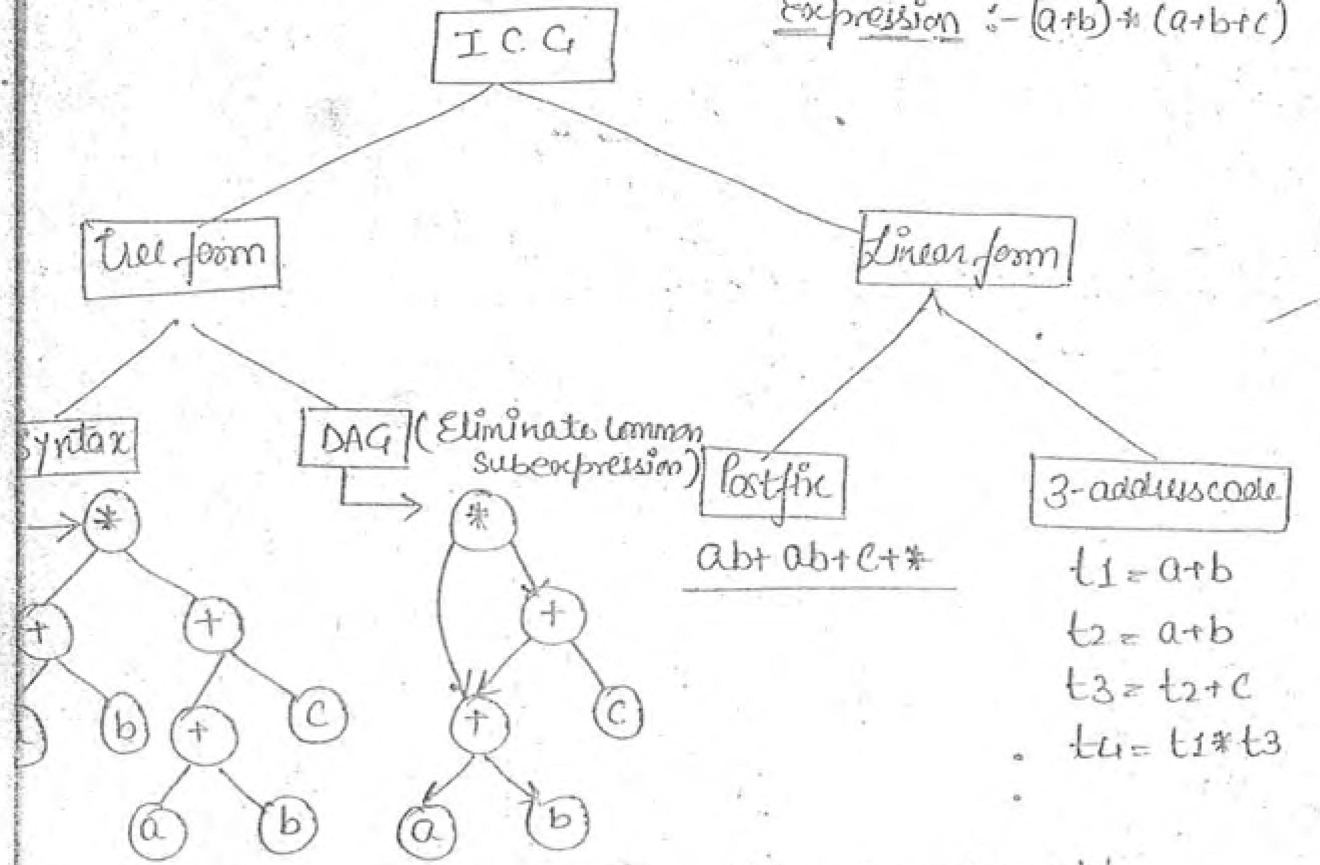
Dated 3. Dec. 10

Chapter No. 4

Intermediate Code Generation

Representation of intermediate code generation

Expression :- (a+b)*(a+b+c)



