UNIT- 1

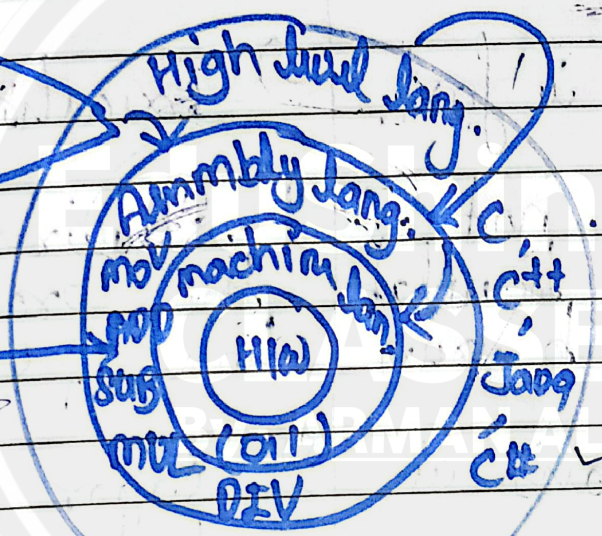
Introduction to Compiler

1. Diff. between Compiler & Interpreter.
2. what are the diff. levels of programming language & diff. types of translators.

Compiler

Interpreter

Assembler



100%

100%

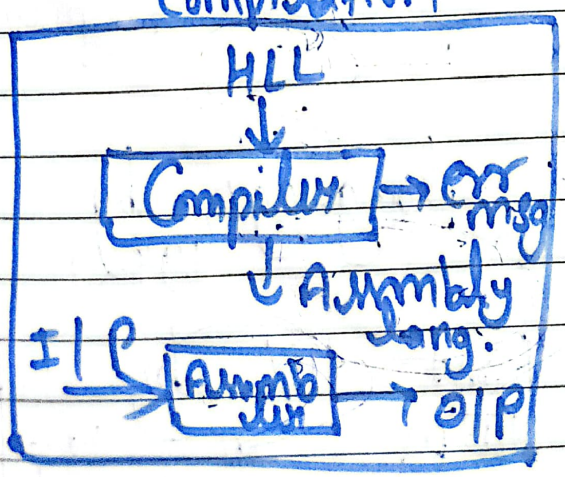
100%

100%

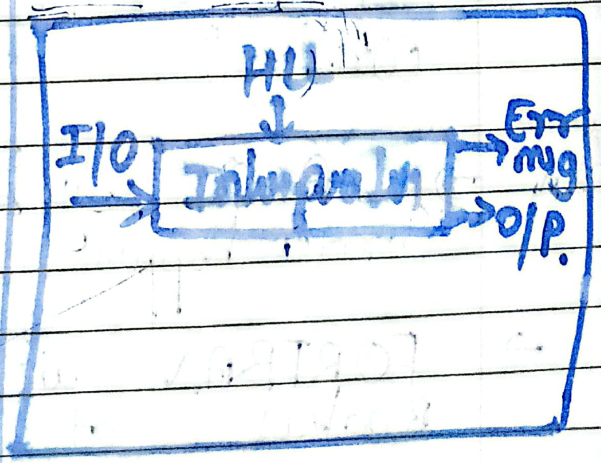
i.e. 4.

Translator -

Compiler -



Interpreter



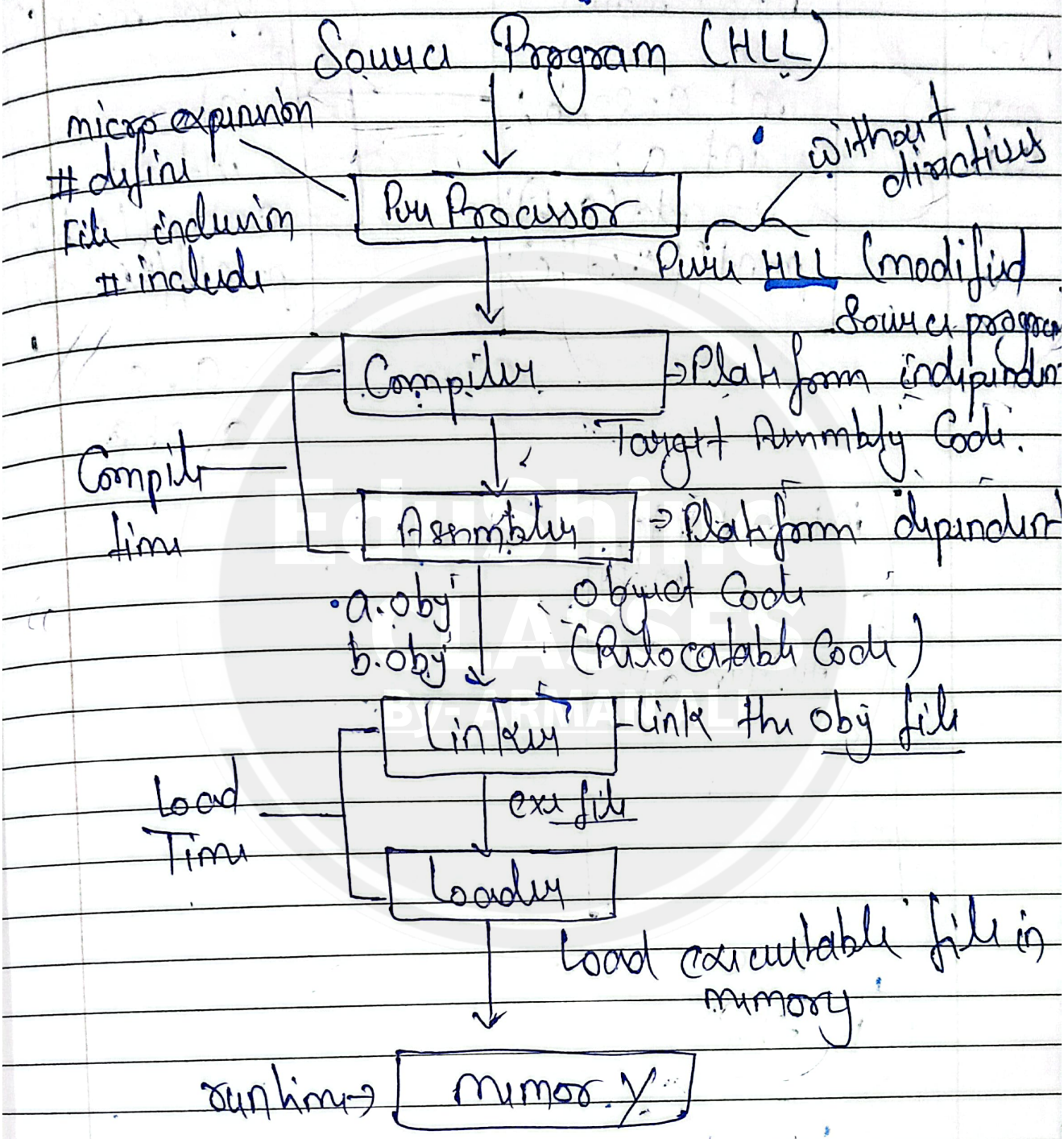
Compiler	Interpreter
1) It translate Source Code into object Code as whole.	1) It translate the Source Code one by one (line by line) & execute immediately
2) It create an object (executable) file.	2) It doesn't.
3) Execution is fast.	3) Execution is slow
4) Program is not required to translate each time to run	4) Requires to translate the program each time to run.
5) Error diagnosis is poor	5) Error diagnosis is better
6) Most HLL use compiler	6) ~~Most~~ HLL use Interpreter

Q3 Which was the first Compiler? how was the first Compiler written?

Ans → Grace Hopper in 1952, System language, by

→ FORTRAN is First HLL by John Backus at IBM in 1957 18 person-year to create it.

* Language Processing System -



Now let's see an easy example —

Pure processor (Header file)

Pure HLL

Not pure HLL

```
#include<stdio.h>
#define b 10;
void main()
{
  int a=20, b;
  int c;
  c = add(a,b);
printf("%d",c);
```

```
void main()
{
  int a=20, b=10;
  int c;
  c = add(a,b);
  printf("%d",c);
}
```

a.c

a.obj

```
int add(int a,
        int b)
{
  return a+b;
}
```

b.c

b.obj

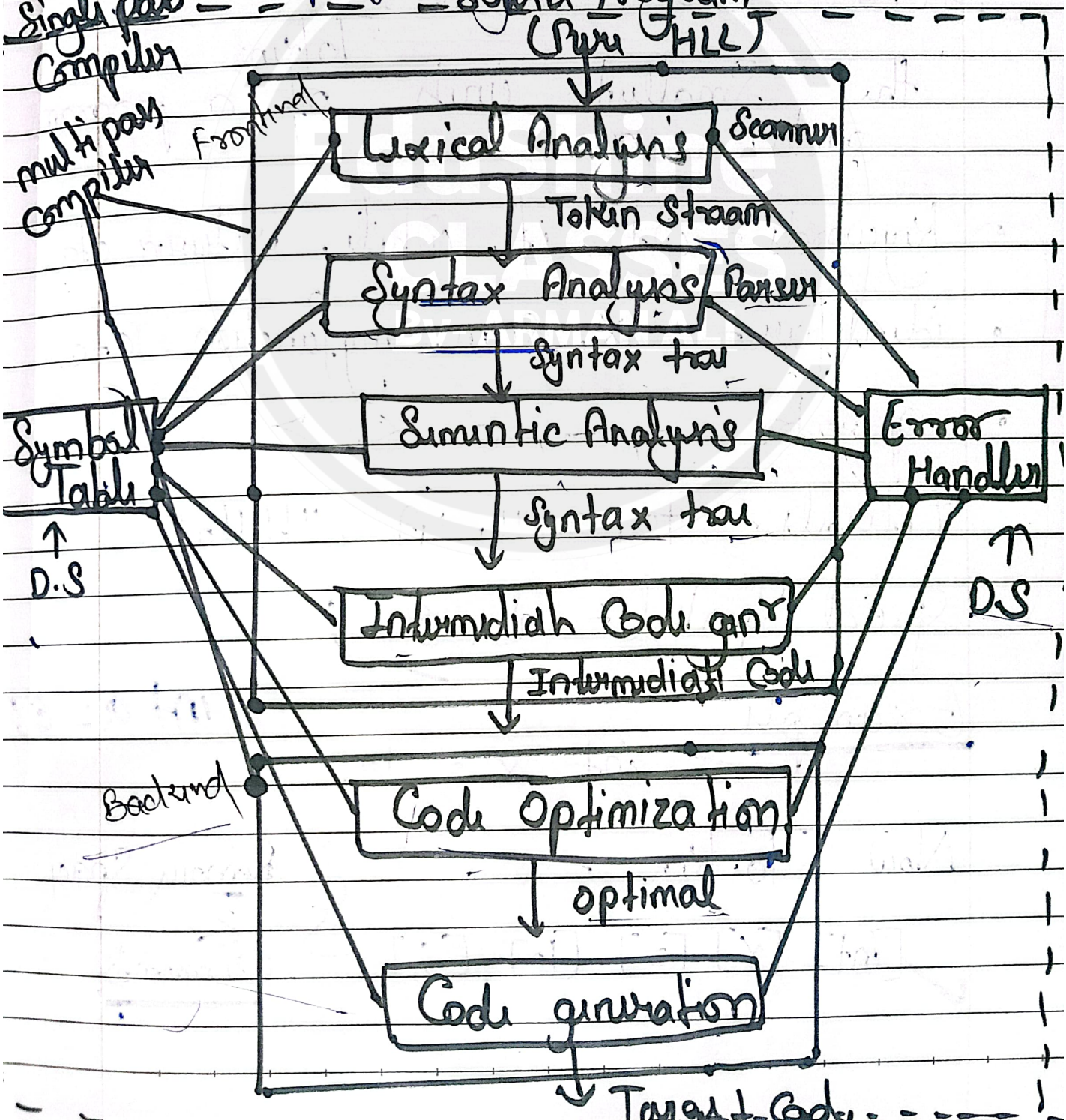
***

* Phases of Compiler & Pass of Compiler -

A Compiler is a program that translate high level Code (like - C, Java, Python) into machine Code that a Computer can understand. This translation process happens in Several phase, each performing a Specific task.

