Page No.
Date

① Compiler Design ②

③ chapters:

(1) Overview of the compiler & its structure
(2) Lexical Analysis
(3) Syntactic Syntax Analysis
(4) Error Recovery
(5) Intermediate-Code Generation
(6) Run-Time Environments
(7) Code Generation and Optimization
(8) Instruction-Level parallelism.

④ Regular Expressions

(1) Construct R.E for the language which consists of exactly two b's over $\Sigma = \{a, b\}$

$$R.E = a^* b a^* b a^*$$

(2) w.a.r.E for the language of all strings of 0's and 1's with an even number of 0's and odd number of 1's.

$$R.E = 1(11)^* + (00)^*$$

⑤ R.E for (1) all strings containing 0's at least one 0 & at least one 1.

$\longrightarrow (0+1)^+$

(2) containing atmost one pair of consecutive 1's.

$$R.E = 0^* 1 1 0^*$$

(3) Containing 0's & 1's both even.
→ $(00+11)^*$

(4) all strings that do not end with 01.

R.E = $(0+1)^*(00+11+10)$

(5) R.E for binary strings with even length.
→ $(aa+ab+ba+bb)^*$

(6) All languages ~~start~~ that do not end with 01. ⊗

(7) All strings of digit that contain no leading 0's.
→ $([1-9][0-9]^*)^*$

⊛ Construct NFA from R.E. (Thompson's Construction).

(1) R.E = ε

ε-NFA → (q0) —ε→ (qf)

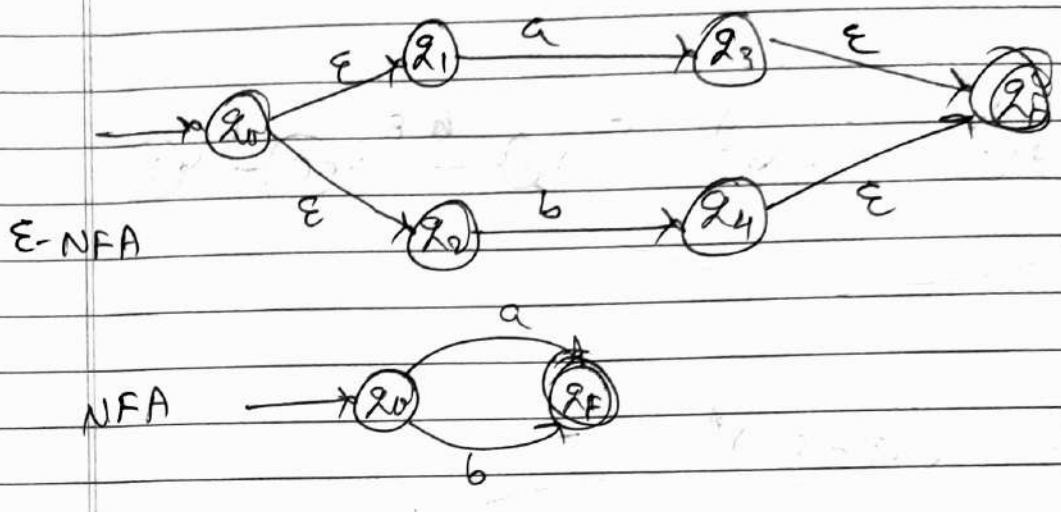
NFA → (qf)

(2) R.E = a

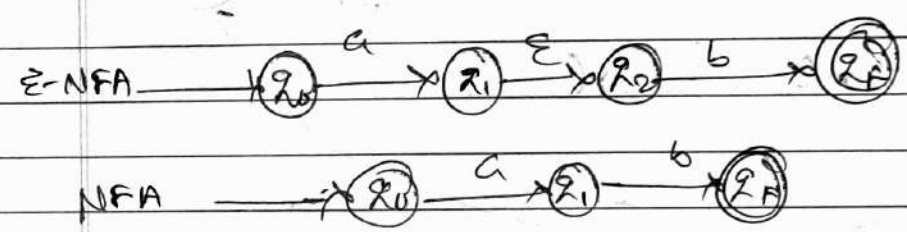
ε-NFA → (q0) —ε→ (q1) —a→ (q2) —ε→ (qf)

NFA → (q0) —a→ (qf)

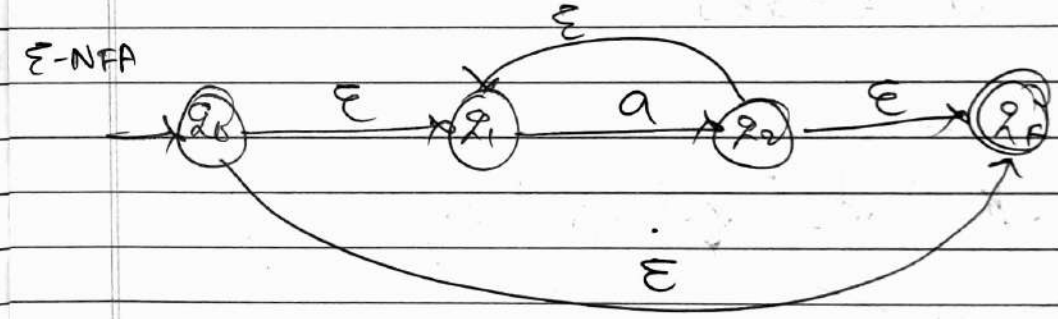
(3) Union Expression (a+b):-

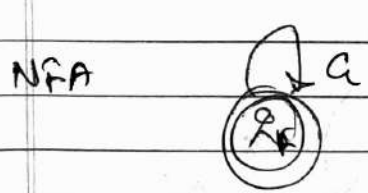


(4) Concatenation (a.b)



(5) Kleene Star operation (a*)



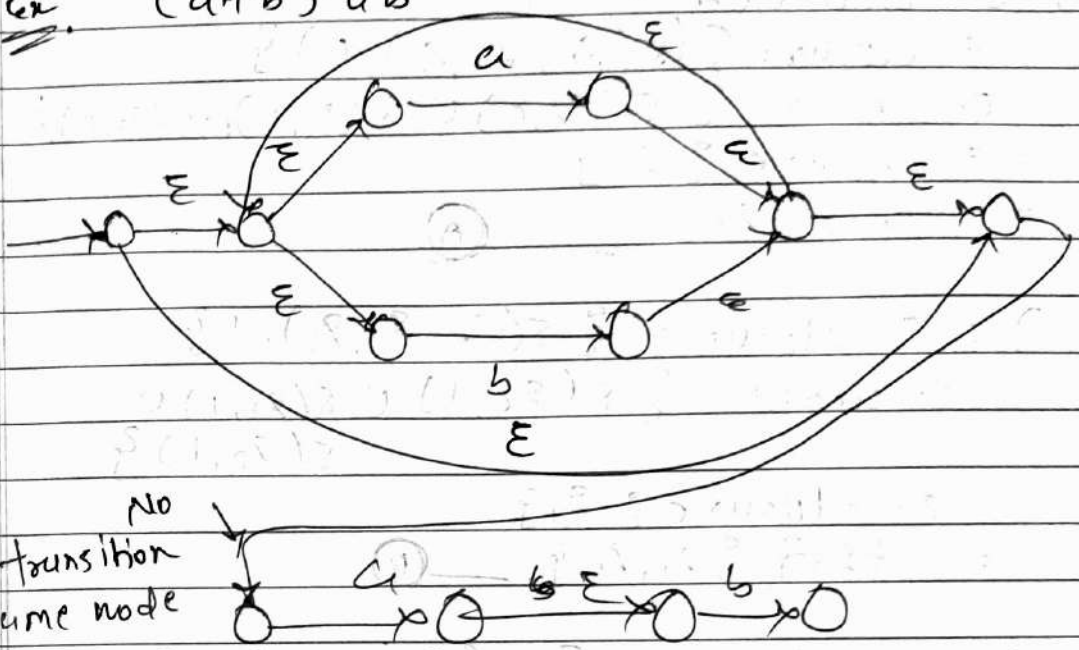


Eg: $a^* b$

Eg: $(a+b)^*$

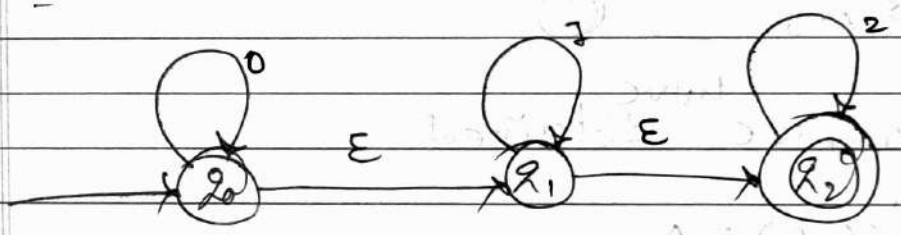
Eg: $a^* b(ab)^*$

Ex. $(a+b)^* ab$



No transition same node

(*) Convert given NFA to its equivalent DFA.



ε-closure (q_0) = { q_0, q_1, q_2 }
ε-closure (q_1) = { q_1, q_2 }
ε-closure (q_2) = { q_2 }.

② Now, we will obtain δ' transition.

Let- ε-closure (q_0) = { q_0, q_1, q_2 } — (A)

$\delta'(A,0) = \varepsilon\text{-closure}\{\delta(A,0)\}$

$= \varepsilon\text{-closure}\{(\delta(q_0,q_1,q_2),0)\}$

$= \varepsilon\text{-closure}\{\delta(q_0,0) \cup \delta(q_1,0) \cup \delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{q_0\}$

$= \{q_0,q_1,q_2\}$ ——— (A)

$\delta'(A,1) = \varepsilon\text{-closure}\{\delta(q_0,q_1,q_2),1\}$

$= \varepsilon\text{-closure}\{\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1)\}$

$= \varepsilon\text{-closure}\{q_1\}$

$= \{q_1,q_2\}$ ——— (B)

$\delta'(A,2) = \varepsilon\text{-closure}\{\delta(q_0,q_1,q_2),2\}$

$= \varepsilon\text{-closure}\{\delta(q_0,2) \cup \delta(q_1,2) \cup \delta(q_2,2)\}$

$= \varepsilon\text{-closure}\{q_2\}$

$= \{q_2\}$ ——— (C)

Thus, we have obtained

$\delta'(A,0) = A$

$\delta'(B,1) = B$

$\delta'(A,2) = C$

- Now we will find transitions on states B and C.

$\delta'(B,0) = \varepsilon\text{-closure}\{\delta(q_1,q_2),0\}$

$= \varepsilon\text{-closure}\{\delta(q_1,0) \cup \delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$

$\delta'(B,1) = \varepsilon\text{-closure}\{\delta(\{q_1,q_2\},1)\}$

$= \varepsilon\text{-closure}\{\delta(q_1,1) \cup \delta(q_2,1)\}$

$= \varepsilon\text{-closure}\{q_1\}$

$= \{q_1,q_2\}$ ——— (B)

$\delta'(B,2) = \varepsilon\text{-closure}\{\delta(q_1,q_2),2\}$

$= \varepsilon\text{-closure}\{\delta(q_1,2) \cup \delta(q_2,2)\}$

$= \varepsilon\text{-closure}\{q_2\}$

$= \{q_2\}$ ——— (C)

Hence, $\delta'(B,0) = \phi$

$\delta'(B,1) = B$

$\delta'(B,2) = C$

Now, For C state transitions are,

$\delta'(C,0) = \varepsilon\text{-closure}\{\delta(q_2,0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$

$\delta'(C,1) = \varepsilon\text{-closure}\{\delta(q_1,0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$

$\delta'(C,2) = \varepsilon\text{-closure}\{\delta(q_2,1)\}$

$= \varepsilon\text{-closure}\{q_2\}$

$= q_2$

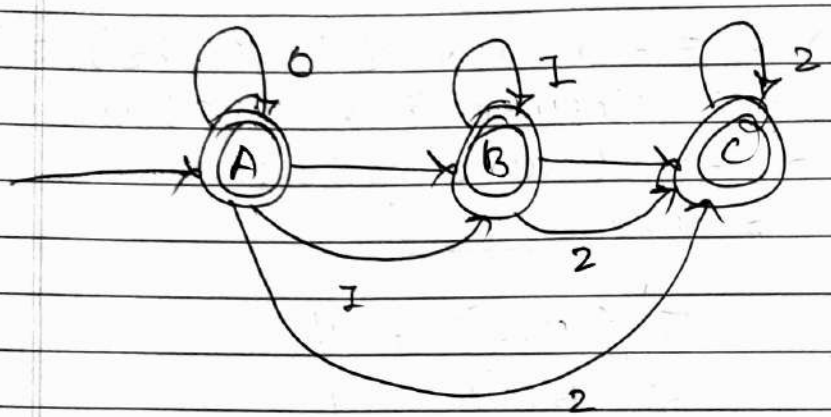
Hence,

Now, Draw the Transition table $\delta'(C,0) = \phi$

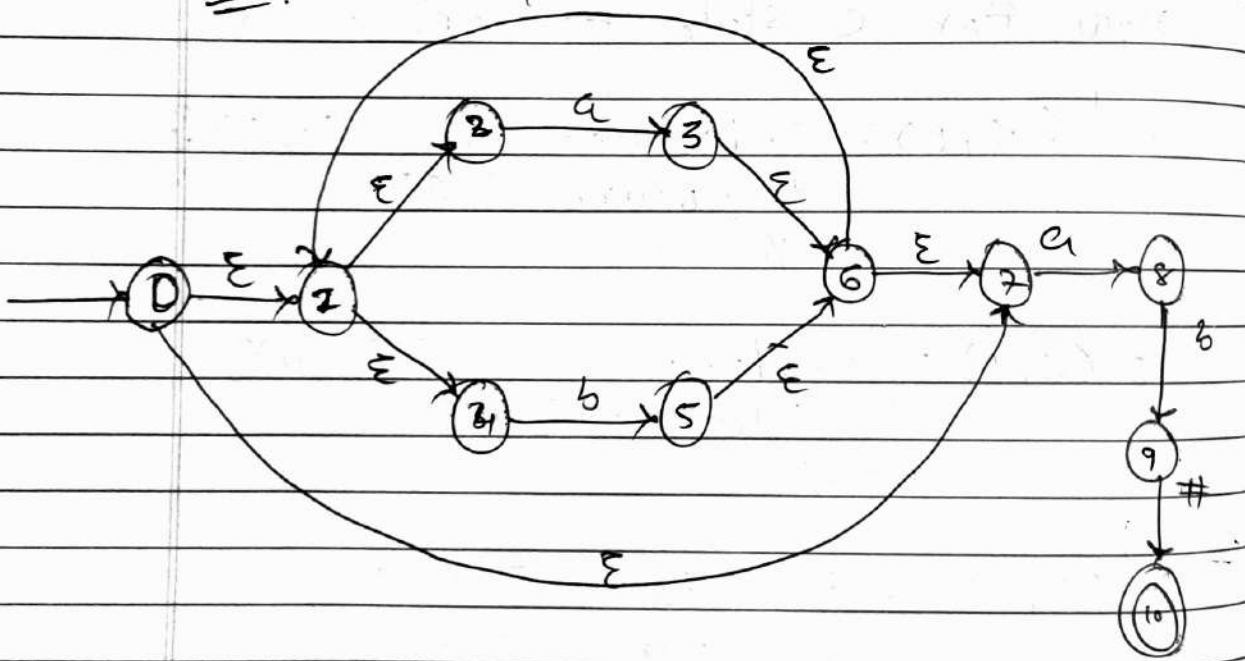
$\delta'(C,1) = \phi$

$\delta'(C,2) = C$

	0	1	2
A	A	B	C
B	ϕ	B	C
C	ϕ	ϕ	C



Ex ② $(a/b)^* ab\#$



Step: ① ↑ NFA

Step: ② ε-closure (0) = {0, 1, 2, 4, 7} — (A)

↑ ε-closure (1) = {1, 2, 4}

ε-closure (2) = {2}, ↑ ε-c(4) = {4}

ε-closure (3) = {3, 6, 1, 2, 4, 7}

ε-closure (5) = {5, 6, 1, 2, 4, 7}

ε-closure(6) = {6, 1, 2, 4, 7}
ε-c(7) = {7}, ε-c(8) = {8}

Page No.
Date

Step: 3 For A = {0, 1, 2, 4, 7}

δ'(A, a) = {3, 8} = ε-closure {3 ∪ 8}
= {1, 2, 3, 4, 6, 7} ∪ {8}
= {1, 2, 3, 4, 6, 7, 8} — (B)

δ'(A, b) = {5} = ε-closure {5}
= {1, 2, 4, 5, 6, 7} — (C)

δ'(A, #) = ∅

For B = {1, 2, 3, 4, 6, 7, 8}

δ'(B, a) = {3, 8} — (B)
δ'(B, b) = {5, 9} = ε-closure {5 ∪ 9}
= {1, 2, 4, 5, 6, 7, 9} — (D)
δ'(B, #) = ∅

For C = {1, 2, 3, 5, 6, 7}

δ'(C, a) = {3, 8} — (B)
δ'(C, b) = {5} — (C)
δ'(C, #) = ∅

For D = {1, 2, 4, 5, 6, 7, 9}

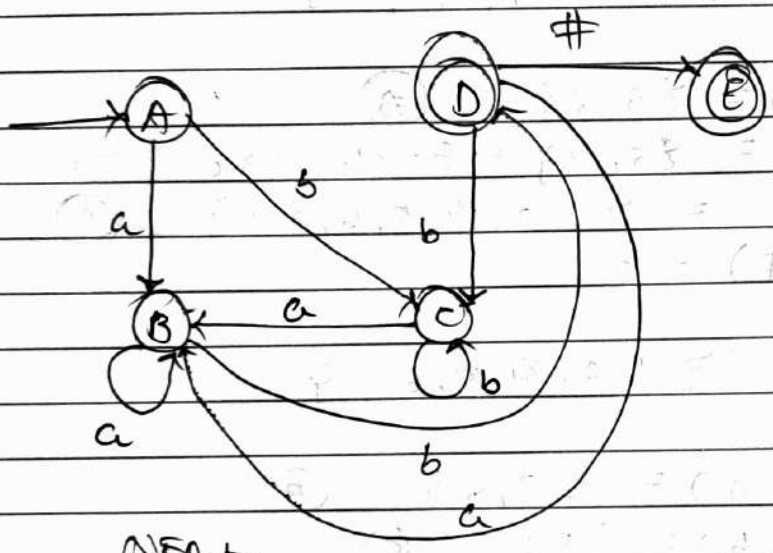
δ'(D, a) = {3, 8} — (B)
δ'(D, b) = {5} — (C)
δ'(D, #) = {10} = ε-closure {10} — (E)

For E = {10}, δ'(E, a) = ∅
δ'(E, b) = ∅
δ'(E, c) = ∅ — Final state

Step: ④ : Transition Table

	a	b	#
A	B	C	—
B	B	D	—
C	B	C	—
D	B	C	E
E	—	—	—

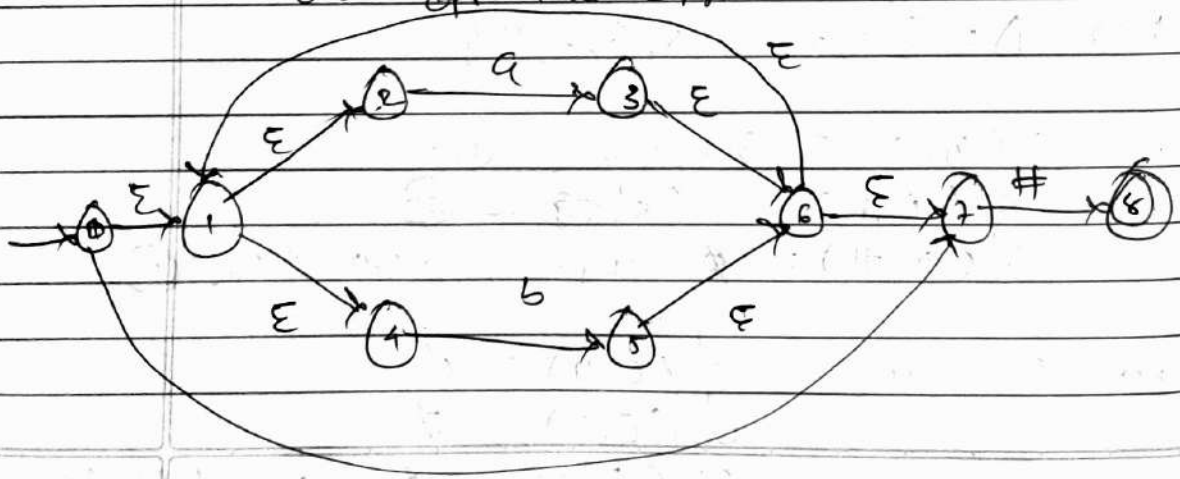
Step: ⑤ : DFA.



NFA to DFA

Eg. ⑦ DFA for (a|b)* #

=> ⓪ or optimize DFA



$\varepsilon(0) = \{0, 1, 2, 4, 7\}$ — (A)

$\delta'(A, a) = \{3\} = \{1, 2, 3, 4, 6\}$ — (B)
$\delta'(A, b) = \{5\} = \{1, 2, 4, 5, 6\}$ — (C)
$\delta'(A, \#) = \{8\} = \{8\}$ — (D)

$\delta'(B, a) = \{3\}$ — (B)
$\delta'(B, b) = \{5\}$ — (C)
$\delta'(B, \#) = \emptyset$

$\delta'(C, a) = \{3\}$ — (B)
$\delta'(C, b) = \{5\}$ — (C)
$\delta'(C, \#) = \emptyset$

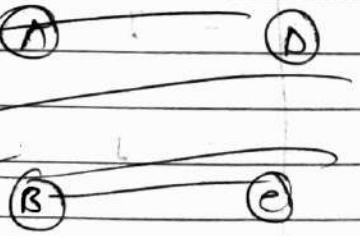
$\delta'(D, a) = \emptyset$
$\delta'(D, b) = \emptyset$
$\delta'(D, \#) = \emptyset$ — Final State.

Step: (3) Transition Table :

	a	b	#
A	B	C	D
B	B	C	–
C	B	C	–
D	–	–	–

Step: (4)

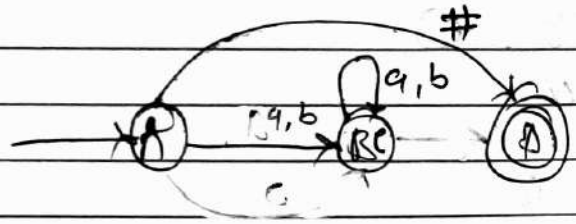
DFA ←→



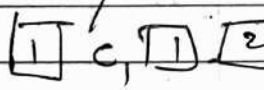
step: (4) optimize the states

Here BC states are same
Now to optimize DFA we will merge that states

	a	b	#
A	B	C	D
BC	B	C	-
D	-	-	-

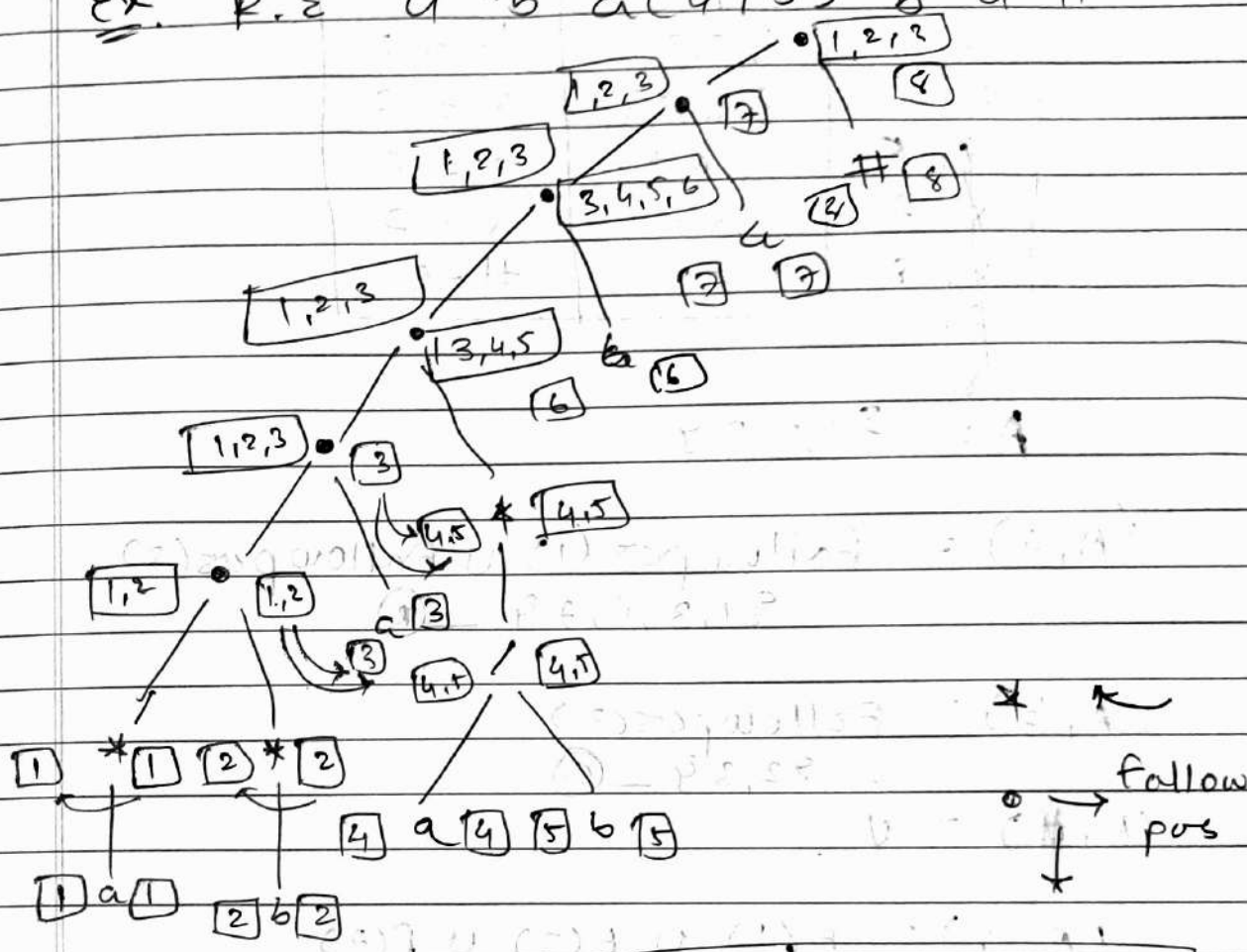


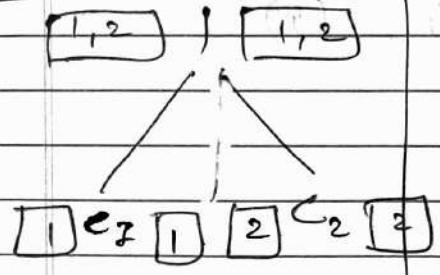
(*) DFA - Rules:

	[1] a [1]	[1] * [1]	[1] • [2] 
			[1] c_1 [1] [2] c_2 [2]
Nullable	- No	- yes	if c_1 (nullable) - f.p(c_1) ∪ F.p(c_2)
f.p	- I	- I	if e_1 (not nullable) - f.p(e_1) ↓ same for l.p
l.p	- I	- I	if c_2 (nullable) - l.p(c_1) ∪ l.p(c_2)

fi c_2 ## nullable l.p(c_2) ←

Ex. R.E $a^*b^*a(a|b)^*b^*a\#$





Node	Followpos
1	{1,3}
2	{2,3}
3	{4,5,6,7}
4	{4,5,6,7}
5	{4,5,6,7}
6	{6,7}
7	{8}
8	—

Reduce with one node.