ab+ ab+c+*

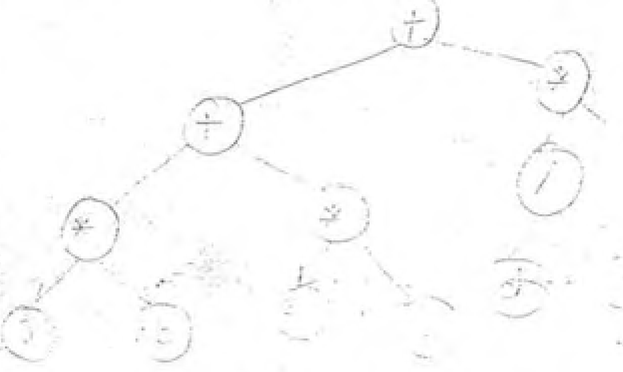
t1 = a+b
t2 = a+b
t3 = t2+c
t4 = t1*t3

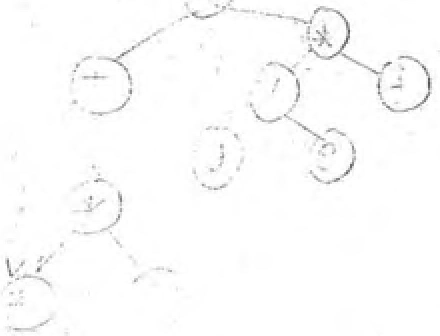
DAG :- atleast one node with indegree '0' and outdegree '0'.

Q.9 (a*b)+(a*b*c)+d/e*f

Soln Syntax tree

DAG





Postfix

ab* ab*c*+ de/f*+

3-address code

$t_1 = a*b$

$t_2 = a*b$

$t_3 = t_2*c$

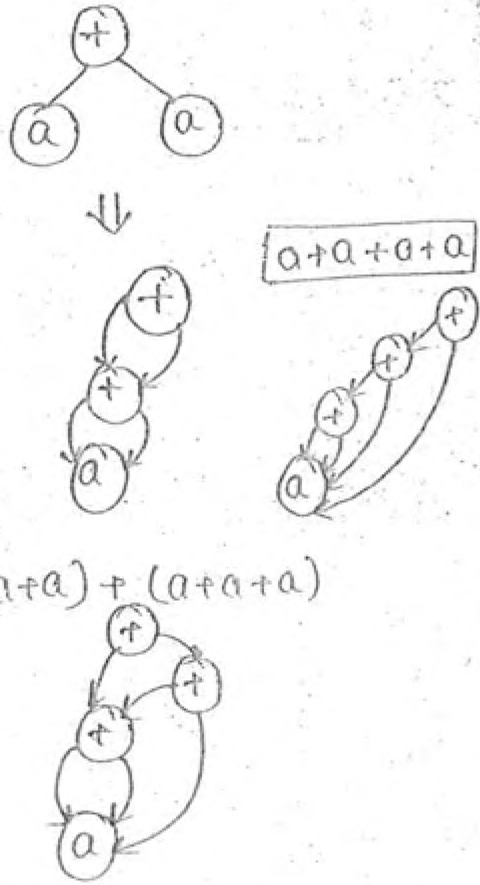
$t_4 = t_1+t_3$

$t_5 = d/e$

$t_6 = t_5*f$

$t_7 = t_4+t_6$

DAG – (a+a)+(a+a)



a+a+a+a

(a+a)+(a+a+a)

Types of three address code

1) $x = y \text{ op } z$

2) $x = \text{op } y$

3) $x = y$

4) $x = *y$

5) $x = \& y$

6) $x = a[i] \Rightarrow *(a+i)$

7) $a[i] = x$

8) goto L (unconditioned jump)

9) if $x<y$ goto L

No three address code

- $x = a[i,j]$
- $x = fun(a,b)$

Q) Construct three address code for the following expression
if a<b then t=1 else e=0

Sol It is not a three address code.
⇓ Conversion in three address code

```
i)   if a<b goto i+3
(i+1) e=0
(i+2) goto i+4
(i+3) t=1
(i+4) —
```