Single pass Compiler

multi pass Compiler



① Lexical Analysis - The first phase of a Compiler, responsible for reading the program character by character and breaking it into meaningful units called 'tokens'.

What are Tokens? Tokens are the smallest units of a program such as -

- Keywords - int, if, while, return etc.
- identifier - x, sum, myfunction etc.
- Operator - +, -, =, <, >
- Literals - 10, 3.14, 'A', "Hello"
- Special Symbol - {, }, ;, ,

Example - int a = 5;

int x = 10;

Now to run -

[int] [x] [=] [10] [;]

Remove Space & Comments

2) Syntax Analysis -

* This phase checks whether the sequence of token follows the grammar of the programming language.
* if the syntax is incorrect, the Compiler gives an error.

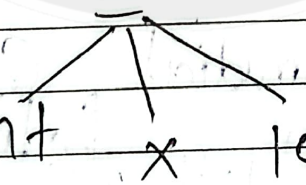
Example -

int x = 10;

int x 10 =;

Parser detects this as an error.

* It uses CFG Rules
* Construct a parse tree to represent the structure of program



3) Semantic Analysis -

* checks the maning of the code & ensure that it makes logical sense. Even if the syntax is correct the maning might be incorrect.

int x = "hello";
10;

Example:

- It stores general variable assignment
- Function checking, Scope checking
- It uses Symbol Table that stores data about variables, functions & their scope.

(4) Intermediate Code Generation:

- Converts the high level code into an intermediate representation (IR) making it easier to optimize and translate into machine code.
- This intermediate code makes the compilation efficient & portable across different machines.

Example:

a = b + c

Now, Intermediate Code (Three address Code (TAC))

$$\begin{bmatrix} T_1 = b + c \\ a = T_1 \end{bmatrix}$$

⑤ Code Optimization –

* Improves the efficiency of the code without changing its meaning.

* This phase removes unnecessary calculations & reduces execution time.

Ex. a = 10
b = c * a
d = e + b
f = d

→ b = c * 10
f = e + b

optimize.

⑥ Code Conversion –

* Converts the optimized intermediate code into machine code (binary code) that the computer understands.

Ex. A = B + C

→ ADD R1, B, C
MOV A, R1

machine Code –

10101000010

int x = 5;

⑦ Symbol Table - Store informations about variables, functions & their attributes.

Example - Every variable declared before use.

Name	Type	Scope	Memory Address
x	int	Global	1000
y	float	Local	1004
sum	func	Global	2000

⑧ Error Handling -
* Detects & reports error at different stages of compilation.

Ex. -
~~if (x)~~
if (x = 5) X should be ("x == 5")

Passes of Compiler -

* In a implementation of Compiler, one or more phases are combined into a module called a 'pass'.

There are two main types -

① Single - Pass Compiler (One time Scanning)

* The Compiler scans the program only once.
* it immediately generates machine code after analyzing each part of the code.
* Faster but less powerful (can't handle complex optimizations).

Ex. : C language, Pascal.

② Multipass Compiler -

* The compiler scans the Source Code multiple times.
* Each pass refines & improves the code.
* Used for better optimization & handling Complex Code.

Common Pass in multi pass Compiler

① First Pass (Lexical & Syntax Analysis)
- Breaks code into token.
- checks for correct Syntax.

② Second Pass (Semantic analysis & intermediate)

③ Third Pass (Optimization & Code Generation)

Ex.
- Java Compiler
- C++ Compiler

Formal Definition of F.A -

$(Q, \Sigma, q_0, F, \delta)$

Types

i) DFA

$(Q, \Sigma, q_0, F, \delta)$

$\delta \rightarrow Q \times \Sigma$

$Q \times \Sigma \rightarrow Q$

ii) NFA

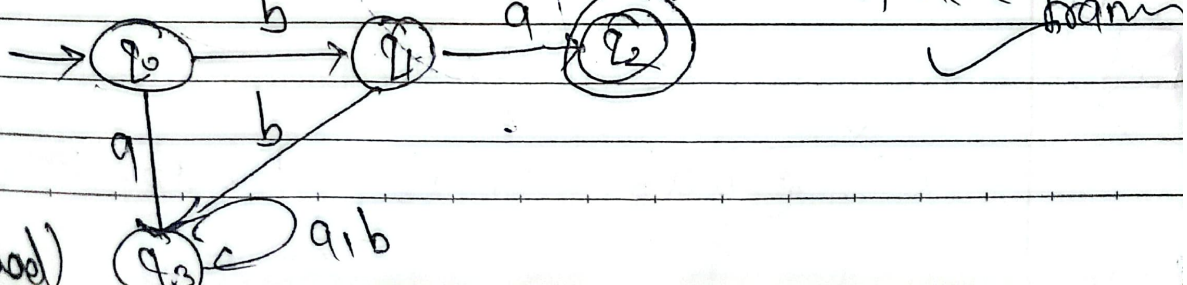
Q1 Design a DFA such that string accepted must start with substring ba $\quad \Sigma = \{a, b\}$

Sol

$\Sigma = \{a, b\}$

$\Sigma^* = \{ba, bab, baa, baba, babb, babbab \text{ ---- }\}$

Diagram -



DFA में हर state से कम से कम 1 edge जाना चाहिए

Study of abstract computing device

What is Automata?

* An automaton, in simple terms is like a little machine that follows a set of rules
* It is often used in computer science and mathematics to model process that follow a certain pattern or behaviour.
* Imagine you're a vending machine, where you put in money and press a button for a particular snack, the vending machine follows a set of rules: if you've put in enough money and press the right button it gives ya snack you want. that's kind of like an Automata
* In Computer Science, automata are often used to discribe how computer process information, they can be used to recognize pattern in text (like finding specific words)

Validate whether a string of characters follows a certain format (like checking if an email id is correctly written)

Diffurnt typs of finite Automata -

Finite Automata without O/P	Finite Automata with o/p
*** ① Deturministic Finite Automata (D.F.A)	① Moore machin
② Non. D.F.A or N.F.A (NDFA)	② Maly machin.
③ Non deturministic finite automata with ε move (ε-NFA or ε-NDFA)	

Finite Automata Rpurnntation -

① Graphical (Transition diagram)

② Tabular (transition table)

③ Mathmatical (Tranition function)

Deturministic Finite Automata -

→ Imagin you have a traffic light it has three state - red, grun and yellow. it follows a fixed Squnce red → grun → yellow → red and so on.

* A Deterministic Finite automata (D.F.A) is a bit like that traffic light but with symbol instead of colors.

* It's simple machine that reads symbol (like letters, or numbers) one at a time and moves between different states based on what it reads.

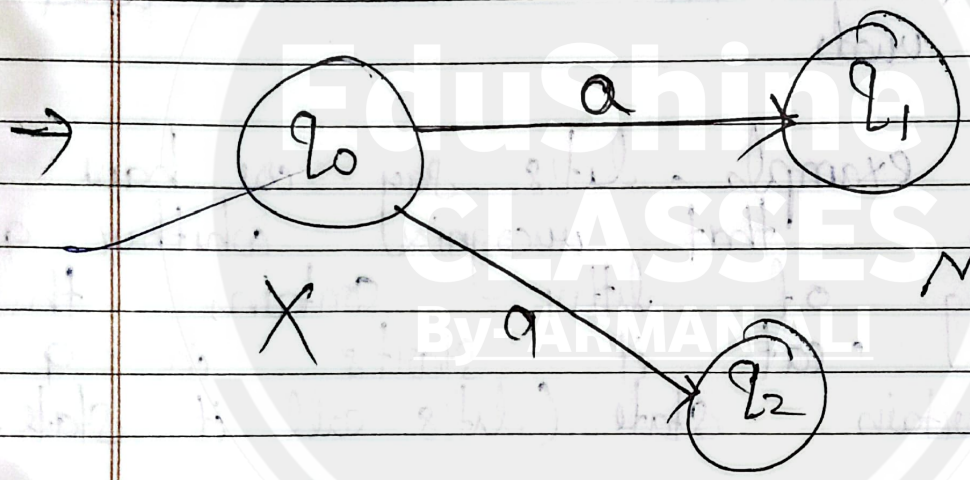
* For example - let's say we have a D.F.A that recognizes whether a string of letter contains the word 'cat' it starts in a certain state (let's call it state 1)

* As it reads each letter of the I/P string, it moves from one state to another. if it reaches the end of the string and is in a specific state (let's call it state 3) it accepts the string as containing "cat" Otherwise, it rejects the string.

* So, D.F.A is essentially a machine that reads symbol and follows predetermined rules to decide whether

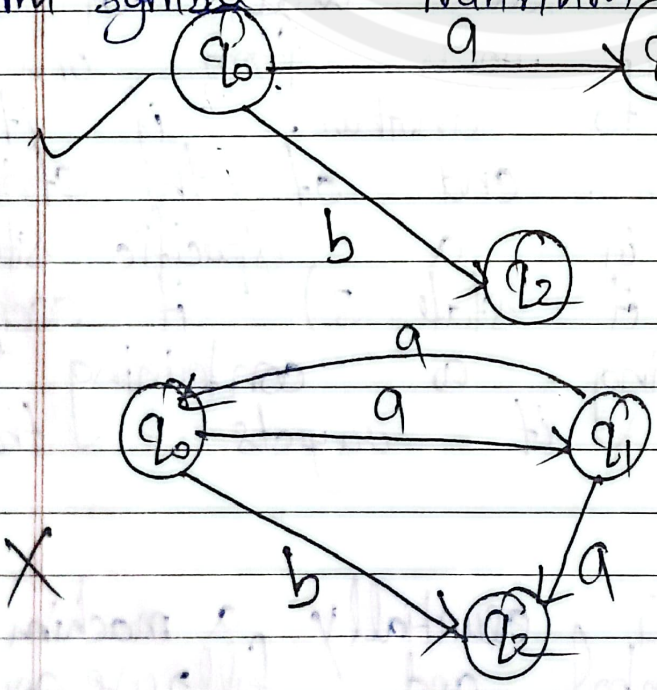
a String belongs to a certain language or follows a certain pattern.

