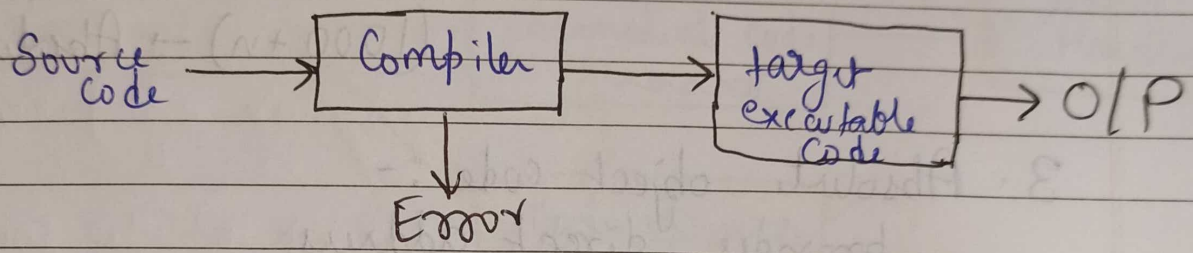
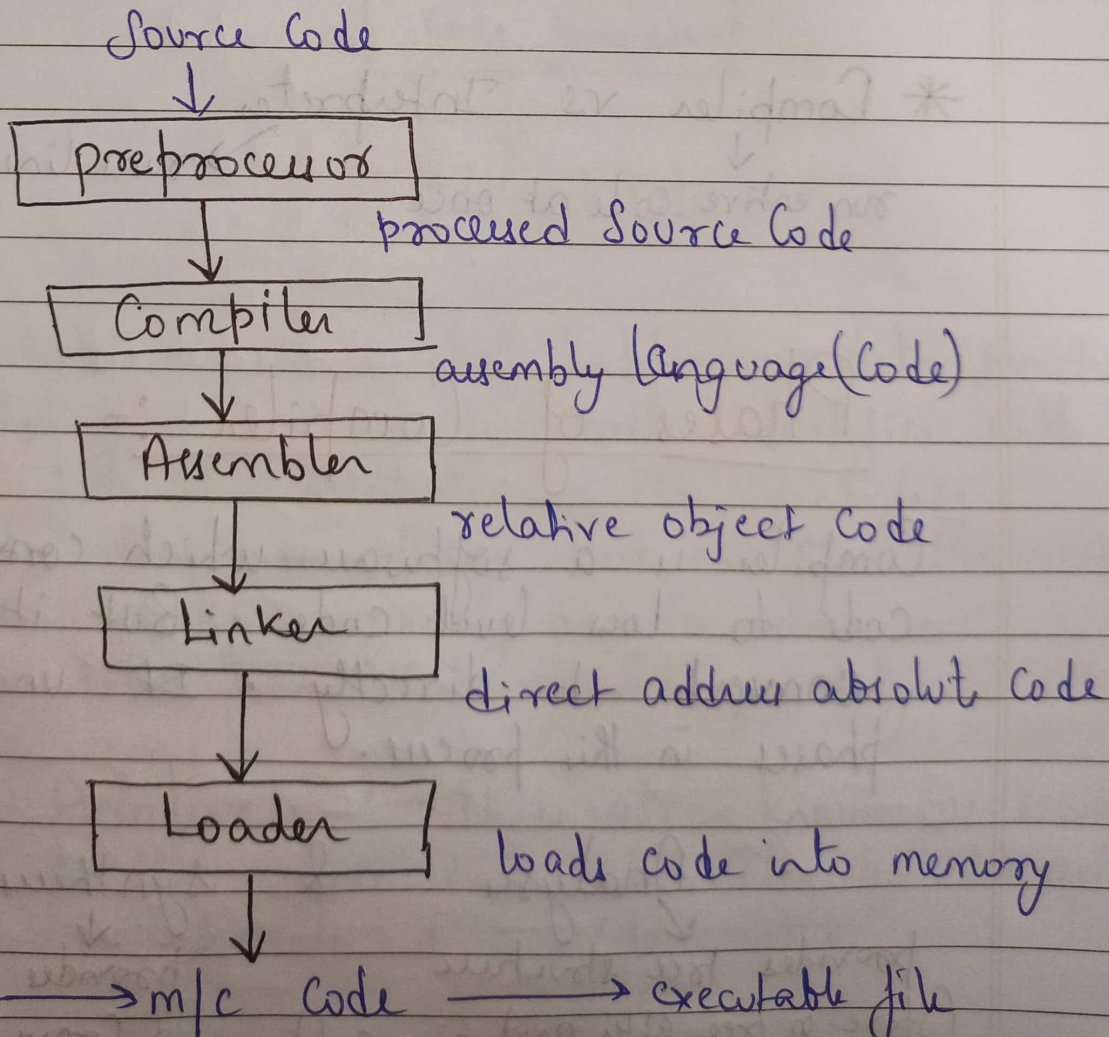
Compiler Design

Unit-1

Compiler :- Compiler is a execution environment that converts the source code to a target code (i.e high level language to machine level language).

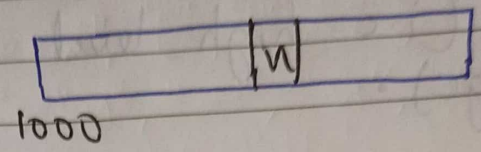


*



1. Assembly Code → in between high & low level language

2. Relocatable object code → provide relative address.



n → Relative address

(1000+n) → Absolute address

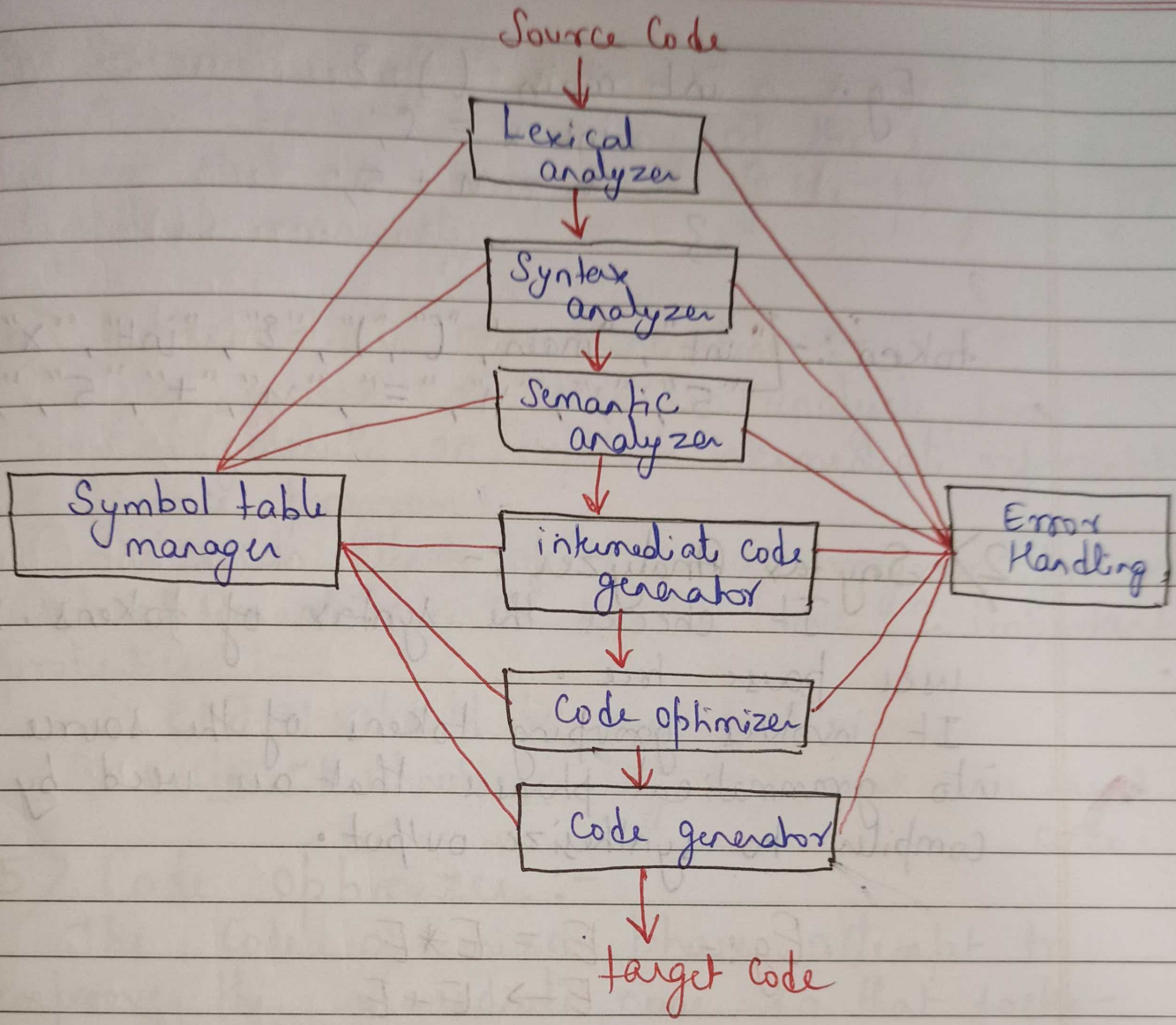
3. Absolute object code :-
provides direct address

* Compiler vs Interpreter
  - Compiler → run entire code at once
  - Interpreter → runs line by line.