⇗ Back patching (filling gaps)

Q) If a<b && c>d then t=1 else e=0

Soln It is not in three address code.
⇓ Conversion in three address code

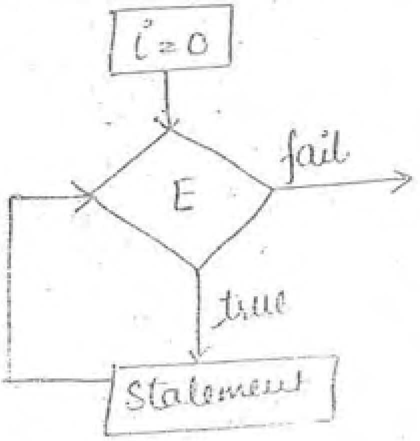
```
i)    if a<b goto i+1
(i+1) if c>d goto i+4
(i+2) e=0
(i+3) goto i+5
(i+4) t=1
(i+5) —
```

```
i)    if a<b goto i+2
(i+1) goto i+3
(i+2) if c>d goto i+5
(i+3) e=0.
(i+4) goto i+6
(i+5) t=1
(i+6) —
```

Q) Construct three address code for 'while' statement in C language.

Soln

```
i=0
while(i<10)
{
  x=a+b*c;
  i++;
}
```

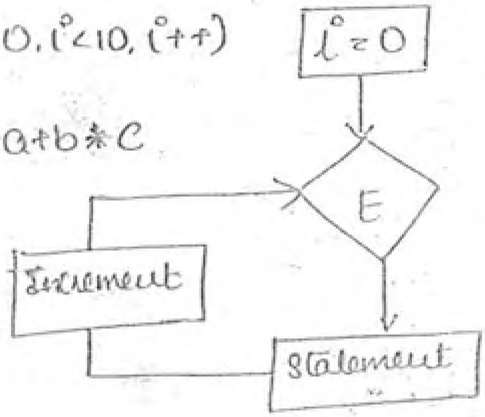


Three address code (condition)

```
     i=0
S)   if i<10 goto S+2   (i if true)
S+1) goto S+7   else outside
S+2) t1=b*C
S+3) t2=a+t1
S+4) x=t2
S+5) i=i+1
S+6) goto S
S+7) —
```

QN Construct three address code for 'for' loop in c language.

Soln
```
for(i=0, i<10, i++)
{
    x = a+b*c
}
```



```
S+0) i=0
S+) if i<10 goto S+2
S+1) goto S+7
S+2) t1 = b*c
S+3) t2 = a+t1
S+4) x = t2
S+5) i = i+1
S+6) goto S
S+7) ___
```

```
for(i=0; i<10; i--)
    a = b+c;
}
If i=0 goto S+1
t1 = a+1
a = t1
```

QN Construct a three address code for switch statement in 'C' language

Soln
```
i=1
switch(i)
{
  case1: x1 = a1+b1*c1
         break;
  case2: x2 = a2+b2*c2
         break;
  default: x3 = a3+b3*c3
}
```

Three address code

```
     i = 1
S)   if(i==1) goto case1
S+1) if(i==2) goto case2
S+2) t1 = b3*c3
S+3) t2 = a3+t1
S+4) x3 = t2
S+5) ___
case1:) t1 = b1*c1
        t2 = a1+t1
        x1 = t2
        goto S+5
case2:) t1 = b2*c2
        t2 = a2+t1
        x2 = t2
        goto S+5
```

Q11 Construct three address code for. $x = a[i][j]$, suppose $a[10][20]$

Sol^n It is not a three address code.

⇓ Conversion in three address code.