"In D.F.A on each input there is only one state which the automata can transition from its current state."



X Not D.F.A

Same Symbol on Transition more than one state पर होता



D.F.A

X Not D.F.A

A D.F.A Consist of 5-tuple
$(Q, \Sigma, \delta, q_0, F)$

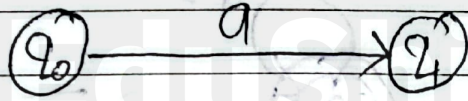
Q - Finite non empty Set of state.

Σ - Finite non empty Set of I/P.

δ - Transition function $Q \times \Sigma \to Q$

eg - $\delta(q_0, a) \to q_1$

$Q \times \Sigma \to Q$



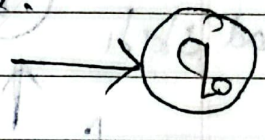
$\therefore Q \times \Sigma$ means

$Q = \{q_0, q_1, q_2\}$ → State

$\Sigma = \{a, b\}$ ← Alphabet

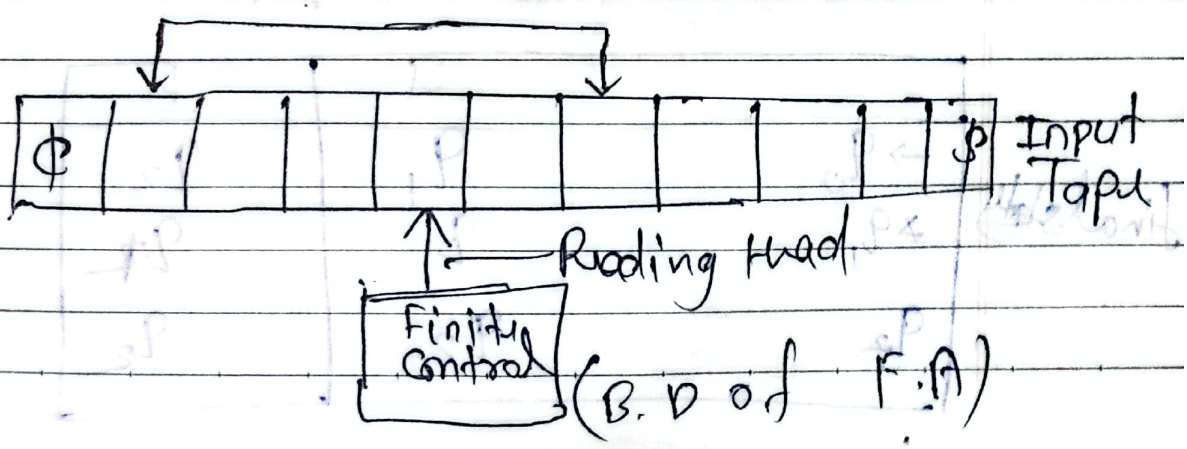
$\delta = 3 \times 2 = 6$ transition func^n

$q_0 \to q_0 \in Q$ is initial state.



F - Final state ($F \subseteq Q$)

* There can be many possible DFAs for a pattern.



Formal Definition of F.A -

$(Q, \Sigma, q_0, F, \delta)$:

Types

i) DFA

ii) NFA

$(Q, \Sigma, q_0, F, \delta)$

$\delta \rightarrow Q \times \Sigma$

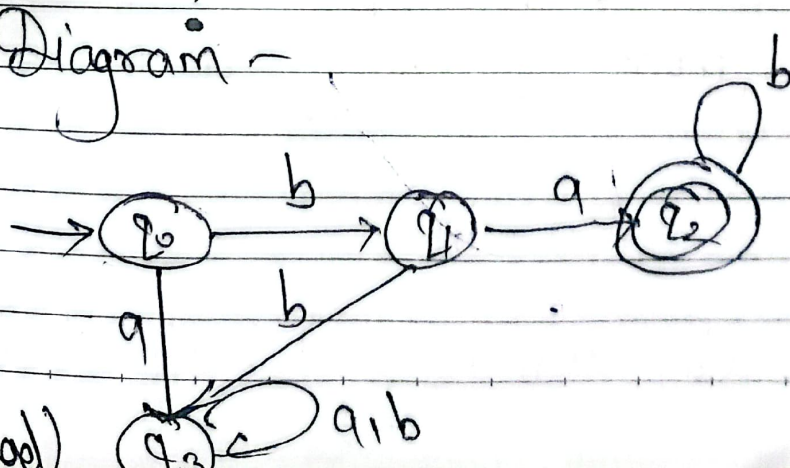
$Q \times \Sigma \rightarrow Q$

Q1 Design a DFA such that string accepted must start with substring ba $\quad \Sigma = \{a, b\}$

Sol $\quad \Sigma = \{a, b\}$

$\Sigma^* = \{ba, bab, baa, baba, babb, babbab ----\}$

Diagram -



DFA में हर state के लिए transition होना चाहिए

Transition Table :-

	a	b
$\rightarrow q_0$	q_3	q_1
q_1	q_2	q_3
$*q_2$	q_2	q_2
q_3	q_3	q_3

Transition Function :-

$\delta(q_0, a) \rightarrow q_3$
$\delta(q_0, b) \rightarrow q_1$
$\delta(q_1, a) \rightarrow q_2$
$\delta(q_1, b) \rightarrow q_3$
$\delta(q_2, a) \rightarrow q_2$
$\delta(q_2, b) \rightarrow q_2$
$\delta(q_3, a) \rightarrow q_3$
$\delta(q_3, b) \rightarrow q_3$

$Q = 4$
$\delta = Q \times \Sigma$
$= 4 \times 2$
$= 8$

$(Q, \Sigma, q_0, F, \delta)$

$(\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{q_0\}, q_2, \delta)$

Q.2 Design a DFA such that string accepted must end with bab $\Sigma = \{a, b\}$

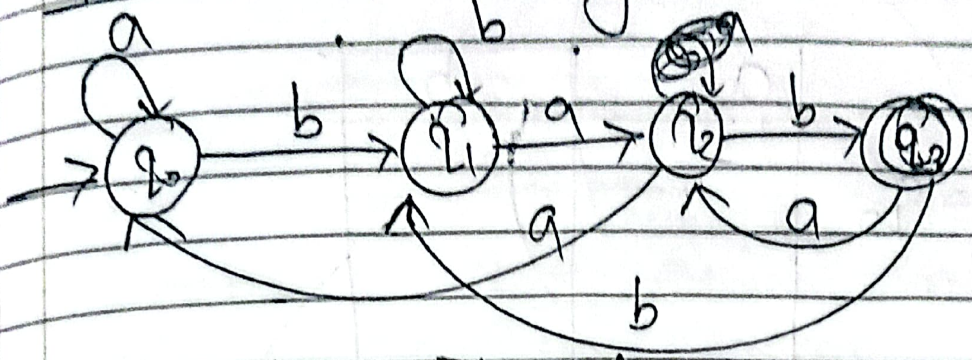
Sol^n

$\Sigma = \{a, b\}$

$\Sigma^* = \{bab, abab, bbab, ababab \ldots\}$

and start with that then dead state ✓ bbbaaabab
end with it dead state ✗

Transition Diagram -



wrong
abaabab
bbabab

Transition Table -

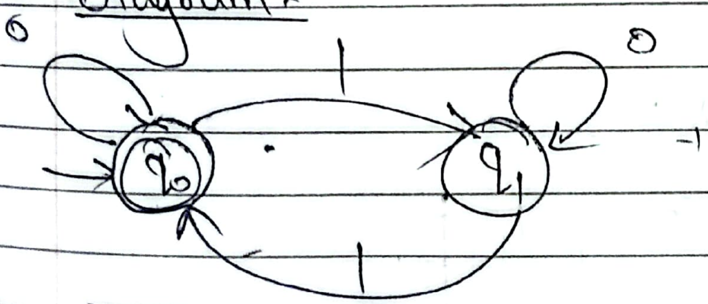
	a	b
$\to q_0$	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_3
$* q_3$	q_2	q_1

Q.3 Design a D.F.A such that string that contain even no of 1's.

Sol/ $\Sigma = \{0, 1\}$

$\Sigma^* = \{\epsilon, 0110, 101, 0101, 10101 1, 0101101101 \text{-----} \}$

Diagram -



01011

Transition Table -

	a	b
$\to q_0$	q_0	q_1
q_1	q_1	q_0

Transition Table

	a	b
→ q_0		

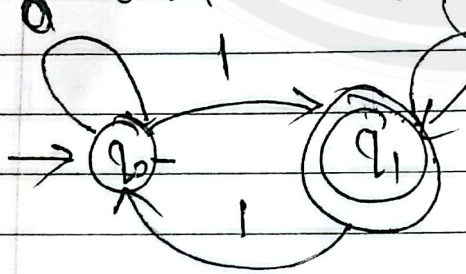
Q 4, Design a D.F.A such that string that contain odd no. of 1's.

Sol^n

$\Sigma = \{0, 1\}$

$\Sigma^* = \{01, 10, 01011, 1011, 01110$
$, 1101011 \text{---} 3$

Transition Diagram -



Transition Table -

	0	1
→ q_0	q_0	q_1
* q_1	q_1	q_0

NFA

$(Q, \Sigma, q_0, F, \delta)$

$\delta : Q \times \Sigma \to 2^Q$

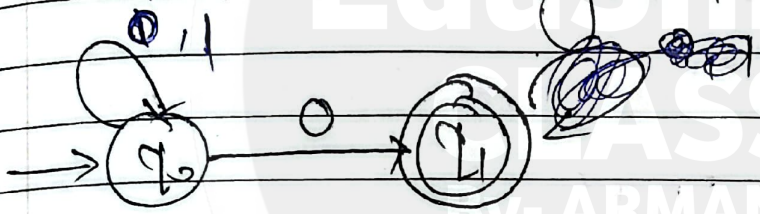
Q.1 Construct NFA over $\{0, 1\}$ accept all set of string end with zero.