Phases of Compiler :-

Compiler is a software which convert high level code to low level code. But it does not convert it directly. It undergoes certain phases in this process.

Analysis & synthesis phase.
- Analysis → provides tree structure (Store in tree after analyzing code)
- synthesis → provides m/c code (Convert tree structure to machine code)



Symbol table :- A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

* Lexical Analyzer :- It is also known as scanner. It reads the entire code character by character and convert it into tokens (lexeme).

Eg.
```
int main () {
    int x = 5;
    n = n + 5;
}
```

token :- ["int", "main", "(", ")", "{", "int", "x", "=", "5", ";", "n", "=", "n", "+", 5, ";", "}"]

2) Syntax Analyzer :-

It checks the syntax of tokens. It uses parse tree.

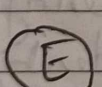
It involves grouping tokens of the source program into grammatical phrases that are used by compiler to synthesize output.

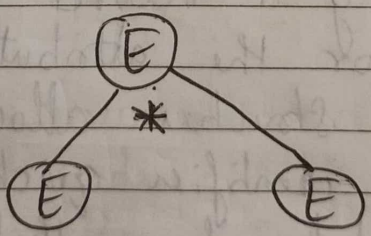
Eg:- E = E*E

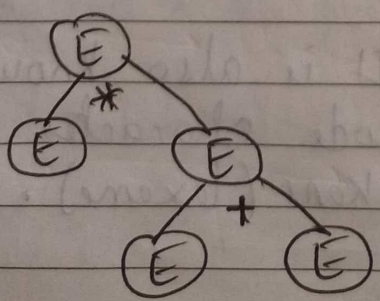
E → E+E

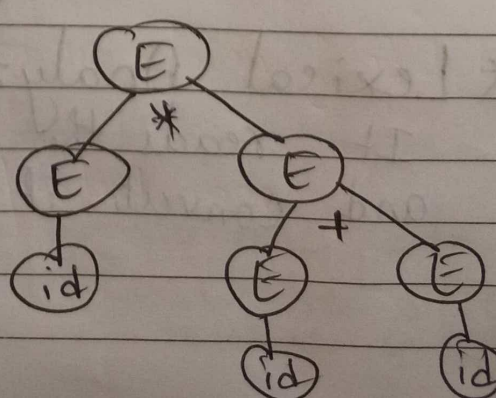
E → id

Generate id * id + id.

1. 

2. 

3. 

4. 

3) Semantic analyzer :-
It checks the meaning of the syntax to ensure that the component of the program fit together manually.

4) Intermediate code Generation :-
After syntax and semantic analysis, some compiler generate an explicit intermediate representation of source program.
We can think of it as a program for an abstract machine. It should have 2 important properties :-
(i) It should be easy to produce
(ii) It should be easy to convert into target code.

5) Code optimizer :-
The code optimizer phase attempt to improve the intermediate code, so that faster-running machine code will result.
It removes the useless symbols without changing the meaning.

6) Code Generation :- The final phase of the compiler is the generation of target code which is machine dependent. It is unique for all the core system (i.e it is different for 32 bit & 64 bit and so on).

Errors in various Analysis phase :-

1) Lexical Analyzer :- Misspelling
Eg. main()

2) Syntax Analyzer :- Not in proper Syntax
Eg. int x = 5
printf("Hello)";

3) Semantics :- Syntax is valid or not (providing meaning or not)
Eg.
int x = 5.5;
char ch = 104.5;

In Short Summary :-

machine independent code
↓
① → info about tokens
↓
② → data types, scope
↓
③ → verify already stored information
↓
④ → Generate intermediate code based on info stored in Symbol table
↓
⑤ → optimize code based on info in Symbol table
↓
⑥ → machine dependent code

* front-end phase :- Machine independent
(analysis) Depend on programming language

Back-end phase :- Machine dependent
(synthesis) Doesn't Depend on programming language

* Passes in Compiler :-

1) One-pass :- Single pass me convert kar deta h.
→ faster but more space

2) Two-pass :- first convert to intermediate then in second pass convert to m/c code.
→ Slower but less space.

Cousin's of Compiler :-

1) Preprocessor :- It produce input to compiler. It may perform following function :-

(a) Macro-processing :- #define PI 3.14
(b) File inclusion :- #include <stdio.h>
(c) Rational preprocessor :- It augments the older language to the new language by the use of built in macros.
(d) Language extensions :-

2) Assemblers :- Some compilers produce assembly code that is passed to assembler which converts the assembly language to Machine level code.

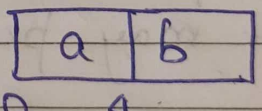
Assembly code is a mnemonic versions of machine code, in which names are used instead of binary codes for operations and names is also given to memory address.

Eg :- MOV a, B1
ADD #2, B1
MOV B1, b

3) Two Pass - Assembly :-

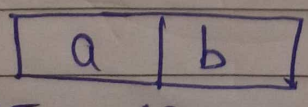
In first pass, all the identifiers that denote storage location are found and stored in symbol table. Basically it checks the code & assign the memory to variable according to requirements.

a 0
b 4



← Assembly Relative addrs.

In Second pass, the assembler scans the input again. This time it translate each operation code into the sequence of bit representing that operation in machine language. The output of this pass is usually relocatable machine code.



← Actual Absolute addrs

```
MOV a, R1          0001 01 00 'OOOOOOOO*
ADD #2, R1  ---->  0011 01 10  00000010
MOV R1, b          0010 01 00  00000100*
```

4) Loader & Link-Editors :-

Usually a program called loader performs the two functions of loading and link-editing. The process of loading consist of taking relocatable machine code, altering the relocatable address and placing the altered instructions and data in memory at proper locations.

The link-editor allows us to make a single program from several files ~~from~~ of relocatable machine code.

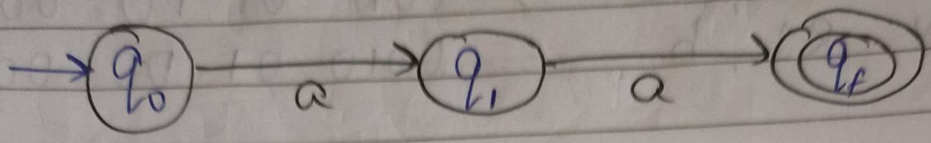
Automata Theory :-

DFA :- 5 tuples :-

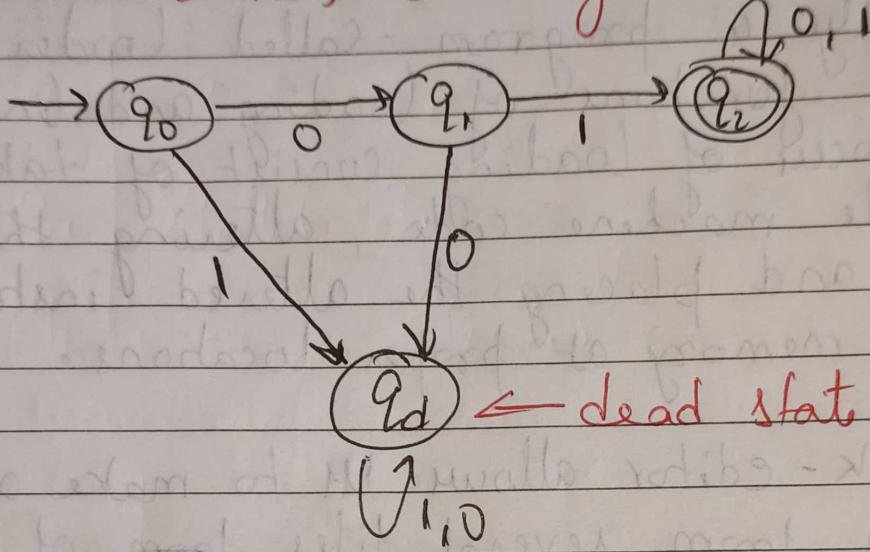
- Q → set of state
- Σ → set of input symbols
- δ → transition functions
- q_0 → Initial state
- F → Final state

$$Q \times \Sigma \rightarrow Q$$

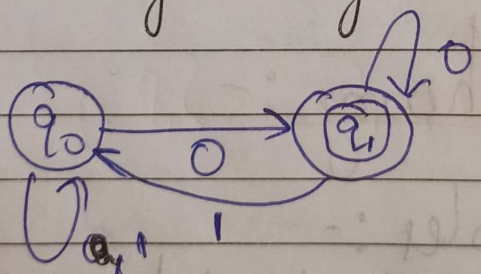
Q aa



Q $\Sigma = \{0,1\}$ starting with 01.