L.p(c1) ∪ L.p(c2)

modified node

1	{1,3}
2	{2,3}
3*	{6,7}
6	{6,7}
7	{8}
8	—

1.	{1, 3}	a = 1
2.	{2, 3}	b = 2
3.	{6, 7}	a = 3
4. 5	{6, 7}	b = 6
5. 7	{8}	a = 7
6. 8	-	# = 8

A = {1, 2, 3}

(A, a) = Followpos(1) ∪ Followpos(3)
= {1, 3, 6, 7} — (B)

(A, b) = Followpos(2)
= {2, 3} — (C)

(A, #) = ∅

(B, a) = F(1) ∪ F(3) ∪ F(7)
= {1, 3, 6, 7, 8} — (D)

(B, b) = F(6)
= {6, 7} — (E)

(B, #) = ∅

(C, a) = F(3) — (E)

(C, b) = F(2) — (C)

(C, #) = ∅

(D, a) = F(1) ∪ F(3) ∪ F(7) — (D)

(D, b) = F(6) — (B)

(D, #) = ∅

(E, a) = F(7) — (E)

(E, b) = F(6) — (B)

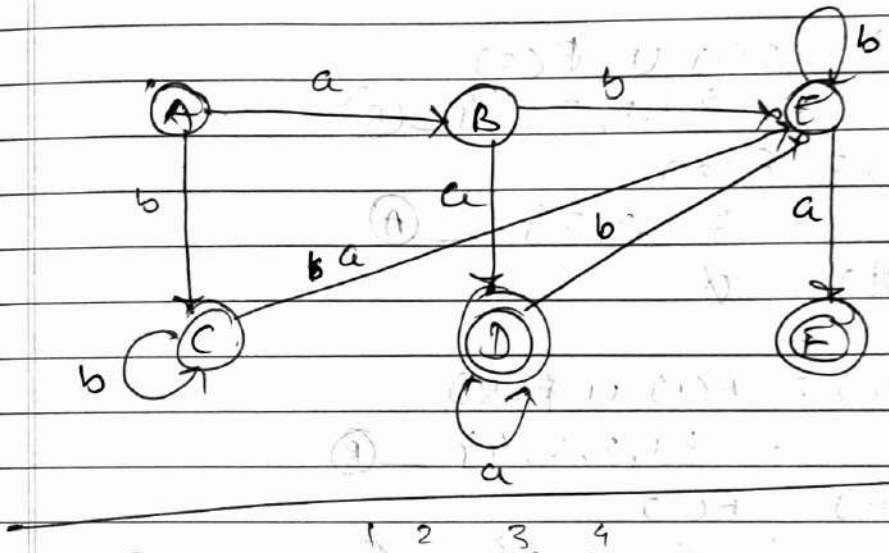
(E, #) = ∅

(F, a) = ∅

(F, b) = ∅

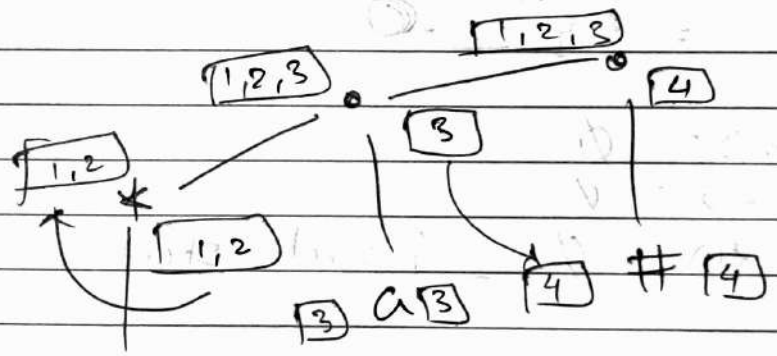
(F, #) = ∅

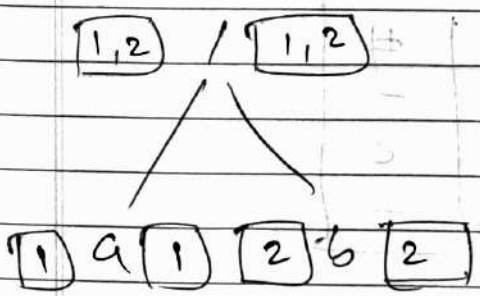
States	a	b
A	B	C
B	D	E
C	E	C
D	D	E
E	F	E
F	–	–



1 2 3 4

Ex. (2) $(a|b)^* a\ \#$





A = {1, 2, 3}

Node	Followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{ }

a = 1
b = 2
a = 3
# = 4

For A

(A, a) = F(1) ∪ F(3)
= {1, 2, 3, 4} — (B)

(A, b) = F(2)
= {1, 2, 3} — (A)

(A, #) = ∅

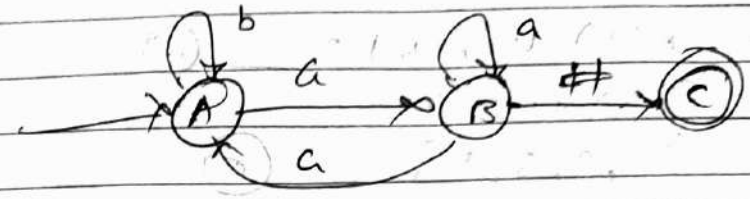
(B, a) = F(1) ∪ F(3)
= {1, 2, 3, 4} — (B)

(B, b) = F(2)
= {1, 2, 3} — (A)

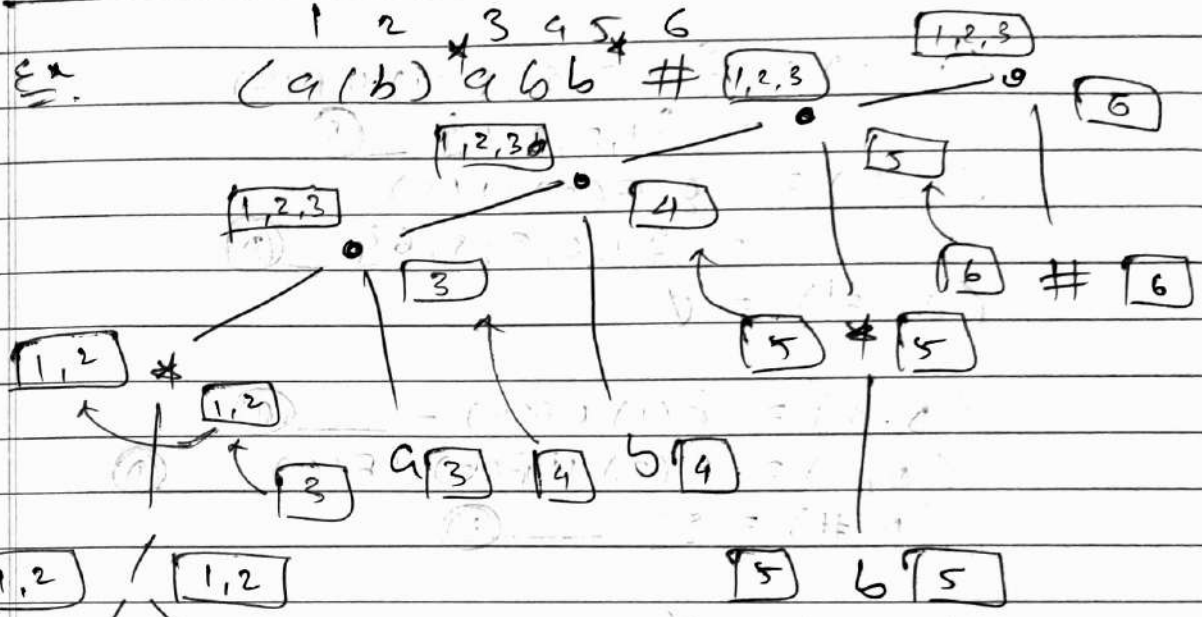
(B, #) = F(4)
= { } — (C)

(C, a) = ∅
(C, b) = ∅
(C, #) = ∅ — final state

	a	b	#
A	B	A	-
B	B	A	C
C	-	-	-



Ex. $(a(b)^*ab^*b\#$ — positions: 1 2 3 4 5 6



Node	Followpos
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6, 5}
6	–

a = 1
b = 2
a = 3
b = 4
b = 5
b = 6
#

A = {1, 2, 3}

(A, a) = F(1) ∪ F(3)
= {1, 2, 3, 4} — (B)

(A, b) = F(2)
= {1, 2, 3} — (A)

(A, #) = ∅

(B, a) = F(1) ∪ F(3) = ___ (B)

(B, b) = F(2) ∪ F(4)
= {1, 2, 3, 5} ___ (C)

(B, #) = ∅

(C, a) = ~~{1, 2, 3,~~ F(1) ∪ F(3)
= {1, 2, 3, 4} ___ (B)

(C, b) = F(2) ∪ (5)
= {1, 2, 3, 5, 6} ___ (D)

(C, #) = ∅

(D, a) = F(1) ∪ F(3) ___ (B)

(D, b) = F(2) ∪ F(5) ∪ F(6) ___ (D)

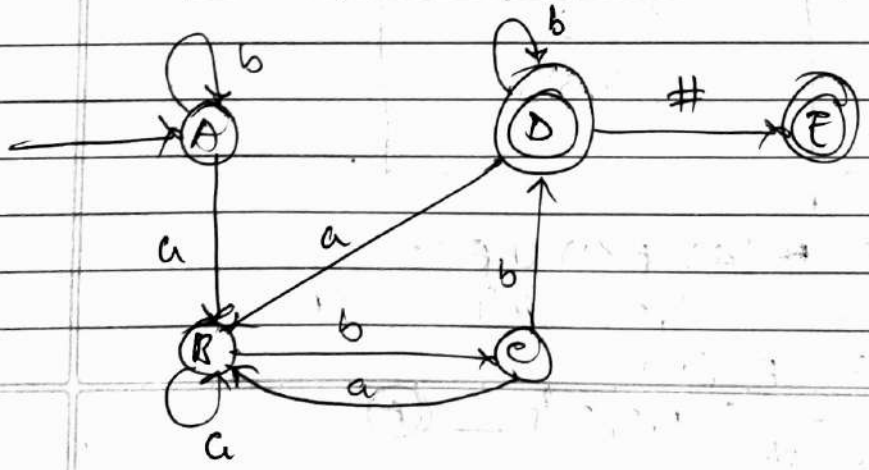
(D, #) = {} ___ (E)

(E, a) = ∅

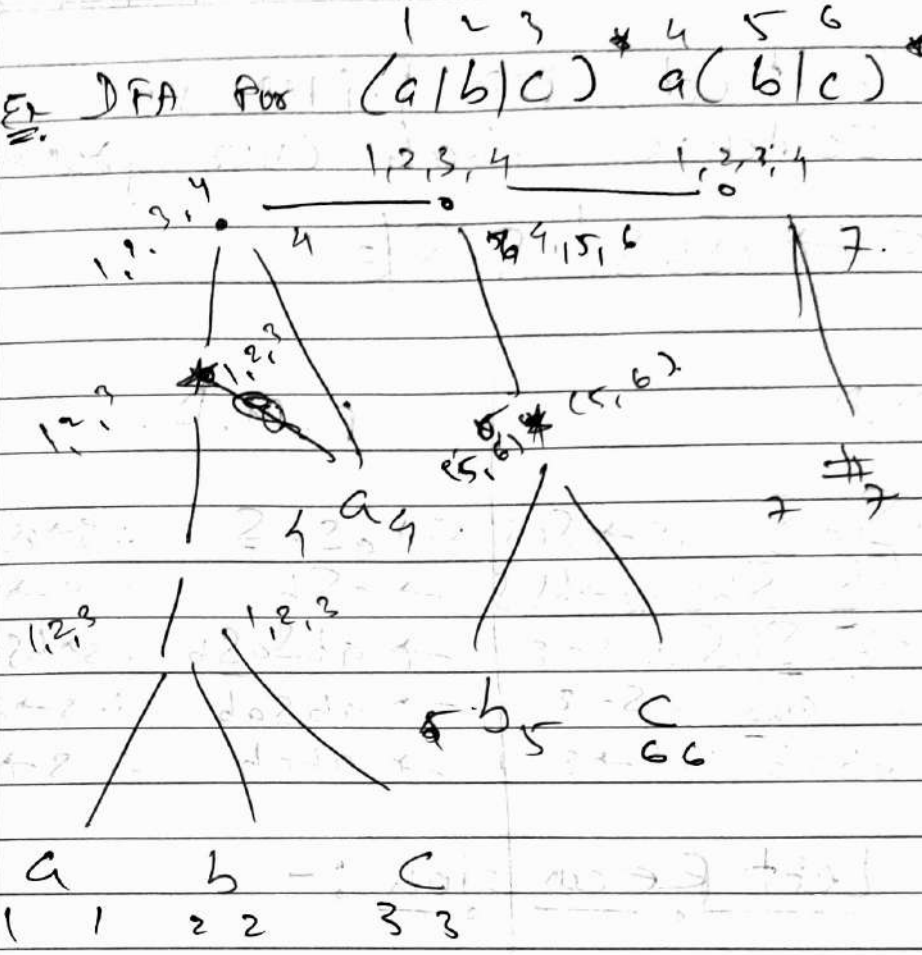
(E, b) = ∅

(E, #) = ∅ ___ final state

	a	b	#
A	B	A	-
B	B	C	-
C	B	D	-
D	B	D	E
E	-	-	-



Ex. DFA for $(a|b|c)^* a(b|c)^* \#$



Node	Followpos
1	{1, 2, 3, 4}
2	{1, 2, 3, 4}
3	{1, 2, 3, 4}
4	{5, 6, 7}
5	{5, 6, 7}
6	{5, 6, 7}
7	{}
0	

⑥ Construct LMD & RMD for. Sentence abab over grammar.

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

L.M.D	R.M.D.
$S \rightarrow aSbS$ ∴ $S \rightarrow aSbS$	$S \rightarrow aSbS$ ∴ $S \rightarrow aSbS$
$\rightarrow abSaSbS$ ∴ $S \rightarrow bSaS$	$\rightarrow aSb$ ∴ $S \rightarrow \varepsilon$
$\rightarrow ab\varepsilon aSbS$ ∴ $S \rightarrow \varepsilon$	$\rightarrow abSaSb$ ∴ $S \rightarrow bSaS$
$\rightarrow abaSbS$ ∴ $S \rightarrow \varepsilon$	$\rightarrow abSab$ ∴ $S \rightarrow \varepsilon$
$\rightarrow abab$ ∴ $S \rightarrow \varepsilon$	$\rightarrow abab$ ∴ $S \rightarrow \varepsilon$

⑦ Left Recursion :-

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

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$

⑧ Remove Left Recursion :-

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Eg. ① $A \rightarrow A+B \mid B$

(α = $+B$, β = B)

$A \rightarrow BA'$

$A' \rightarrow +BA' \mid \varepsilon$

Ex. ② $E \to E+T \mid T$

(α = $+T$, β = T)

$E \to TE'$

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

Ex. ③ $A \to AB \mid Ac \mid aB \mid cd$

(α_1 = B, α_2 = c, β_1 = aB, β_2 = cd)

$A \to aBA' \mid cdA'$

$A' \to BA' \mid cA' \mid \varepsilon$

Ex. ④ $A \to ABd \mid Aa \mid a$

(α_1 = Bd, α_2 = a, β = a)

$A \to aA'$

$A' \to BdA' \mid aA' \mid \varepsilon$

---

⑤ Left Factoring :-

Ex. ① $A \to ABd \mid Aa \mid a$ (α_1 = Bd, α_2 = a, β = a)

$B \to Be \mid b$

$A \to aA'$

$A' \to BdA' \mid aA' \mid \varepsilon$

$B \to bB'$

$B' \to eB' \mid \varepsilon$

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

$A \to \alpha A' \mid \varepsilon$

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

$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \beta_1 \mid \beta_2$ ~~$A \to A\alpha_1 \mid A$~~

Ex. (6)

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid \varepsilon$

~~$S \to Aa \mid b$~~

~~$A \to$~~

if we replace S with Ac|b then

$A \to Ac \mid Aad \mid bd \mid \varepsilon$

$\alpha_1 \quad \alpha_2 \quad \beta_1 \quad \beta_2$

$A \to bdA' \mid \varepsilon A'$

$A' \to cA' \mid adA' \mid \varepsilon$

Ans: $S \to Aa \mid b$

$A \to bdA' \mid A'$

$A' \to cA' \mid adA' \mid \varepsilon$

Ex. (7)

~~$S \to Aa \mid b$~~

~~$A \to A$~~

$S \to aBDh$

$B \to Bb \mid c$

$D \to EF$

$E \to g \mid \varepsilon$

$F \to f \mid \varepsilon$

$S \to aBDh$

$B \to cB'$

$B' \to bB' \mid \varepsilon$

$D \to EF$

$E \to g \mid \varepsilon$

$F \to f \mid \varepsilon$

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

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

④ Left Factoring :-

if we have

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \gamma_1 \mid \gamma_2 \ldots$

Then we replace it with

$A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \ldots$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \ldots$

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \gamma_1, \gamma_2$

$A \to \alpha A' \mid \gamma_1 \ldots$

$A' \to \beta_1 \mid \beta_2 \mid \ldots$

Ex. ① $S \to C+E \mid C*E \mid C \mid E$

(α = C, β_1 = +E, β_2 = *E, β_3 = E)

$S \to CS'$

$S' \to +E \mid *E \mid \varepsilon$

Ex. ② $S \to iEtS \mid iEtSeS \mid a$

(α = iEtS, β_1 = eS, γ = a)

$S \to iEtSS' \mid a$

$S' \to \varepsilon \mid eS$

Ex ③ $S \to aAd \mid aB$

(α = a, β_1 = Ad, β_2 = B)

$A \to a \mid ab$

$B \to ccd \mid ddc$

$S \to aS'$

$S' \to Ad \mid B$

$A \to aA'$

$A' \to \varepsilon \mid b$

$B \to ccd \mid ddc$

Page No.
Date

Ex. (4)

$S \to \underset{\alpha}{aab} \mid \underset{B_1}{aa}\underset{B_2}{A} \mid \underset{\gamma}{c}$

$A \to \varepsilon \mid C$

$S \to aaS' \mid c$

$S' \to b \mid A$

$A \to C$

Ex. (5)

$A \to \underset{\alpha}{aA}\underset{B_1}{B} \mid \underset{\gamma}{aA} \mid a$

$B \to bB \mid b$

$A \to aAA' \mid a$

$A' \to B \mid \varepsilon$

Ex. (6)

$A \to \underset{\alpha}{a}\underset{B_1}{AB} \mid \underset{B_2}{aA} \mid \underset{B_3}{a}$

$B \to \underset{\alpha}{b}\underset{B_1}{B} \mid \underset{B_2}{b}$

$A \to aA'$

$A' \to AB \mid A \mid \varepsilon$

$B \to bB'$

$B' \to B \mid \varepsilon$

Ex. (7)

$A \to \underset{\alpha}{ad} \mid \underset{B_1}{a} \mid \underset{B_2}{ab} \mid \underset{B_3}{ab}\underset{B_4}{c} \mid \underset{\gamma}{b}$

$A \to bA' \mid b$

$A' \to d \mid \varepsilon \mid b \mid bc$

⑧ First & Follow Table :-

(1) $S \to aBDh$
$B \to cC$
$C \to bC \mid \varepsilon$
$D \to EF$
$E \to g \mid \varepsilon$
$F \to f \mid \varepsilon$

Follow: Top to down & NO ε.

If ε → add to the set. Bottom to up (First)

First

Symbols	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}

(2) $S \to A(B \mid cbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \varepsilon$
$C \to h \mid \varepsilon$

Symbols	First	Follow
S	{d, g, h, ε, b, a}	{$}
A	{d, g, h, ε}	{g, h, $}
B	{g, ε}	{h, $, a, g}
C	{h, ε}	{$, b, g, h}

(3) $S \rightarrow ABC$
$A \rightarrow a | b | \varepsilon$
$B \rightarrow c | d | \varepsilon$
$C \rightarrow e | f | \varepsilon$

Symbols	First	Follow
S	{a, b, c, d, e, f, ε}	{$}
A	{a, b, ε}	{c, d, e, f, $}
B	{c, d, ε}	{e, f, $}
C	{e, f, ε}	{$}

(4) $E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow (E) | id$

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

(5) $S \rightarrow kMNOP$
$k \rightarrow kB | \varepsilon$
$M \rightarrow m | \varepsilon$
$N \rightarrow n$
$O \rightarrow o | \varepsilon$
$P \rightarrow p | \varepsilon$

Symbols	First	Follow
S	{k, m, n}	{$}
K	{k, ε}	{m, n}
M	{m, ε}	{n}
N	{n}	{o, p, $}
O	{o, ε}	{p, $}
P	{p, ε}	{$}

(6) ~~E → TE'~~
~~E' → +TE'~~

S → aAB | bA | ε
A → aAb | ε
B → bB | ε

Symbols	First	Follow
S	{a, b, ε}	{$}
A	{a, ε}	{b, $}
B	{b, ε}	{$}

(7) S → iCtSA | a
A → eS | ε
C → b

Symbols	First	Follow
S	{i, a}	{e, $}
A	{e, ε}	{e $}
C	{b}	{t}

(8) S → Aa | bAc | Bc | bBa
A → d
B → d

Symbols	First	Follow
S	{d, b}	{$}
A	{d}	{a, c}
B	{d}	{a, c}

(9). $S \to AB$
$A \to Ca \mid \varepsilon$
$B \to BaAc \mid c$
$C \to b \mid \varepsilon$

Here, $B \to BaAc \mid c$ → left recursion

Symbols	First	Follow
S	{a, c, ε}	{$}
A	{b, ε}	{a, c}
B	{a, c}	{$, a}
C	{b, ε}	{$, a}

after removing left recursion.

$S \to AB$
$A \to Ca \mid \varepsilon$
$B \to cB'$
$B' \to aACB'$
$C \to b \mid \varepsilon$

જો B ને કોઈ કોલમો નહી થાવું જોઈ તેથી જ S નો Follow લેવો પડે B નો કોલમો છે.

Symbols	First	Follow
S	{b, c, a}	{$}
A	{b, ε, a}	{b, ε, a}
B	{c}	{$}
B'	{a, ε}	{$}
C	{b, ε}	{a, $}

(10) $S \to A$

$A \to aB \mid Ad$

$B \to bBC \mid f$

$C \to g$

removing left recursion

$A \to Ad$ f

$S \to A$

$A \to aBA' \mid Ad$

$A' \to dA' \mid \varepsilon$

$B \to bBC \mid f$

$C \to g$

Symbols	First	Follow
S	{a}	{$}
A	{a}	{$}
A'	{d, ε}	{$}
B	{b, f}	{d, g}
C	{g}	{d, g}

Predictive Parsing (LL(1)) Table :-

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

(11) $S \to 1AB \mid \varepsilon$

$A \to 1AC \mid 0C$

$B \to 0S$

$C \to 1$

Symbols	First	Follow
S	{1, ε}	{$}
A	{1, 0}	{0, 1}
B	{0}	{$}
C	{1}	{0, 1}

⊛ predictive Parsing table :-

	0	1	$
S		$S \to 1AB$	$S \to \varepsilon$
A	$A \to 0C$	$A \to 1AC$	
B	$B \to 0S$		
C		$C \to 1$	

- This grammar is LL(1) Because every State in LL(1) Table has one entry.

(12) $S' \to S$

$S \to aA \mid b \mid cB \mid d$

$A \to aA \mid b$

$B \to cB \mid d$

Symbol	First	Follow
S'	{a, b, c, d}	{$}
S	{a, b, c, d}	{$}
A	{a, b}	{a, b, c, d, $}
B	{c, d}	{a, b, c, d, $}

⊛ predictive Parsing Table :-

	a	b	c	d	$
S'	S'→S	S'→S	S'→S	S'→S	
S	S→aA	S→b	S→cB	S→d	
A	A→aA	A→b			
B			B→cB	B→d	

⊛ Shift Reducing Parser :-

(1)
E → E-E | E*E | id [input string "id1-id2*id3"]

stack	input buffer	Parsing Action
$	id1-id2*id3 $	shift
$id1	-id2*id3 $	Reduce by E→id
$E	-id2*id3 $	shift
$E-	id2*id3 $	shift
$E-id2	*id3 $	Reduce by E→id
$E-E	*id3 $	shift
$E-E*	id3 $	shift
$E-E*id3	$	Reduce by E→id
$E-E*E	$	Reduce by E→E*E
$E-E	$	Reduce by E→E-E
$E	$	Accept

(2) S → TL;
T → int | float
L → L, id | id

input string
↓
int id, id;

Stack	Input buffer	Parsing Action
$ int	int id,id;$	shift
$ int	id,id;$	R.b T→int
$ T	id,id;$	shift
$ Tid	id;$	R.b L→id
$ TL	id;$	shift
$ TL,	id;$	shift
$ TL,id	;$	R.b L→L,id
$ TL	;$	shift
$ TL;	$	R.b S→TL;
$ S	$	Accept

⊛ Operator Precedence Parsing :-

$$A \to \varepsilon \ (\otimes)$$

Two Rules
(1) No production on the right side is ε. (⊗)

(2) two adjacent non-terminals at right hand side are not considered.

$$A \to BC \ (\otimes)$$

Ex. $E \to EAE \mid (E) \mid -E \mid id$
$A \to + \mid - \mid * \mid / \mid \wedge$

Now, here EAE is not considered that's why we will convert into an equivalent operator precedence grammar.

id * + − $

+ > >

E → E+E | E−E | E*E | E/E | E^E

E → (E) | −E | id

(1) Precedence Relation Table :-

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

Rule for Stack

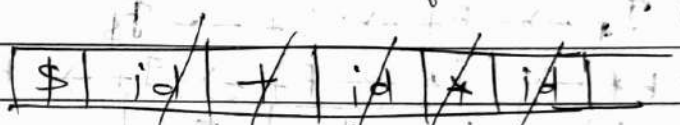
IF (S <· L) → {push}

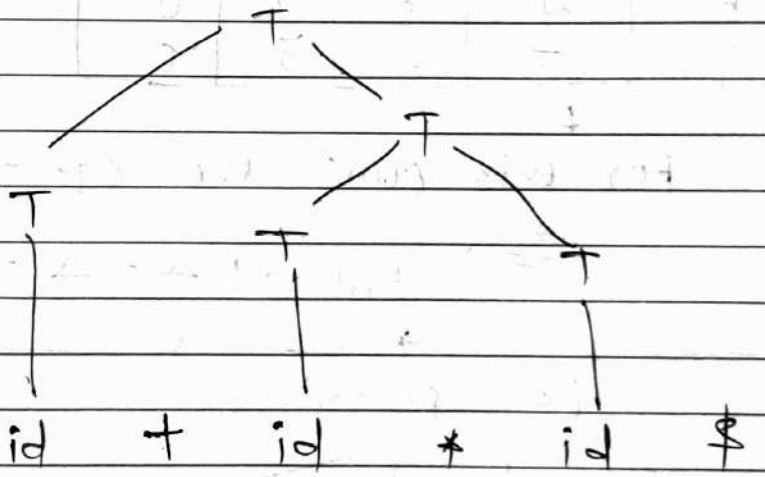
IF (S ·> L) → {pop}

fid f+ f* f$

S → Top of the Stack Operator

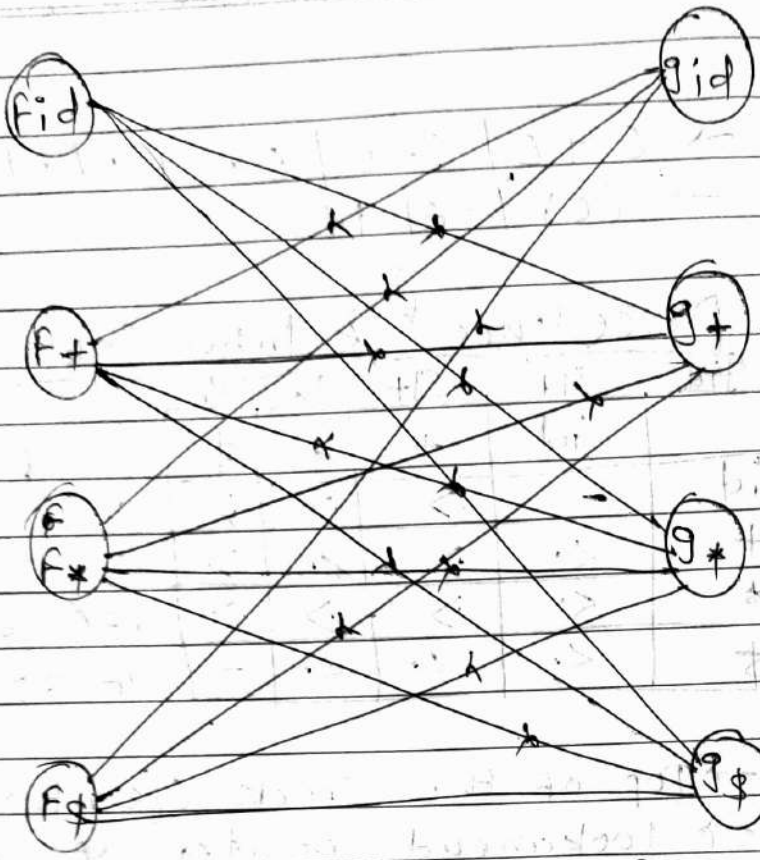
L → lookahead pointer of stack.





Now here operator precedence size is the issue that is n×n = n².

Now, to solve this problem we will make operator precedence function table



fid → g* → F+ → g+ → F$

gid → F* → g+ → F+ → g+ → F$

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

↓

For this count the ~~eg~~ → edges

eg.