Sol^n

$\Sigma = \{0, 1\}$

$\Sigma^* = \{10, 100, 110, 1010, 11010, 1011010 \ldots\}$

Transition Diagram



Transition Table

	0	1
$\to q_0$	q_0, q_1	q_0
q_1	—	—

Q Construct NFA for the language which accept all the string in which third symbol from the right end is always a $\Sigma = \{a, b\}$

Application of Finite Automata -

* **D.N.A Sequence Analysis** - Finite automata can be used to search for patterns in D.N.A sequences.

* **Robotics** - Finite automata can be used to model and control robot behaviour in certain scenarios.

* **Pattern recognition** - Finite automata can be used in image and signal processing for pattern recognition tasks.

Limitation of Finite Automata -

* **Limited memory** - Finite automata have a finite no. of states.

* **Limited Expressiveness** - Finite automata can only recognize regular languages.

* Diffrunca between D.F.A & N.D.F.A-

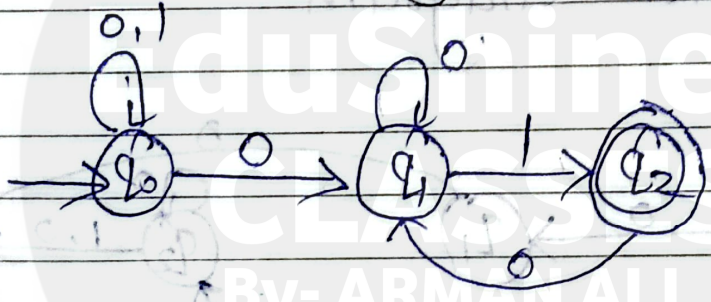
D.F.A	N.F.A
① Each transition lead to exactly one state called deterministic	① A transition lead to a subset of state i.e some transition can be non-deterministic.
② Accept input if the last state is final	② It accept input if one of the states is final
③ It requires more space	③ It require less space.
④ Backtracking is allowed in D.F.A	④ Backtracking is not possible.
⑤ Empty string transition are not run in D.F.A	⑤ it Permit empty string transition.
⑥ D.F.A is a subset of N.F.A	⑥ Need to convert N.F.A to D.F.A in the design of compiler
⑦ $\delta: Q \times \Sigma \to Q$	⑦ $\delta: Q \times \Sigma \to 2^Q$
⑧ $\delta(q_0, a) = \{q_1\}$	⑧ $\delta(q_0, a) = \{q_1, q_2\}$
⑨ D.F.A is difficult to Construct	⑨ N.F.A is easier to Construct.

Q.2 Design an N.F.A $\Sigma = \{0,1\}$ which accept all string ending '01'.

Sol: given, $\Sigma = \{0,1\}$

$L = \{01, 0001, 10101, 0101 \cdots\}$

Transition diagram.



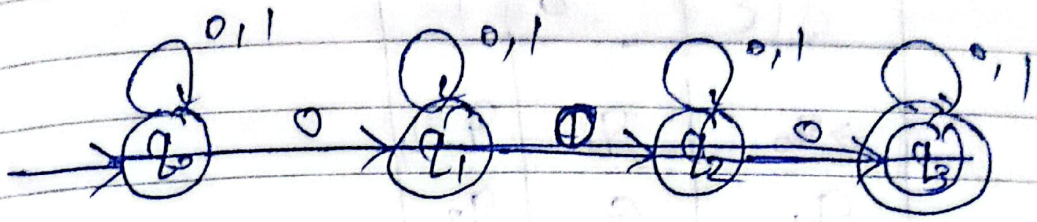
Q. Design a N.F.A, $\Sigma = \{0,1\}$ which accept all string in which the third symbol from the right end is always '0'.

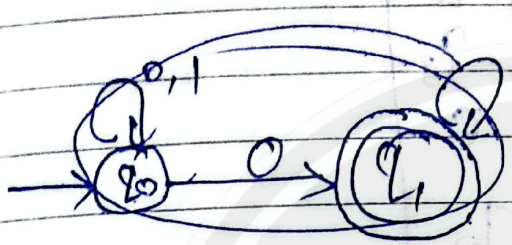
Ans: given $\Sigma = \{0,1\}$

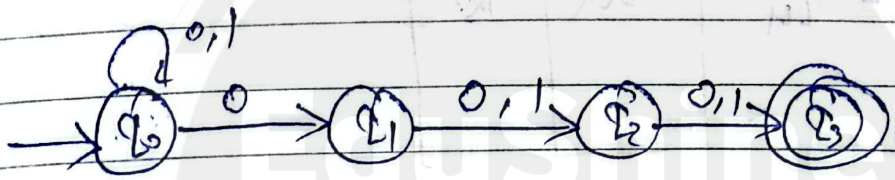
$L = \{01010, 11011, 01011, 10010 \cdots, 010, 011 \cdots\}$

wrong
101 x

Transition diagram :-





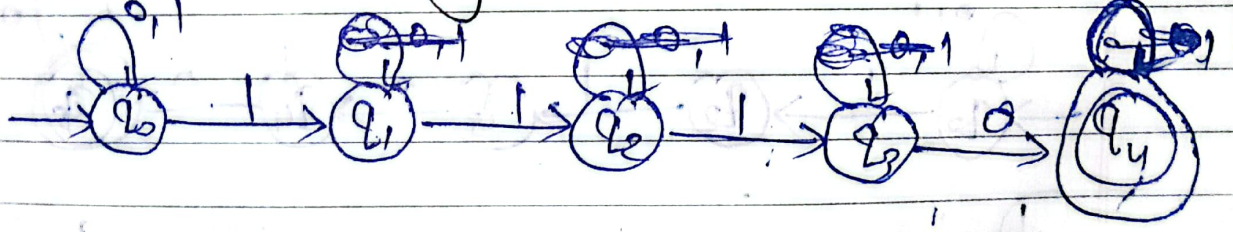


Trans
Design an N.F.A which accept all the string that contain a substring '1110'

given $\Sigma = \{0, 1\}$

$L = \{1110, 001110, 11110, 011100 \ldots\}$

Transition diagram -



Transition table.

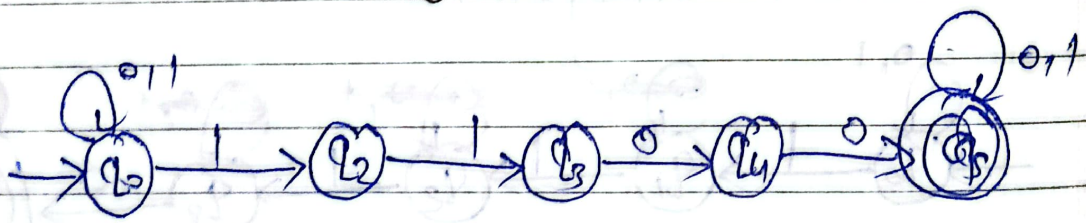
δ | Σ	0	1
→ q_0	q_0	q_0, q_1
q_1	∈	q_2
q_2	∈	q_3
q_3	q_4	∈
* q_4	q_4	q_4

Q Design an NFA with $\Sigma = \{0, 1\}$ in which '11' is followed by '00'

A given $\Sigma = \{0, 1\}$

$L = \{1100, 01100, 11100 \cdots\}$

Transition diagram.



Q. Design an N.F.A with $\Sigma = \{0, 1\}$ which accept all the string of length '2'

Minimization of D.F.A -

The Process of elimination of state whose presence or absence doesn't affect the language accepting capability of D.F.A is called minimization of D.F.A and the result is minimal D.F.A commonly known as MFA

Note: M.F.A is unique for a language.

Dead state - It is basically created to make the System Complete, can be defined as a state from which there is no transition possible to the final state.

Note - One dead state is Sufficient to Complete the functionality.

Unreachable state - Unreachable states are the states that are not reachable from the initial state of D.F.A for any input String.

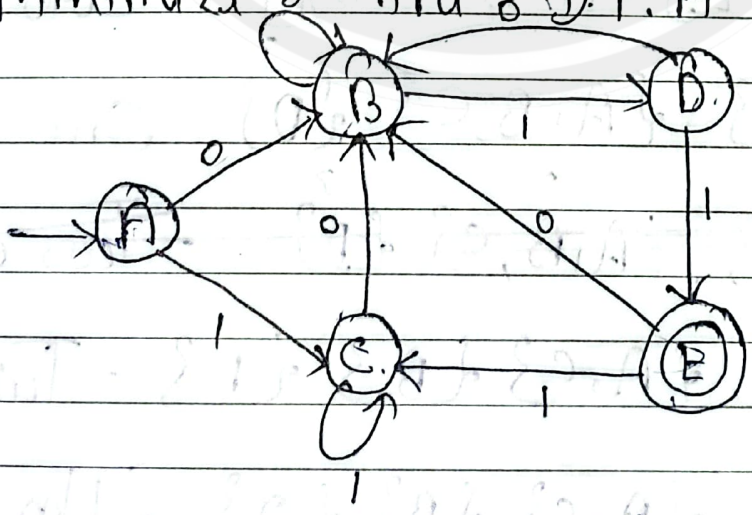
Equivalent state -

In the minimization of D.F.A, equivalent states are states that can be merged without changing the language recognized by the D.F.A.

Equivalence class - (π)

An equivalence class refers to a grouping of states that are equivalent to each other. This means that all states within an equivalence class behave identically with respect to the language recognized by this D.F.A.

Q) Minimize the D.F.A



Soln

Step-1 check unreachable state if any then remove.

Here no any unreachable state from initial state.