$x = a[i][j] = *(*(a+i)+j)$

$t1 = i*20$ $a[5,10]$

$t2 = t1+j$ $5*20 = 100$

$x = a[t2]$ $+10$

110

Representations of three address code

1) Quadroples

2) Triples

3) Indirect Triples

Expression :- $-(a+b)*(a+b*c)$

1) Quadraples → Advantage - Can move the result.
Disadvantage - More space

S.No.	OP	OP1	OP2	Result
1	+	a	b	t1
2	–	t1		t2
3	*	b	c	t3
4	+	a	t3	t4
5	*	t2	t4	t5

Memory (Quadraples)

OP1	OP2	result
a	b	t1
t1		t2
b	c	t3
a	t3	t4
t2	t4	t5.

2) Triples

S.No.	OP	OP1	OP2
1	+	a	b
2	–	(1)	
3	*	b	c
4	+	a	(3)
5	*	(2)	(4)

Advantage :-

* Less space.

* Can't move the result at desired place. → disadvantage

3) Indirect touples

→ If there is a requirement, then we can move the result to some another location by copying the same values.

Advantages

* less space is required.

* Results can be move.

Q4 (a+b) * (a+b+c) * d/e + f

Sol^n Quadraples

S.No.	OP.	OP1	OP2	Result
1	+	a	b	t1
2	+	a	b	t2
3	+	t2	c	t3
4	*	t1	t3	t4
5	*	t4	d	t5
6	/	t5	e	t6
7	+	t6	f	t7

Triples

S.No.	OP	OP1	OP2
1	+	a	b
2	+	a	b
3	+	(2)	c
4	*	(1)	(3)
5	*	(4)	d
6	/	(5)	e
7	+	(6)	f

Indirect Triples

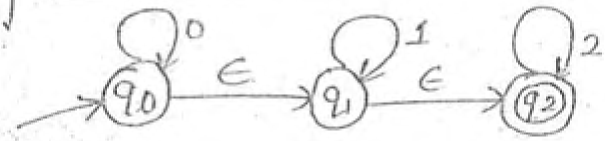
S.No.	OP	OP1	OP2	Copy
1	+	a	b	
2	+	a	b	
3	+	t2	c	
4	*	t1	t3	500
5	*	t4	d	
6	/	t5	e	
7	+	t6	f	

~~ated~~
~~6-Dec-2010~~

Chapter No. 5

CODE OPTIMIZATION

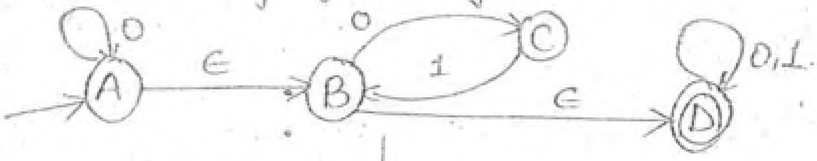
ϕ E-NFA ⇒ NFA



Sol^n Conversion

	0	1	2
→q0*	q0,q1,q2	q1,q2	q2
q1*	ϕ	q1,q2	q2
q2*	ϕ	ϕ	q2

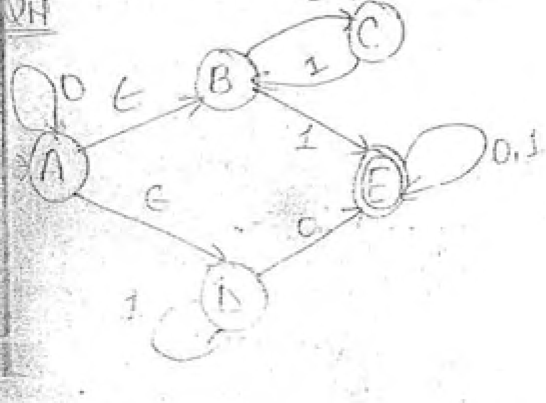
Q^N Construct NFA for following E-NFA:-



Sol^n

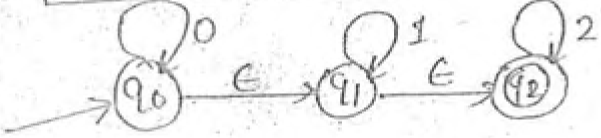
	0	1
→A*	A,B,C,D	D
B*	C,D	D
C	ϕ	B,D
*D	D	D