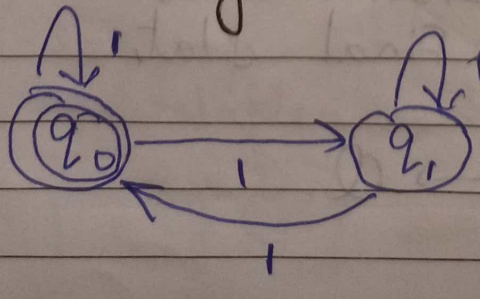


Q All string ending with 0.

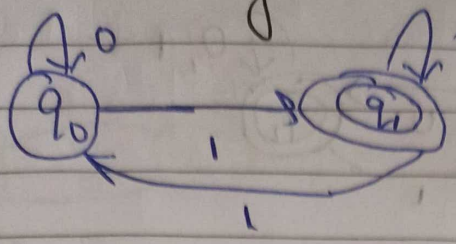


	0	1
→ q_0	q_1	q_0
q_1	q_1	q_0

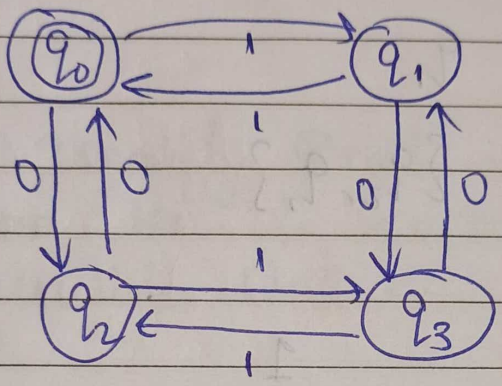
Q Even No. of 1's :-



Q. Odd no. of 1's :-



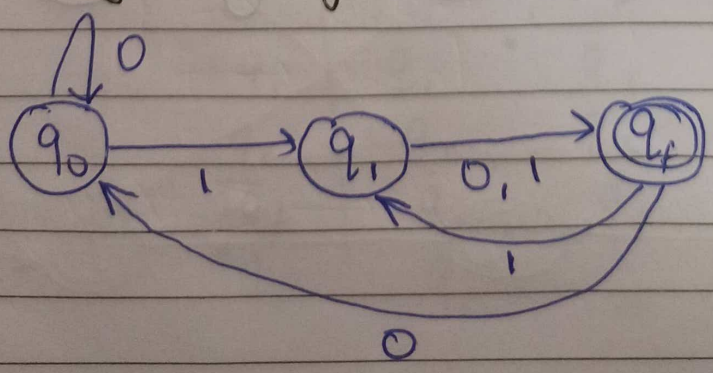
Q Even no. of 1's & even no. of 0's :-



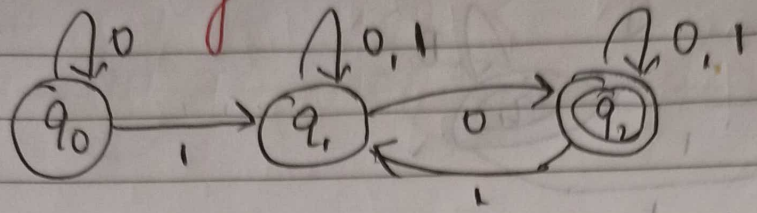
Practice Problems :-

1) Even no. of 0's or even No. of 1's.
2) odd no. of 1's and even no. of 0's.
3) odd no. of 1's or even no of 0.

Q. DFA for string with 2nd last bit as 1.

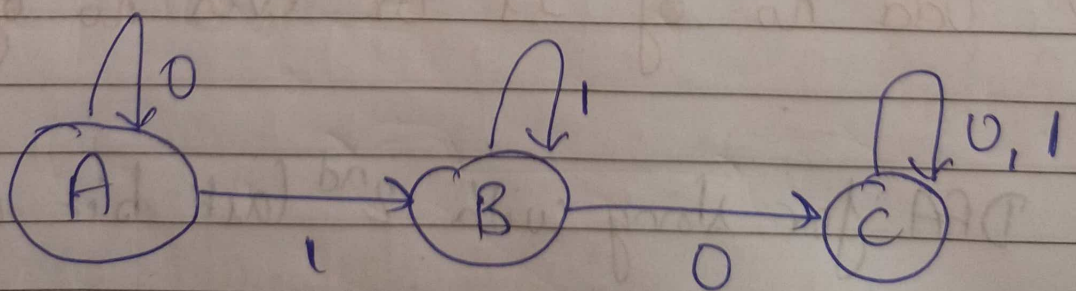


Converting NFA to DFA :-

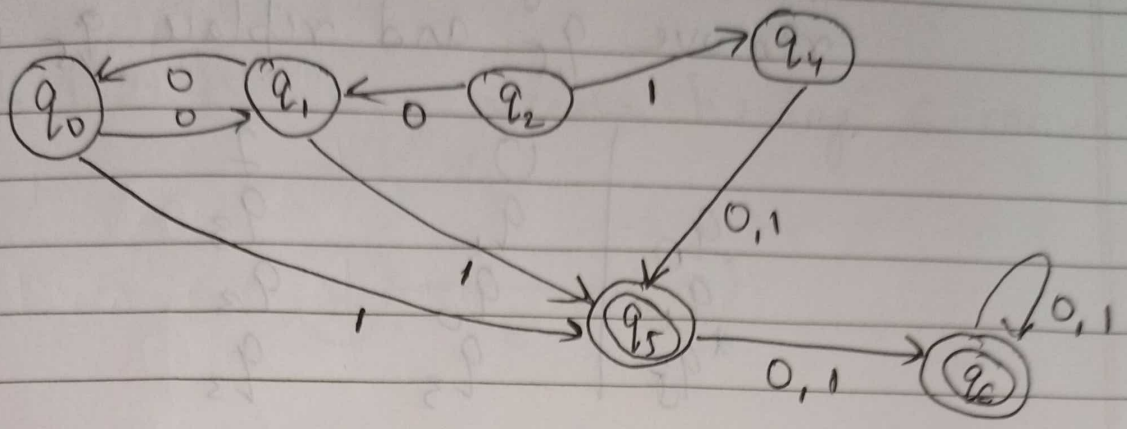


	0	1
q_0	q_0	q_1
q_1	$\{q_1, q_2\}$	q_1
q_2	q_2	$\{q_2, q_1\}$

		0	1
(A)	q_0	q_0	q_1
(B)	q_1	$\{q_1 q_2\}$	q_1
(C)	$q_1 q_2$	$\{q_1 q_2\}$	$\{q_1 q_2\}$



Minimization of DFA :-



Step 1 → Identify Dead state & Inaccesible state
Here, there is no dead state but there is 2 inaccesible state.

Step 2 :- Draw Transition table will all the states except dead & inaccesible one.

	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

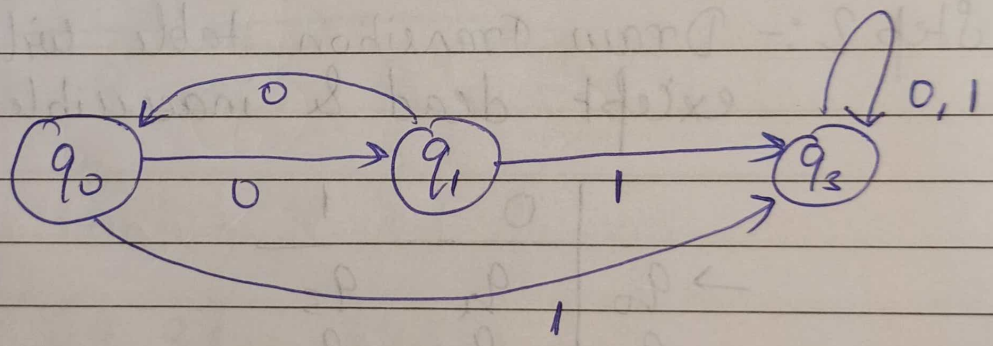
Step 3 :- Creat partition between final stat and all other state.

	0	1
→ q_0	q_1	q_3
q_1	q_0	q_3
* q_3	q_5	q_5
* q_5	q_5	q_5

Step 4 :- Since q_3 and q_4 both have same transition for 0 and 1. So, we can remove q_5 and replace q_5 with q_3.