Step 2 Check any dead state (Don't write)

Step 3 check final state - Write - Y

Transition table -

	0	1
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C
or (E)		

Now Minimized D.F.A -

$\Pi_0 = \{\{E\}, \{A, B, C, D\}\}$ - Two equivalent

$\Pi_1 = \{\{E\}, \{A, B, C\}, \{D\}\}$ One equivalence

~~$\Pi_2 = \{\{E\}, \{A, B, C\}\{D\}\}$ - Two equivalent~~

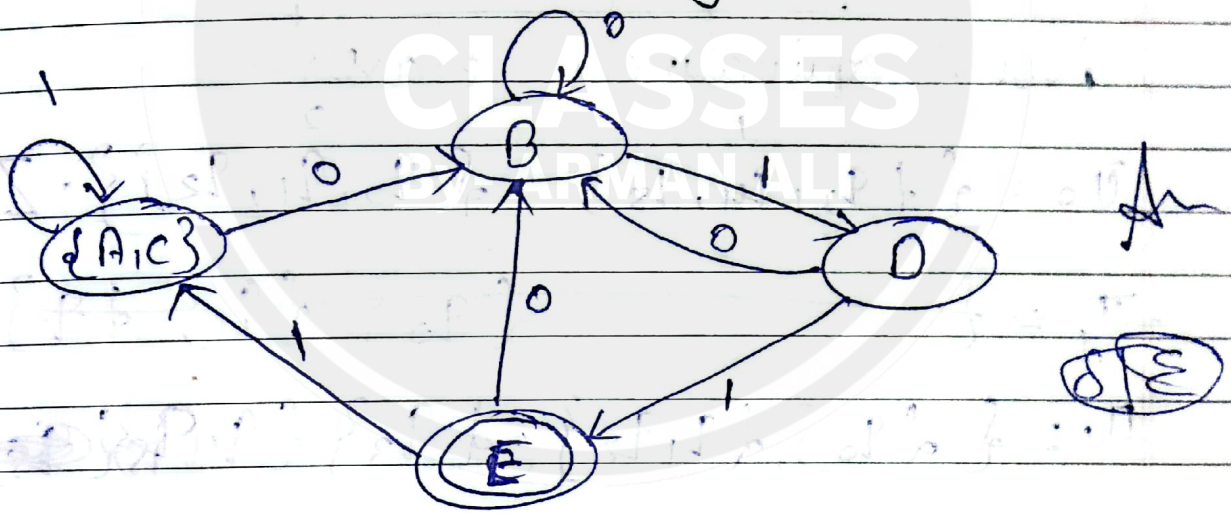
$\Pi_2 = \{\{E\}, \{A, C\}\{B\}\{D\}$ - Two equivalent

$\Pi_3 = \{\{E\}, \{A, C\}\{B\}\{D\}$ - Three

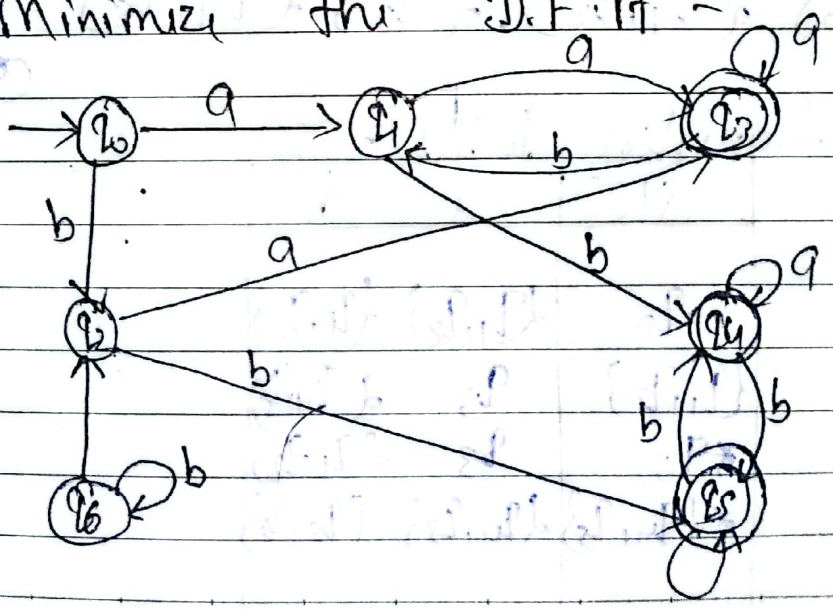
Minimized Transition Table :-

δ | Σ	0	1
→ {A,C}	B	{A,C}
B	B	D
D	B	E
* E	B	{A,C}

Minimize transition diagram -



Q.2 Minimize the D.F.A -



Now Transition table -

$\delta \mid \Sigma$	a	b
$\to q_0$	q_1	q_2
q_1	q_3	q_4
q_2	q_3	q_5
$*q_3$	q_3	q_1
$*q_4$	q_4	q_5
$*q_5$	q_5	q_4
q_6	q_2	q_6

✓ — No un reachable

Now equivalency classes -

$\pi_0 = \{\{q_3, q_4, q_5\}, \{q_0, q_1, q_2\}\}$ - Zero eqn

~~$\pi_1 = \{ \quad \{q_0\} \{q_1\} \{q_2\}$~~

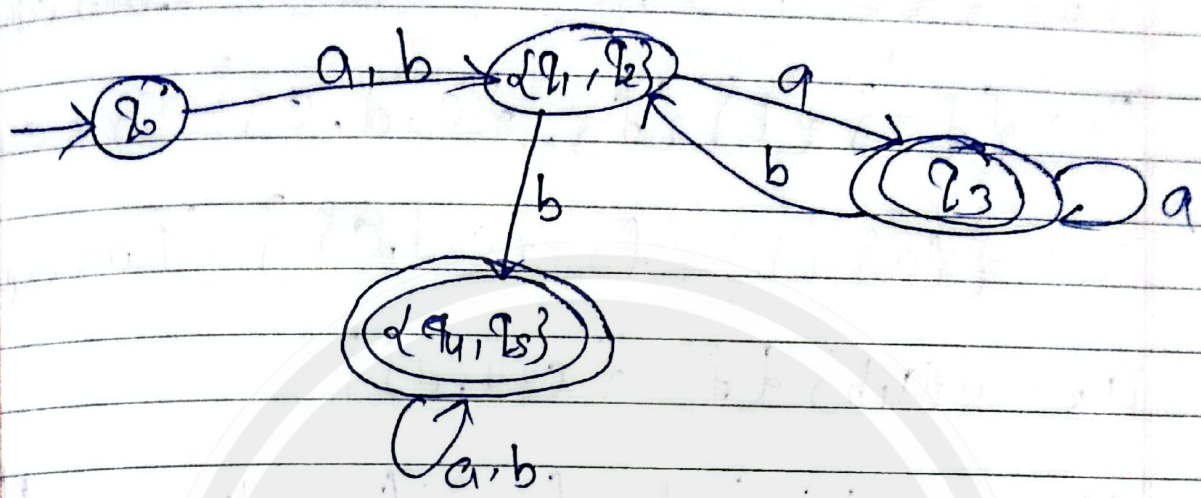
$\pi_1 = \{\{q_0\}, \{q_1, q_2\}, \{q_3\} \{q_4, q_5\}\}$

$\pi_2 = \{\{q_0\}, \{q_1, q_2\}, \{q_3\} \{q_4, q_5\}$ - Two equivalence

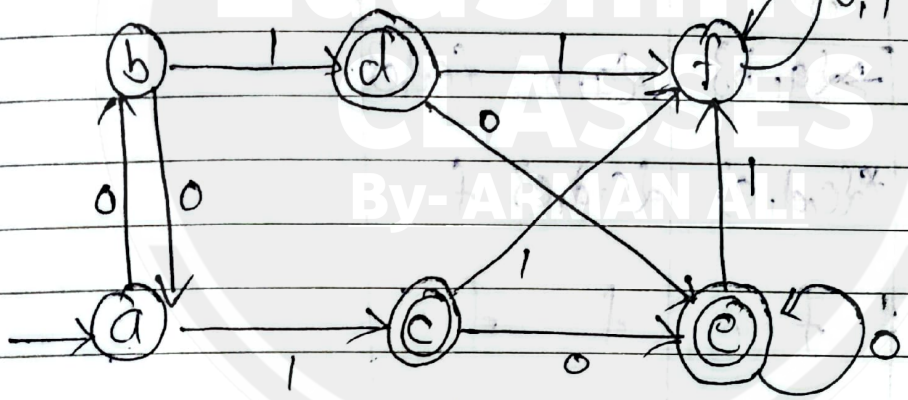
Minimize transition table -

$\delta \mid \Sigma$	a	b
$\to q_0$	$\{q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	q_3	$\{q_4, q_5\}$
$*q_3$	q_3	$\{q_1, q_2\}$
$*\{q_4, q_5\}$	$\{q_4, q_5\}$	$\{q_5, q_4\}$

Now Minimize transition diagram -



Q3 Minimize the D.F.A. -



A Transition table.

δ | Σ	0	1
→ a	b	c
b	a	d
* c	e	f
* d	c	f
* e	e	f
f	f	f

Now equivalence classes -

$\pi_0 = \{\{c, d, e\}^1, \{a, b, f\}^2\}$ - Zero equivalence

$\pi_1 = \{\{a\}^1 \{b\}^2 \{f\}^3, \{c, d, e\}^4\}$ - One equival.

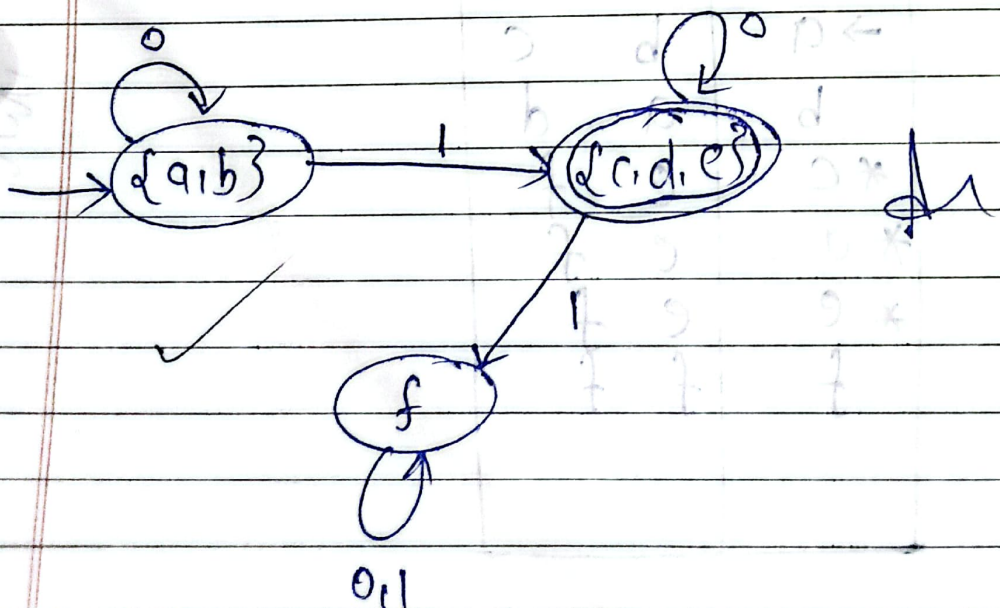
$\pi_2 = \{\{a\} \{b\} \{f\}, \{c, d, e\}\}$ - Two equivalr.