Q^N

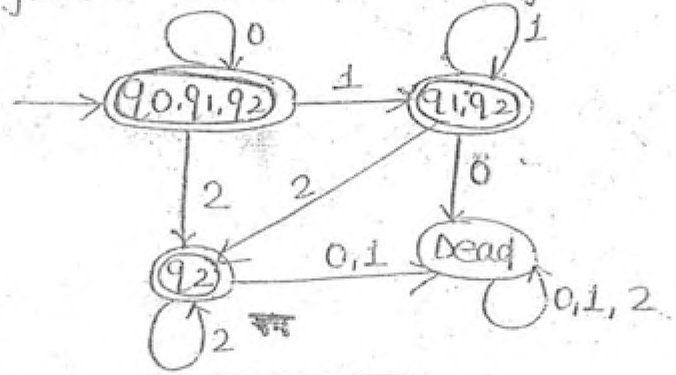


	0	1
→*A	A,B,C,D,E	D,E
B	C	E
C	ϕ	B
D	E	D
*E	E	E

ф $\boxed{\epsilon\text{-NFA} \rightarrow \text{DFA}}$



- Find ε-closure to starting state, then-



ф Quotient Operation

If L_1 is regular and L_2 is also regular, then L_1/L_2 is also regular.

$$\boxed{L_1/L_2 = \{x \mid \text{if } xy \in L_1 \text{ and } y \in L_2\}.}$$

Exp:- $L_1 = \{b^2, b^4, b^6, b^8, \text{---}\}$

$L_2 = \{b\}$

$L_1/L_2 = \{b, b^3, b^5, b^7, \text{----}\}$

$L_3 = \{a\}.$

$L_1/L_3 = \{\ \}$

$L_2/L_1 = \{\ \}.$

Exp:- $L_1 = \{101, 011, 0010, 00\}.$

$L_2 = \{0, 1\}$

$L_3 = \{00\}.$

$L_1/L_2 = \{001, 0, 10, 01\}.$

$L_1/L_3 = \{\epsilon\}$

$L_3/L_2 = \{0\}.$

Note 1 $\boxed{\text{In } L_1/L_2, \text{ if } L_2 \text{ contain } \epsilon, \text{ then } L_1/L_2 = L_1 \cup \{\ \}.}$

Note-2 $\boxed{\text{If } L \text{ is non empty, then } \frac{\Sigma^*}{L} = \Sigma^*}$ If L is empty the $\frac{\Sigma^*}{L} = \{\}$. (Nomatching)

Note-3 $\boxed{\text{If } L \text{ is non-empty, then- } \frac{L}{\Sigma^*} = \text{all the prefixes of } L.}$

$$\frac{a}{(a+b)^*} = \epsilon, a$$

$$\frac{TOC}{(A+B+\cdots Z)^*} = \epsilon, T, TO, TOC = \text{all the prefixes of } L.$$

Code Optimization

- * Loop optimization
- * Strength Reduction
- * Redundency Elimination
- * Dead code elimination
- * Constant folding
- * Copy propagation
- * Algebric simplification

Loop Optimization

⇓

① Loop invarient (code motion)

② loop unrolling (Decreasing testcases)

③ Loop jamming (Loop combine)

1 Loop Invarient

```
i=0
while(i ≤ 10,00,000)
{
          10        20
   x = [Sin(A) * Cos(B)] * i
   i = i+1;
}
```

→ not varying (invariant)

⇓ take invariant Code Outside the while loop.

```
i=0
t= Sin(A)* Cos(B)
while ( i≤ 1,00,000)
 {
   x= t*i;
   i= i+1;
 }.
```

2 <u>Loop Unrolling</u> (Decreasing test cases).

```
while ( i≤ 10,00,000)
  {
    x[i]=i;
     i++;
  }.
```

10,00,000 test cases

⇒

```
while (i≤ 10,00,000)
 {
   x[i]= i;
   i++;
   x[i]= i;
   i++;
 }
```

5,00,000 test Cases.