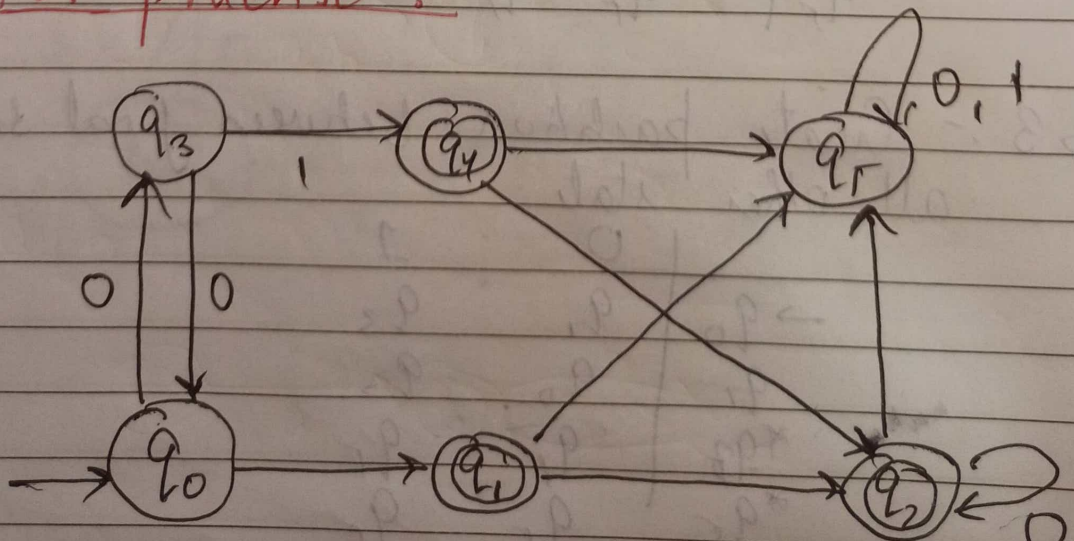
	0	1
q_0	q_1	q_3
q_1	q_0	q_3
*q_3	q_3	q_3

Step 5 :- Now, since there is no other pair of states with same transition, so, we stop and draw DFA for current transition table.



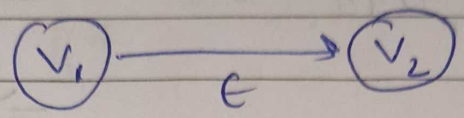
For practise :-

Q.



Converting ϵ-NFA to NFA :-

Step 1 :- Identify vertex having ϵ-move as V_1 and V_2.

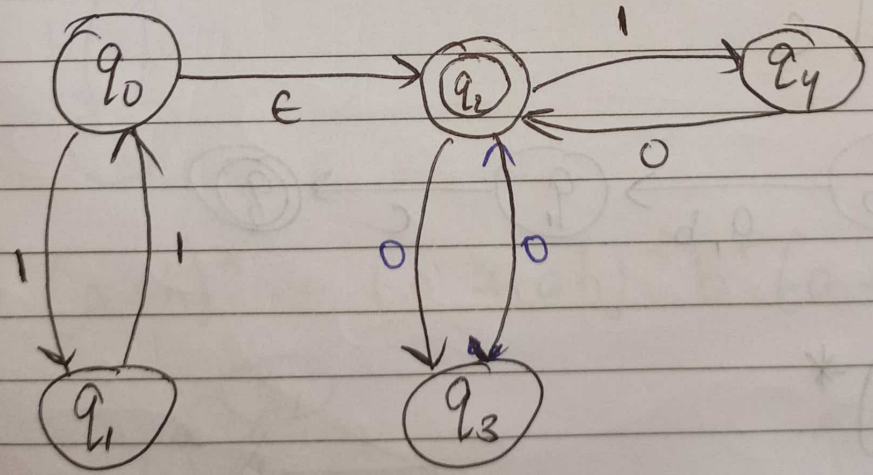


Step 2 :- Copy all the moves of V_2 ~~and~~ to V_1. Delete ϵ-edge.

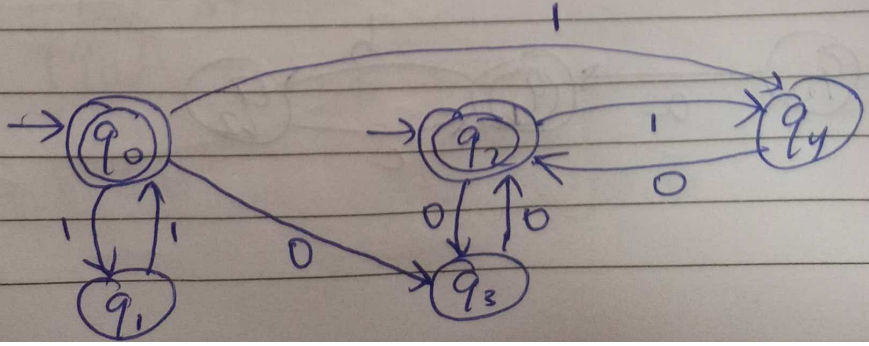
Step 3 :- If V_1 is initial stat make V_2 initial stat

Step 4 :- If V_2 is final stat make V_1 final stat.

Q.



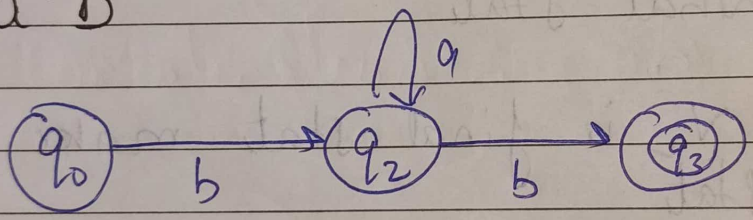
⇒



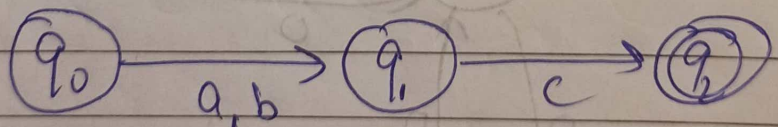
Regular Expression :-

1. $L = \{\epsilon, a, aa, \ldots\}$ $\Rightarrow a^*$
2. $L = \{a, aa, aaa, \ldots\}$ $\Rightarrow a^+$
3. $L = \{\}$ $\Rightarrow \phi$
4. $L = \{\epsilon\}$ $\Rightarrow \epsilon$