$\pi_3 = \{\{a, b\} \{f\} \{c, d, e\}\}$ -

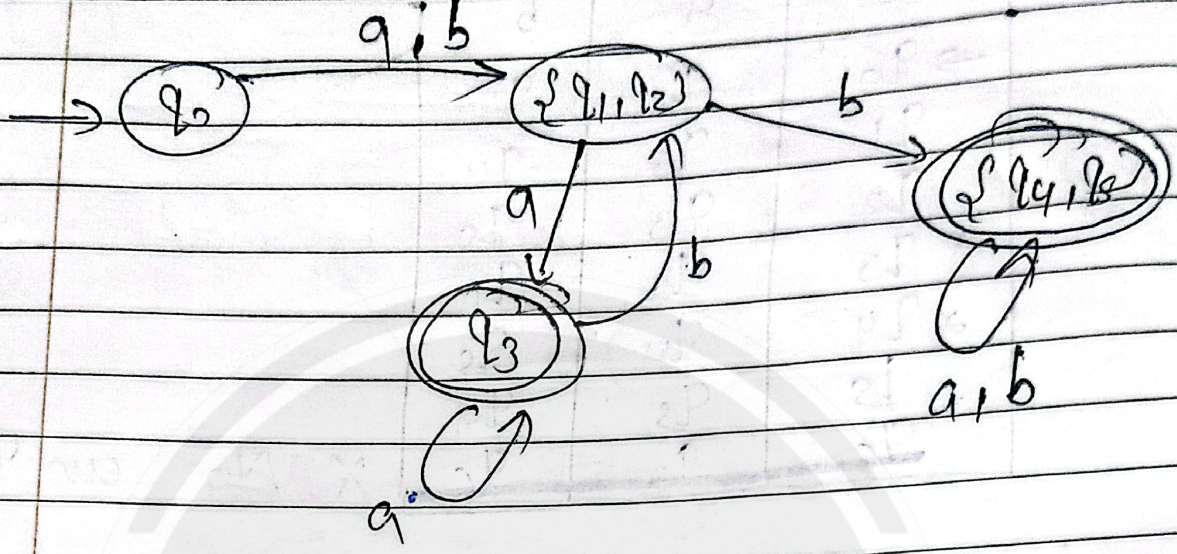
Now minimized transition diagram table

δ | Σ	a	1
→{a, b}	{a, b}	{c, d, e}
*{c, d, e}	{c, d, e}	f
f	f	f

Transition diagram -



Minimize Transition diagram -



Mon Question in TAFL Copy -

Q. How will you group the phases of compiler?

The phases of a compiler are generally grouped into two main categories

① Analysis phase (Frontend)

② Synthesis phase (Backend)

int x = "hello"; int x 10 =;

*** Que what are the various errors that occurs in the compilation process?

① Lexical Error (During lexical analysis)
↳ Occurs due to invalid Tokens or characters. Ex, int @num (@ x)

② Syntax Error (During Syntax analysis)

③ Semantic Error (During Semantic analysis)

④ Logical Error (Detected at Runtime) -

⑤ Runtime Error

⑥ Linking Error (During Linking phase)

Que/ Whats are the classification of Compiler?

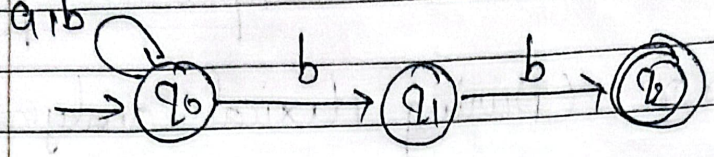
Based on Execution Method
- Native Compiler = Converts Code for the same platform
- Cross Compiler → Converts Code for diff platform.
- Source to Source Compiler (Transpiler) → Convert Code from one lang to another lang.

Based on Compiler process
- Single pass Compiler → Process the source code in one pass (fast but less optimize)
- Multipass Compiler → process the code in multiple stages (better optimization & error handling)

Based on Target language
- Machine Code Compiler → convert source code direct into machine code.
- Intermediate Code Compiler like (java)

NFA To D.F.A →

Q.



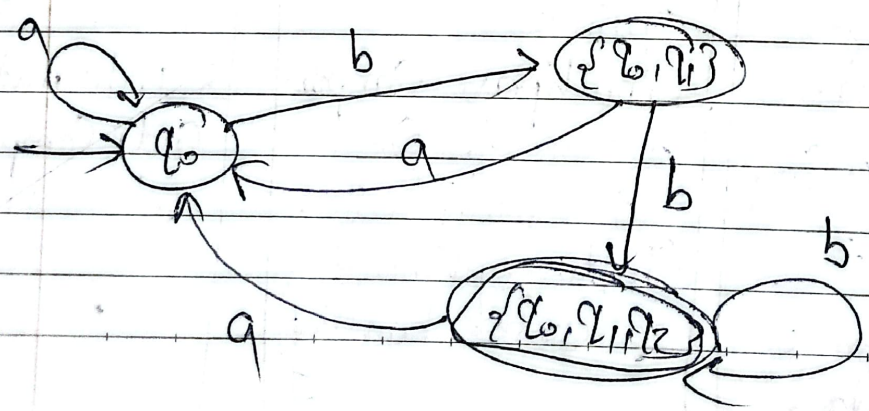
Sol: Transition Table of NFA.

δ | Σ	a	b
→ q_0	q_0	q_0, q_1
q_1	—	q_2
* q_2	—	—

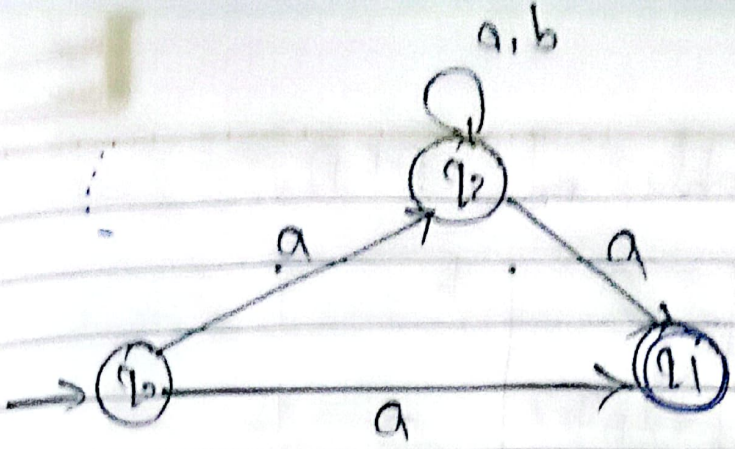
Now, Transition Table of DFA

δ | Σ	a	b
→ q_0	q_0	$\{q_0, q_1\}$
$\{q_0, q_1\}$	q_0	$\{q_0, q_1, q_2\}$
* $\{q_0, q_1, q_2\}$	q_0	$\{q_0, q_1, q_2\}$

D.F.A



Q 2



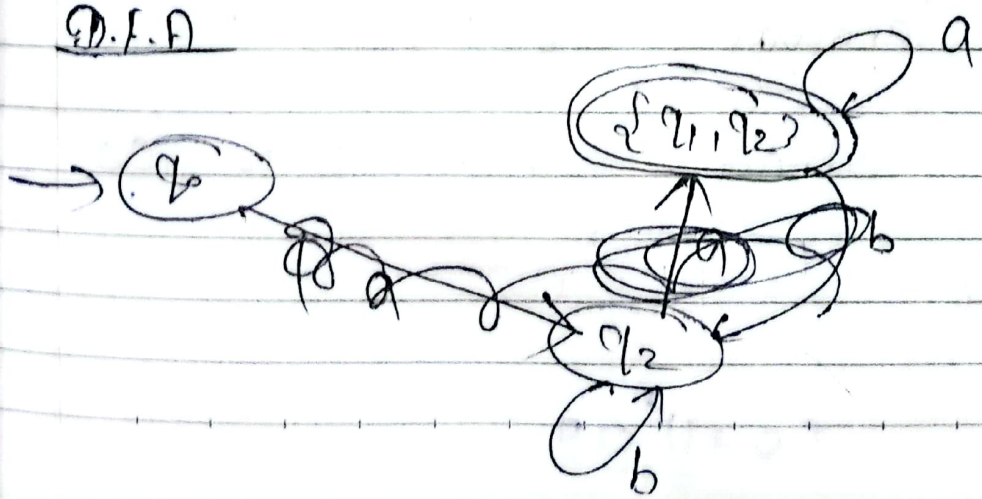
Sol^n Transition table of NFA :

δ / Σ	a	b
→ q_0	q_2, q_1	–
* q_1	–	–
q_2	q_2, q_1	q_2

Transition table of D.F.A –

δ / Σ	a	b
→ q_0	$\{q_2, q_1\}$	–
q_2	$\{q_2, q_1\}$	q_2
* $\{q_2, q_1\}$	$\{q_2, q_1\}$	q_2

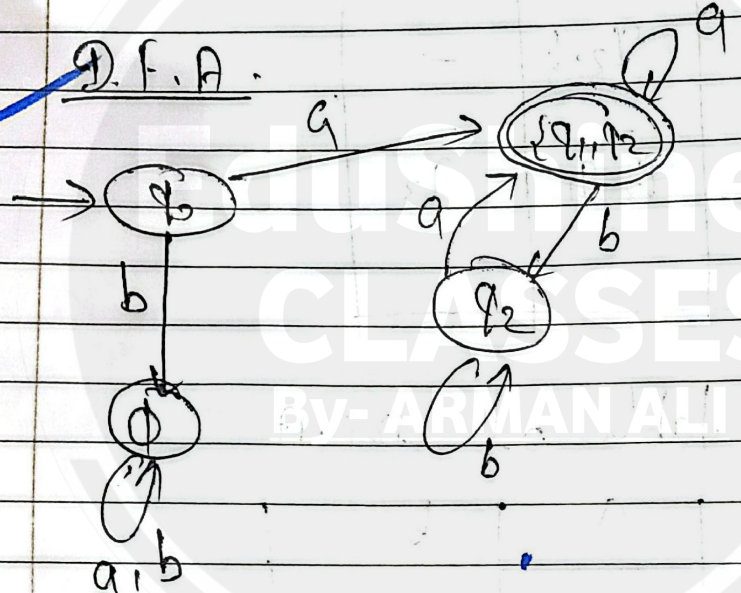
D.F.A



Transition table of DFA

δ | Σ	a	b
→ q_0	$\{q_1, q_2\}$	–
* $\{q_1, q_2\}$	$\{q_1, q_2\}$	q_2
q_2	$\{q_1, q_2\}$	q_2
∅	∅	∅

D.F.A.



* Regular language -
* Regular expression -

Q. Find the R.E for the following language - Σ = {0,1}

Q1 all string end with '00'

R.E :- $(0+1)^* 00$

$(0+1)^* 00$

The language accepted by Finite automata can be easily described by Simple expressions called 'Regular expression'

* It is the most effective way to represent any language.

* The language accepted by some regular expression are referred to as 'regular language'.

* A Regular expression can also be described as a sequence of pattern that defines a string.

* Regular expressions are used to match character combination in string. String searching algorithm used this pattern to find the operation on string.