Fid → 1 → 2 → 3 → 4

LR parsing

LR parser | SLR parser | CLR parser | LALR parser

eg. A → .TT

LR(0) items

eg. A → .TT, $

LR(1) items

look ahead

② LR parser :-

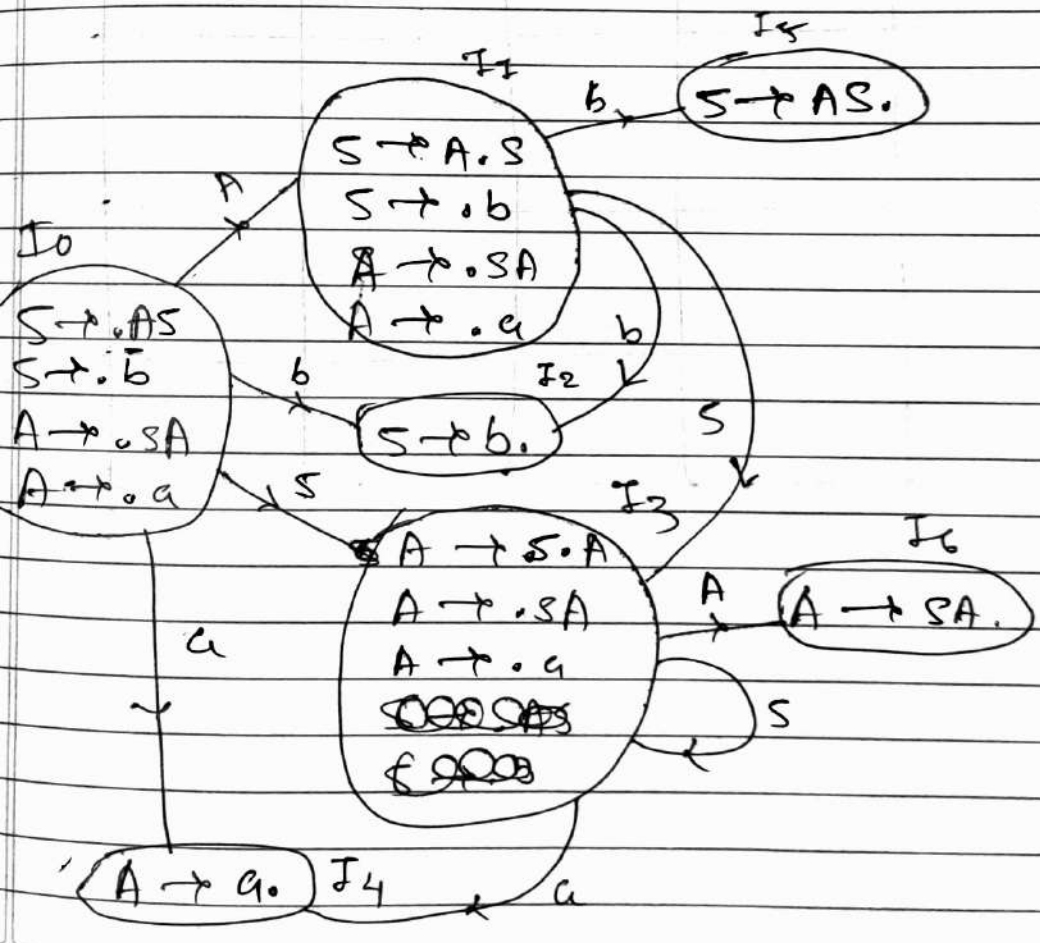
Eg. $S \to AS \mid b$ Find closure (I) & goto (I).
$A \to SA \mid a$

→ First we convert into augmented grammar. using • dot operator.

$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$
} closure (I) ——— I_0

$S' \to S$
$S \to AS \mid b$
$A \to SA \mid a$

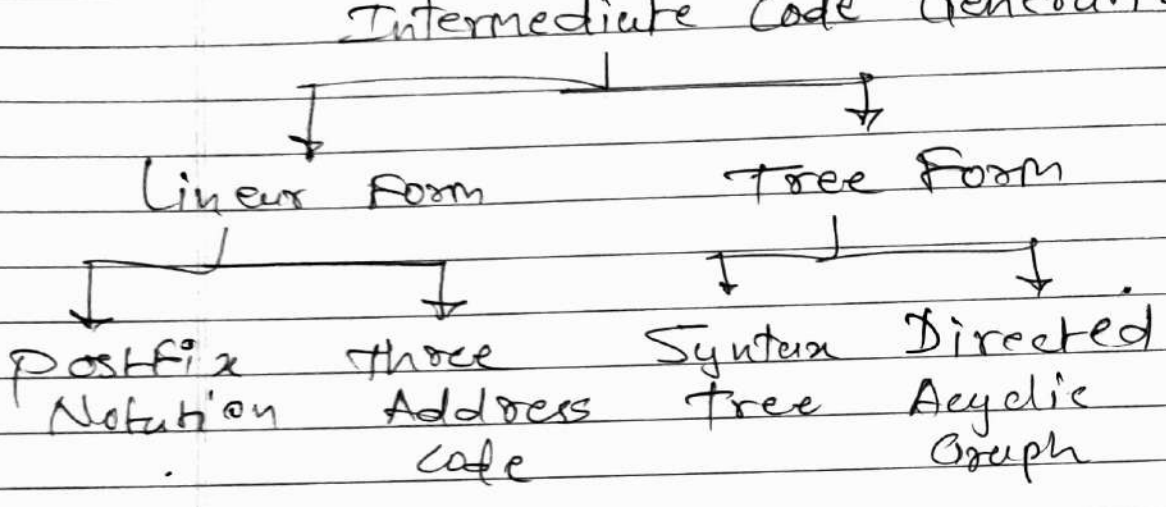
Now, we apply goto on each symbol in I_0.



	Action			GOTO	
	a	b	$	S	A
0	S4	S2		3	1
1		S		3	
2					
3					
4					
5					
6					

④ Intermediate Code generation :-

(1) Postfix Notation
(2) Three Address Code
(3) Syntax Tree / DAG



Eg. $(a+b) * (a+b+c)$

postfix → $(ab+) * (a+b+c)$

$(ab+) * (ab++c)$

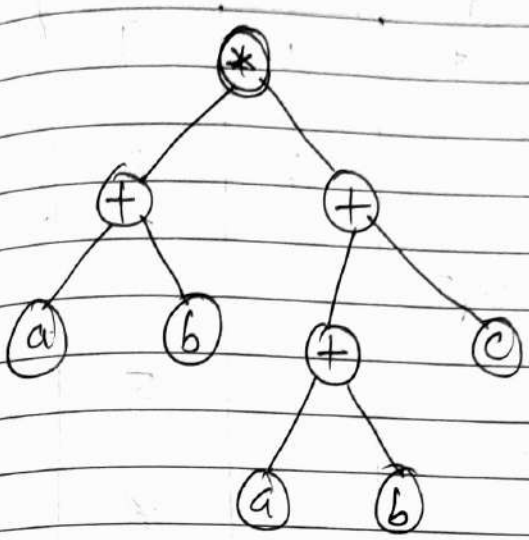
$(ab+) * (ab+c+)$

$(ab+ ab+c+) *$

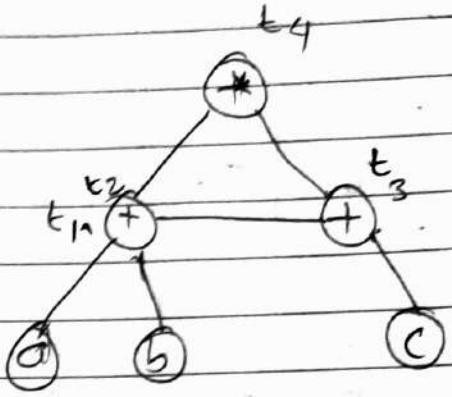
Three Address code →

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

Syntax Tree

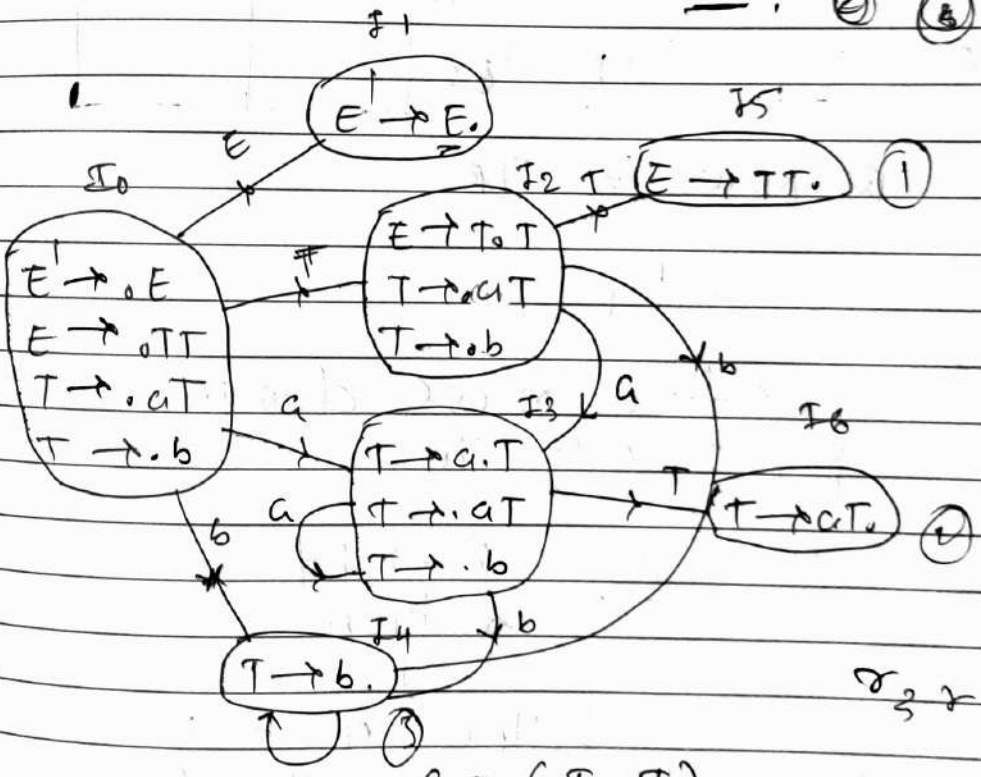


DAG



② LR(0) Parser ex :-

$E' \to E$ ⓪
$E \to TT$ ①
$T \to aT / b$ ② ③



GOTO (I_2, T)

Ex
⇒.

	Action			Goto	
	a	b	$	E	T
0	s_3	s_4		1	2
1			accept		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

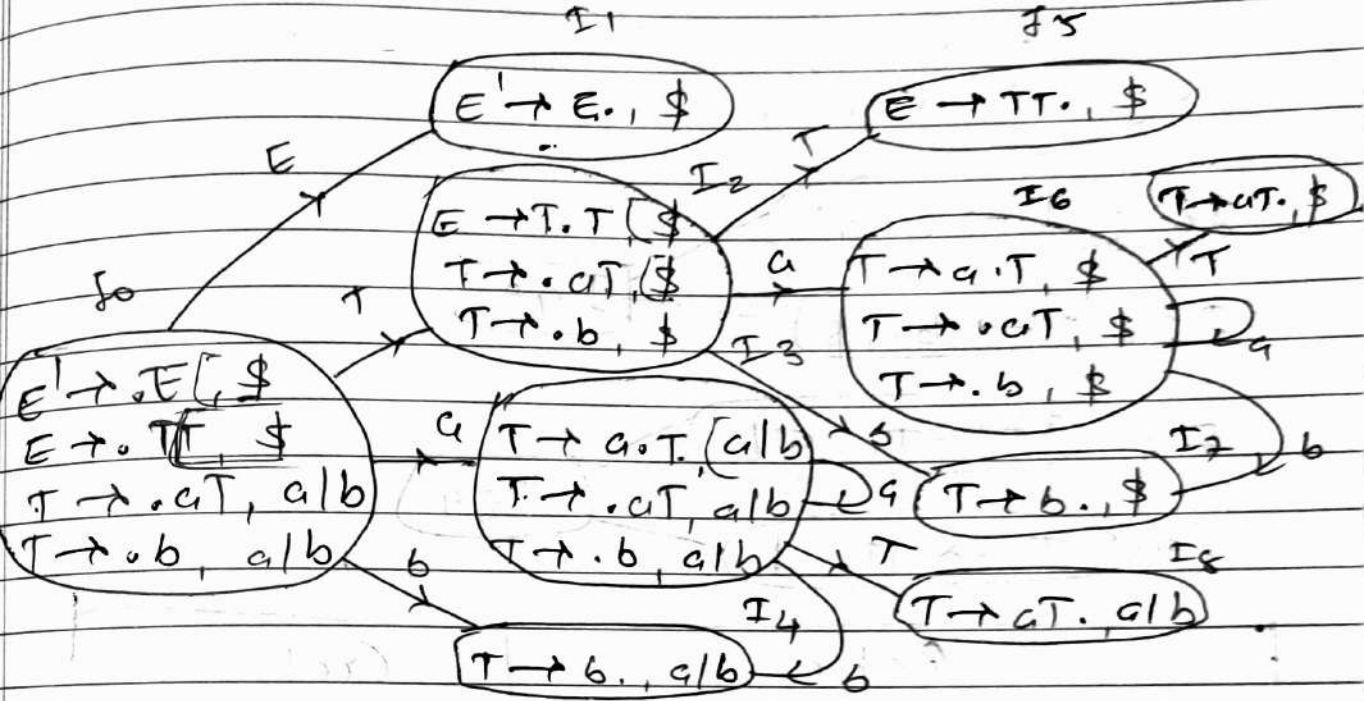
② Now in SLR only Reduce move change & put only in the Eg. $F \rightarrow id$ & then

Follow of F is the field in which you have to fill r move

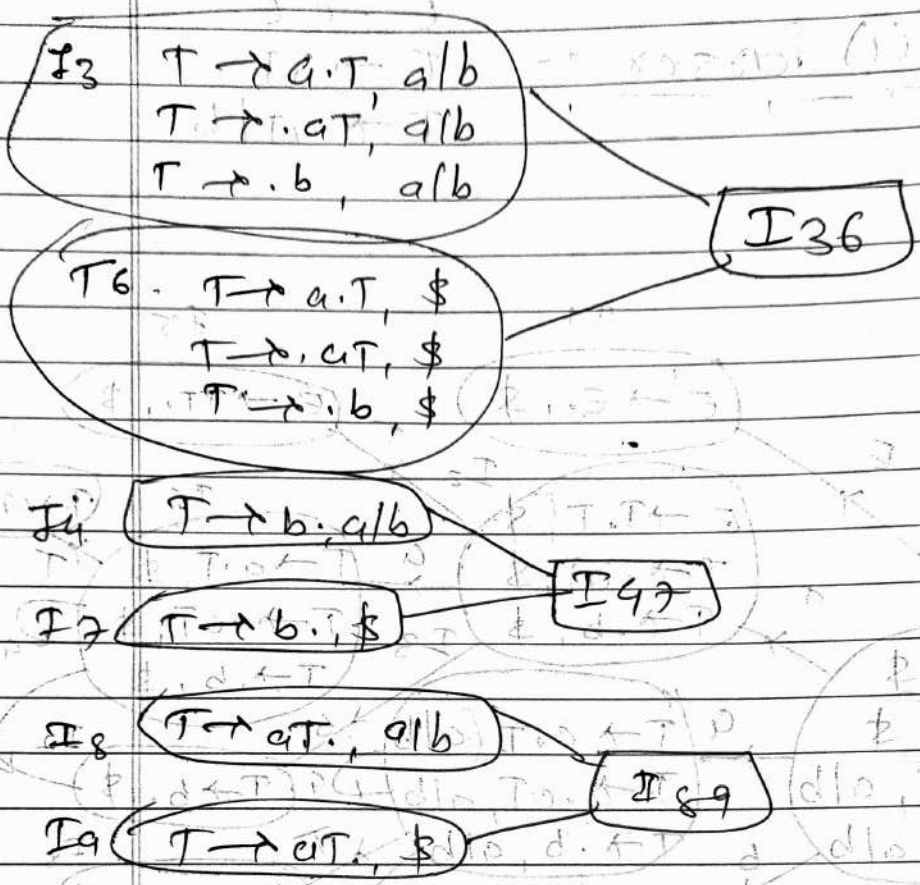
LR(1) items = LR(0) item + lookahead

(8) CLR(1) Parser :-

$E' \to E$
$E \to TT$
$T \to aT \mid b$



	Action			GoTo	
	a	b	$	E	T
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	γ_3	γ_3			
5			γ_1		
6	S_6	S_7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		



	Action			GoTo	
	a	b	$	E	T
0	S_{36}	S_{47}		1	2
1			Accept		5
2	S_{36}	S_{47}			89
36	S_{36}	S_{47}			
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Page No.
Date

Q.① Write a brief note on input buffering.

- There are two techniques for input buffering:

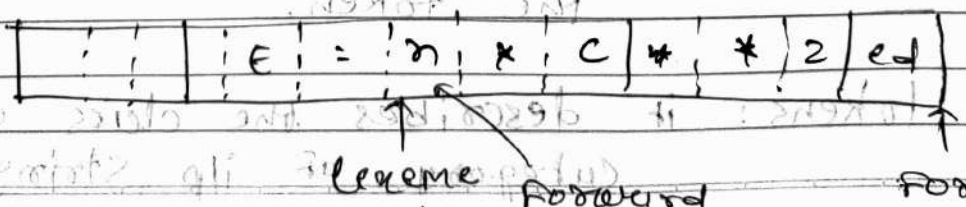
(1) Buffer pairs
(2) Sentinels

[1] Buffer Pairs:

- The lexical analysis phase - scan input from left to right.
- Scan one character at a time.

- One - input buffers:
- There are many situation where we need to look atleast one character ahead.

- Two input Buffer
- Buffer divided into two character.
- N is number of character on one block.



- lexeme begin → marks the beginning of
- forward - scans ahead pattern match found.

[2] Sentinels:

- if we use the scheme of buffer pairs we must check, each time pointer that we have not moved off one of the buffers;

- if we do, then we must reload the other buffer, thus, for each character read, we make two tests.

- we can combine the buffer-end test with the test for the character if we extend buffer to hold a sentinel character at the end.

Q.(2) Explain lexeme patterns & tokens.

- Lexemes: sequence of character in the sentence program that are matched with the pattern of the token.

- patterns: set of rules that describes the token.

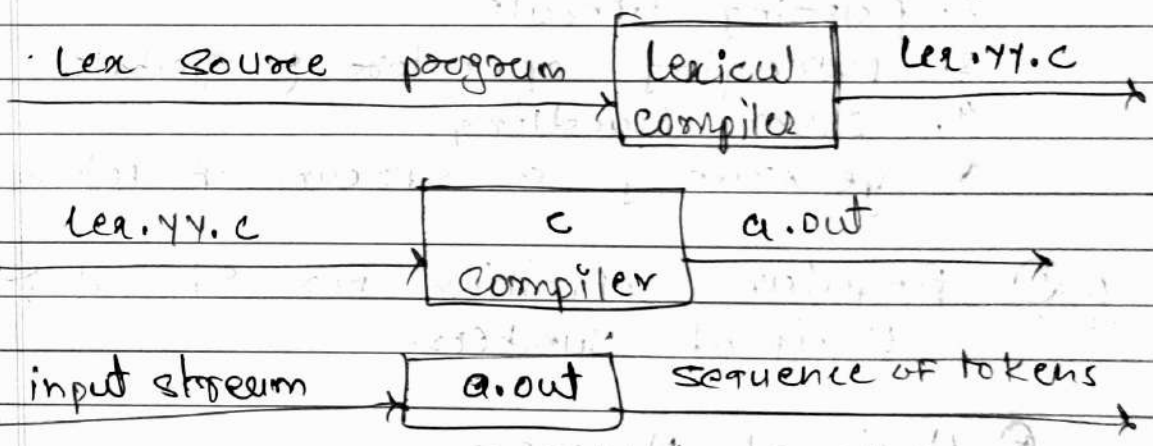
- tokens: it describes the class of the category of i/p string.

Eg. $a = b + 20;$

lexems : a, =, b, +, 20, ;
token : id1, op1, id2, op2, num2, punctu.
pattern letter, =, +, -, letter, +, -, digit.

Q.(3) write a short note on lex tool.

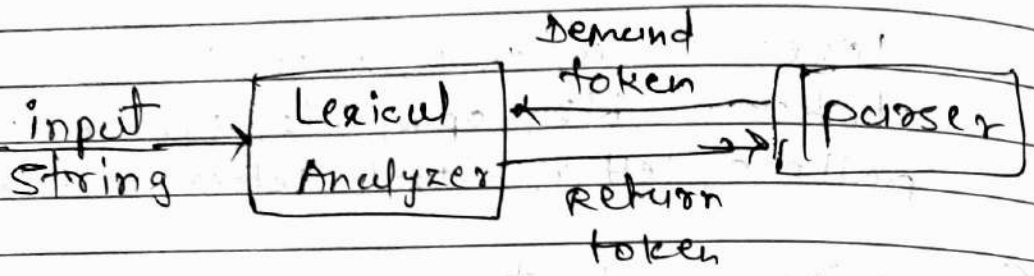
- Lex is a tool or a computer program that generates lexical Analyzer.
- The lex tool itself is a compiler.
- The lex compiler takes the input & transforms that input patterns.
- it is commonly used with YACC (yet Another compiler). it was written by mike lesk & eric schmidt.



(*) Block
— *. Diagram of lex (*)

Q.(4) Role of Lexical Analyzer.

→



- takes input string & produces the stream of tokens.
- it eliminates blanks & comments.
- it generates symbol table which stores information about identifires, constants

: key Roles :

1. Tokenization
2. parsing literals
3. managing operators & symbols
4. Error handling.
5. generating a stream of tokens.

Q.(5) Regular Definition For Signed & Unsigned Numbers.

(a) Unsigned Numbers :

- the Unsigned numbers don't have any sign for representing negative numbers. so, the unsigned numbers are always positive.

- By default, the decimal number representation is positive.
- There is no sign bit in Unsigned Num so it can only represented by its magnitude.

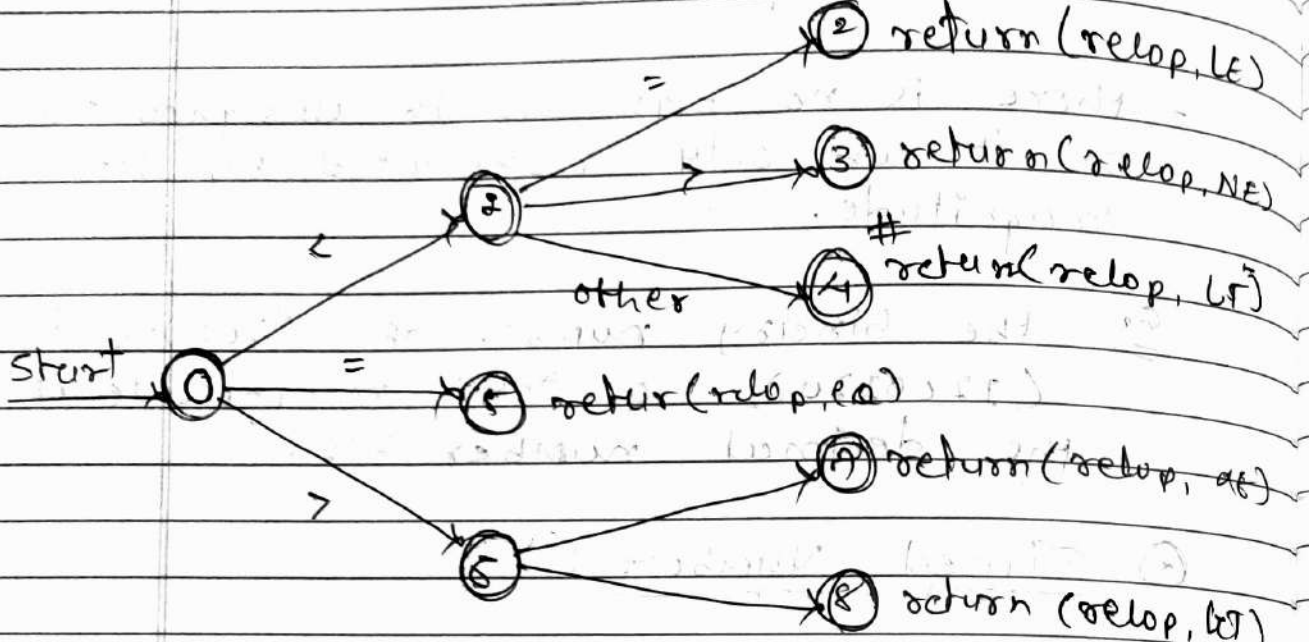
Ex: The binary number of $(702)_{10}$ is $(1200220)_2$, a 7-bit magnitude of the decimal number 702.

(*) Signed Numbers:

- The signed numbers have a sign bit so that it can differentiate positive and negative integer numbers.
- The signed binary number technique has both the sign bit & magnitude of the number.
- There are following types of representation of signed binary numbers:

(1) Sign-magnitude form
(2) 1's complement
(3) 2's complement

Q.6. Draw the transition Diagram for relational operators.



Q.(7) write a Regular Expression for.

=.

(1) the languages of all strings that do not end with 01.

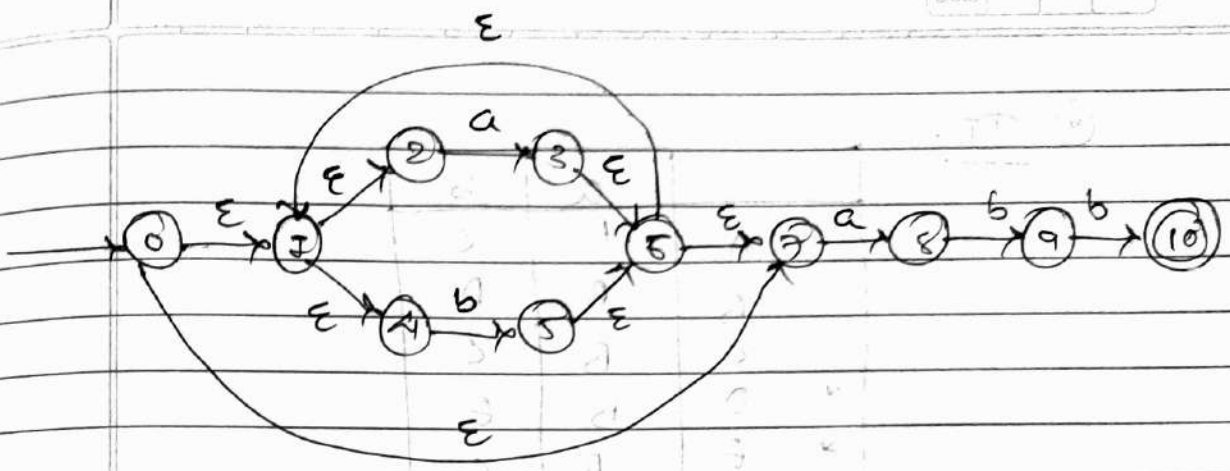
(2) All strings of digit that contains no leading 0's.

→ $(0+1)^*(0|11+00+10)$

→ $1(0+1)^*$

Q.(8) constructs non-deterministic F.A by using Thompson's construction for following regular expression: $(a|b)^*abb$

Show the sequence of move node in processing the input string ababbab.



- ε- closure = {0, 1, 2, 4, 7} — (A).

move (A, a) = {3, 8}
ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} — (B).

move (A, b) = {5}
ε-closure (5) = {1, 2, 4, 6, 7} — (C).