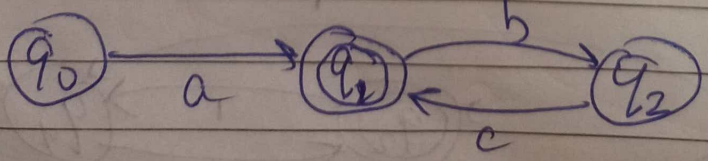
Q. ba^*b



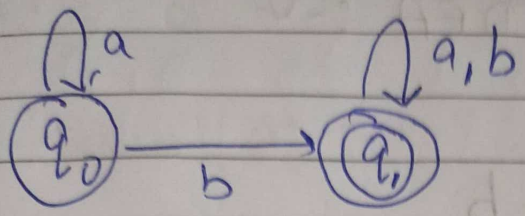
Q. $(a+b)c$



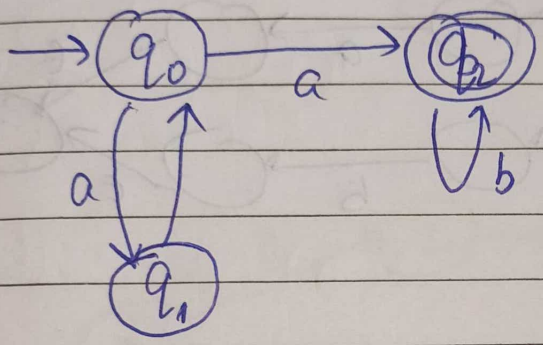
Q. $a(bc)^*$



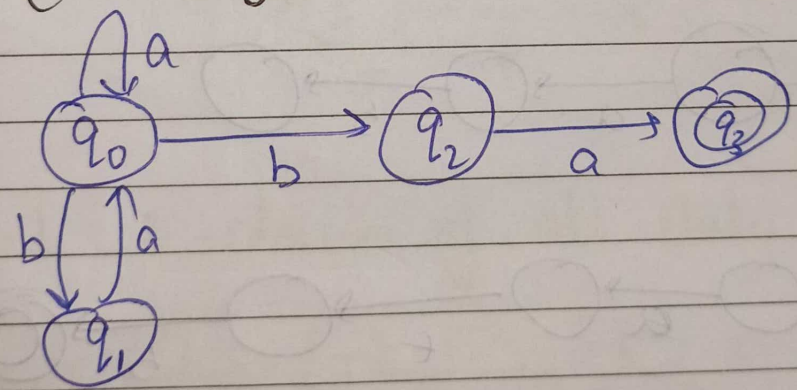
Q. $a^* b (a+b)^*$



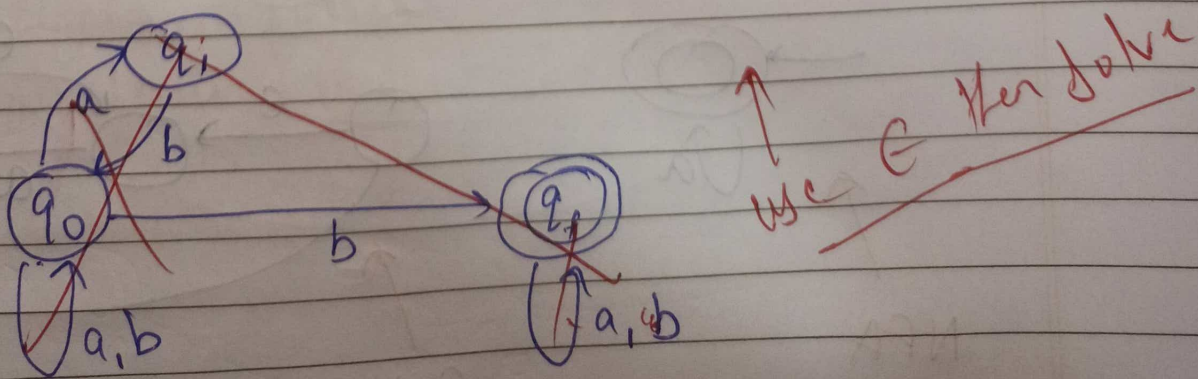
Q. $(ab)^* a b^*$



Q. $(a + ba)^* ba$

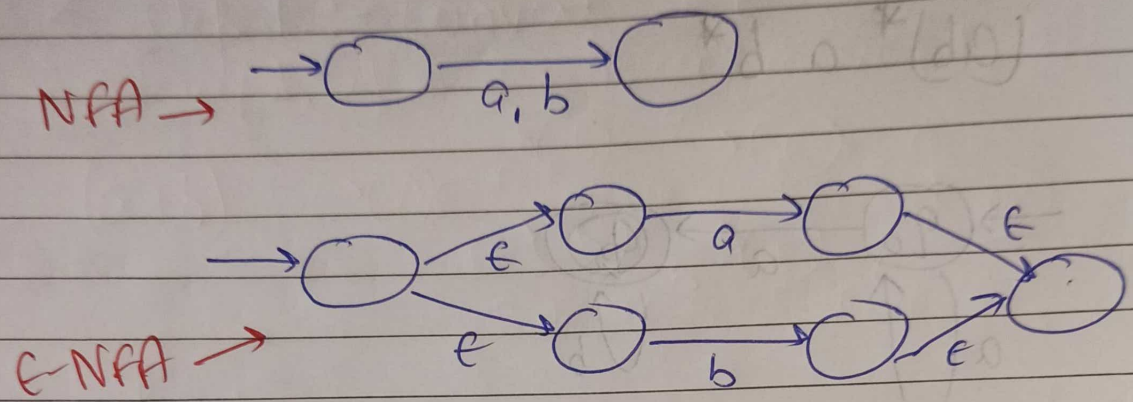


Q. $(a+b)^* + (a + ab)^* b^+ (a+b)^*$

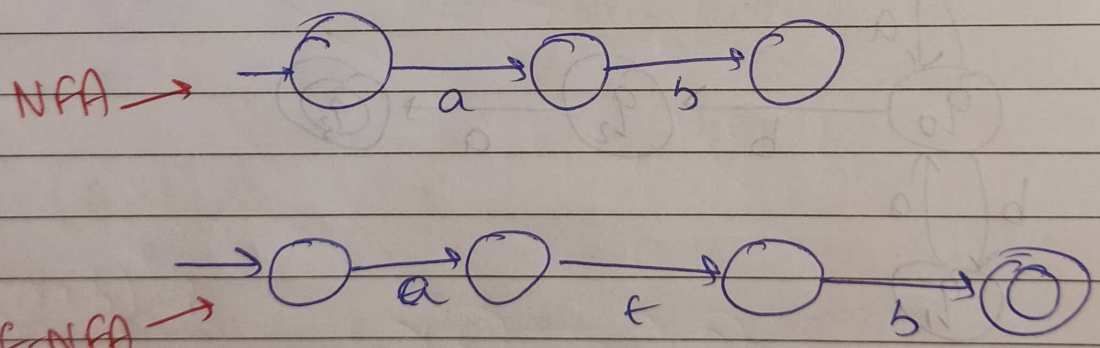


Thompson's Algorithm for Converting RE to DFA :-

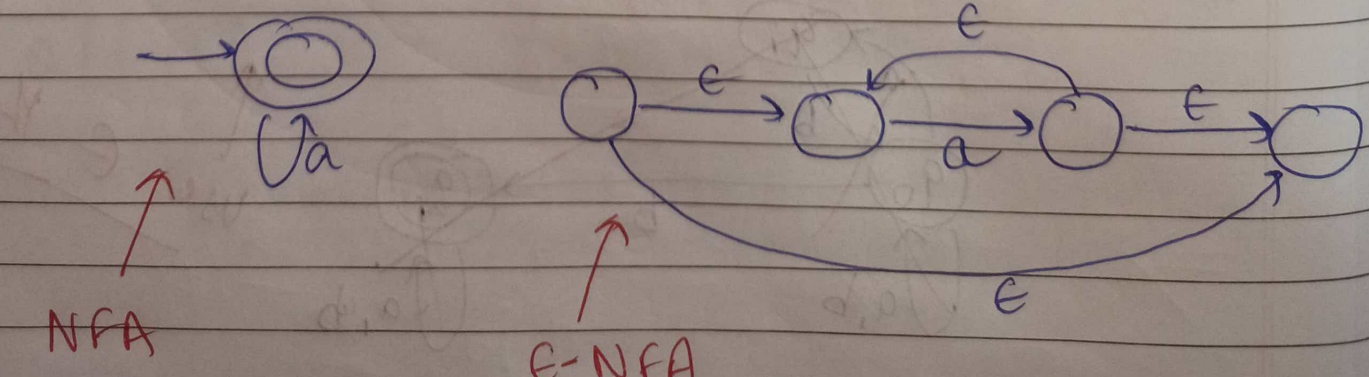
1. Union :- a + b

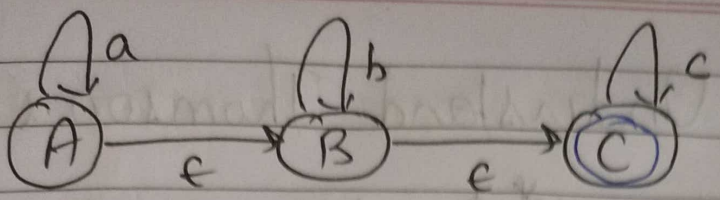


2. Concatenation :- ab



3. Closure :- a*



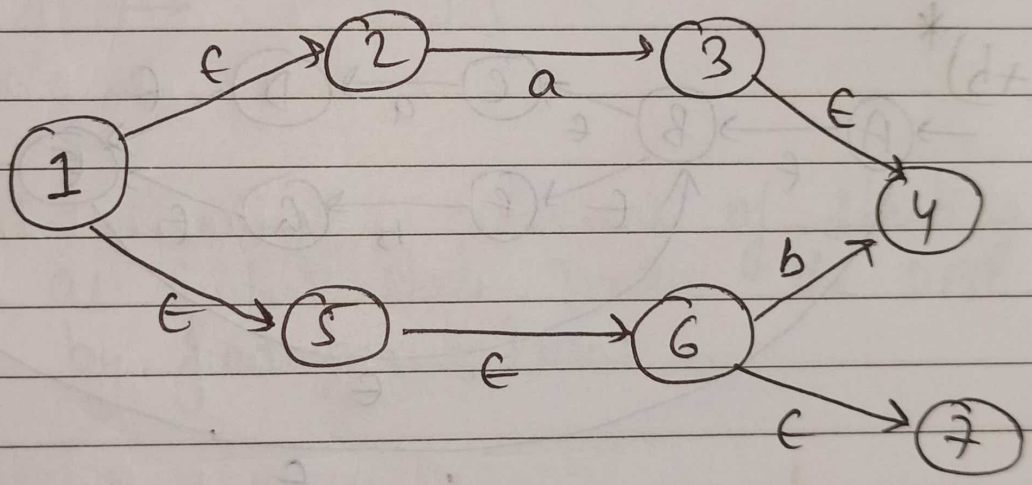


ϵ-closure (A) :- {A, B, C}

ϵ-closure (B) :- {B, C}

ϵ-closure (C) :- {C}

Q.



find ϵ-closure of all state

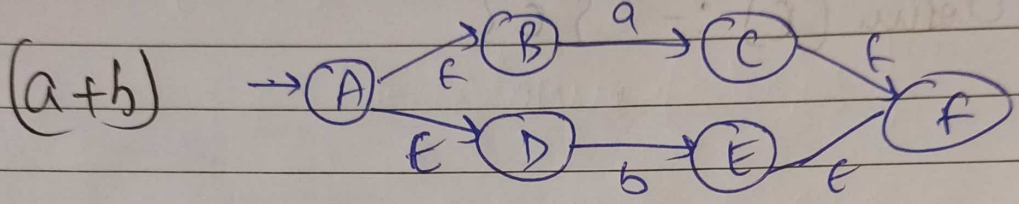
⇒ ϵ-closure (1) :- {1, 2, 5, 6, 7}
ϵ-closure (2) :- {2}
ϵ-closure (3) :- {3, 4}
ϵ-closure (4) :- {4}
ϵ-closure (5) :- {5, 6, 7}
ϵ-closure (6) :- {6, 7}
ϵ-closure (7) :- {7}