In a Regular expression x* means 0 or more occurance of x. it can generate -
(ε, x, xx, xxx, xxxx ...)

b) all string beginning with 0 end with 1

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

c) $L = \{\wedge, 11, 1111, 111111 \cdots\}$

R.E = $(11)^*$

d) All strings have exactly two 0

R.E = $1^*01^*01^*$

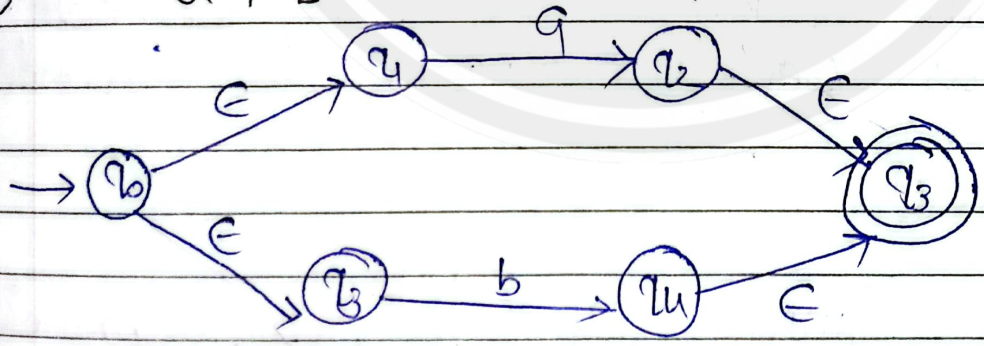
e) All string contain exactly atleast '00'

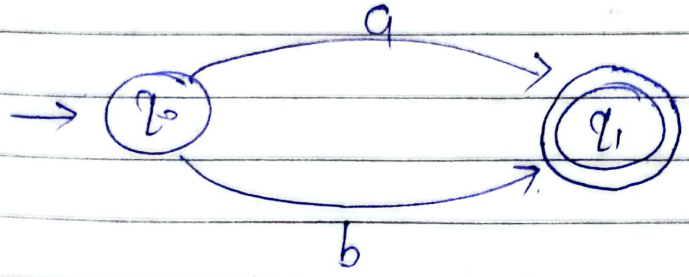
R.E = $(0+1)^*0(0+1)^*0(0+1)^*$

---

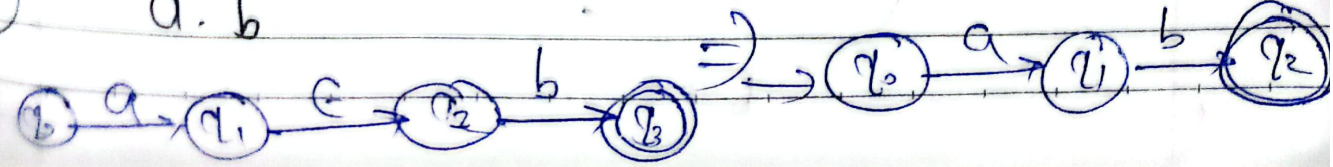
* Convert the R.E to Epsilon NFA.

① a + b

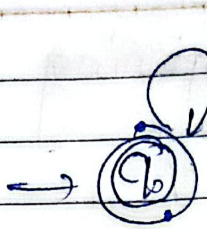




② a . b



③ . a^* -

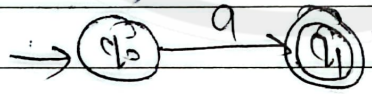


Q. Explain Thompson's construction -

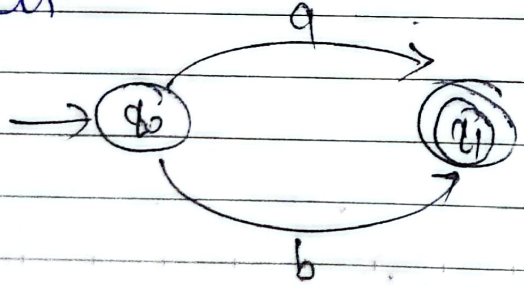
* It is an algorithm for transforming a regular expression to equivalent NFA
* Following rules are defined for a regular expression as a basis for the construction -

→ The NFA representing the empty string is -

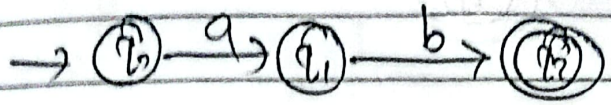
→ if the R.E just a character then it can be represented



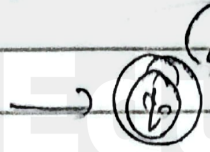
→ The union operator is represented by a choice of transitions from a node thus a+b can be repre- as -



→ Concatination simply involves connecting one NFA to the other thus ab can be represented as -

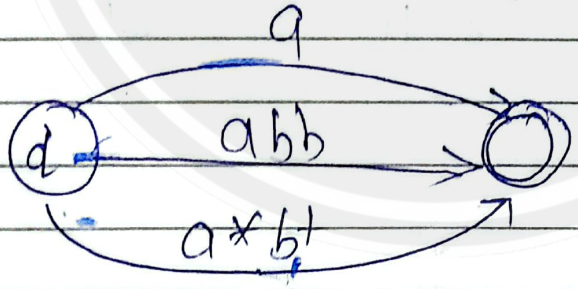


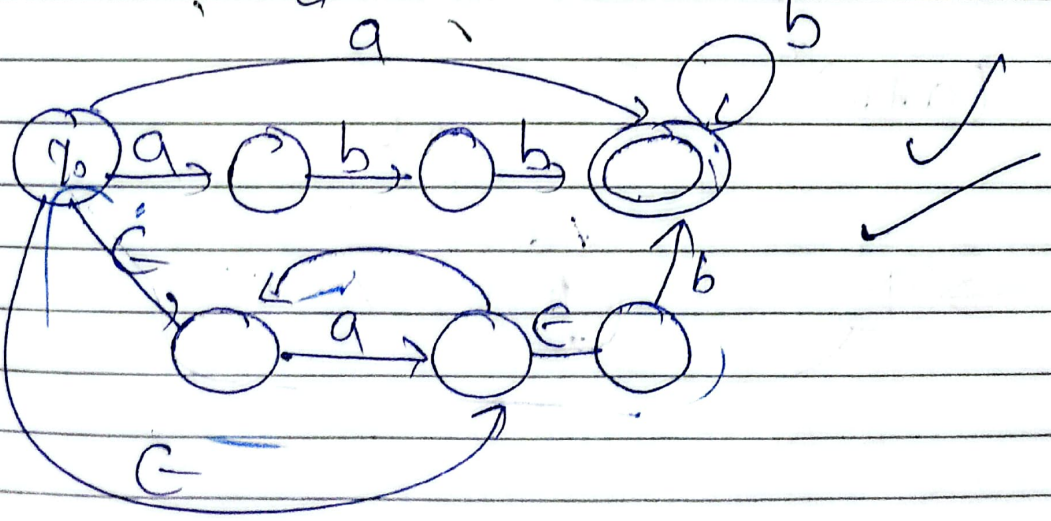
→ The Kleen closure must allow for taking zero or more instances of the letter from the input thus a^* looks like