B = {1, 2, 3, 4, 6, 7, 8}

move (B, a) = {3, 8}
ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} — (B)

move (B, b) = {5, 9}
ε-closure (5, 9) = {1, 2, 4, 5, 6, 7, 9} — (D)

C = {1, 2, 4, 5, 6, 7}

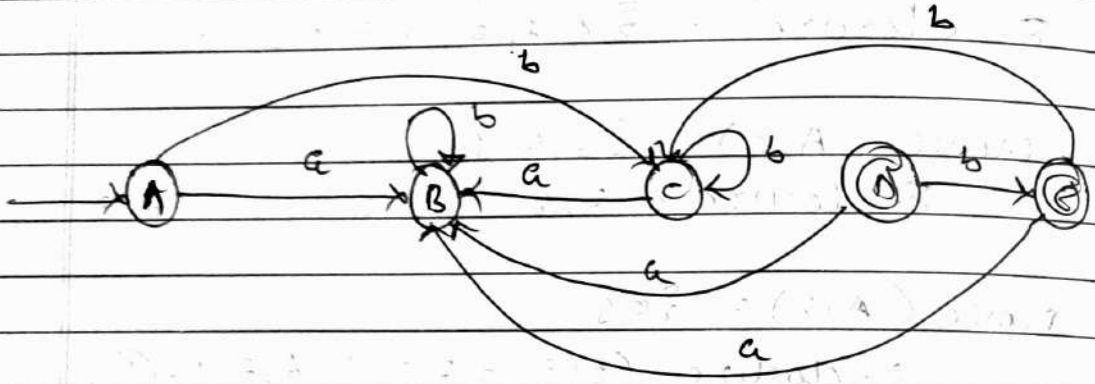
move (C, a) = {3, 8} = {1, 2, 4, 6, 7, 8} — (B)
move (C, b) = {5} = {1, 2, 4, 5, 6, 7} — (C)

move (D, a) = {3, 8} = {1, 2, 4, 6, 7, 8} — (B)
move (D, b) = {5} = {1, 2, 4, 5, 6, 7} — (E)

move (E, a) = {3, 8} = {1, 2, 4, 6, 7, 8} — (B)
move (E, b) = {5} = {1, 2, 4, 5, 6, 7} — (C)

(*) TT:

		a	b
	A	B	C
	B	B	D
	C	B	C
*	D	B	E
*	E	B	C



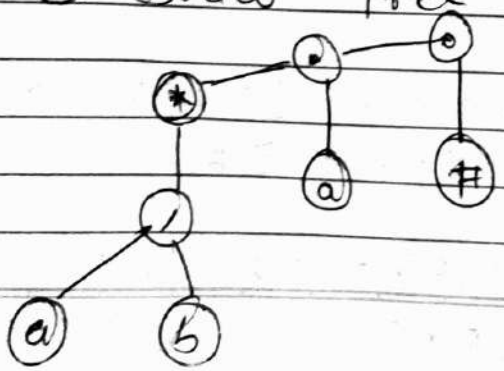
Q.(9) Draw DFA for the following regular expression using firstpos(), lastpos() and followpos() functions. (a/b)*a.

- We append the given string with the symbol # at the end.

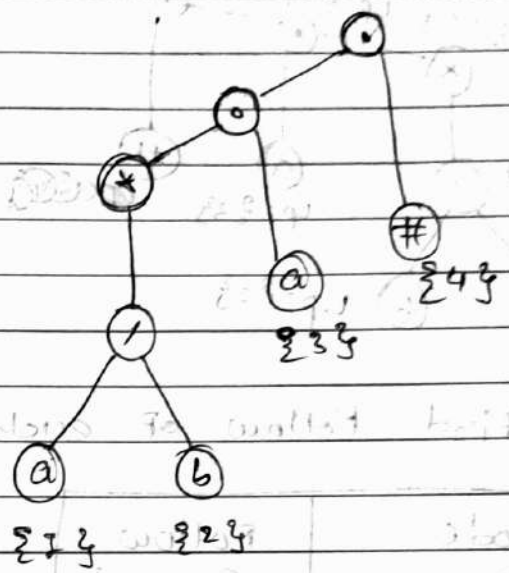
∴ the string becomes,

→ (a/b)*a#.

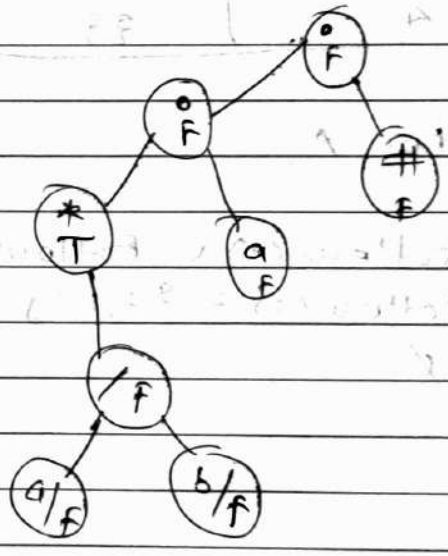
Step: (1) Draw Tree



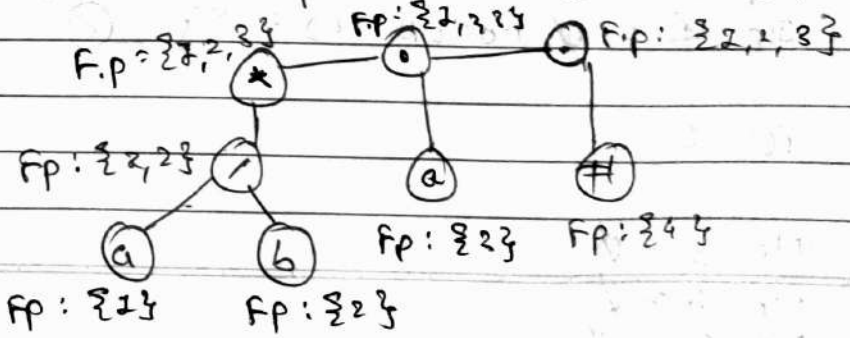
step: ② Number of leaf Node.



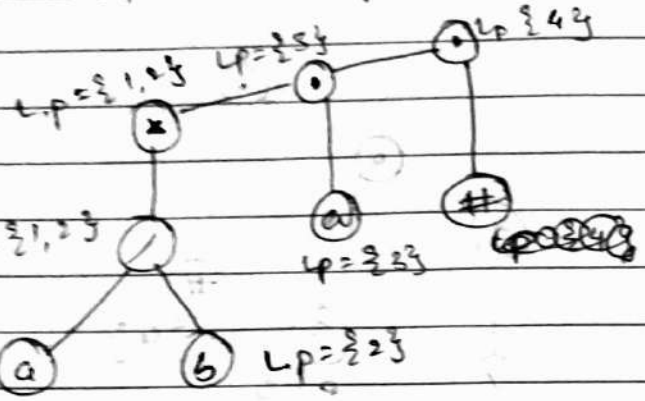
step: ③ Find nullable of each node.



step: ④ Find firstpos at each node.



Step: ⑤ : Find lastpos of each node.



Step: ⑥ : Find Follow of each node.

node	Follow
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{}

step: ⑦ : For A,

1p a = Follow(1) ∪ Follow(3) = {1,2,3,4} — (B)
1p b = Follow(2) = {1,2,3} — (A)
1p # = ∅

For B,

1p a = Follow(1) ∪ Follow(3) = {1,2,3,4} — (B)
1p b = Follow(2) = {1,2,3} — (A)
1p # = Follow(4) = {} — (C)

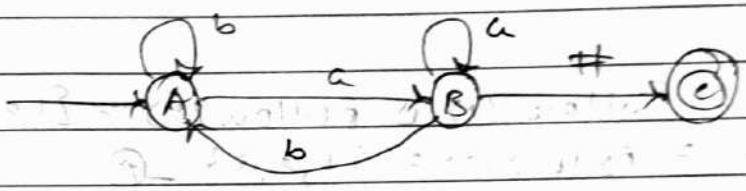
For C,

1p a = ∅
1p b = ∅
1p c = ∅

∴ TT:

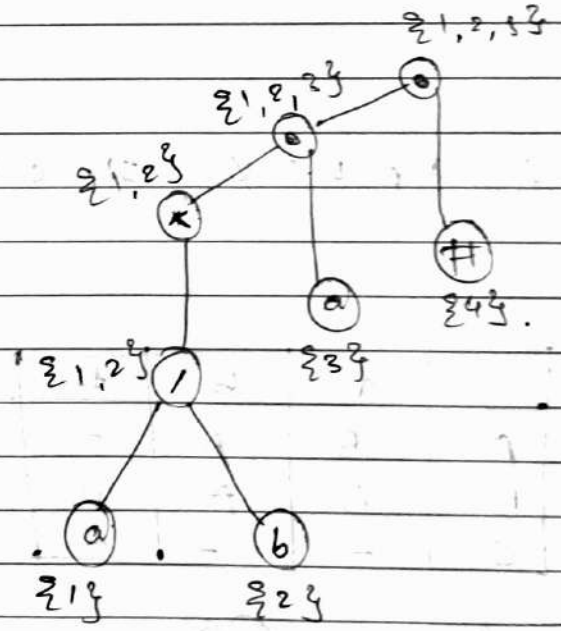
	a	b	#
A	B	A	-
b	B	A	C
C	-	-	-

DFA:



Q.(11) construct a DFA without constructing NFA for the following regular Expression $(a|b)^*a\#$.

- Appending string with #.
String becomes $(a|b)^*a\#$.



(*) ~~Desi~~ Tree (*)

First root = {1,2,3} — (A)

Follow(1) = {1,2,3}
Follow(2) = {1,2,3}
Follow(3) = {4}
Follow(4) = {} — null state

For A,

I/p a = Follow(1) ∪ Follow(3) = {1,2,3,4} — (B)
I/p b = Follow(2) = {1,2,3} — (A)
I/p # = ∅

For B,

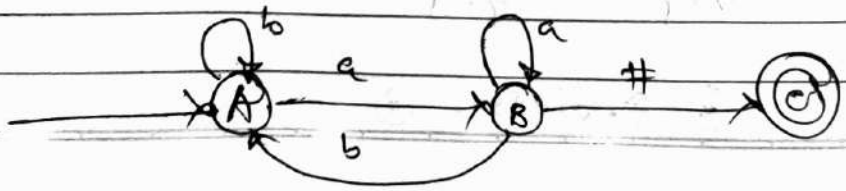
I/p a = Follow(1) ∪ Follow(3) = {1,2,3,4} — (B)
I/p b = Follow(2) = {1,2,3} — (A)
I/p # = Follow(4) = {} — (C)

For C,

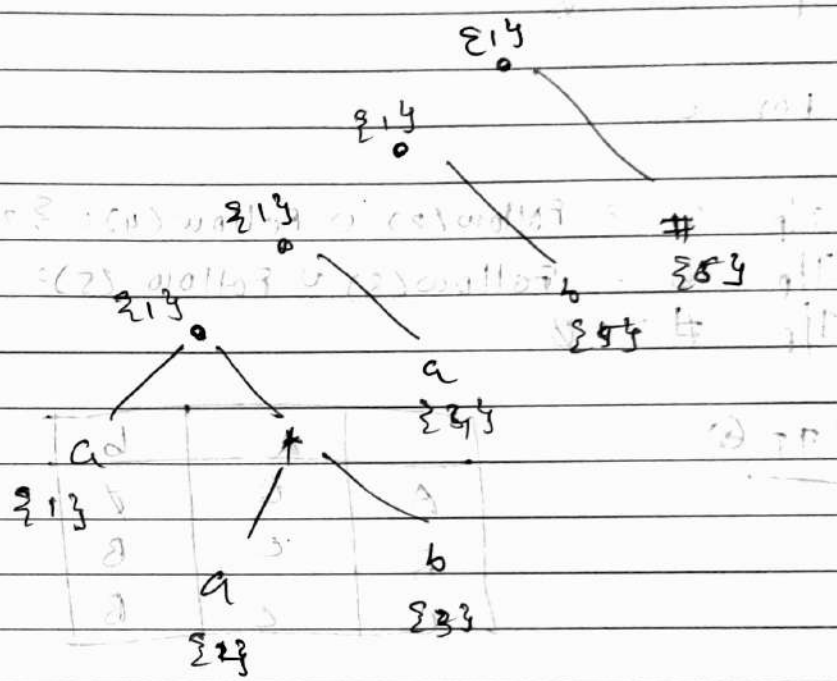
I/p a = ∅, I/p b = ∅, I/p c = ∅.

(C) II (A):

	a	b	#
A	B	A	-
B	B	A	C
C	-	-	-



Q. (16) Construct a DFA without constructing NFA for the following r.e: $a(a+b)^*ab$.



First root = {1}

Follow(1) = {2,3,4}
Follow(2) = {2,3,4}
Follow(3) = {2,3,4}
Follow(4) = {5}
Follow(5) = {} null state

For A,

fp a = Follow(1) = {2,3,4} ——— (B)
fp b = ∅
fp # = ∅

For B,

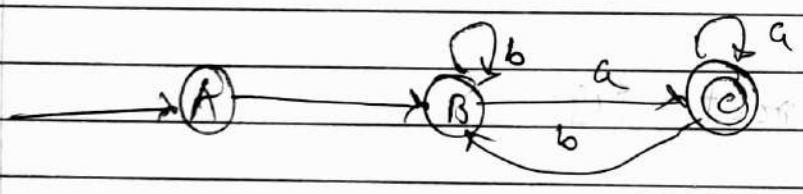
I/p a = Follow(4) = {2,3,4,5} → (C)
I/p b = Follow(3) = {2,3,4} → (B)
I/p # = ∅

For C,

I/p a = Follow(2) ∪ Follow(4) = {2,3,4,5} → (C)
I/p b = Follow(3) ∪ Follow(5) = {2,3,4,5} → (B)
I/p # = ∅

(C TT B)

	a	b
A	B	∅
B	C	B
C	C	B

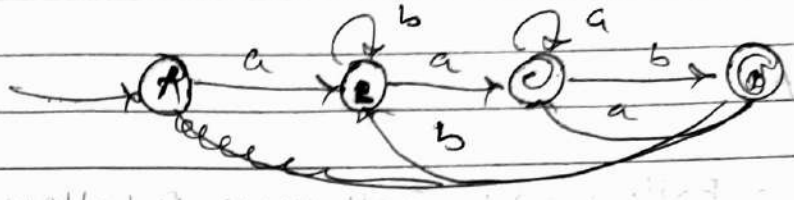


For D :

I/p a = Follow(3) ∪ Follow(4) = → (C)
I/p b = Follow(3) = {2,3,4} → (B)
I/p # = ∅

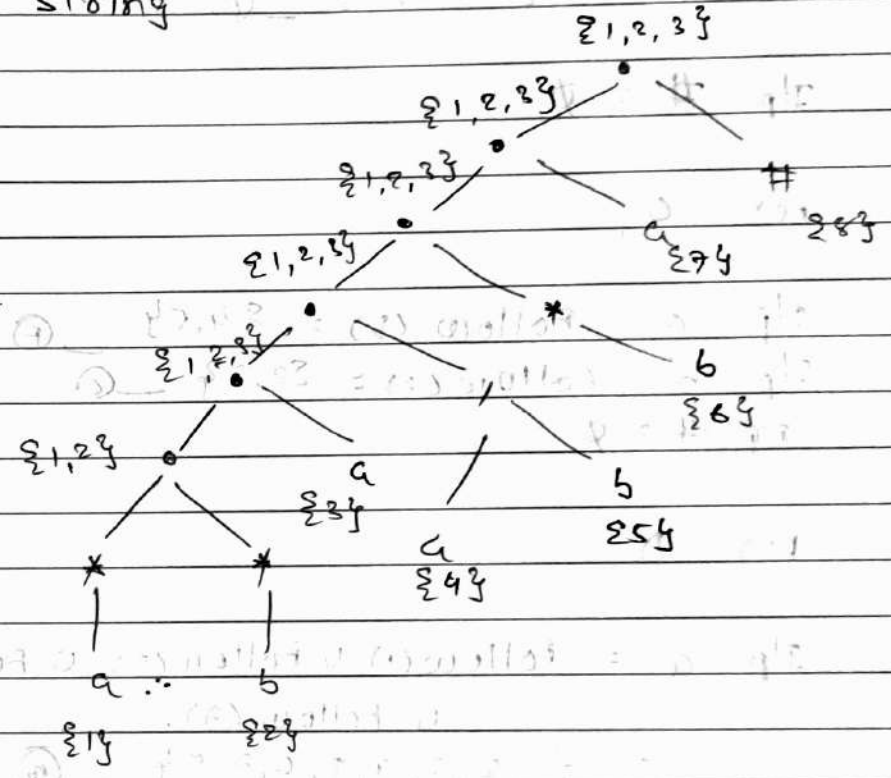
(TT)

	a	b	#
A	B	-	-
B	C	B	-
C	C	D	-
D	C	B	-



Q.(12) Construct DFA without constructing NFA for following r.e : $a^*b^* \in a(a|b)^*b^*a\#$.

- given string



First root = {1,2,3} — (A)

Follow(1) = {1,2,3}
Follow(2) = {2,3}
Follow(3) = {4,5}
Follow(4) = {6,7}
Follow(5) = {6,7}
Follow(6) = {6,7}
Follow(7) = {8}
Follow(8) = {} = null state.

For A,

I/p a = Follow(1) ∪ Follow(3) ∪ Follow(4)
= {1, 2, 3, 4, 5, 6, 7} ——— (D)

I/p b = Follow(2) ∪ Follow {5}
= {2, 3, 6, 7} ——— (E)

I/p # = ∅

For C,

I/p a = Follow(3) = {4, 5} ——— (F)
I/p b = Follow(2) = {2, 3} ——— (G)
I/p # = ∅

For D,

I/p a = Follow(1) ∪ Follow(3) ∪ Follow(4)
∪ Follow(7).
= {1, 2, 3, 4, 5, 6, 7, 8} ——— (H)

I/p b = Follow(2) ∪ Follow(5) ∪ Follow(8)
= {2, 3, 6, 7} ——— (I)

I/p # = ∅.

For E,

I/p a = Follow(3) ∪ Follow(7) = {4, 5, 8} ——— (J)
I/p b = Follow(2) ∪ Follow(6) = {2, 3, 6, 7} ——— (K)
I/p # = ∅.

For F,

I/p a = Follow(4) = {6, 7} ___ (I)

I/p b = Follow(5) = {6, 7} ___ (J)

I/p # = ∅

For G,

I/p a = Follow(1) ∪ Follow(3) Follow(4) ∪ Follow(7)
= {1, 2, 3, 4, 5, 6, 7, 8} ___ (G)

I/p b = Follow(2) ∪ Follow(5) ∪ Follow(6)
= {2, 3, 6, 7} ___ (E)

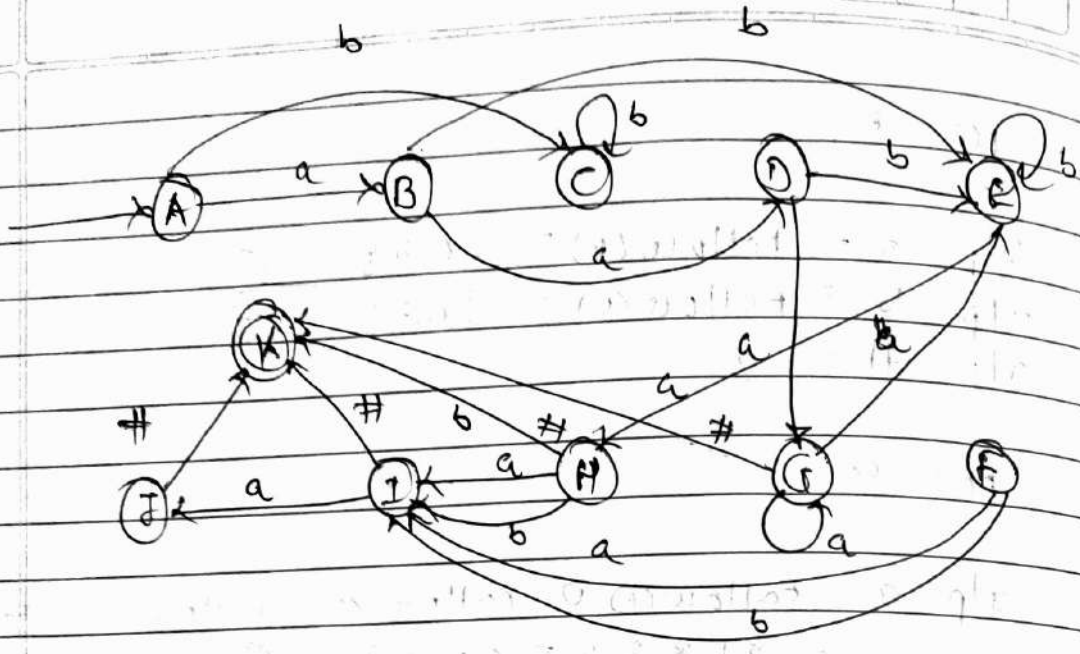
I/p # = ∅ = Follow(8) = {} ___ (H)

For K,

I/p a = ∅, I/p b = ∅, I/p # = ∅.

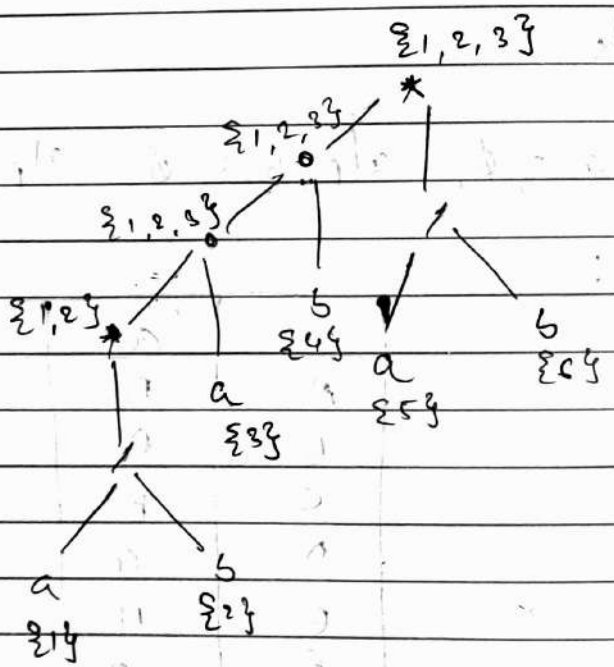
∴ (IT) :

	a	b	#
A	B	C	-
B	D	E	-
C	F	C	-
D	G	E	-
E	H	E	-
F	I	I	-
G	G	E	K
H	I	I	K
I	J	I	K
J	-	-	K
K	-	-	-



Q.(13) Construct DFA for following R.E without constructing NFA & optimize same.
(a|ε)* ab (a|b)*.

=.



First root = {1,2,3} ——— (A)

Follow (1) = {1,2,3}
Follow (2) = {1,2,3}
Follow (3) = {4}
Follow (4) = {5,6}
Follow (5) = {5,6}
Follow (6) = {5,6}.

For A,

I/p a = follow(1) ∪ follow(3) = {1, 2, 3, 4} — (B)
I/p b = For ∅
I/p ε = follow(2) = {1, 2, 3} — (A)

For B,

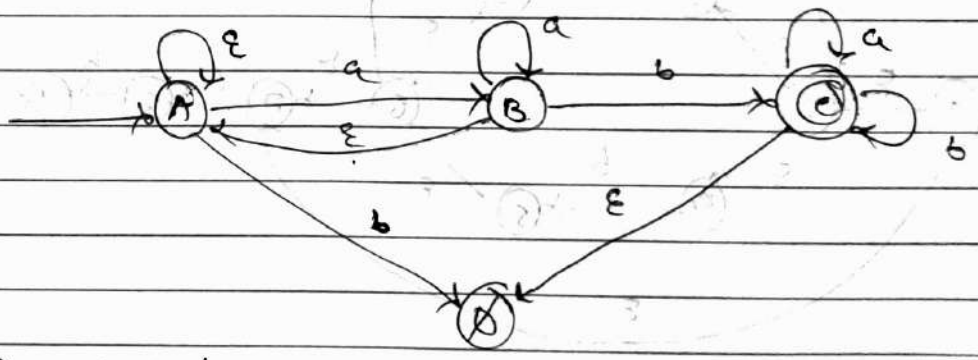
I/p a = follow(1) ∪ follow(3) = {1, 2, 3, 4} — (B)
I/p b = follow(4) = {5, 6} — (C)
I/p ε = follow(2) = {1, 2, 3} — (A)

For C,

I/p a = follow(5) = {5, 6} — (C)
I/p b = follow(6) = {5, 6} — (C)
I/p ε = ∅.

(2) TT (A)

	a	b	ε
A	B	–	A
B	B	C	A
* C	C	C	–



(3) optimizing DFA (3)

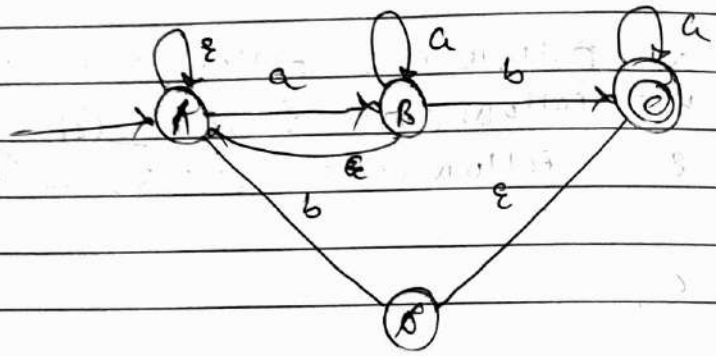
→ {A, B, ∅} = Non accepting state
{C} = Accepting state.

For state A & B.

$\delta(A, a) = B$ $\quad \delta(B, a) =$
$\delta(A, b) = \emptyset$ $\quad \delta(B, b) =$ } Not matching.
$\delta(A, \varepsilon) = A$ $\quad \delta(B, \varepsilon) =$

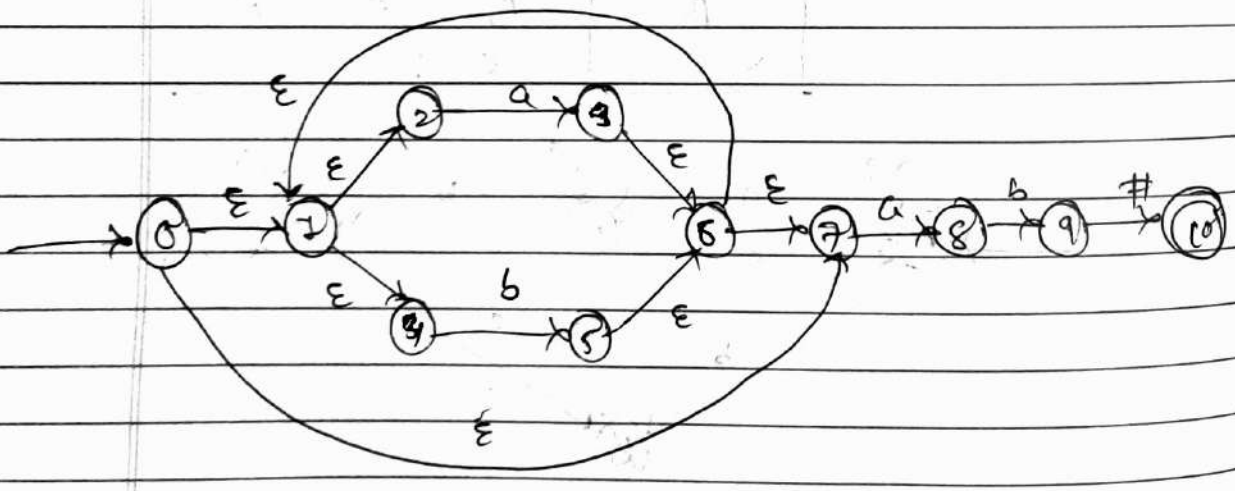
- So, we can't merge any state.

DFA:



Q.(14) Construct the NFA using Thompson's Notation for following regular Expression and then construct to DFA:

$(a/b)^* abb\#$.



ε-closure {0) = {0, 1, 2, 4, 7} ___ (A).

→ move (A, a) = {3, 8}

ε-closure (3, 8) = {1, 2, 4, 3, 4, 6, 7} — (B)

move (A, b) = {5}

ε-closure = {1, 2, 4, 5, 6, 7} — (C)

move (A, #) = ∅

→ move (B, a) = {3, 8}

ε-closure (3, 8) = {1, 2, 3, 4, 6, 7, 8} — (B)

move (B, b) = {5, 9}

ε(5, 9) = {1, 2, 4, 5, 6, 7, 9} — (D)

move (B, #) = ∅

→ move (C, a) = {3, 8}

ε(3, 8) = {1, 2, 3, 4, 6, 7, 8} — (B)

move (C, b) = {5}

ε(C, a) = {1, 2, 4, 5, 6, 7, 9} — (C)

move (C, #) = ∅.