Problem to Understand Thomson's Method :-

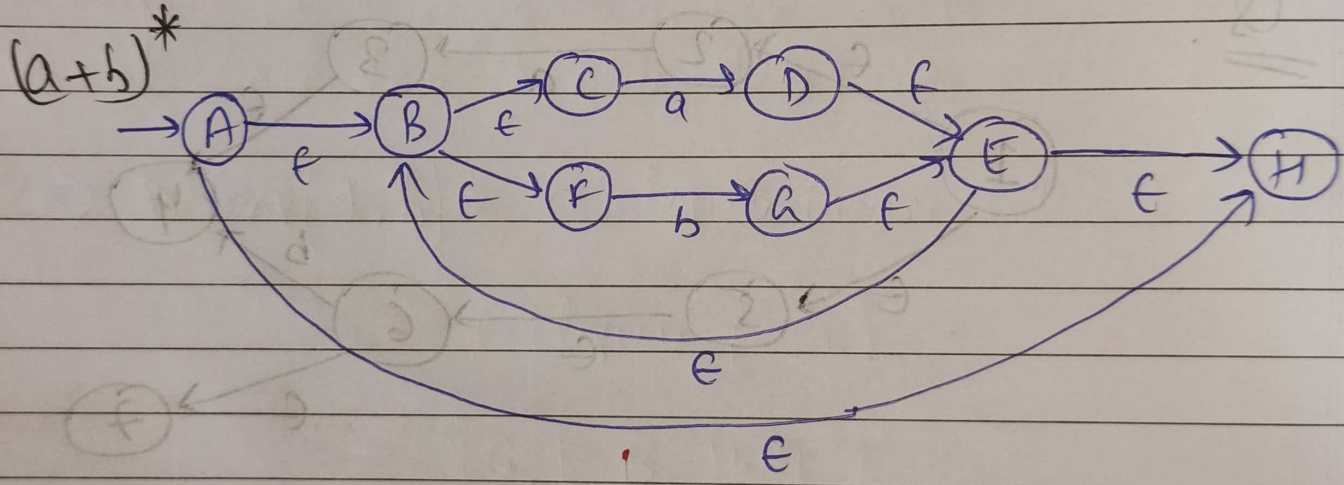
Q Convert $(a+b)^*a$ to DFA.

⇒ Step 1 :- Convert RE to ϵ-NFA

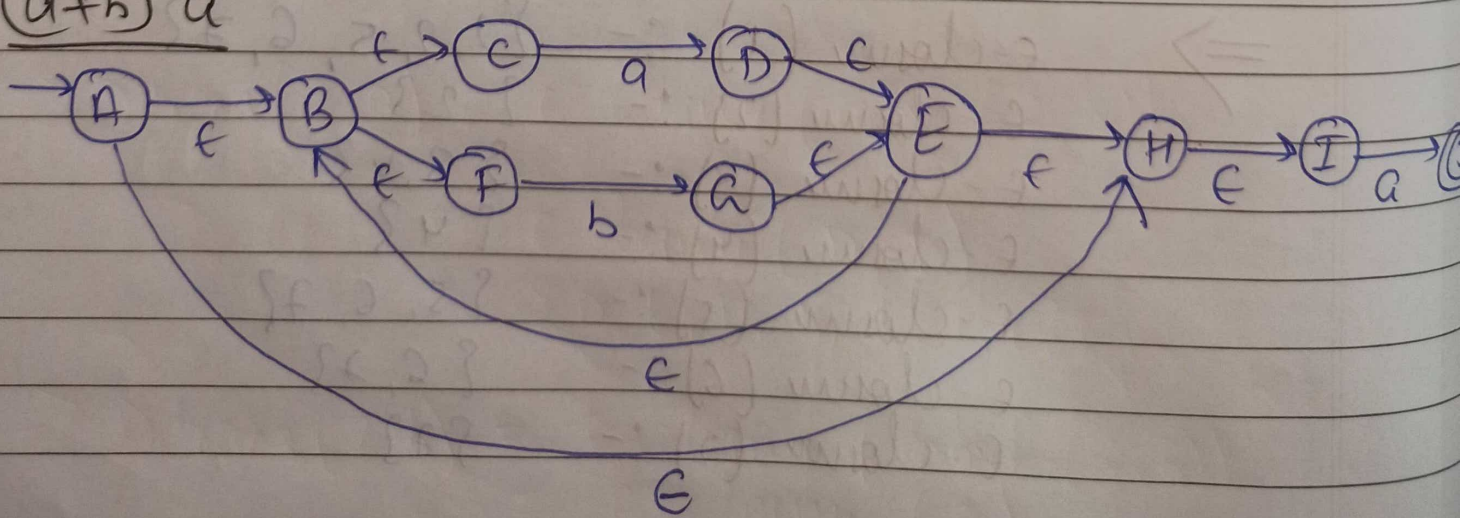
$(a+b)$



$(a+b)^*$



$(a+b)^*a$



Step 2 :- Find ε-closure of all states :-

A → {A, B, C, F, H, I}
B → {B, C, F}
C → {C}
D → {D, E, B, C, F, H, I}
E → {E, H, I, B, C, F}
F → {F}
G → {G, E, B, C, F, H, I}
H → {H, I}
I → {I}
J → {J}

Step 3 :- Taking ε-closure of starting state as initial state Draw the DFA table by finding DTrans.

ε(NFA) State	DFA State	a	b
→ {A, B, C, F, H, I}	1	2	3
{D, E, H, B, I, C, F, J}	2	2	3
{G, E, B, H, I, C, F}	3	2	3

final as it contain J

Formula :-

DTrans(State, input_Symbol) = ε-closure(move(State, input_Symbol))

⇒ Dtrans(1, a) = ε-closure(move(1, a))

= ε-closure(D, J)

= {D, E, H, B, I, C, F, J}

↑ since it is a new state we call it (2)

⇒ Dtrans(1, b) = ε-closure(move(1, b))

= ε-closure(G)

= {G, E, B, H, I, E, F}

↑ since it is a new state we call it (3).

⇒ Dtrans(2, a) = ε-closure(move(2, a))

= ε-closure(D, J)

= {D, E, H, B, I, C, E, J}

⇒ Dtrans(2, b) = ε-closure(move(2, b))

= ε-closure(a)

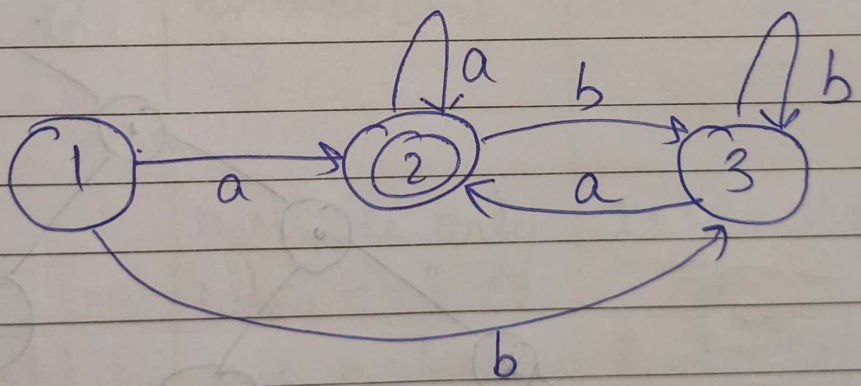
= {a, E, B, H, I, E, F}

$\Rightarrow$ Dtrans (3, a) = ϵ-closure(move(3, a))
= ϵ-closure (D, I)
= { D, E, H, B, I, C, F, I }

$\Rightarrow$ Dtrans (3, b) = ϵ-closure (move (3, b))
= ϵ-closure (G)
= { G, E, B, H, I, E, F }

↳ Since there is no new state available now, so we stop here.

Step 4 :- Draw DFA from the calculated table.