3 <u>Loop Jamming</u> (Loop Combine).

```
i=1;
while (i≤ 10,00,000)
 {
   x= a+b*i;
   i= i+1;
 }
```

10,00,000 Comp

```
i=1;
while ( i ≤ 10,00,000)
 {
   y= c+d*i
    i++;
 }.
```

10,00,000 Comp

⇒

```
i=1
while (i≤10,00,000)
 {
   x= a+b*i;
   y= c+d*i;
     i++;
 }.
```

⇓ **Strength Reduction** :- Replacing costlier operation by less cost Operation or replacing lower speed operator to the higher speed operator

Exp:- n ⇒ *
⇓ ⇓
2*n Left shift

4*i ⇒ * (lesstime)
⇓ ⇓
i+i+i+i +

Constant Folding :-

Fold all the constants and give one equivalent value.

$$a = b + \boxed{5+10+15+25}$$

$$\Downarrow$$

$$a = b + \boxed{55}$$

Copy Propagation :- Unnecissarily Don't propagate the constant by copying one by one into another variable.

Exp:-
```
PI = 3.14
x = PI
y = x*100
Z = 100
a = y/Z
```

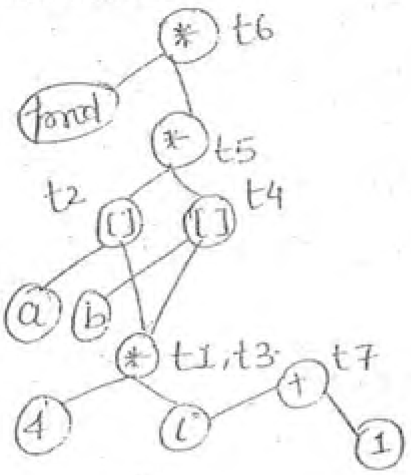
Redundancy Elimination :- Use DAG Data Structure

```
A = b+C
B = 2+b+3+C
C = C+1+b
```

⇓

```
A = b+C
B = 5+A
C = A+1
```

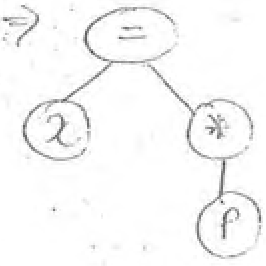
Exp-2
```
t1 = 4*i
t2 = a[t1]
t3 = 4*i
t4 = b[t3]
t5 = t2*t4
t6 = prod*t5
t7 = i+1
```
⇒



⇓

```
t3 = 4*i
t2 = a[t3]
t4 = b[t4]
t5 = t2*t4
t6 = prod*t5
t7 = i+1
```

x = *P ⇒



Dead Code Elimination :-

Exp
```
x = t1;
a[t1] = t2
b[t2] = a[t1]
printf(b[t2])
```
⇒
```
a[t1] = t2
b[t2] = a[t1]
printf(b[t2])
```
⇒ x is not at all useful.

§ **Algebric simplification**

A = A*1 }
B = B+0 } => don't use these type of operators

GATE Problems

QH Consider the following C pgm-

```
for(i=1; i<N; i++)
 {
   for(j=1; j<N; j++)
    {
      if(i%2)
       {
         x+= 4*j + 5*i
         y+= 7+4*j;
       }.
```

Then which one of the following is false:-

a) above pgm contain loop invariant.

b) above pgm contain common subexpression elimination.

c) above code contain strength reduction

d) None of the above.