→ move (D, a) = {3, 8}

ε(3, 8) = {1, 2, 3, 4, 6, 7, 8} — (B)

move (D, b) = {5}

ε(5) = {1, 2, 4, 5, 6, 7} — (C)

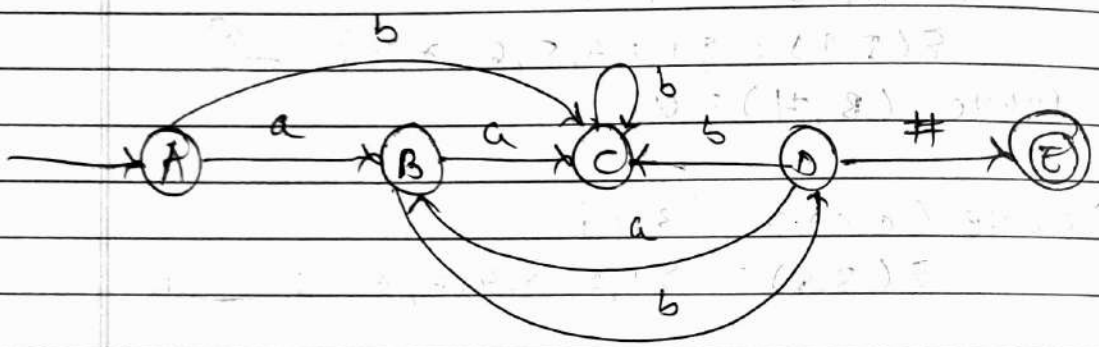
move (D, #) = {10}

ε(10) = {10} — (E)

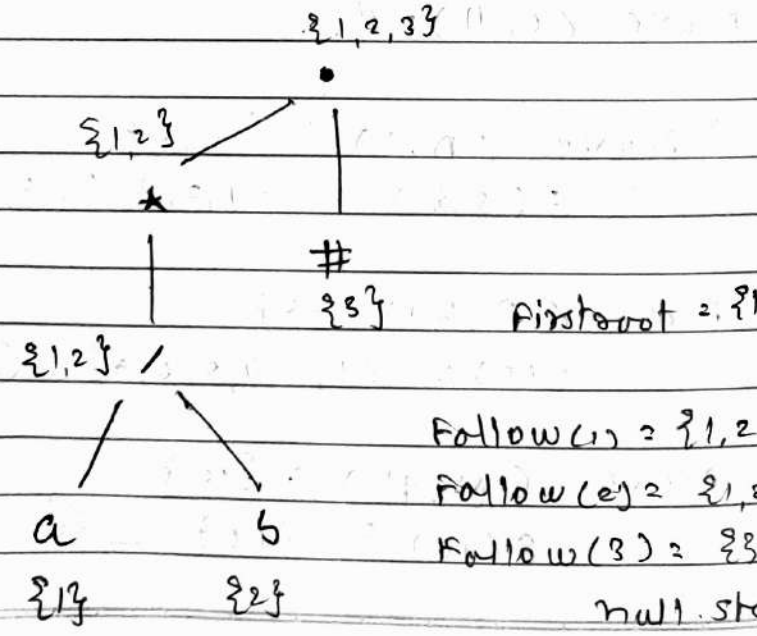
→ move (E, a) = ∅
move (E, b) = ∅
move (E, #) = ∅.

⊛ TT ⊛

	a	b	#
A	B	C	∅
B	B	D	∅
C	B	C	∅
D	B	C	E
E	∅	∅	∅



Q. (15) optimize the DFA for R.E: (a/b)* #.



Firstpos = {1,2,3}

Follow(1) = {1,2,3}
Follow(2) = {1,2,3}
Follow(3) = {}
null state.

→ For A,

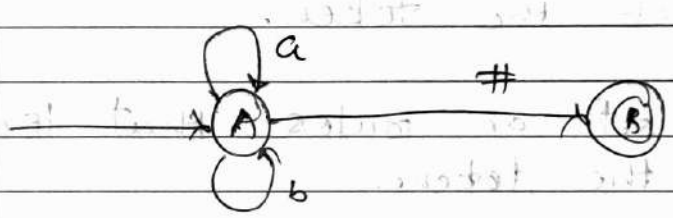
Ip a = Follow (1) = {1, 2, 3} ___ (A)
Ip b = Follow (2) = {1, 2, 3} ___ (A)
Ip # = Follow (3) = {} ___ (B)

→ For B,

Ip a = ∅
Ip b = ∅
Ip # = ∅

(*) IT. (*)

	a	b	#
A	A	A	B
B	-	-	-



- Optimizing DFA:

1. A is the initial state.
2. B is the only state since the r.e allows any number of a and b followed by.

- So, the DFA is already in its minimal form. and there are no ~~for~~ further r.e.

Q.(18) Define token, lexeme and pattern. Identify the lexeme that make up the token for the following code.

```
Const  p = 10;

if (a<p)
 {
    a++;
     if (a==5);
     continue;
 }
```

- lexeme: Sequence of characters in the source program that is matched with the pattern of the token.

- pattern: set of rules that lexemes the tokens.

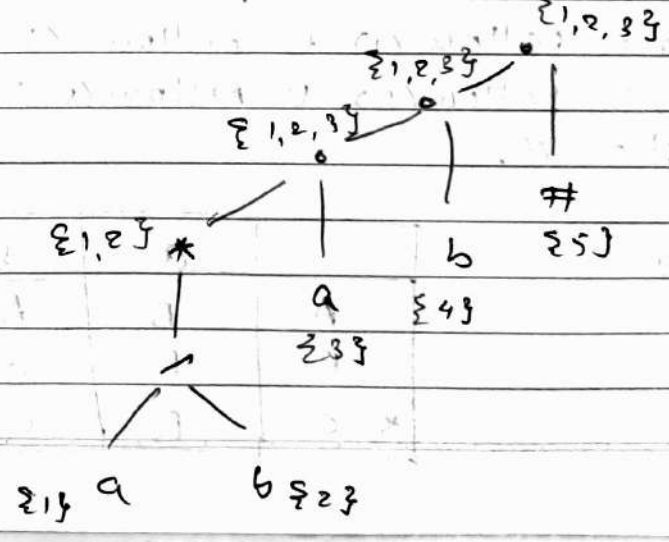
- Tokens: It decisions the class or category.

- Tokens: and lexemes in the provided code.

1. 'const' (keyword Token)
2. 'p' (Identifier token)
3. ':=' (Assignment operator token)
4. '10' (Integer token).

5. ';' (semicolon token)
6. 'if' (keyword token)
7. '(' (left parenthesis token)
8. 'a' (Identifier parenthesis token)
9. '<' (less than operator token)
10. 'p' (Identifier token)
11. ')' (Right parenthesis token)
12. '{' (left Brase token)
13. 'a' (Identier token)
14. '++' (Increment operator token)
15. ';' (semicolon token)
16. 'if' (Identifier token)
17. '(' (left parenthesis token)
18. 'a' (Identifier token)
19. '==' (Equality operator token)
20. '5' (Integer literal Token)
21. ')' (Right parenthenthesic token)
22. 'continue' (keyword token)
23. ';' (semicolon token)
24. '}' (Right Brace token).

Q (19) Construct deterministic finite automata without NFA for following r.e :
$(a|b)^* abb^*$.



First root : = {1, 2, 3} ——— (A)

Follow (1) = {1, 2, 3}
Follow (2) = {1, 2, 3}
Follow (3) = {4}
Follow (4) = {5}
Follow (5) = {5}

- For A,

I/p a = Follow (1) ∪ Follow (3)
= {1, 2, 3, 4} ——— (B)

I/p b = Follow(2) = {1, 2, 3} ——— (A)
I/p # = ∅.

- For B,

I/p a = Follow (1) ∪ Follow(3) = {1, 2, 3, 4} ——— (B)
I/p b = Follow(2) ∪ Follow(4) = {1, 2, 3, 5} ——— (C)
I/p # = ∅.

- For C,

I/p a = Follow(1) ∪ Follow (3) = {1, 2, 3, 4} ——— (B)
I/p b = Follow(2) ∪ Follow (5) = {1, 2, 3, 5} ——— (C)
I/p # = ∅.

(x) TT. (x)

	a	b
A	B	A
B	B	C
* C	B	C

Q.(18) write a r.e definition for:

1. write the language of all strings containing at least one 0 and at least one 1.

2. the language of all string that do not containing 01.

3. the language of all strings which are containg 0's and 1's both are even.

4. the language of all strings which are starting with 1 and ending with 0.

→ (1) = $(0+1)^*(01+10)(1+0)^*$

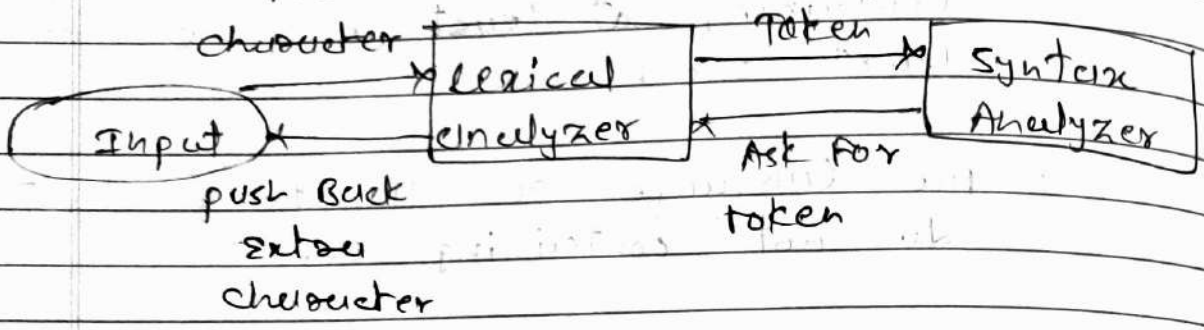
(2) $(0+1)^*(00+10+11)$

(4) $1(0+1)^*0$

Q.(19) what is lexical analysis? which are the token performed by lexical analyzer.

- It is the first phase of the compiler also known as a scanner. It converts the high level input program into a sequence of tokens.

- It can be implemented with the Deterministic Finite automata.



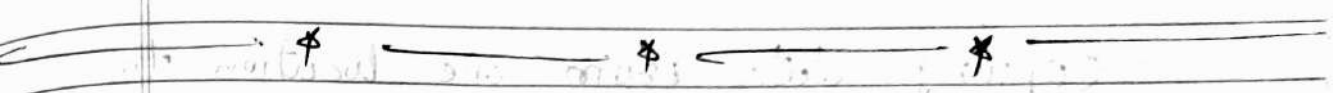
- task performed by lexical analyzer.
- It is responsible for removing the white space and Comments from the source program.
- It is corresponds to the error message with the source program.
- it helps to identify the tokens.
- it input characters are read by the lexical analyzer from the source code.

Q. (20) write R.E the Following Language.

1. All Strings of 0's and 1's that do not Contains.
2. All strings of 0's and 1's that every 1 is Followed by 00.

(1) $(0+10)^*$

(2) $((0+1)00)^*$.



⑧ phases of compiler :-

(1) Lexical Analysis
(2) Syntax Analysis
(3) Semantic Analysis
(4) intermediate code Generation
(5) Code optimization
(6) Code generation

error report & recovery } front end

Symbol table } back end

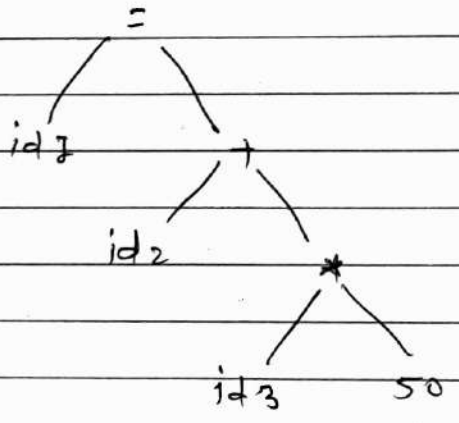
Ex. (1) L.A :- Sum := oldsum + Rate * 50

(2)

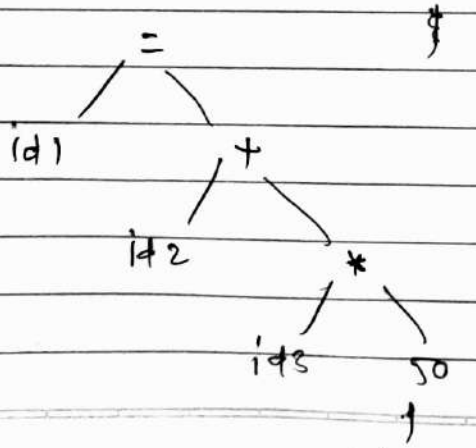
o.p =

id 1	identifier 1
id 2	identifier 2
=	equal sign
+	plus sign
id 3	identifier 3
50	Number 50

(2) S.A :-



(3) S.A :-

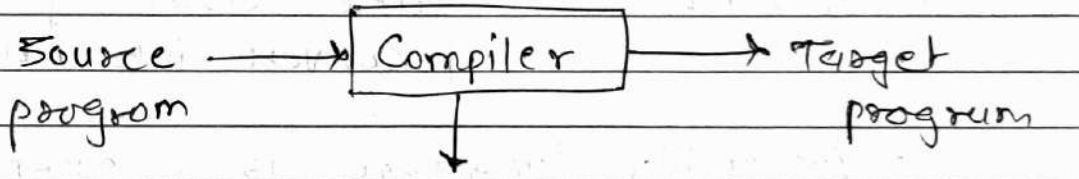


(4) i.c.g :- $t_1 := inttoreal(50)$
$t_2 := id3 * t_1$
$t_3 := id * t_2$
$id1 := t_3$

(5) c.o :- $t_1 := id3 * 50.0$
$id1 := id2 + t_1$

(6) c.g :-
```
MOVF   id3, R2
MOVF   #50.0, R2
MOVF   id2  R2
ADDF   R2, R1
MOVF   R1, id2.
```

(7) Compiler :-



Func^n :-

- convert one program to another
- convert such a way that the generated code should be easy to understand
- preserve meaning of source code.
- error report & recovery
- compilation must be done efficiently.

① Context of a Compiler :-

(1) Preprocessor :- produces input to Compiler.

Func :- (1) Macro processing
- define macros
(2) File inclusion
- include files
(3) Rational preprocessor
- provide built-in macro for construct statement like while & if.
(4) Language Extensions
- add capabilities to lang.

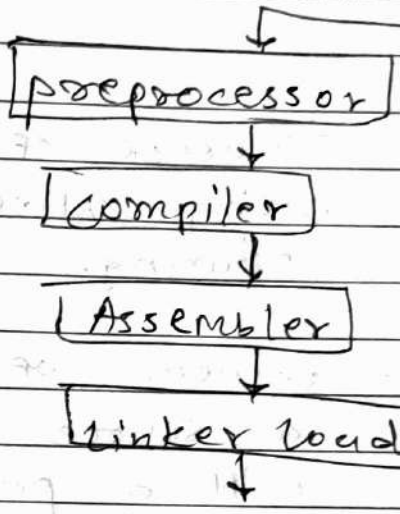
(2) Assembler :- take assembly language & convert into machine code.

- Assembly language is a mnemonic version of machin code, in which names are used instead of binary ~~code~~ op codes.

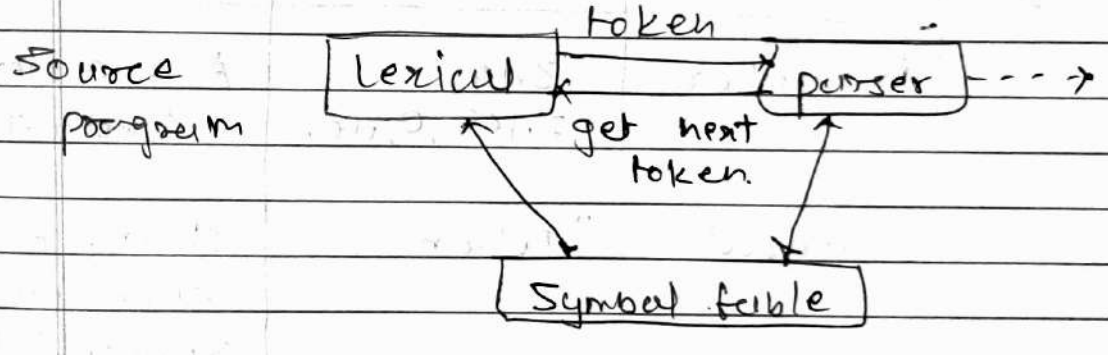
(3) Linker :- make single program from a several files of relocatable machine code.

- these file may have the result of several different compilation

(4) Loader :- taking relocatable m.code, alter relocatable address & place altered instructions & data in memory.



(2) Lexical Analyzer :-



(3) Communication between Scanner & parser (3)

Func" :- implemented by making subroutine.
"receive" "get next token" msg & point.
red input character until next token
- Stripping out comments & white space in the form of blanks, tabs & newline characters.

- Some lexical Analyzer are divided in two phases Scanning & lexical Analysis → for simple task, for complex task.

① Token :- sequence of characters having collective (similarity) meaning.

② Lexeme :- sequence of character in a source program matched with a pattern.

③ pattern :- set of rules associated with token.

Ex

token	Lexeme	pattern
Number	3.14, 2.168, 0, etc.	Any numeric constant
Literal	"Rohan"	string of character between " ".

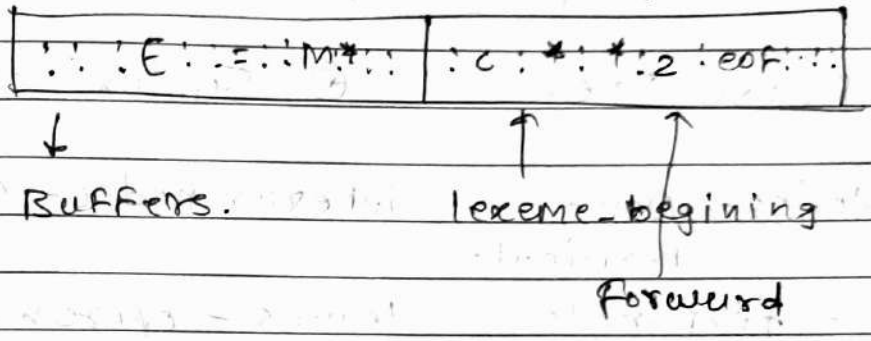
Ex total = Sum + 10

identifier operator id2 operator constant

	pattern
tokens : total, =, sum, +, 10	identifier, operator, identifier, operator, constant.
lexemes : total, =, sum, +, 10	

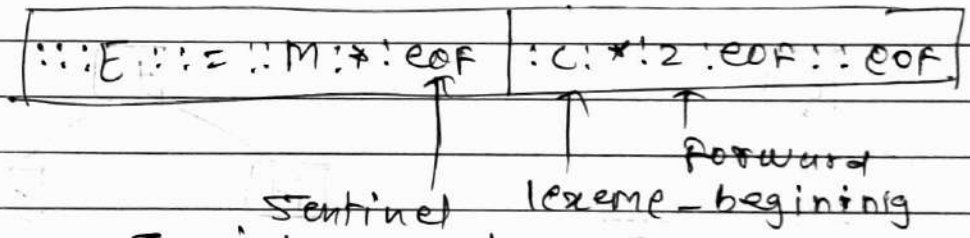
① Input buffering techniques :-

(1) Buffer pair - reduce amount of
(2) Sentinels overhead of I. Analysis input scanning.



pointer lexeme_begining - marks beginning of current lexeme
pointer Forward - scan ahead until a pattern match found

② Buffer pair ②

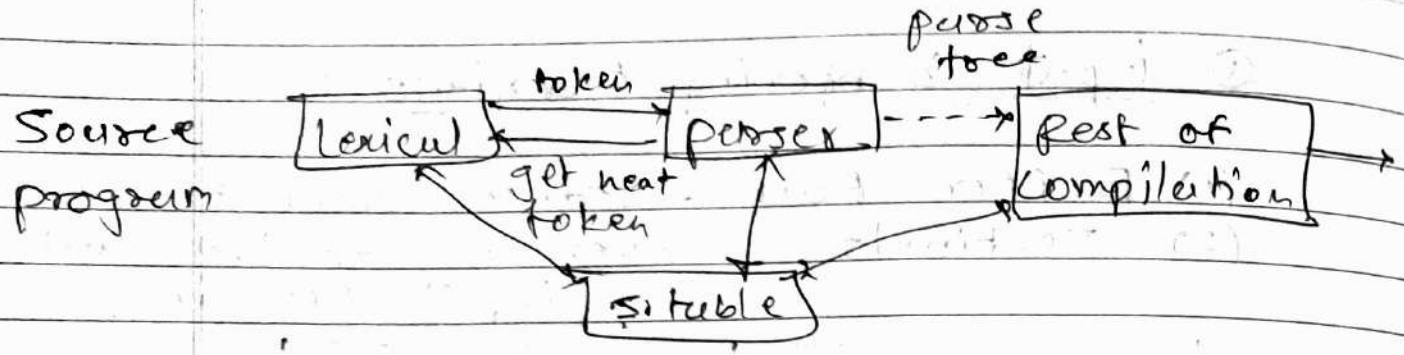


Special character (which is not a part of program).

③ Sentinels at end of each buffer ③

④ Parser :-

get token → make parse tree
if true
else give error msg.



① parse tree : ② Syntax tree

Interior Nodes - Non-terminals, interior nodes - operators

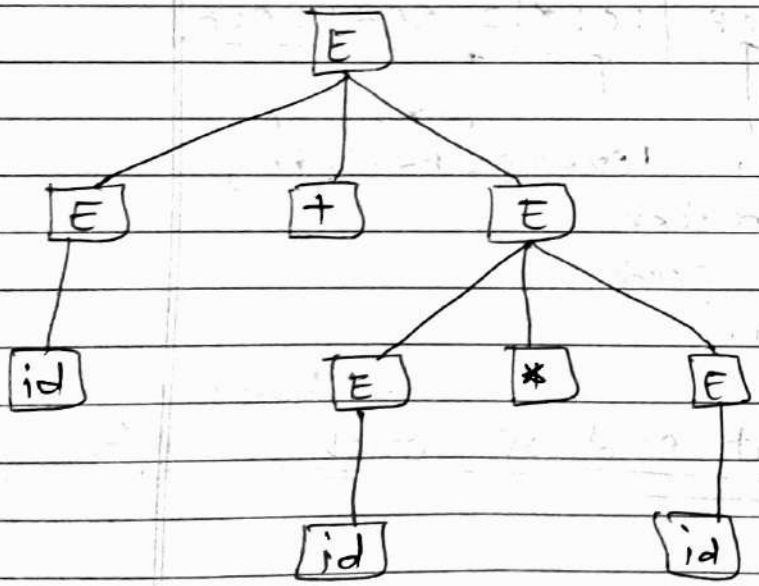
leaves - terminals leaves - operands

↓ ↓

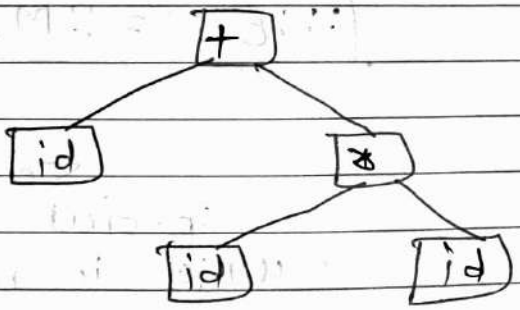
make concrete Syntax of program make abstract Syntax of program

G : $E \to E+E$
$E \to E*E$
$E \to id$

→ String : a + b * c

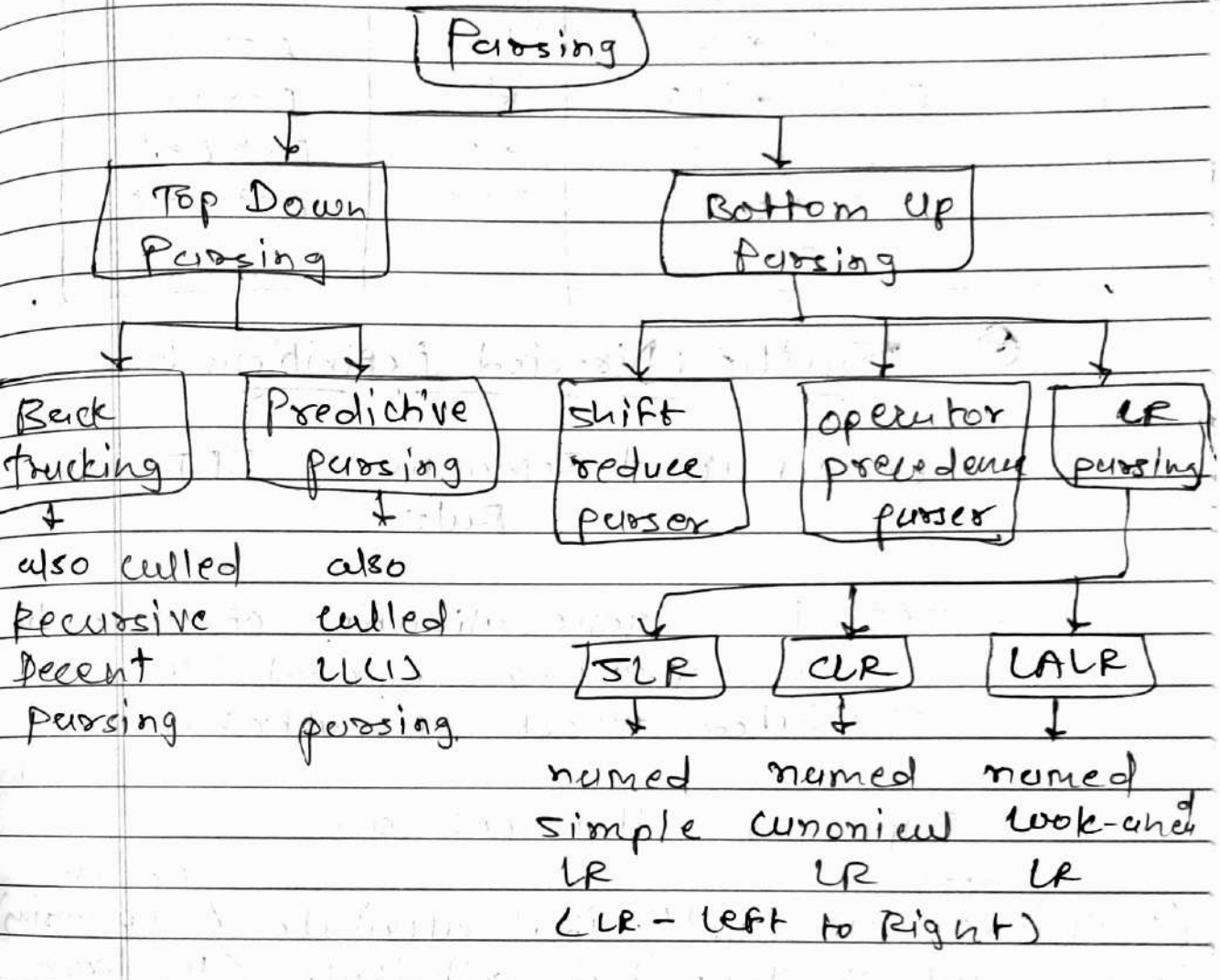


① parse tree ①



② syntax tree ②

① Types of Parsing :-



(LR - Left to Right)

---

② (*) Handle - substring of a string that matches right side of production & whose reduction to the non-terminal is one step along the reverse of R.M.D.

③ Handle :- process of discovering handle & reducing it to appropriate left hand side non-terminal.
pruning

Right sentential form	Handle	reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

⑤ Syntax Directed Definitions :-

- Grammar + Semantic Rules = SDT

- SDD is a generalization of CFG in which grammar symbol has an associated set of attributes.