Question for practise :-

⟶ $(ab)^* + ab$

⟶ $(a + ab)^*$

Direct Method for Converting RE to DFA :-

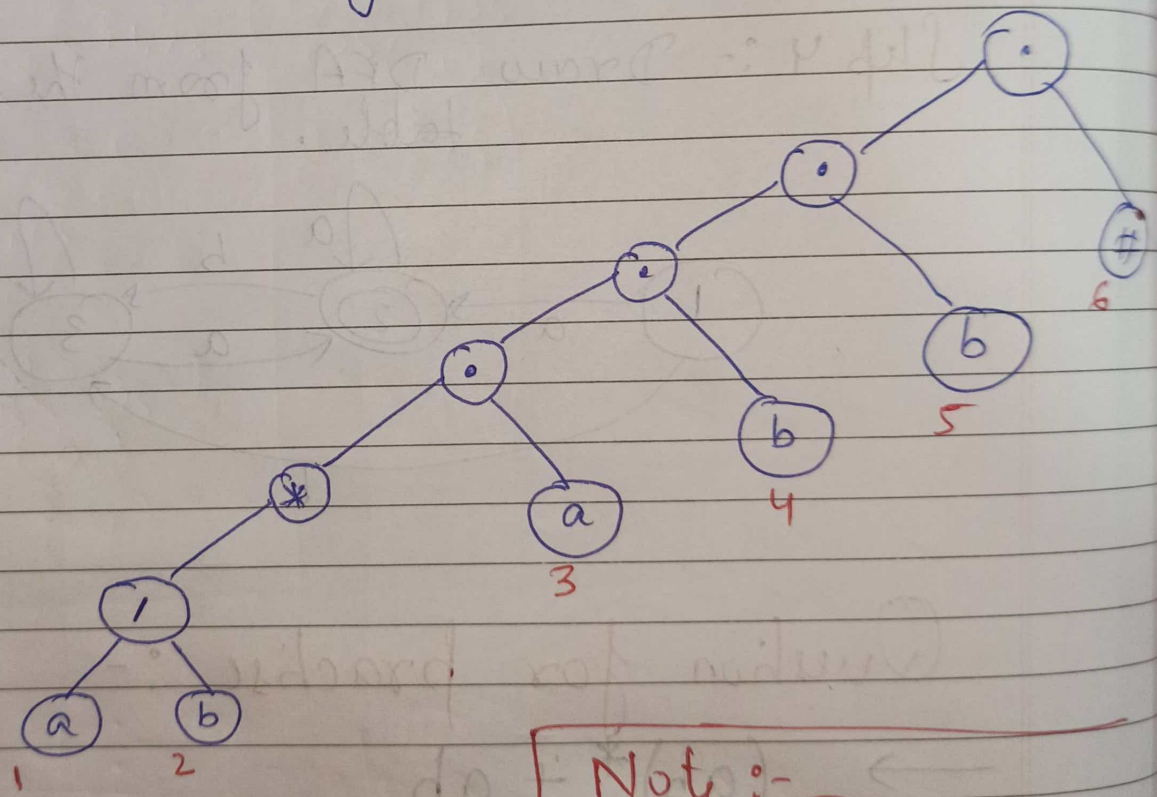
Q $(a/b)^*abb$

Step 1 :- Augment # at the end of regular expression.

$r' = (a/b)^* abb\#$

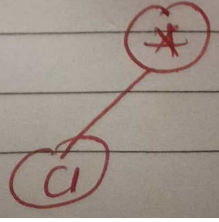
$\Rightarrow (a/b)^* \cdot a \cdot b \cdot b \cdot \#$

Step 2 :- Create Syntax tree for r'.

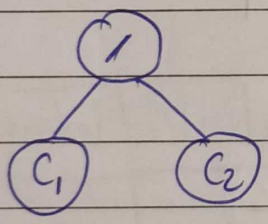
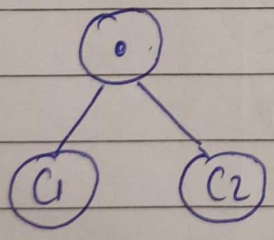
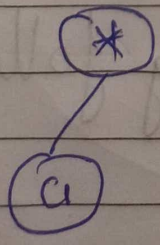


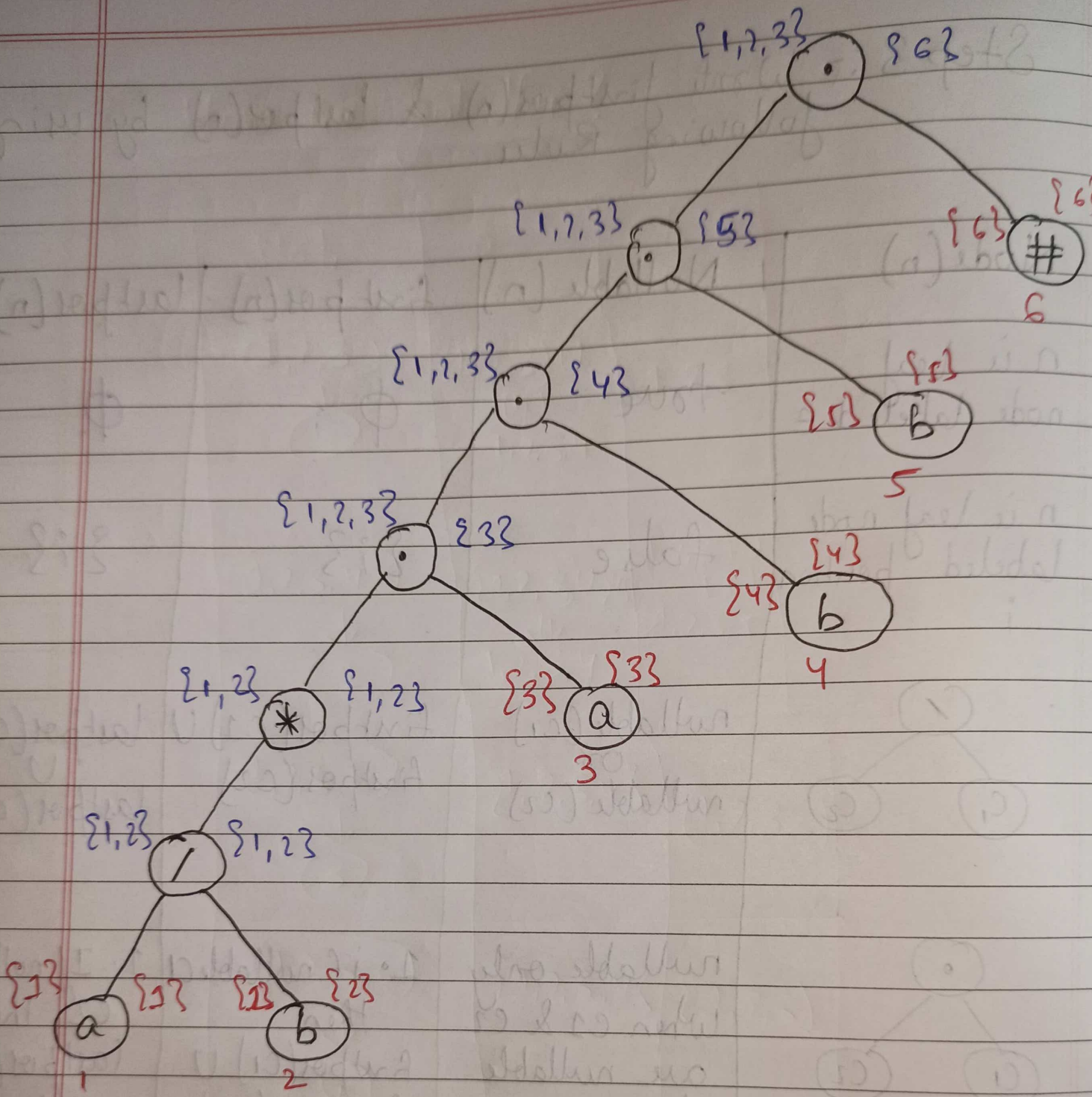
Note :-

(*) ka ek hi child hota h.



Step 3:- Write firstpos(n) & lastpos(n) by using following Rule.

S.No	Node(n)	Nullable (n)	first_pos(n)	lastpos(n)
1.	n is leaf node labeled ε	true	ϕ	ϕ
2.	n is leaf node labeled position i	false	{i}	{i}
3.		nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
4.		nullable only when c1 & c2 are nullable else not nullable	1. if nullable(c1) then firstpos(c1) ∪ firstpos(c2) 2. Else firstpos(c1)	1. If nullable c2 then lastpos(c1) ∪ lastpos(c2) 2. Else lastpos(c2)
5.		true	firstpos(c1)	lastpos(c1)



Step 4 :- Create a followpos(n) table by using Rules of computing followpos.