- Common subexpression = 4*j
- strength Reduction = j+j+j+j
- for(i=1; i<N; i++)

```
      if(i%2)
  for(j=1; j<N; j++)
   {
     x+= 4*j + 5*i
     y+= 7+4*j
   }
```

QH Multiplication of a positive integer by a power of 2, can be replaced by left shift, which executes faster on most of the processors. This is an example of:-

a) Loop unrolling

b) strength Reduction

c) Dead code Reduction

d) None of above

Q19
```
i=1, j=0;
while (j≤n)
{
   i = 2*i   } N+1
   j = j+1;  }
}
```

for the above pgm, involving integers i, j and n, which one of the following is loop invariant:-

i) $i = j+1$

ii) $i = (j+1)^2$

iii) $j = 2^i$

iv) ✓ $i = 2^{j+1}$

⇓

$i = (2)^{j+1}$

Q4.20 S → AB|CA

B → BC|AB

A → a

C → aB|b

Sol^n Reduced form

① Eliminate all the states or variables which are not reachable from start symbol.

↳ S → AB|CA

B → BC|AB

A → a

C → aB|b

② Eliminate those variables and productions, which are unnecessary

S → ~~AB~~|CA

~~B → BC|AB~~

A → a

C → ~~aB~~|b

⇒

S → CA

A → a

C → b.

ϕ LA

Syntax

Semantic

I.C.G.

C.O.

T.C.G.

Chapter No.5

RUN TIME ENVIRONMENT

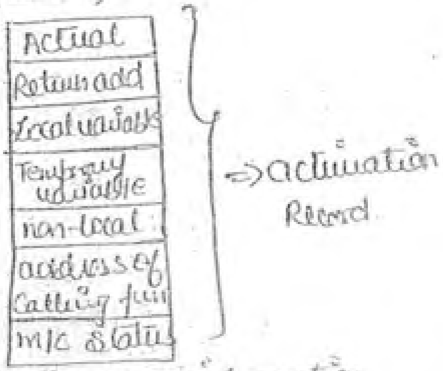
Environment
(Binding)

a
5000
memory
location

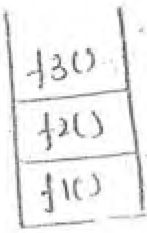
⇒ variable will be allocated to the multiple location at runtime. Variable will not changed.

f1() → f2() → f3()

(Activation record)



Control stack



⇒ all the current active function of the system in same order.

⇒ All of activation record first enter to the control stack

(This are the information that should be to f1() before the control is going to f1() → f2()

Storage Allocation

i) Static storage Allocation
ii) Stack storage Allocation
iii) Heap storage allocation

→ memory created only once (static variable) = Compilation time memory allocation ⇓ Can't be allocated at run time

* Static Storage Allocation
→ Memory is allocated at compilation time only
→ Bindings do not change at run time
→ One activation record for procedure
→ Recursion is not supported
→ Size of the object must be known as compile time itself (one time allocation of add)
→ Data structures can not be created dynamically (not allocated and deallocated dynamically

Exp:—

F(1)
F(2)
F(3)

2 Stack Storage Allocation:—

→ Whenever is a function is called, activation record is created and pushed it into the stack.

→ whenever a function ends, activation record is poped out from the stack.

→ At the time of running memory location will change.

→ Locals are bound to new activation record.

F(1)
F(2)
F(3)

⇒ 6

Disadvantage:—

Locals can not retained when activation ends i.e. function is over.

3 Heap Allocation

* Allocations and deallocation may be done in any order

Code Optimization
Runtime Environment } ⇒ Book

—— × —— × —— × —— × —— × ——

DBMS
1. NF
2. Relational Algebra
3. Transaction

CN
1. Data link layer
   - → Stop n wait
   - → Go back n
   - → Selective reject

   } → Tokenning
→ Jamming } →

2. N/W layer
   - → Subnetting
   - → Supernetting

3. Transport layer

OS
- → process magmt
- → Scheduling
- → Synchronization
- → memory magmt
- → Page replacement
- → Disk scheduling

Algo

□ Notes

DS

□ Notes

Compiler

□

Digital

Complete.

CO
- i) pipelining
- ii) Addressing modes
- iii) memory magmt
- iv) floating point

Maths
- → matrices
- → Graph
- → Rel^n and fun^n
- → lattices
- → group theory

Apti

4. Web Technology

XML
HTML } W3school.com

5. S/W Engg
- → Cyclomatic
- → Cocomo model

COMPILER