- Types of attributes are,

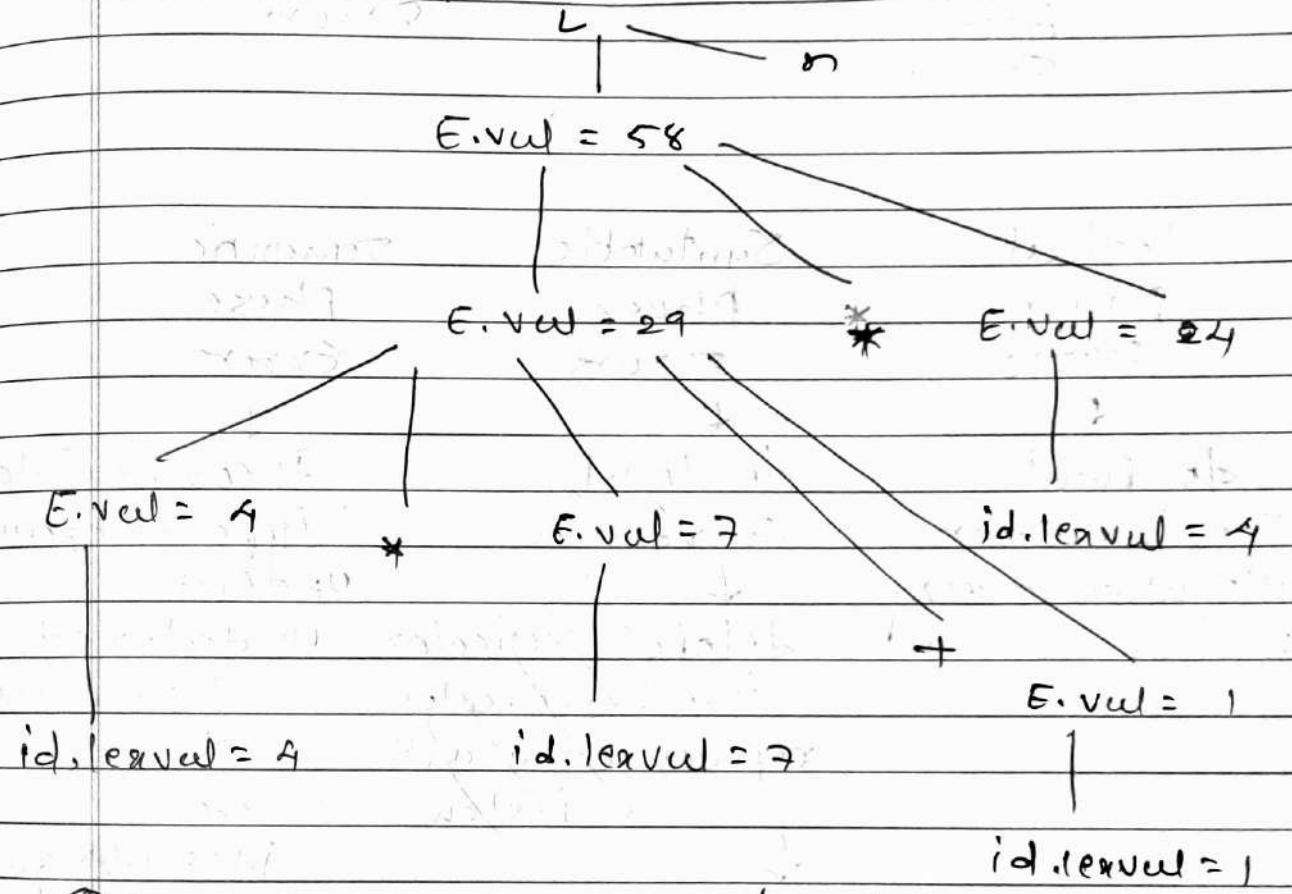
(s-attributed SDT) (1) Synthesized attribute (bottom up parsing)

(s-attributed + L-attributed (2) Inherited attribute (top down parsing with restriction of left siblings)

Ex. Annotated parse tree : $3 * 5 + 4$ n.

Annotated parse tree → show value of attribute at each node.
+
→ process of computing attribute values at node.

production	Semantic Rules
$L \to En$	Print (E.val)
$E \to E + E$	E.val = E.val + E.val
$E \to E * E$	E.val = E.val * E.val
$E \to id$	E.val = id.lexval



② Annotated parse tree for (4 * 7 + 1) * 2n ②

④ Error Recovery Strategies

(1) Panic Mode
(2) Phrase Level Recovery
(3) Error Production
(4) Global Correction

Types of Errors

Compile time Errors

Runtime Errors

Lexical phase Errors
↓
Spelling Errors
illegal characters
numeric constants

Syntactic phase errors
↓
do local correction
↓
delete semicolon
insert semicolon
replace comma with semicolon
↓
Errors in structure
missing operators.
Unbalanced parenthesis's
Ex
printf("%d", a);
No ;

Semantic phase Errors
↓
Incompatible type of operand
Undeclared Variables.
↓
Ex
id1 = id2 * 30
↓
Not Compa-

⊛ Function of Error Handler = Error Detection + Error Report + Error Recovery.

③ Intermediate code Generation :-

(1) Postfix Notation
(2) Three Address code
(3) Syntax tree & DAG.

Ex $a = b * -c + b * -c$

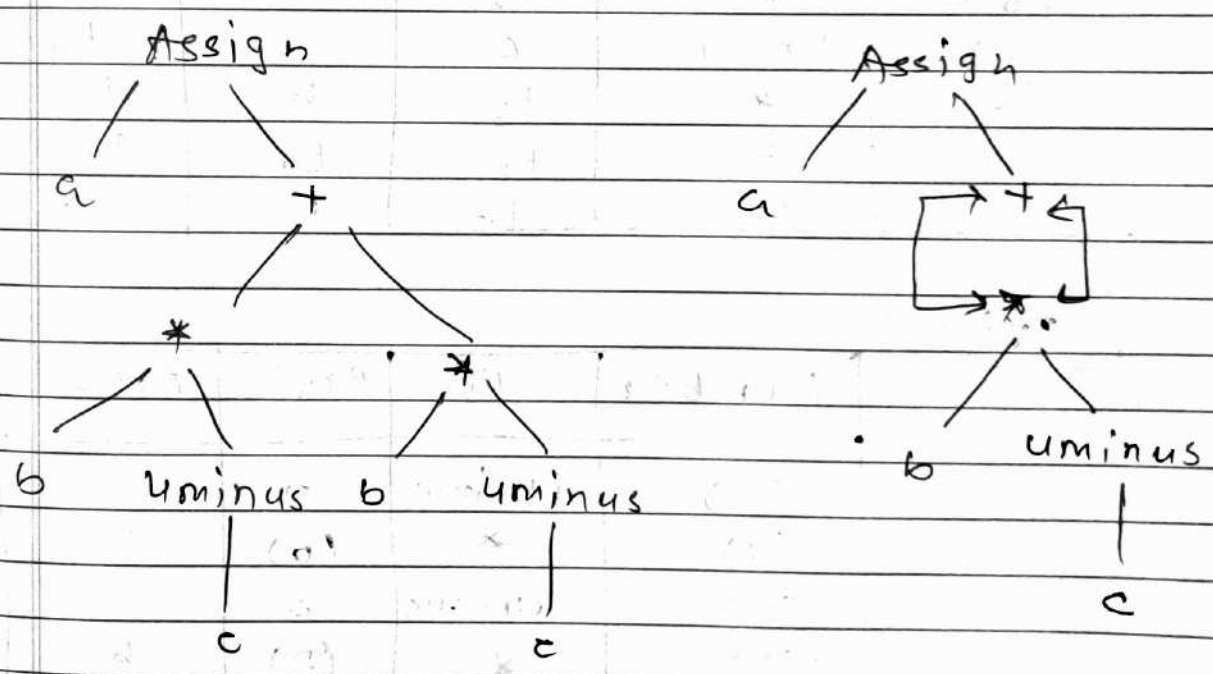
postfix → $ab = * -c + b * -c$

$ab = -c * + b * -c$

$ab = -cb + * -c$

$ab = -cb + -c *$

Syntax tree & DAG



④ Syntax tree ⑤ ⑥ DAG ⑦

Ex. for TAC :

$X := -a * b + -a * b$

$t_1 := -a$
$t_2 := t_1 * b$
$t_3 := -a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$X = t_5$

Quadruple

Number	OP	Arg 1	Arg 2	result
(0)	Uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	Uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		X

Triple

Number	OP	Arg 1	Arg 2
(0)	Uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

Indirect Triple

Num	OP	Arg1	Arg2
(0)	Uminus	a	
(1)	*	(11)	
(2)	Uminus	a	b
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	x	(15)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

⊛ Activation Record :-

- Execution of Procedure - A.R
- contains all necessary information required to call a procedure
- Function call occurs → new A.R created → pushed into stack → procedure completed → popped out from stack.

Fields of Activation Record →

1. return values
2. Actual Parameters
3. Control link
4. Access link
5. Machine status
6. Local Variable
7. Temporary values

RACAMLT.

⊛ Activation Record ⊛

⊛ Activation Tree :-

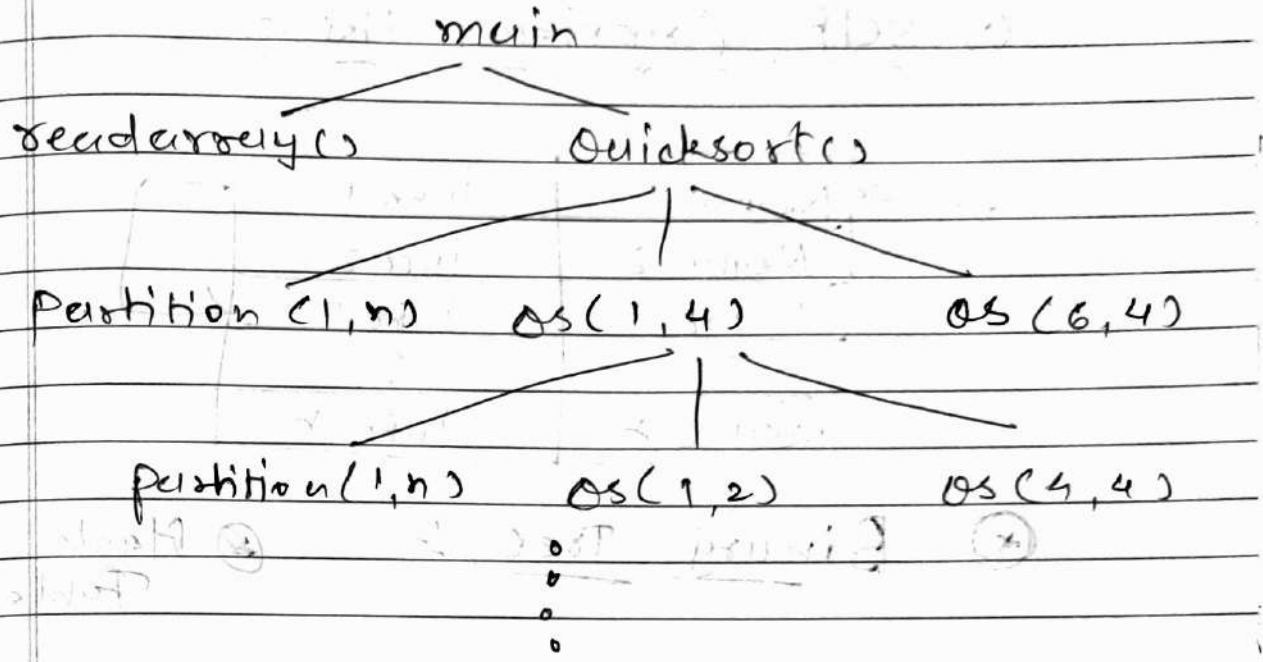
- tree structure that represents func" calls made by a program during execution.
- A.R process --- popped out.

⊛ properties :-

- Each node represents an activation of a procedure.
- root shows — activation main
- procedure 'x' is parent of pro. 'y' if & only if control flow procedure x to procedure y.

Ex.

```
main (){
    int n;
    readarray();
    quicksort (1,n);
}

Quicksort (int m, int n) {
    int i = partition (m,n);
    quicksort (m, i-1);
    quicksort (i+1, n);
}
```



② Symbol Table with Data Structures :-

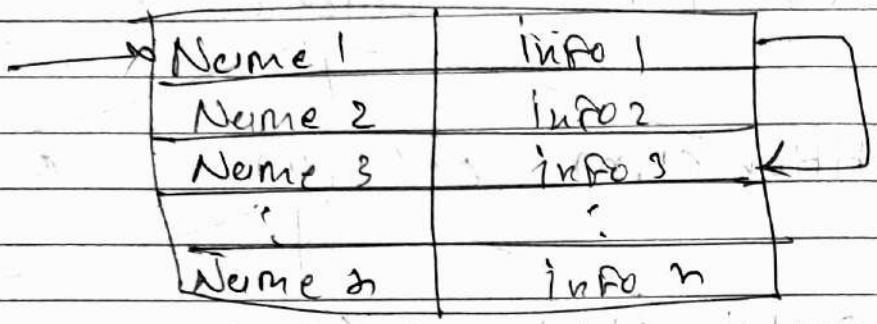
③ Data structures for Symbol table :-

(1) List Data Structure
(2) Self Organizing List
(3) Binary Tree.
(4) Hash Table

④ List Data Structures :-

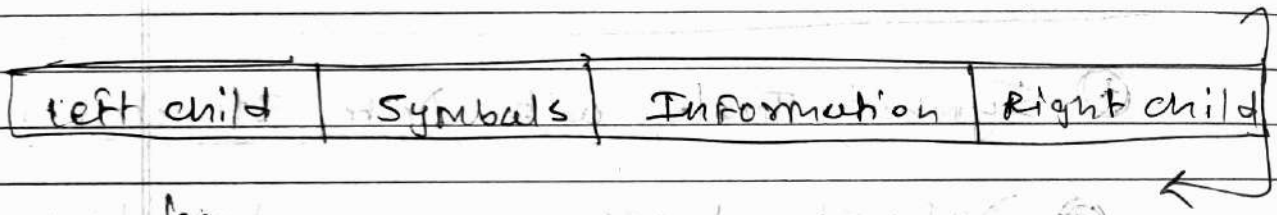
Name 1	info 1
Name 2	info 2
Name 3	info 3
Name 4	⋮
Name n	info n

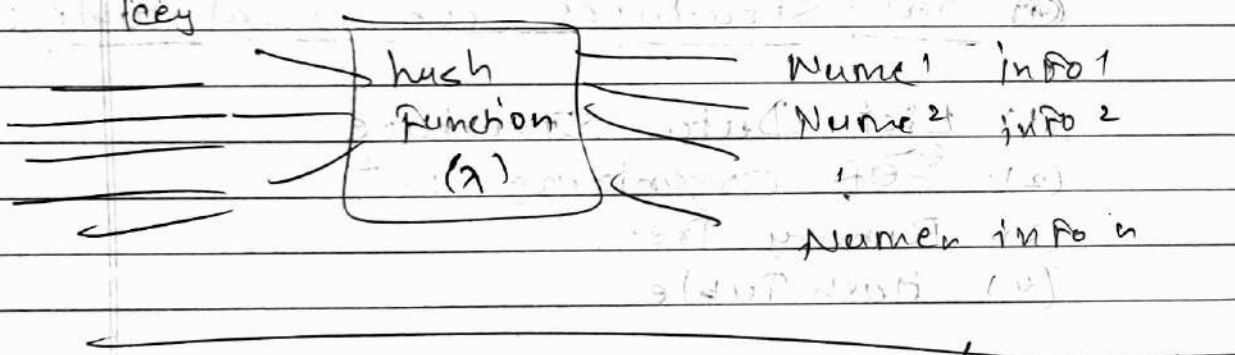
⊛ Self Organizing list :-



⊛ Binary Tree :-

⊛ Hash Table :-





⊛ Code Optimization :- (Machine independent)

(1) Compile time Evaluation :
- Folding : (22/7 → 3.14)
- Constant Folding (pi = 3.14)

(2) Common sub Expression elimination

(3) Code Movement

(4) Reduction in strength (* → +)

(5) Dead code Elimination

(6) Copy Propagation.

① Issues in Code Generator :-

(1) Input to code generation
(2) Target program → Absolute m.lang
Relocatable m.lang
Assembly m.lang.
(3) Memory Management
(4) Instruction selection → mov..
(5) Register allocation
(6) Choice of evaluation
(7) Approaches to code generation.

② Loop Optimization :-

(1) code motion
(2) Induction Variable elimination
(3) Strength Reduction
(4) loop unrolling
(5) loop Jumping

③ Peephole Optimization :- (machine dependent)

(1) Redundant load & store elimination
(2) constant folding
(3) Strength reducing
(4) Null Agebric Expressions
(5) combine operations
(6) Dead code elimination

④ DAG :- Leaves → Variable / constants
Interior nodes → Operator.

Nodes are also given a sequence of identifiers for labels to store the computed values.

⑤ Basic Block :-

input: three address code
output: list of basic blocks with each tac in exactly one block.

(1) determine set of leaders
↓
(1) first statement
(2) target conditional/unconditional goto statement
(3) Any statement immediately Follow goto statement.

⑥ Code scheduling Constraints :-

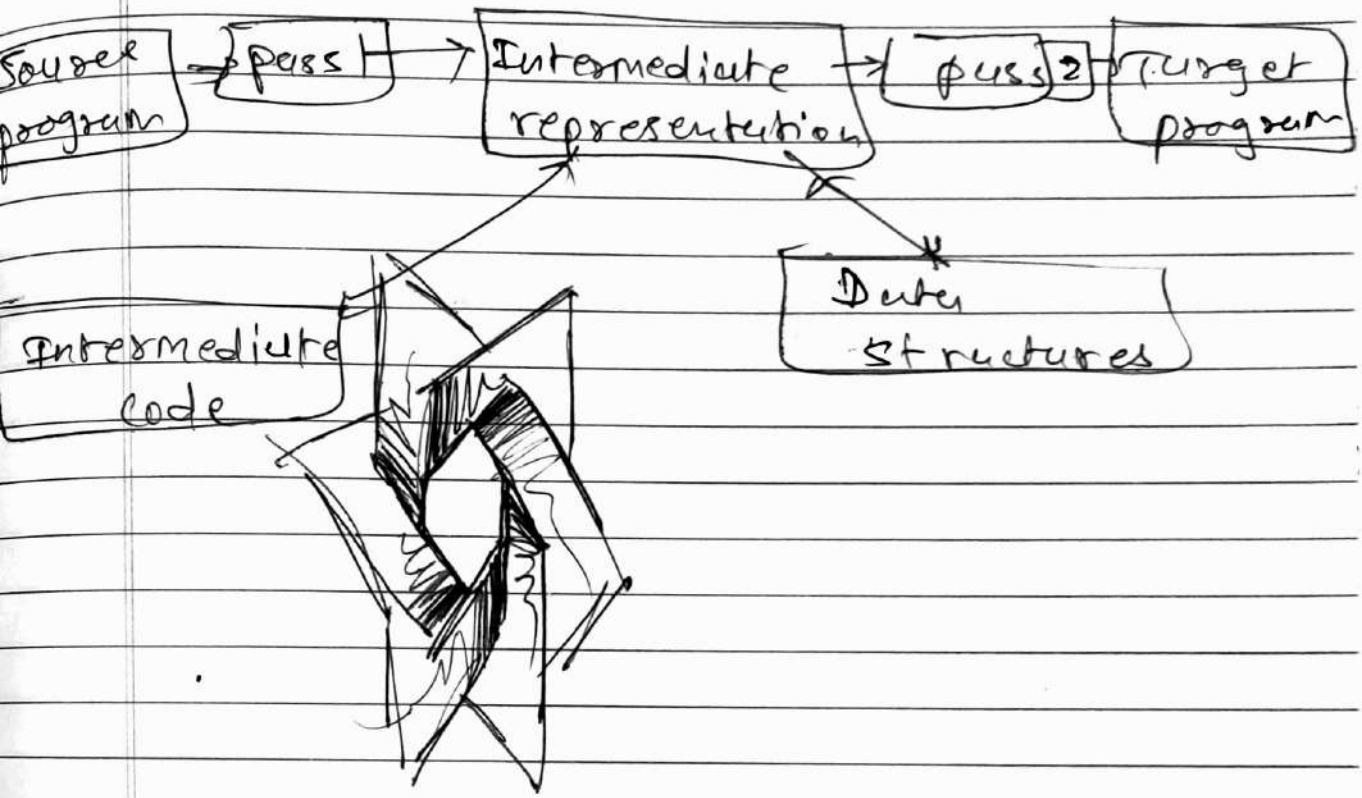
(1) Control dependence graph (same operations)
(2) Data dependence graph (same results)
(3) Resource Constraints (not oversubscribe resources)

⑦ Basic-Block Scheduling :-

(1) Data-Dependence Graph :-
↓
$G = (N, E)$ → Node → representing
Edge → representing

- Each n in N a resource
- reservation table RT_n whose
- Each edge e in E is labled with delay de indicating first node must be issued

(2) List Scheduling of Basic Blocks:
(3) Prioritized Topological Orders.



operations in the machine instructions.
data-dependence constrains among the operations.

value is simply resource-reservations.

no earlier source node.

② Error Recovery Questions ②

Q.① Explain Error recovery strategies used by parser. (7)

OR

Explain error recovery strategies.

OR

Explain error Recovery Strategies in compiler in brief.

Q.② Discuss the functions of error handler. (3)

Q.③ Explain the following with example. (4)

(1) Lexical phase Error
(2) Syntactic phase Error

Q.④ Explain panic mode recovery strategy. (3)

Q.① Ans :-

- There are mainly 4 e.r.s :-

(1) Panic Mode
(2) phrase Level recovery
(3) Error production
(4) Global generation.

- [1] Panic Mode :-

- In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found.

- Synchronizing tokens are delimiters such as semicolon or end.

- These tokens indicate an end of the statement.

- If there is less number of error in the same statement then this strategy is best choice.

```
fi() ---> Scan entire line otherwise
  {       Scanner will return fi as
            valid identifier
  }
```

[2] Phrase Level Recovery :-

- In this method, on discovering an error parser performs local correction on remaining input.

- The local correction can be :
  (1) Replacing comma by semicolon
  (2) Deletion of semicolons
  (3) Inserting missing semicolons.

- This type of local correction is decided by compiler designer.

- This method is used in many error-replacing compilers.

[3] Error Production :-

- If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with error productions that generate the erroneous constructs.

- Then we use the grammar augmented by these error production to construct a parser.

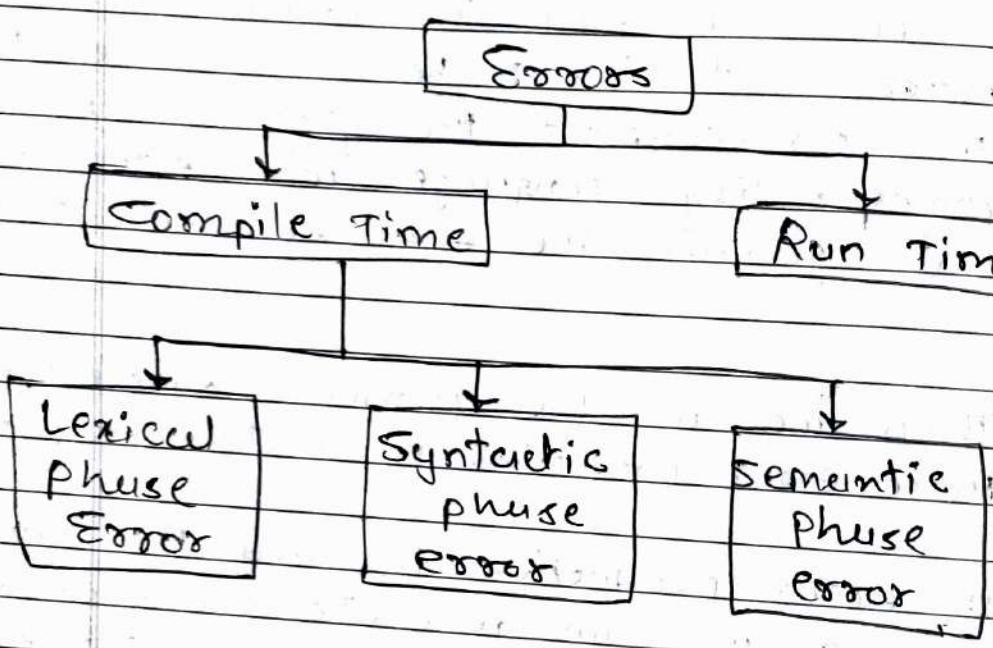
- If error production is used then, during parsing we can generate appropriate error message & parsing can be continued.

[4] Global Correction :-

- Given an incorrect input string x & grammar G, the algorithm will find a parser tree for a related string y, such that number of insertions, deletions, & changes of token required to transform x into y is as small as possible.

- Such methods increase time & space requirements at parsing time.
- Global Correction is thus simply a theoretical concept.

Q. ② Types of Errors :-



⑧ Lexical Phase Error :-

- Lexical phase error can be detected during lexical analysis phases.
- Typical lexical phase errors are :
  (1) Spelling errors
  (2) Exceeding length of identifier or numeric constants.
  (3) Appearance of illegal characters.

Ex :

```
Fi ()
{

}
```

- in above code 'Fi' cannot be recognized as a misspelling of keyword if rather lexical analyzer will understand that it is an identifier and will return it as valid identifier.

- Thus, misspelling causes errors in token formation.

(*) Syntax Phase Error :-

- Syntax error appear during syntax analysis phase of compiler.

- Typical syntax phase error are :

  (1) Errors in Structure.
  (2) Missing operators
  (3) Unbalanced parenthesis

- The parser demands for token lexical analyzer & if the tokens do not satisfies the grammatical rules of programming language then the syntactical errors get raised.

Eg:
printf("Hello World!!!");

Error:
Semicolon missing

④ Semantic phase Error :-

- semantic phase error detected during semantic analysis phases.
- Typical semantic phase errors are!

(1) Incompatible types of operands
(2) Undeclared Variables
(3) Not matching of actual argument with Formal argument

Eg: id1 = id2 + id3 * 60
↓
(directly we can't perform multiplication due to incompatible types of Variables)

⑤ Function of Error Handler :-

- The task of the Error Handling process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error.

- During this whole process processing time of the program should not be slow.

⊛ Functions of Error Handler :-

(1) Error Detection
(2) Error Report
(3) Error Recovery

Error handle = Error Detection +
Error Report +
Error Recovery.

⊛ Intermediate Code Generation :-

Q.① Translate following arithmetic expression.
(a * b) + (c + d) - (a + b)

① Quadruples :-

OP	opr1	opr2	Result
*	a	b	t_1
+	c	d	t_2
+	t_1	t_2	t_3
+	a	b	t_4
-	t_3	t_4	t_5

$t_1 = a * b$
$t_2 = c + d$
$t_3 = t_1 + t_2$
$t_4 = a + b$
$t_5 = t_3 - t_4$

② Triple :-

op	opr1	opr2
*	a	b
+	c	d
+	(1)	(2)
+	a	b
-	(3)	(4)

③ Indirect Triple :-

I00	(0)
I01	(1)
I02	(2)
I03	(3)
I04	(4)

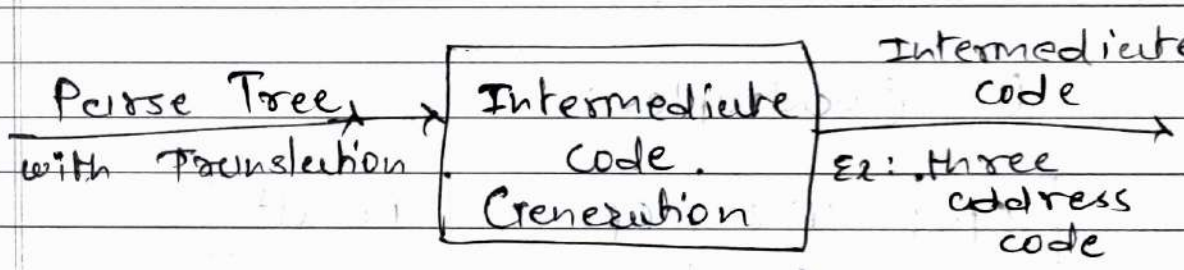
Q.② What is Intermediate form of code? What are the advantages of it?

- Intermediate code can translate the source program into the machine program.

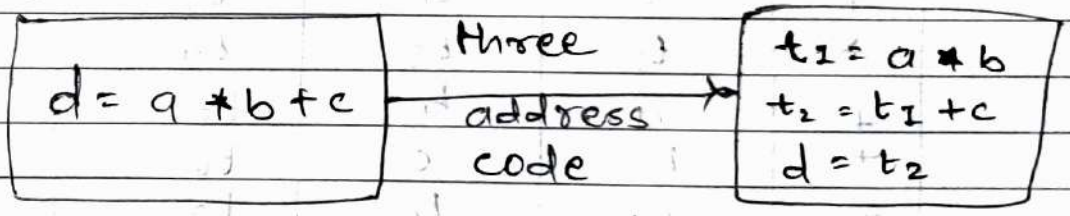
- Intermediate code is generated because the compiler can't generate machine code directly in one pass.

- First it converts the source program into intermediate code, which performs efficient generation of machine code further.

- The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph, three address codes, Quadruples & Triples.



Ex:



(*) Advantages of Intermediate code Generation :-

(1) Easy to Implement
(2) Facilities code optimization
(3) platform independence

(4) code reuse
(5) Easier debugging.