Rules for Computing followpos :-

1. If (n is a star * node)

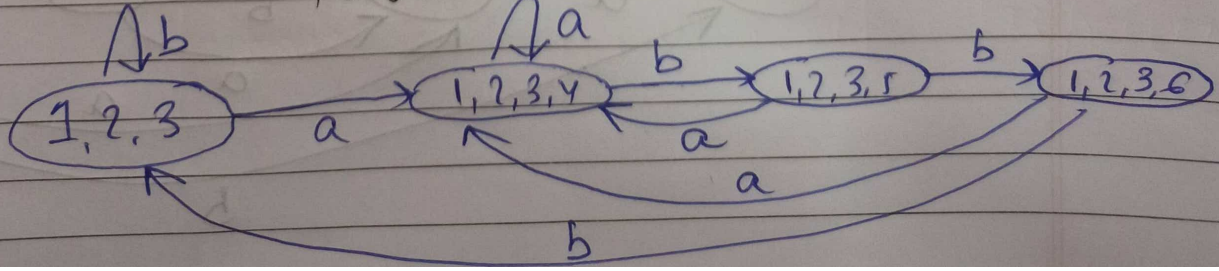
   for (each i in lastpos(n1))
   followpos(i) = followpos(i) ∪ firstpos(n);

2. If (n is a cat (·) node)

   for (each i in lastpos(c1))
   followpos(i) = followpos(i) ∪ firstpos(c2)

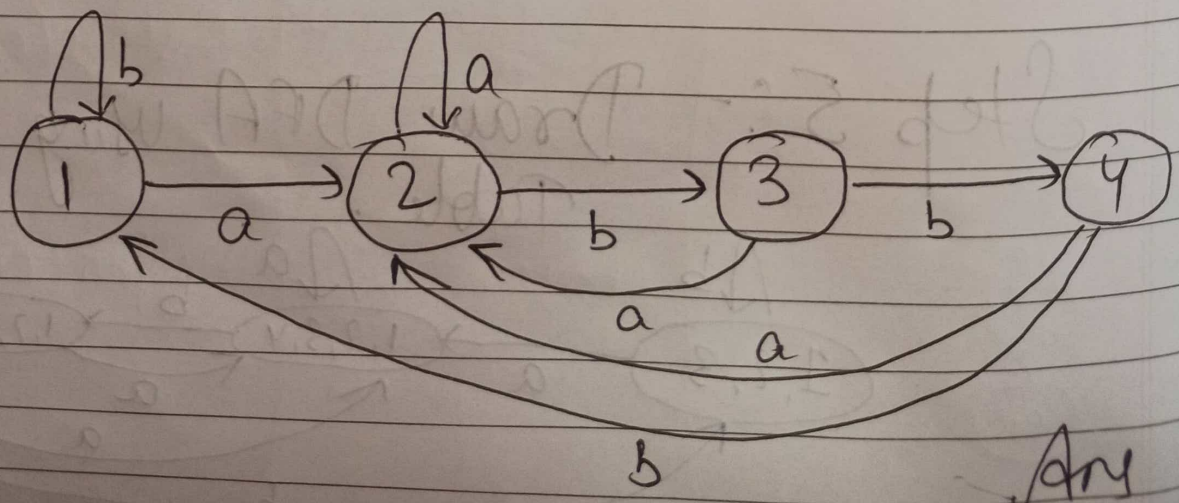
Symbol	Node	followpos
a	1	{1,2,3}
b	2	{1,2,3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	—

Step 5:- Draw DFA using followpos Table.



How we got this ?

State	a	b
(1,2,3) → 1	(1,2,3) ∪ (4) = 1,2,3,4	1,2,3
1,2,3,4 → 2	{1,2,3} ∪ {4} ⇓ 1,2,3,4	(1,2,3) ∪ (5) = 1,2,3,5
1,2,3,5 → 3	{1,2,3,4}	(1,2,3,6)
1,2,3,6 → 4	{1,2,3,4}	{1,2,3}



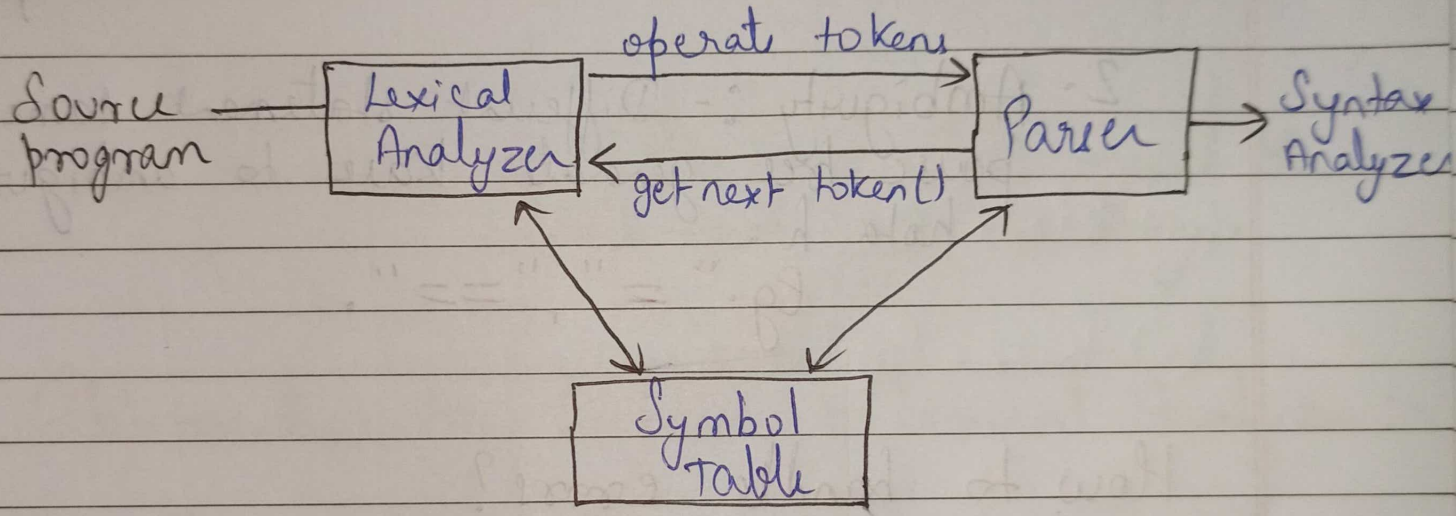
Any

Problem for practise :-

→ $(a+b)^* + (a \cdot c)^*$

→ $(a+b)^* \cdot (a \cdot b)^*$

Lexical Analyser



* Lexical Analyzer does :-

(i) Scanning :-

(ii) Tokenization :-

a. Lexeme (characters)

Eg. int a+b;

→ i, n, t, a, +, b, ;

b. Pattern (rules).

c. Tokens → meaningful words generate karega

int, a, +, b.

* Parser :- Lexical analyzer ko request karta h ki next line ke tokens generate kare.

Why we need lexical analyzer?
→ generate tokens & identify typing error.

Issues in Lexical Analysis?

1. Lookahead : Basically difficulty in deciding where to stop.

Eg. a + b + c = d + e
↑
in this it is not easy to identify where to stop.

2. Ambiguity :- Different statement same sa pause hee generate kare to ambiguity hota h.

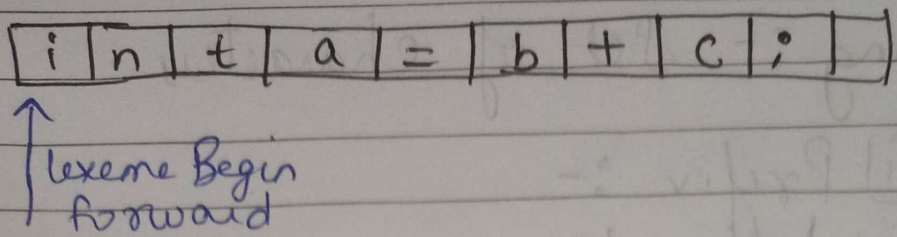
Eg. " = " , " == ".

How to handle errors?

→ ~~Insert~~ Delete one char → intn → delete
→ ~~Delete~~ Insert one char → print → Insert
→ Replace one char → int → int
→ transpose two character → itn → int
swap.

Input Buffering :-

Buffer → temporary space where we store data.

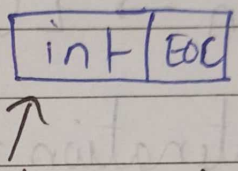


2

2 pointer work on this.

→ Lexeme Begin which identify 1st character of the word

→ forward which moves forward in search of whitespace.



Buffer space which match the word with our directory.

Approaches to Implement Lexical Analyzer :-

1) LEX TOOLS

2) Basic Programming Language

3) Assembly Language

Specification of Tokens :-

1. String
2. Language
3. Regular Expression

1. String :- Set of characters

(i) Prefix :-
(ii) Suffix :-
(iii) Substring :-

2. Language :- $L = \{0, 1\}$, $S = \{a, b, c\}$

(i) Union → $L \cup S = \{0, 1, a, b, c\}$

(ii) Concatenation → $LS = \{0a, 0b, 0c, 1a, 1b, 1c\}$

(iii) Klene Closure → L^*

(iv) ~~Klene~~ Positive closure

3. Regular Expression :- used to represent regular language

Q. R.E for second element as 0.
⇒ $(0+1)0(0+1)^*$

Q. RE for first element as 1 & last element as 0
$1(0+1)^*0$

Q. RE for either start with 0 or end with 1.
⇒ $0(0+1)^* + (0+1)^*1$.