Q.(3) Translate the expression:

$-(a+b) * (c+d) + (a+b+c)$

$t_1 = a+b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2 * t_3$
$t_5 = a+b$
$t_6 = t_5 + c$
$t_7 = t_4 + t_6$

(1) Quadruples:

Op	opr 1	opr 2	Result
	e		
+	a	b	t_1
-	t_1		t_2
+	c	d	t_3
*	t_2	t_3	t_4
+	a	b	t_5
+	t_5	c	t_6
+	t_4	t_6	t_7

(2) Triple :-

#	OP	opr 1	opr 2
(0)	+	a	b
(1)	-	(1)	
(2)	+	c	d
(3)	*	(2)	(3)
(4)	+	a	b
(5)	+	(5)	c
(6)	+	(4)	(6)

(3) Indirect Triple :-

#	Statement	
200	(0)	(0)
201	(1)	(1)
202	(2)	(2)
203	(3)	(3)
204	(4)	(4)
205	(5)	(5)
206	(6)	(6)
207	(7)	(7)

Q. (4) $a = (a + b * c) * (b * c) + (b + c) \wedge a$

$t_1 = a + b$
$t_2 = t_1 * c$
$t_3 = a * t_2$
$t_4 = b * c$
$t_5 = t_3 * t_4$
$t_6 = b + c$
$t_7 = t_5 + t_6$
$t_8 = t_7 \wedge t_8$ $t_9 = a = t_8$

Three Address code.

Page No.
Date

Q.5 Explain Quadruple, Triple and Indirect Triple with example.

(1) Quadruple :-

- it is a structure which consists of 4 fields namely op, arg1, arg2 & result.

- op denotes the operator
- arg1 & arg2 denotes the two operands
- Result is used to store result.

(2) Triples :- easy to rearrange code.

- this representation doesn't make use of extra memory variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg2 & arg2.

↓

difficult to rearrange code.

(3) Indirect Triples :-

- This representation makes use of pointers to the listing of all references to computations which is made seperately & stored.

- It is similar in utility as compared to quadruple representation but requires less space than it.

Ex. (1) Quadruple :-

$t_1 = a * b$
$t_2 = t_1 + c$
$t_3 = t_2 \wedge D$

$a * b + c \wedge D$

OP	arg 1	arg 2	Result
*	a	b	t_1
+	t_1	c	t_2
$\wedge$	t_2	D	t_3

(2) Triple :-

#	OP	arg 1	arg 2
(0)	*	a	b
(1)	+	(1)	c
(2)	$\wedge$	(2)	D

(3) Indirect Triple :-

#	Statement
200	(0)
201	(1)
202	(2)

Q.⑥ Importance of Three Address code or (Intermediate code.)

(1) Easy to Implement :-

- Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.

(2) Facilitates code optimization :-

- Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance & efficiency of the generated code.

(3) Platform Independence :-

- Intermediate code is platform-independent, meaning that it can be translated into machine code.

(4) Code Reuse :-

- Intermediate code can be reused in the future to generate code for other platforms or languages.

(5) Easier debugging :-

- Intermediate code can be easier to debug than machine code, as it is closer to the original source code.

Q.⑧ Translate Following Expression:

$$-(a*b)+(c+d)-(a+b+c+d)$$

$t_1 = c+d$

$t_2 = a+b$

$t_3 = t_2 + t_2$

~~$t_4 = -t_3$~~ $t_4 = c+d$

~~$t_5 = c+d$~~ $t_5 = t_4 - t_3$

$t_6 = a+b$

$t_7 = t_6 + t_5$

$t_8 = -t_7$

(1) Quadruples :-

op	arg1	arg2	Result
+	c	d	t_1
+	a	b	t_2
+	t_1	t_2	t_3
+	c	d	t_4
−	t_4	t_3	t_5
*	a	b	t_6
+	t_6	t_7	t_7
Uminus	t_7		t_8

(2) Triples :-

#	Op	arg1	arg2
(0)	+	c	d
(01)	+	a	b
(2)	+	(1)	(2)
(3)	+	c	d
(4)	−	(4)	(3)
(5)	*	a	b
(6)	+	(6)	(7)
(7)	Uminus	(7)	

(3) Indirect Triples :-

#	Statement
700	(0)
701	(1)
702	(2)
703	(3)
704	(4)
705	(5)
706	(6)
707	(7)

Q. (9) (10) .

(*) Run Time Environments :-

Q. (1) Explain stack Allocation & Activation Record Organization in brief.

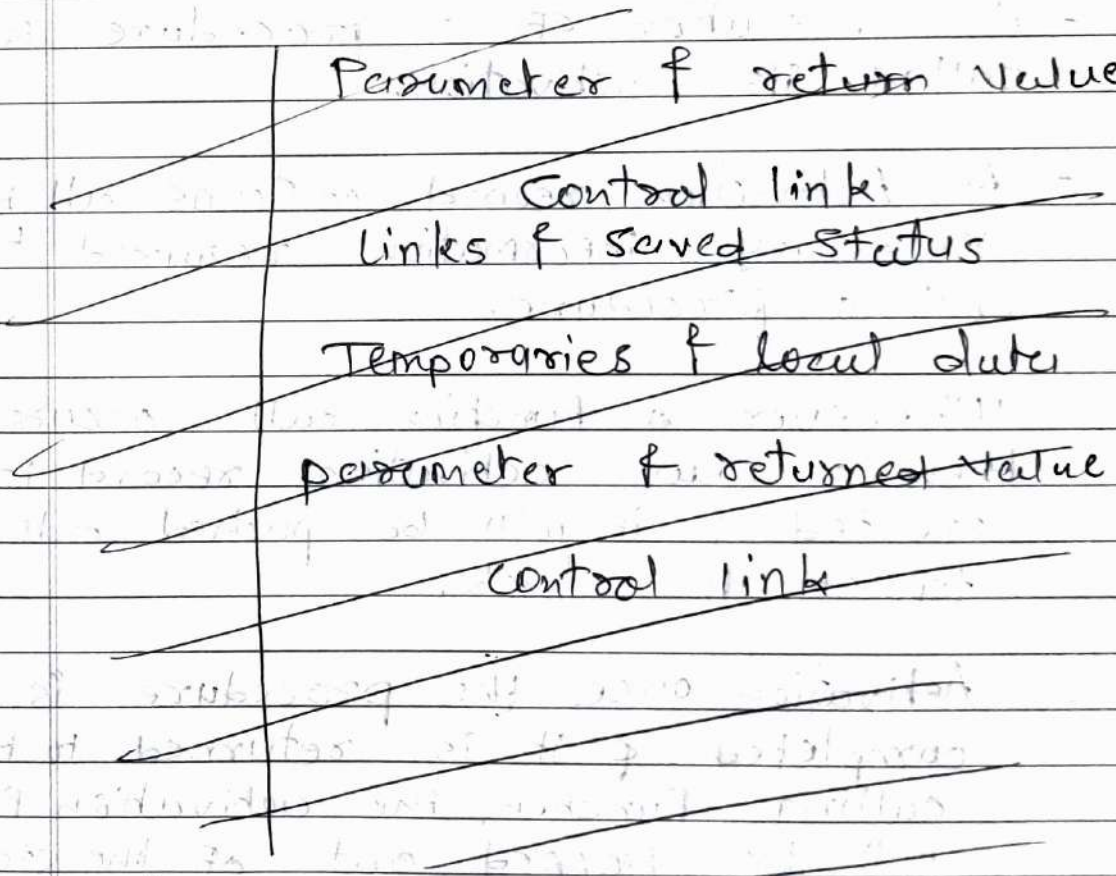
(*) stack Allocation :-

- Stack allocation is commonly known as Dynamic allocation.
- Dynamic allocation means the allocation of memory at run-time
- Stack is a data structure that follows the LIFO principle so whenever there is multiple activation record created

it will be pushed in the stack as activations begin & ends.

- local Variables are bound to new storage each time whenever the activation record begins because the storage is allocated at runtime every time a procedure or function call is made.

- ~~procedures are implemented by generating what are known as calling sequences in the target code.~~



- When the activation ~~record gets popped~~ out, the local ~~variables values~~ get

Ex. void sum (int a, int b) {

```
int ans = a+b;
cout << ans;
```

}

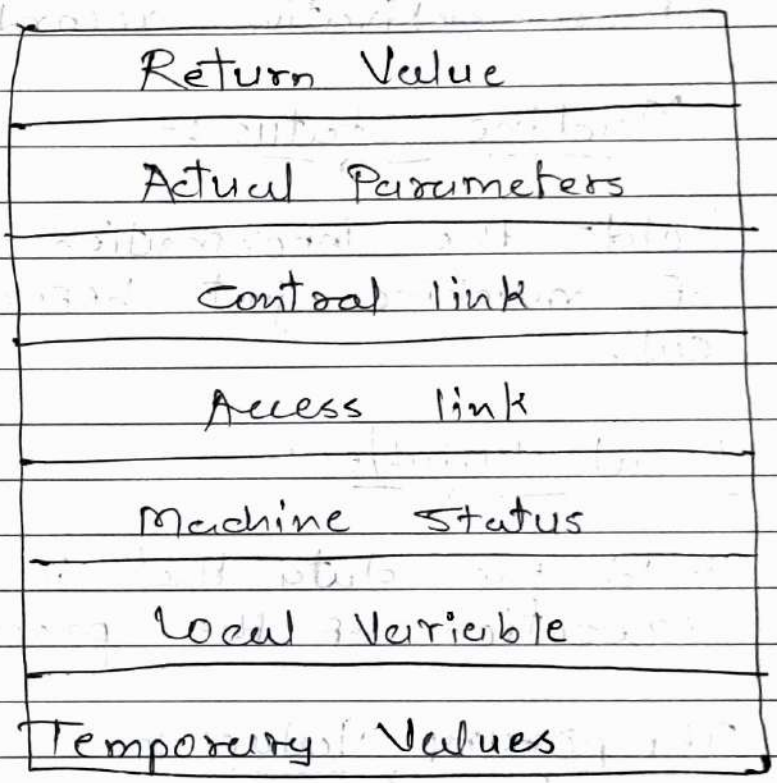
→ When we call the sum function in the example above, memory will be alloted for the variable ans. dynamically

⊛ Activation Record :-

- The execution of a procedure is called its activation.
- An Activation record contains all the necessary information required to call a procedure.
- Whenever a function call occurs, then a new activation record is created & it will be pushed onto the top of the stack.
- Activation once the procedure is completed & it is returned to the calling function, the activation func^n will be popped out of the stack.

- Activation Record includes some fields which are :-

Return Values, Actual Parameters, Control link, Access link, Machine status, Local Variables, Temporary Values.



(1) Return Value :-

- Used by the called procedure to return a value to calling procedure.

(2) Actual Parameters :-

- This field holds the information about the actual parameters.

(3) Control link :-

- points to activation record of caller.

(4) Access link :-

- refers to non-local data held in other activation records.

(5) Machine Status :-

- holds the information about status of machine just before the func^n call.

(6) Local Variables :-

- hold the data that is local to the execution of the procedure.

(7) Temporary Values :-

- stores the values that arise in the evaluation of an expression.

Q.② Differentiate static Vs Dynamic M. Allocation.

No.	Static M.A	Dynamic M.A
1.	- In SMA, Variables get allocated permanently till the program executes.	In DMA, Variables get allocated only if your program unit gets active.

No.	Static M.A	Dynamic M.A
2.	SMA is done before program execution.	DMA is done during program execution.
3.	It uses stack for managing the static allocation of memory.	It uses heap for managing the dynamic allocation of memory.
4.	It is less efficient.	It is more efficient.
5.	In SMA, there is no memory re-usability.	In DMA, there is memory re-usability & memory can be freed when not required.
6.	In SMA, once the memory is allocated, the memory size can not change.	In DMA, when the memory is allocated the memory size can be changed.
7.	In SMA, we can not reuse the unused memory.	In DMA, we can reuse the unused memory.
8.	In SMA, Execution is faster than DMA.	In DMA, execution is slower than SMA.
9.	Memory is allocated at compile time.	Memory is allocated at Run time.
10.	Ex. generally used for for array	generally used for linked list

Q.③ Symbol Table Management.

- The symbol table is defined as the set of Name and Value pairs.

- ST is an important data structure created and maintained by the compiler in order to keep track of semantics of Variables.

- it is built-in lexical & syntax analysis phases.

- it is used by the compiler to achieve compile-time efficiency.

⊛ Items to be stored into symbol table are :-

1. Variable Names
2. constants
3. procedure Names
4. Function Names
5. Literal constants & strings.
6. Compiler generated temporaries.
7. Lables in source langu

⊛ operations of Symbol Table :-

1. allocate
2. Free
3. lookup
4. insert
5. set-attribute
6. get-attribute

Q.(4) Explain Various Parameter Passing methods.

- There are two types of parameters, Formal parameters & Actual Parameters.
- And based on these parameters passing methods, the common methods are:

(1) call by Value
(2) call by reference
(3) Copy restore
(4) call by name

(1) Call by Value :-

- This is the simplest method of parameter passing.
- The call by Value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.
- The operations on Formal parameters do not change the Values of a parameter.

(2) call by Reference :-

- This method is also called as call by address or call by location.

- The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter.

- Inside the function, the address is used to access actual argument used in the call.

- It means the changes made to the parameter affect the passed argument.

---

[3] Copy Restore :-

- This method is a hybrid between call by value & call by Reference.

- This method is also known as copy-in-copy-out or values result.

- The calling procedure calculates the value of actual parameter & it then copied to activation record for the called procedure.

- During Execution of called procedure, the actual parameters value is not affected.

- If the actual parameter has L-value then at the return the value of formal parameter is copied to actual parameter.

[4] Call by Name :-

- This is less popular method of parameter passing.
- Procedure is treated like macro.
- The procedure body is substituted for call in caller with actual parameters substituted for formals.
- The local names of called procedure and names of calling procedure are distinct.
- The actual parameters can be surrounded by parenthesis to preserv then integrity.

Q.(5) What is activation Record ?

Q.(6) What are the limitations of static storage allocation ?

- Static storage Allocation is not highly scalable.
- Static storage allocation is not very efficient.
- The size of the data must be known at the compile time.

Q.① Explain Activation Record. How is task divided between calling & called program for stack updating?

⊛ Activation Record :-

Static

⊛ Storage Allocation :- Calling Sequences

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a runtime support package.

- Since the bindings do not change at run-time, every time a procedure is activated, its names are bounded to the same storage location.

- All compilers for languages that use procedures, functions or methods as units of user define actions manage at least part of their run-time memory as a stack.

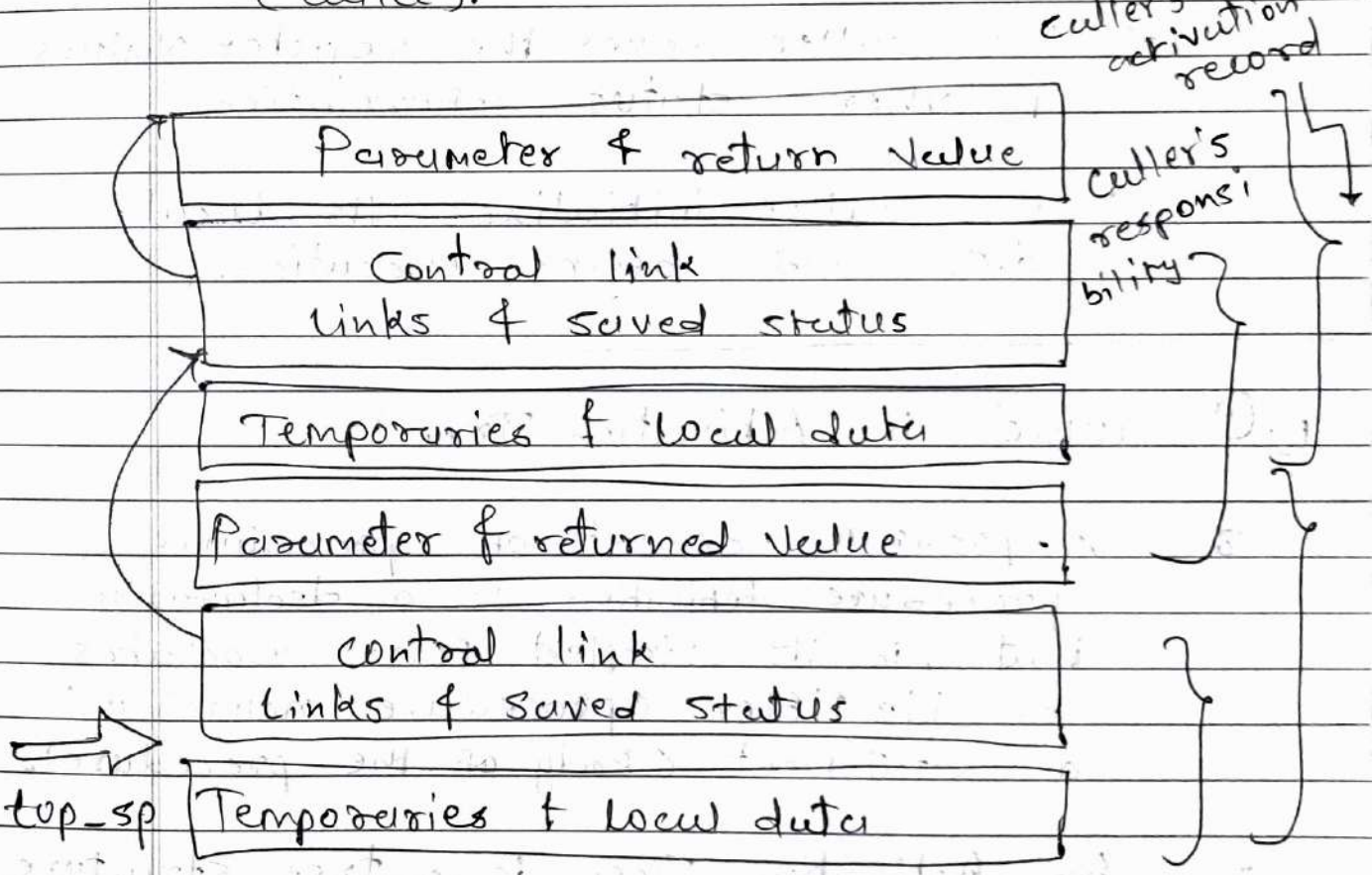
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, the space is popped off the stack.

⊛ Calling Sequences :-

- procedures are implemented by generating what are known as calls sequences in the target code.

- A call sequence allocates an activation record and enters the information into its fields. execution.

- The code is calling sequence of often divided between the calling procedure (caller) and procedure is is calls (callee).



* task dividing between calling & called program *

Page No.
Date

(8) → The calling sequence & its division between caller & callee are as follows:

(1) The caller evaluates & actual parameters.

(2) The caller stores a return address and the old value of top-sp into the callee's activation record. The caller then increments the top-sp to the respective positions.

(3) The callee saves the register values & other status information.

(4) The callee initializes its local data and begins execution.

Q. (9) What is Activation Tree?

3 - A program consists of procedures, a procedure definition is a declaration that, in its simplest form, associates an identifier (procedure name) with a statement (body of the procedure).

2. - An Activation Tree is a tree structure that represents function calls made by a program during Execution.

2. - when a function is called a new activation record is pushed to the stack

and popped from the stack when the function returns.

4. (a) properties of Activation Tree :-

(1) Each Node represents an activation of a procedure.

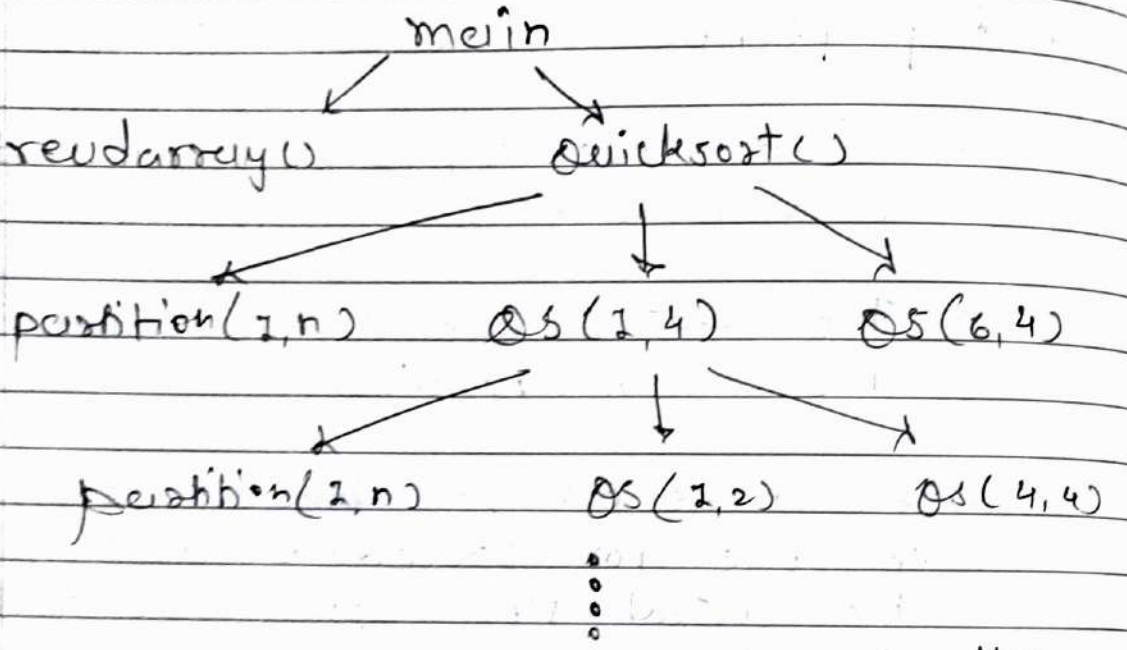
(2) The root shows the activation of the main function.

(3) The node for procedure 'x' is the parent of node for procedure 'y' if & only if the control flows procedure x to procedure y.

Ex :- Consider the following program of Quicksort.

```
main () {

    Int n;
    readarray ();
    quicksort (1, n);

}

quicksort (int m, int n) {

    Int i = partition (m, n);
    quicksort (m, i-1);
    quicksort (i+1, n);
}
```

∴ The activation tree for this program will be:



- First main function as the root then main calls readarray and quicksort.
- Quicksort in turn calls partition & Quicksort again.
- The flow of control in a program corresponds to a pre-order depth-first traversal of the activation tree which starts at the root.

Q. (3) Write diff between stack & Heap memory allocation.

No.	Stack	Heap
1.	Memory is allocated in a continuous block.	Memory is allocated in any random order

No:	Stack	Heap
2.	Allocation & De-allocation is done Automatic by compiler instructions.	Manual by the programmer.
3.	cost is less.	Cost is more.
4.	Implementation is easy.	Implementation is more.
5.	Access time is faster.	Access time is slower.
6.	Main issue is shortage of memory.	main issue is memory fragmentation
7.	Stack is fixed size	Re-sizing is possible.
8.	SMA is prefered in an array	HMA is preffered in linked list
9.	Small size than Heap memory :	Larger than stack memory.

Q. (15) Symbol Table with two data structures suitable for it.

(*) Data structures for Symbol Table :-

(1) List data structure
(2) Self organizing List
(3) Binary Tree (4) Hash Table.

[1] List Data Structure :-

- The name can be stored with help of starting index & length of each ~~time~~ name.

- Linear list is a simplest kind of mechanism to implement the s.table.

- In this method an array is used to store names & associated information.

- New Names can be added in the order as they arrive.

- The list data structure using array is given below:

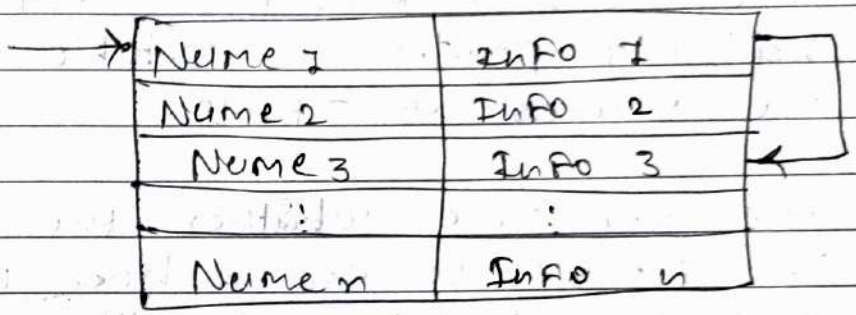
Name 1	Info 1
Name 2	Info 2
Name 3	Info 3
Name ...	~~Info~~ ...
:	:
Name n	Info n

[2] Self organizing list :-

- This symbol table implementation is using linked ~~list~~ list. A link field is added to each record.

- we search the records in the order pointed by the link of field.

- The pointer "First" is maintained to point to first record of the symbol table.



[3] Binary Tree :-

- When the organization symbol table is by means of binary tree, the node structure will as follows:

- The left child field stores the address of previous symbol.

- Right child field stores the address of next symbol.

- The symbol field is used to store the name of the symbols.

- Information field is used to give information about the symbol.

Left child	Symbols	Information	Right child

[4] Hash Table :-

- In Hashing scheme two tables are maintained - a hash table & s.table.
- The hash table consists of k entries from 0, 1 to k-1. These entries are basically pointers to s.table pointing to the names of s.table.
- The determine whether the 'Name' is in s.table. we use a hash function 'h' such that h(name) will result any integer between 0 to k-1. we can search any name by position = h(name)
- Using this position we can obtain the exact locations of name in s.table.
- Advantage of hashing is quick search is possible & the advantage is that hashing is dis complicated to implement.

⑤ Code Generation and optimation :-

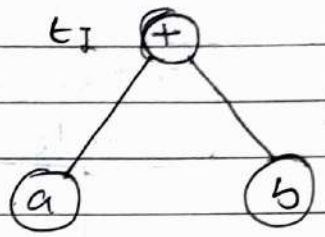
Q.① Construct a DAG for (a+b)*(a+b+c)

TAC ⇒ $t_1 = a + b$

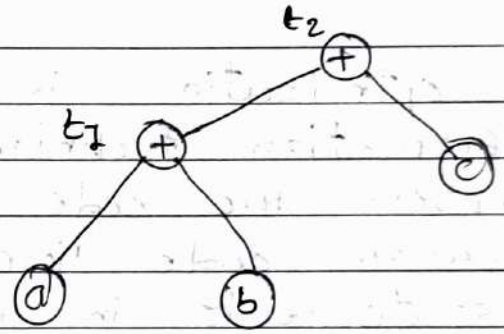
$t_2 = t_1 + c$

$t_3 = t_1 * t_2$

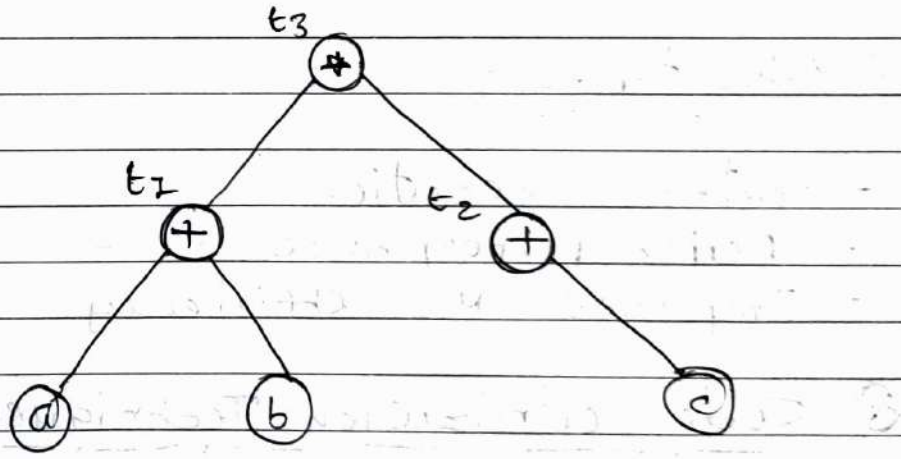
step : ①



step : ②



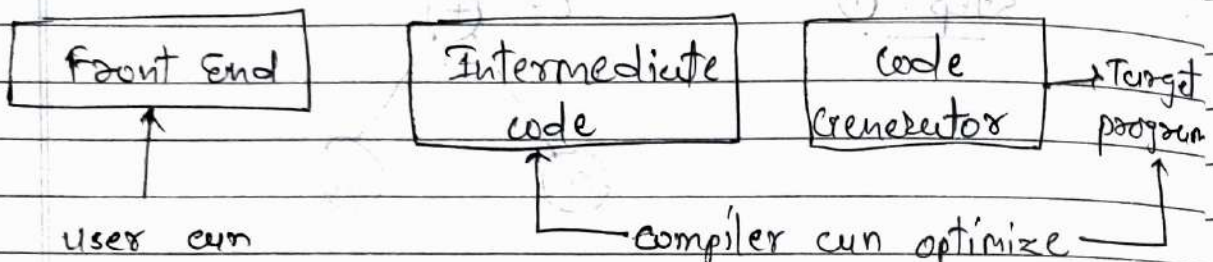
step : ③



- This is a Direct Acyclic Graph of $(a+b) * (a+b+c)$.

Q.② Explain Any Three address code optmization techniques with example.

⊛ Code optmization :-



- code optmization is a program transformation technique which, tries to improve the code by eliminating unnecessary code lines & arranging the statements in such a sequence that speed up the execution without wasting the resources.

⊛ Advantages :-

- Faster Execution
- Better Performance
- Improves the efficiency

⊛ Code optmization Techniques :-

(1) Compile time evaluation
(2) Common sub expressions elimination
(3) code movement or code motion
(4) Reduction in Strength
(5) Dead code elimination.
(6) Copy propagation

[1] Compile time Evaluation :-

- Compile time evaluation means shifting of computations from run time to compile time.

- There are two methods used to obtain the compile time evaluation.

* Folding :-

- in the folding technique the computation of constant is done at compile time instead of run time.

Ex : length = (22/7) * d

- Here folding is implied by performing the computation of 22/7 at compile time.

* constant propagation :-

- In this technique the value of variable is replaced and computation of an expression is done at compilation time.

Ex : pi = 3.14; r = 5;
Area = pi * r * r;

- Here at the compilation time the value of pi is replaced by 3.14 & r by 5 then computation of 3.14 * 5 * 5 is done during compilation.

[2] Common Sub Expressions elimination :-

- The common sub expression is an expression appearing repeatedly in the program which is computed previously.

- If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.

Ex.

$t_7 := 4 * i$
$t_2 := a + 2$
$t_3 := 4 * j$
$t_4 := 4 * i$
$t_5 := n$
$t_6 := b[t_4] + t_5$

→ eliminate this for code optimization

[3] Copy propagation :-

- Copy propagation means use of one variable instead of another.

Ex.

```
X = pi;
area = X * r * r;
```

↓

```
area = pi * r * r;
```

[4] Code Movement or Code Motion :-

- Optimization can be obtained by moving some amount of code outside the loop & placing it just before entering in the loop.

- it won't have any difference if it executes inside or outside the loop.

- This method is also called loop invariant computation.

Ex:

```
while (i<= max-i)
{
sum = sum+a[i];
}
```

Before optimization

```
N= max-1;
while (i<=N)
{
sum = sum + a[i];
}
```

After optimization.

[5] Reduction in Strength :-

- priority of certain operators is higher than others.

- For instance strength of * is higher than +.

- In this technique the higher strength operators can be replaced by lower strength operators.

Ex :

$A = A * 2$ $\qquad$ $A = A + A$

Before optimization $\qquad$ After Optimization.

---

[6] Dead code Elimination :-

- the variable is said to be dead at a point in a program if ~~we~~ the value contained into it is never been used.

- the code containing such a variable supposed to be a dead code.

Ex :

```
i = 0;
if (i == 1)
{
    a = x + 5;
}
```

} Dead code

- if statement is a dead code ie this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

Q.③ Explain various Issues in design of code generator.

- Issues in code Generation are :

(1) Input to code generator
(2) Target program
(3) Memory management
(4) Instruction selection
(5) Register allocation
(6) Choice of evaluation
(7) Approaches to code generation.

[1] Input to code generator :-

- Input to the code generator consists of the intermediate representation of the source program.

- Types of intermediate language are:

  (1) postfix notation
  (2) Three address code
  (3) Syntax trees or DAGs

- The detection of semantic error should be done before submitting the input to the code generator.

- The code generation phase requires complete error free intermediate code as an input.

[2] Target program :-

- The output may be in form of:

(1) Absolute machine language :-

- Absolute machine language program can be placed in a memory location and immediately execute.

(2) Relocatable machine language :-

- The subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution.

(3) Assembly language :-

- producing an assembly language program as output makes the process of code generation easier, then assembler is require to convert code in binary form.

(3) memory management :-

- Mapping names in the source program to address of data objects in run time memory is done cooperatively by the front end & the code generator.

- We assume that a name in a three-address statement refers to a symbol table entry for the name.

- From the symbol table information, a relative address can be determined for the name in a data area.

[4] Instruction Selection :-

- Ex : the sequence of statements

```
a := b + c
d := a + e
```

- would be translated into.

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
MOV  a, R0
ADD  e, R0
MOV  R0, d
```

```
MOV  b, R0
ADD  c, R0
ADD  e, R0
MOV  R0, d
```

- So, we can eliminate redundant statements.

[5] Register Allocation :-

- The use of registers is often subdivided into two sub problems:

- During register allocation, we select the set of variables that will reside in registers at a point in the program.

- During a subsequent register assignment phase, we pick the specific register that a variable will reside in.

- Finding an optimal assignment of registers to variables is difficult, even with single register value.

- Mathematically the problem is NP-complete.

[6] Choice of evaluation :-

= The order in which computations are performed can affect the efficiency of the target code.

- Some computation orders require fewer registers to hold intermediate results than others.

- picking a best order is another difficult, NP-complete problem.

[7] Approaches to code generation :-

- The most important criterion for a code generator is that it produces correct code.

- The design of code generator should be in such a way so it can be implemented, tested & maintained easily.

Q. (4) Explain three loop optimization techniques.

- Loop optimization is the process of increasing execution speed and reducing the overheads associated with loops.

- Loop optimization is a machine independent optimization. whereas peephole optimization is a machine dependent optimization technique.

- Decreasing the Number of instructions in an inner loop improves the running time of a program even if the amount of code outside that loop is increased.

(A) Loop Optimization Techniques :-

(1) Code motion (Frequency Reduction)
(2) Induction Variable Elimination
(3) Strength Reduction
(4) Loop invariant Method
(5) Loop Unrolling
(6) Loop Jamming (7) Loop Spliting

(1) Code Motion (Frequency Reduction):-

- In Frequency Reduction, the amount of code in the loop is decreased. ~~A statement~~

Ex

```
while (i<100)                    t = sin(x)/cos(x);
{                                while (i<100){
a = sin(x)/cos(x)+i;      ⟶        a = t+i;
  i++;                             i++;
}                                }
```

(2) Induction Variable elimination:-

- if the value of any variable in any loop gets changed every time, then such a variable is known as induction variable.

Ex

```
B1                               B1
i := i+1                         i := i+1
x := 3*i                  ⟶     x := x+4
y := a[x]                        y := a[x]
if y<15, goto B2                 if y<15, goto B2
```

(3) Strength Reduction:-

- Strength reduction deals with replacing expensive operations with cheaper ones like multiplication is costlier than addition.

Page No.
Date

```
                                    t = 3 * x + 7;
Ex.  while (x<10)                   while (x<10)
     {                      →       {
     y := 3*x+1;                    y = t;
     a[4] := a[y]-2;                a[4] := a[y]-2;
     x := x+2;                      x = x+2;
     }                              t = t+6;
                                    }
```

[4] Loop Invariant Method :-

- In the loop Invariant method, the expression with computation is avoided inseed the loop.

Ex.

```
For(int i=0; i<10; i++)             s = x*y;
{                                   For(int i=0; i<10; i++)
t = i+(x*y);            →           t = i+s;
...                                 ...
end;                                end;
}
```

[5] Loop unrolling :-

- loop unrolling is a loop transformation technique that helps to optimization the execution time of a program.

Ex.
```
For(int i=0; i<3; i++)              printf("hello");
{                                   printf("hello");
printf("hello");        →           printf("hello");
}
```

(c) Loop Jamming :-

- Loop Jamming is combining two or more loops in a single loop.

Ex

```
for (int i=0; i<5; i++)
    a = i+5;

for (int i=0; i<5; i++)
    b = i+10;
```

⟶

```
for (int i=0; i<5; i++)
{
    a = i+5;
    b = i+10;
}
```

(d) Loop Fission :-

- Loop Fission improves the locality of reference, in loop Fission a single loop is divided into multiple loops over the same index range.

Q.5 Peephole Optimization :-

- Peephole optimization is a type of code optimization performed on a small part of the code. That small part of instructions code on which optimization is performed is known as peephole or window.

- The peephole optimization is machine dependent optimization.

⊛ Objectives of peephole optimization :-

- to improve performance
- to reduce memory footprint
- to reduce code size

⊛ Peephole optimization Techniques :-

(1) Redundant load & store Elimination
(2) constant Folding
(3) Strength Reduction
(4) Null sequences / simplify Algebric Expressions
(5) Combine operations
(6) Decode elimination

---

[1] Redundant load & store elimination :-

- this eliminates redunduncies.

Ex:

```
y = x + 5;          y = x + 5;
i = y;      ⟶      w = y * 3;
z = i;
w = z * 3;
```

---

[2] constant folding :-

- the code that can be simplified by the user itself, is simplified.

Ex. Initial code: Optimized code:

```
X = 2 * 3;        X = 6;
```

---

[3] Strength Reduction :-

- The operators that consumes higher execution time are replaced by the operators consuming less execution time.

Ex. `y = *X * 2;` → `y = X + X` or `y = X << 2;`
`y = x / 2;` → `y = x >> 1;`

---

[4] Combine operations :-

- Several operations are replaced by a single equivalent operation.

Ex. `r2 := r1 x 2;`
`r3 := r2 x 1;` → `r3 := r1 + r1;`

---

[5] Null Sequences / simplify Algebric Expressions:

- Useless operations are deleted.

Ex.
```
a := a + 0;
a := a * 1;
a := a/1;
a := a - 0;
```

[5] Deadcode elimination :-

- Deadcode refers to portions of the program that are never executed or do not affect the program's observable behaviour.

Ex:

```
a = 0;
b = 2;
c = 3;

for (int i=0; i<5; i++)
{
printf ("hello");
}
```

→ here,

```
a = 0;
b = 2;
c = 3;
```

is a Deadcode.

---

(6) Directed Acyclic Graph :-

- A DAG for basic block is a directed acyclic graph with the following labels on nodes:

1. The leave of graph are labeled by unique identifier and that identifier can be variable names or constants.

2. Interior nodes of the graph is labeled by an operator symbol.

3. Nodes are also given a sequence of identifiers for labels to store the computed value.

- DAGs are type of data structure. it is used to implement transformations on basic blocks.

- DAG provides a good way to determine the common sub-expression.

⊛ Algorithm for Construction of DAG :-

Input: it contains a basic block

Output: it contains the following information:

- Each node contains a label. For leaves, the label is an identifier.

- Each node contains a list of attached identifiers to hold the computed values

Ex ==

$T_1 = a + b$

$T_2 = T_1 + c$

$T_3 = T_1 \times T_2$

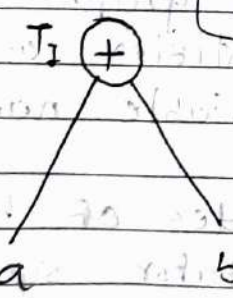
- Three cases of DAG:

case-(1) $X = Y \text{ op } Z$

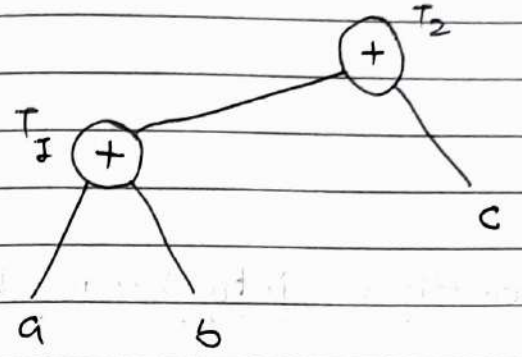
case-(2) $X = \text{op } Y$

case-(3) $X = Y$

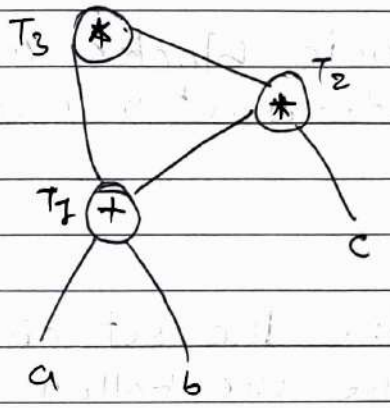
Step: ① $T_1 = a + b$



step : ② $T_2 = T_1 * c$



step : ③ $T_3 = T_1 \times T_2$



Q. ④ Basic Block and Flow Graph :-

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning & leaves at the end without halt or possibility of branching except at the end.

- The following sequence of three-address statements form a basic block:

$t_1 := a * a$
$t_2 := a * b$
$t_3 := 2 * t_2$
$t_4 := t_1 + t_3$
$t_5 := b * b$
$t_6 := t_4 + t_5$

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements.

output: A list of basic blocks with each three-address statement in exactly one block.

method:

1. We first determine the set of leaders. For that we use the following rules:
   1. The first statement is leader
   2. Any statement that is the target of a conditional or unconditional goto is leader.
   3. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader & all statements up to but not including the

next leader or the end of the program.

Ex:

```
begin
    prod := 0;
    i := 1;
    do
        prod := prod + a[t1] * b[t2];
        i := i+1;
    while i<= 20
end
```

(*) TAC :-

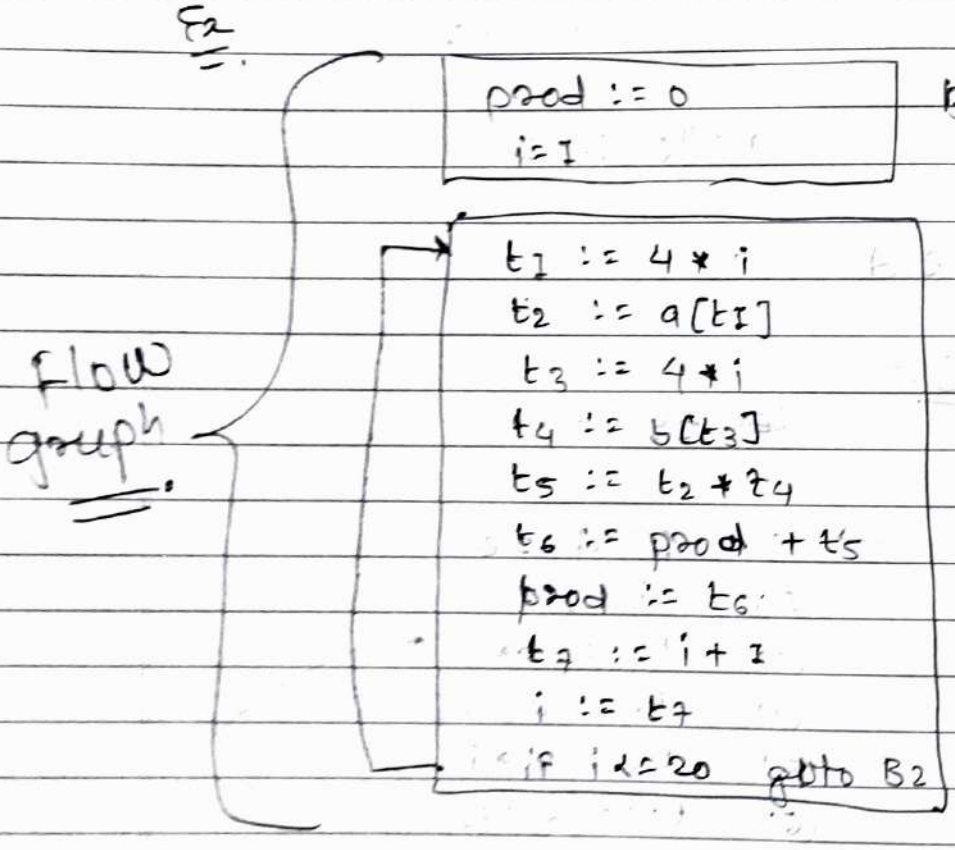
(1) prod := 0
(2) i := 1
(3) t_1 := 4 * i
(4) t_2 := a[t_1]
(5) t_3 := 4 * i
(6) t_4 := b[t_3]
(7) t_5 := t_2 * t_4
(8) t_6 := prod + t_5
(9) prod := t_6
(10) t_7 := i + 1
(11) i := t_7
(12) if i <= 20 goto (3)

④ Flow Graph :-

- We can add flow-of-control information to the set of basic blocks making up a program by constructing a direct graph called a flow Graph.

- Nodes in the flow graph represent computations, and the edges represent the flow of control.

Ex :-



5. ⑤ Transformation on Basic Blocks :-

- A Number of transformations can be applied to basic block without changing the set of expressions computed

by the block.

- many of these transformations are useful for improving the quality of the code.

- Types of transformations are:

(1) Structure Preserving Transformation
(2) Algebric Transformation

(*) Structure Preserving Transformation :-

(1) common sub-expression elimination
(2) Dead-code elimination
(3) Renaming of temporary variables
(4) Interchange of two ~~1st~~ independent adjacent statements.

(*) Algebric Transformation :-

(1) simplify expression
(2) Replacing expensive operations with cheaper.

Q.9 Explain Simple code Generator & Code Generation Algorithm.

- the code generation strategy generates target code for a sequence of three address statement.

- it uses function getReg() to assign register to variable

- the code generator algorithm uses descriptors to keep track of register contents & address for names.

- Address descriptor stores the location where the current value of the name can be found at run time.

  - the information about locations can be stored in the symbol table & is used to access the variables.

- Register descriptor is used to keep track of what is currently in each register.

  - As the generation for the block progresses the registers will hold the values of computation.

④ Code Generation Algorithm :-

- the algorithm take a sequence of three-address statements as input.

  - For each three-address statement of the form x := y op z perform the various actions.

  - Assume L is the location where the output of operation y op z is stored.

1. Invoke a function getReg() to find out the location L where the result of computation y op z should be stored.

2. Determine the present location of 'y' by consulting address description for y if y is not present in location L then generate the instruction MOV y', L to place a copy of y in L.

3. present location of z is determined using step 2' & the instruction is generated as OP z', L.

4. if L is register then update it's descriptor that it contains value of x. update the address descriptor of x to indicate that it is in L.

5. if the current value of y or z have no next uses or not live on exit from the block or in register then after alter the register descriptor to indicate that after execution of x := y op z those register will no longer contain y or z.

(*) Generating code for Assignment Statement

- the assignment statement

d := (a-b) + (a-c) + (a-c) can be translated into following TAC:

Statement	Code Generated	Register Descriptor	Address descriptor
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 & memory

* * *

Statement	Code Generated	Register Descriptor	Address descriptor Register descriptor
$t_1 := a - b$	MOV a, R0 SUB b, R0	R0 contains t_1	t_1 in R0
$t_2 := a - c$	MOV a, R1 SUB c, R1	R0 contains t_1 R1 contains t_2	t_1 in R0 t_2 in R1
$t_3 := t_1 + t_2$	ADD R1, R0	R0 contains t_3 R1 contains t_2	t_2 in R1 t_3 in R0
$t_4 := t_3 + t_2$	ADD R1, R0 MOV R0, t_4	R0 contains t_4	t_4 in R0 t_4 in R0 & memory

⊛ Chapter: ⑧ : Instruction-Level Parallelism ⊛

Q. ① Code scheduling constraints :-

- Code scheduling is a form of program optimization that applies to the machine code that is produced by code generator.

- Code scheduling subject to three kinds of constraints :

(1) Control dependence constraints :-

- All the operations executed in the original program must be executed in the optimized one.

(2) Data Dependence constraints :-

- The operations in the optimized program must produce the same result as the corresponding ones in the original program.

(3) Resource constraints :-

- The schedule must not oversubscribe the resources one the machine.

- The scheduling constraints guarantee that the optimized program produces the same result as the original.

Q.② Basic-Block Scheduling :-

- There are three types of Basic-Block Scheduling:

(1) Data-Dependence Graphs
(2) List scheduling of Basic Blocks
(3) Prioritized Topological Orders

(1) Data-Dependence Graphs :-

- We represent each basic block of machine instructions by a data-dependence graph, $G = (N, E)$, having a set of nodes N representing the operations in the machine instructions in the block & a set of directed edges in E representing the data-dependence constraints among the operations.

- The nodes & edges of G are constructed as follows:

(1) Each operation n in N has a resource-reservation table RT_n, whose value is simply the resource-reservation table associated with the operation type of n.

(2) Each edge e in E is labeled with delay de indicating that the destination node must be issued no either earlier

Page No.
Date

then de clocks after the source node is issued.

- Suppose

(2) List scheduling of Basic Blocks :-

- the simplest approach to scheduling basic blocks involves visiting each node of the data-dependence graph in "prioritized topological order".

- since there can be no cycles in a data-dependence graph, there is always at least one topological order for the nodes.

- List scheduling does not backtrack; it schedules each node once & only once.

[3] Prioritized Topological Orders :-

- it uses a heuristic priority function to choose among the nodes that are ready to be ~~shed~~ scheduled next.

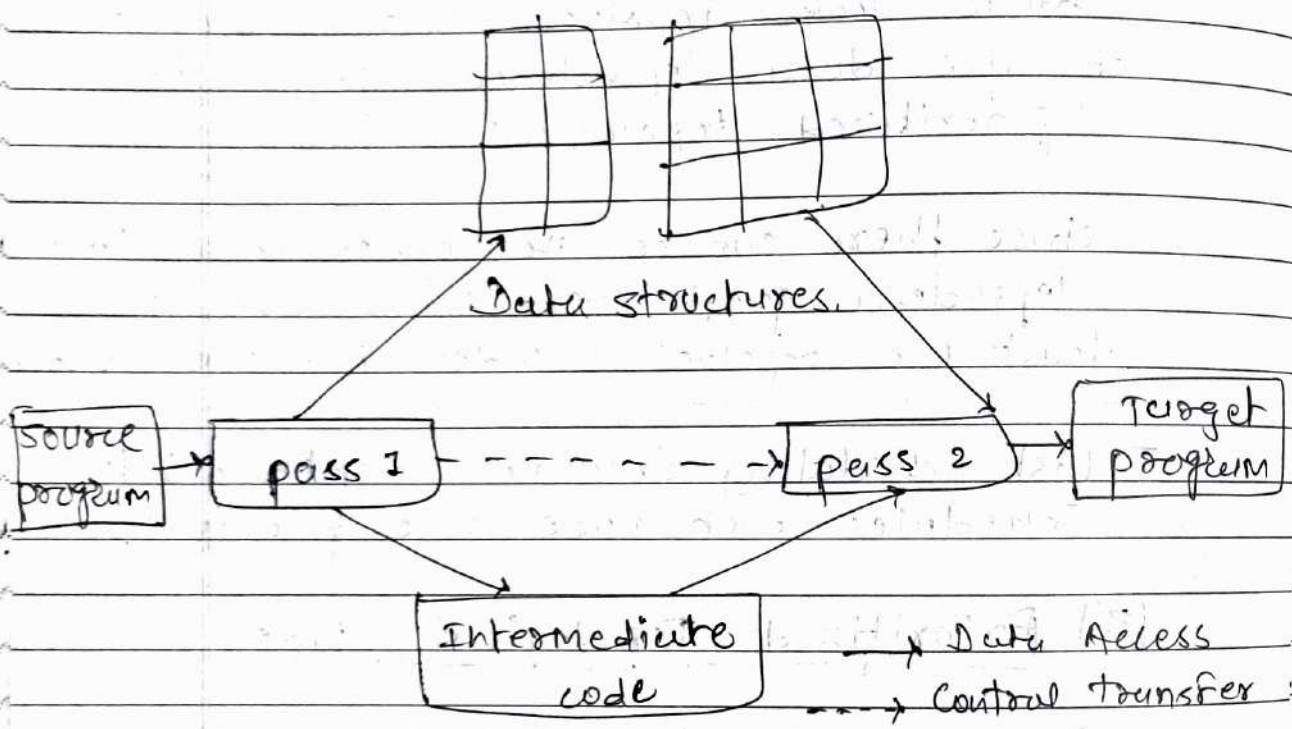
Q.(3) Pass Structure of Assembler :-

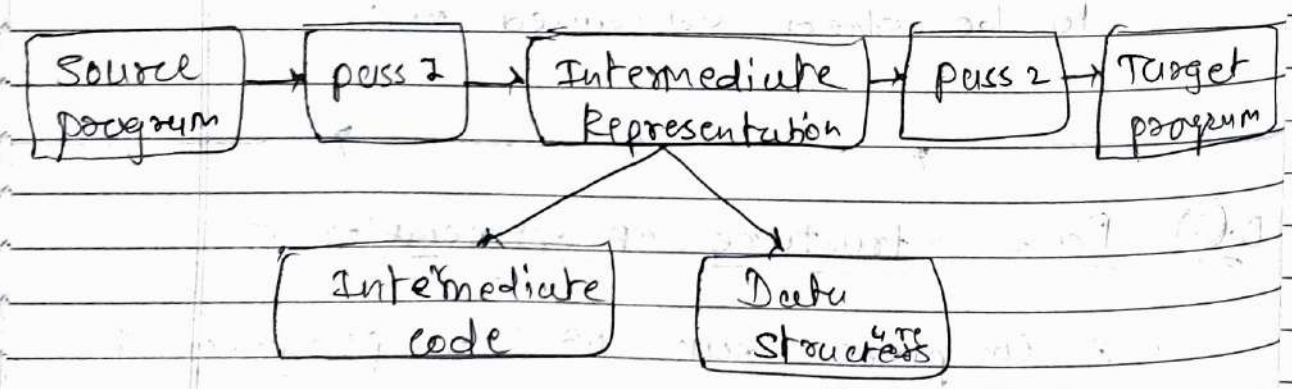
- A complete scan of the program is called pass.

- Types of assembler :

(1) Two pass Assembler (Two pass Translation)
(2) Single pass Assembler (single pass Translation)

(1) Two Pass Assembler :-





- The first pass performs analysis of the source program.

- The first pass performs location Counter processing and records the Address of Symbols in the Symbol Table.

- it constructs intermediate representation of the source program.

- intermediate representation consist of Following two components:

  (1) Intermediate Code
  (2) Data structures

- The second pass synthesizes the target program by using address information recorded in the symbol table.

- Two pass translation handles a forward reference to a symbol naturally because the address of each symbol would be known before program synthesis begins.

→ Use of a Symbol that precedes its definition in a program.

---

(2) One pass Assembler :-

- A one pass Assembler requires one scan of the source program to generate machine code.

- Location Counter processing, Symbol table construction & target code generation proceed in single pass.

- The issue of forward references can be solved using a process called back patching.

✱ Chapter : 3 : Some Questions ✱

Attributes.

① Differentiate Synthesized & Inherited

No	Synthesized Attributed	Inherited Attributes
(1.)	An Attribute is said to be synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.	An Attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node
(2.)	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.

(3)	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, & n's siblings
(4)	it can be evaluated during a single bottom-up traversal of parse-tree.	it can be evaluated during a single top-down & sideways traversal of parse tree.
(5)	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited Attributes can't be contained by both, it is only contained by non-terminals.
(6)	Synthesized Attribute is used by both S-attributed SDT & L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
(7)	Ex. E.Val ⇏ F.Val E.Val ↑ F.Val	Ex. E.Val = F.Val E.Val ↓ F.Val

Q.② Differentiate S-attributed & L-attributed SDT.

No.	S-attributed SDT	L-attributed SDT
(1)	if SDT uses only Synthesized attributes, it is called as S-attributed SDT.	if SDT uses both Synthesized & Inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
(2)	S-attributed SDT are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.	Attributes in L-attributed SDT are evaluated by depth-first & left-to-right parsing manner.
(3)	Semantic actions are places in right most place of RHS.	Semantic actions are placed any where in RHS.
(4)	Ex. $A \rightarrow BC \{ A.a = B.a, C.a \}$	Ex. $S \rightarrow ABC$

Page No.
Data

Q (3) Explain Handle & Handle Pruning :-

(*) Handle :-

- The Handle is the substring that matches the body of a production whose reduction represents one step along with the reverse of a Rightmost derivation.

- The Handle of the right sequential form Y is the production of Y where the string S may be found & placed by A to produce the previous right sequential form in RMD (Right most Derivation) of Y.

- Sentential $S \Rightarrow a$ here 'a' is called Sentential form : form 'a' can be mix of terminals & non-terminals.

Ex $S \rightarrow aSa | bSb | \varepsilon$

Derivation :

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbSbba \Rightarrow abbbbba$

(*) Left-Sentential & Right-sentential form (*)

- A left-Sentential form is a sentential form that occurs in the left most

derivation of some sentence.

- A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

(*) Handle contains two things:

(1) Production
(2) Position.

(*) Handle Pruning :-

- Removing the children of the left-hand side non-terminal from the parse-tree is called Handle Pruning.

- A rightmost derivation is reverse can be obtained by handle pruning.

(*) Steps to Follow :-

- Start with a string of terminals 'w' that is to be parsed.

- Let $w = Y_n$, where Y_n is the n^{th} right sequential form of unknown RMD.

- To reconstruct the RMD in reverse, locate handle B_n in Y_n. Replace B_n with LHS of some $A_n \to B_n$ to get $(n-1)^{th}$ RSF Y_{n-1}. Repeat.