Q. Construct the NFA for the R.E $a \mid abb \mid a^*b^+$ by using thompson

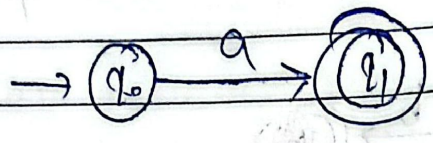
Sol^n



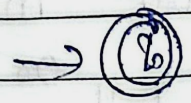


Q Convert the R.E to NFA

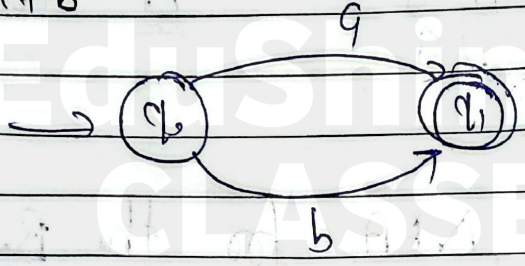
Q.1 a) a -



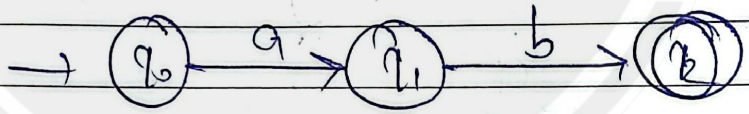
b) Epsilon ε



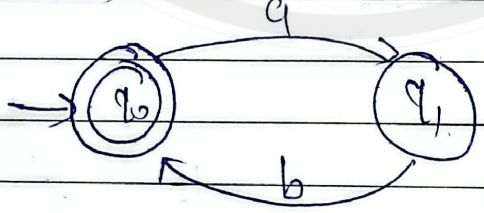
c) a+b



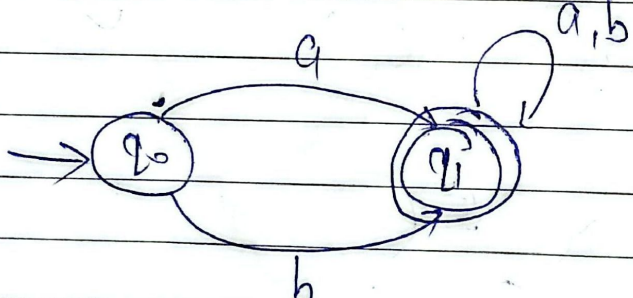
d) a.b



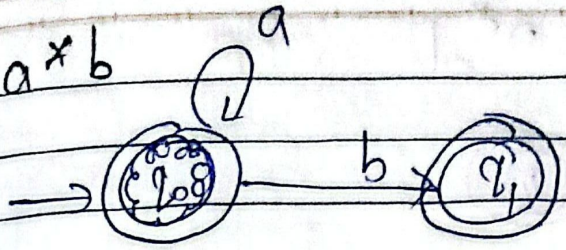
e) $(a.b)^*$



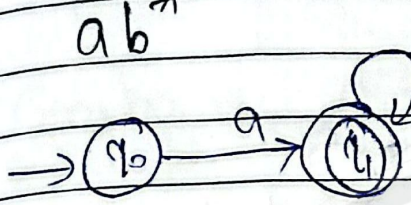
f) $(a+b)^*$



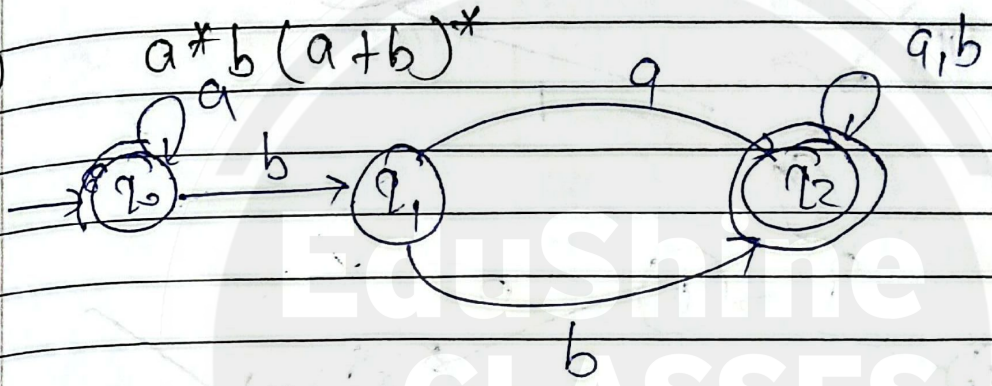
g) a^*b



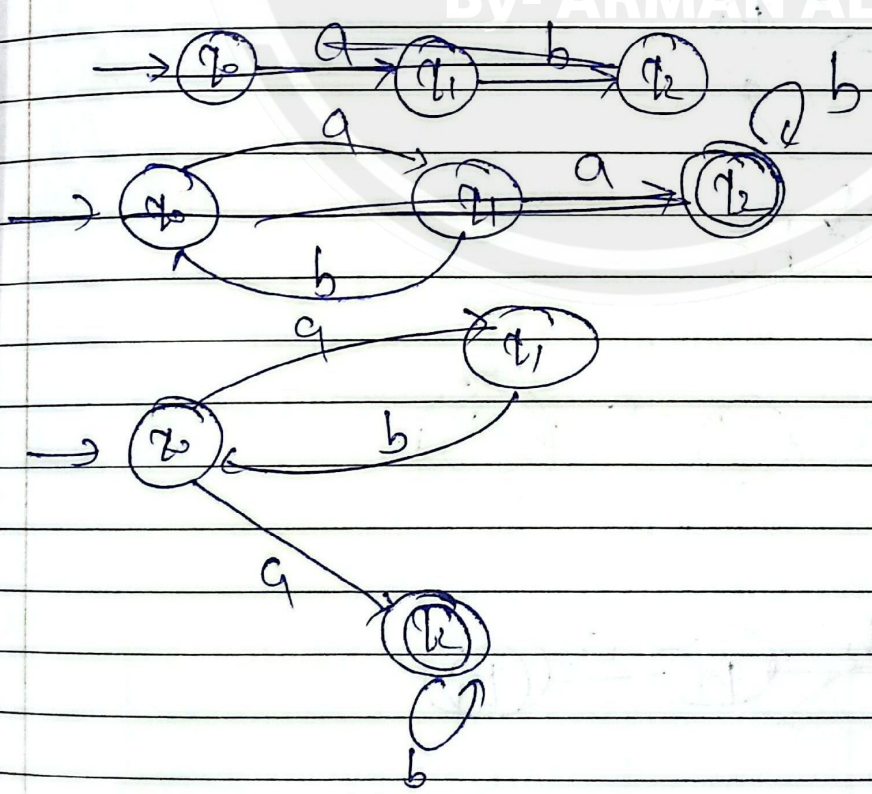
h) ab^*



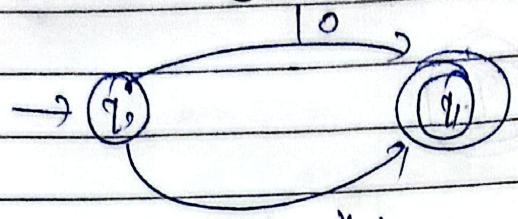
i) $a^*b(a+b)^*$



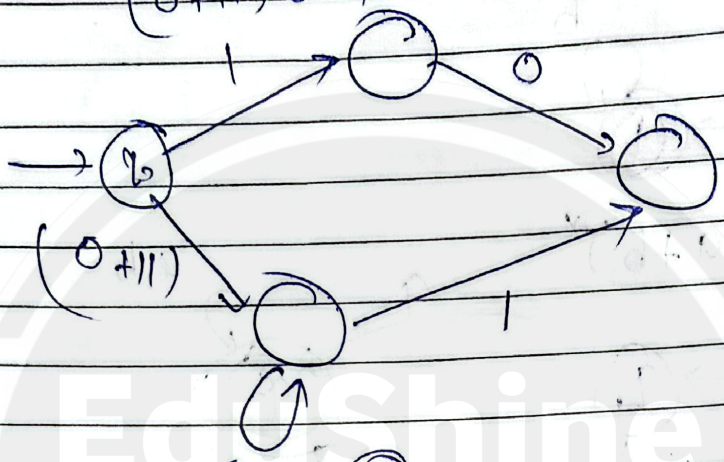
j) $(a.b)^*ab^*$

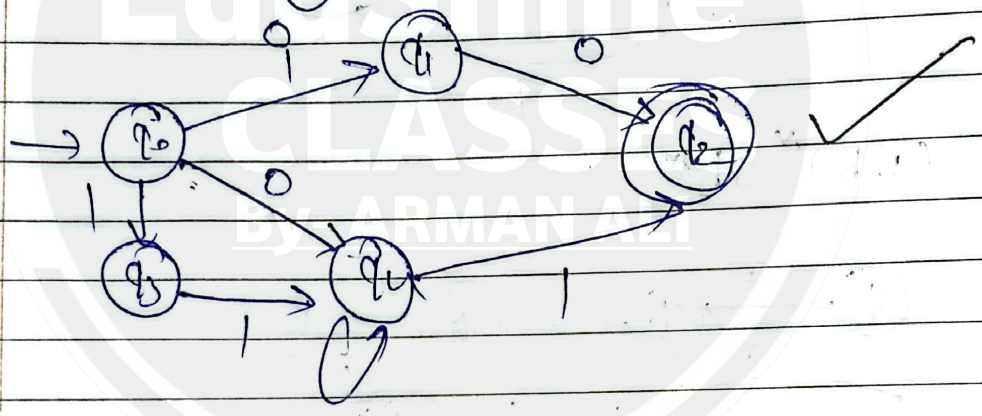


k) $10 + (0+11)\,0^*1$

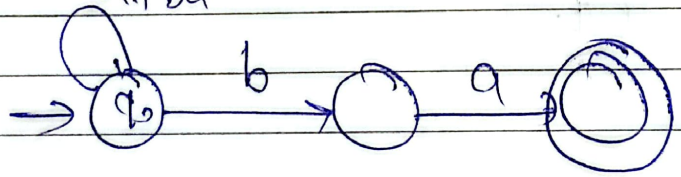


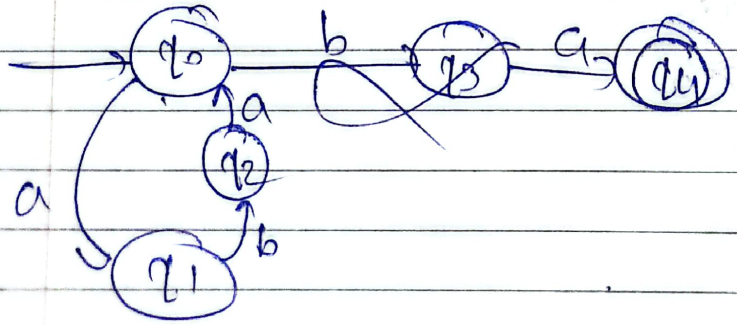
$(0+11)\,0^*1$

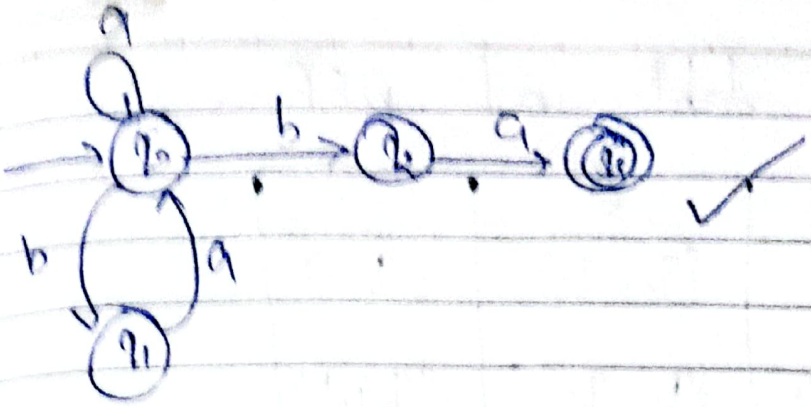




l) $(a+ba)^*ba$



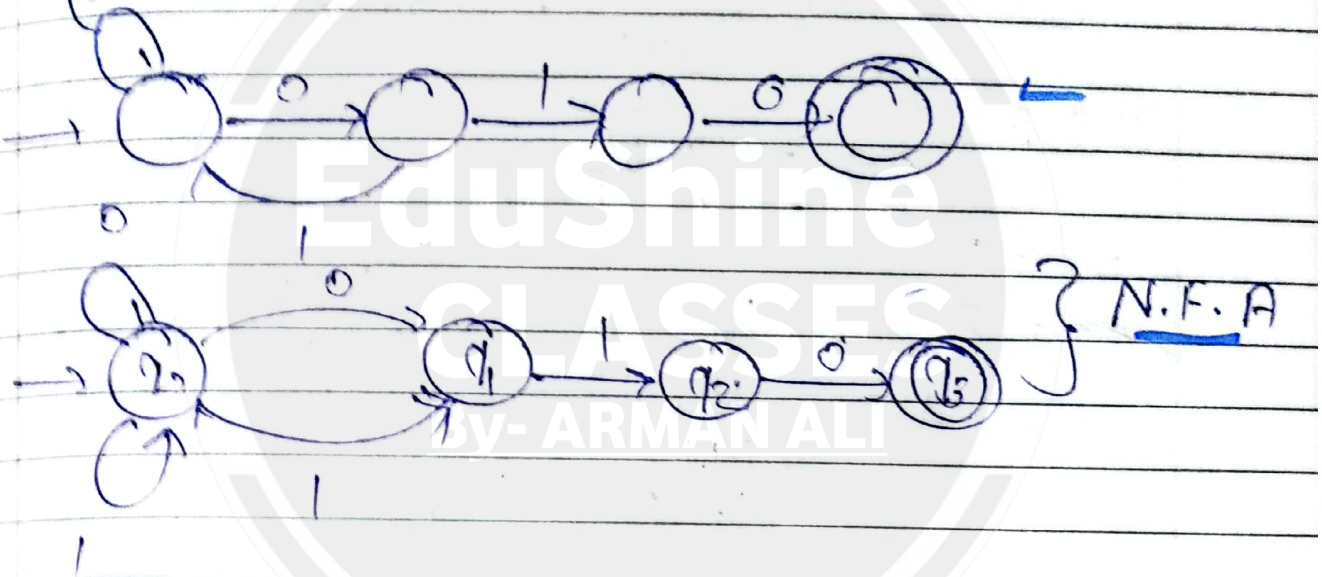




***

Q Construct NDFA for R.E $(0+1)^*(0+1)$

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



Transition table of NF.A

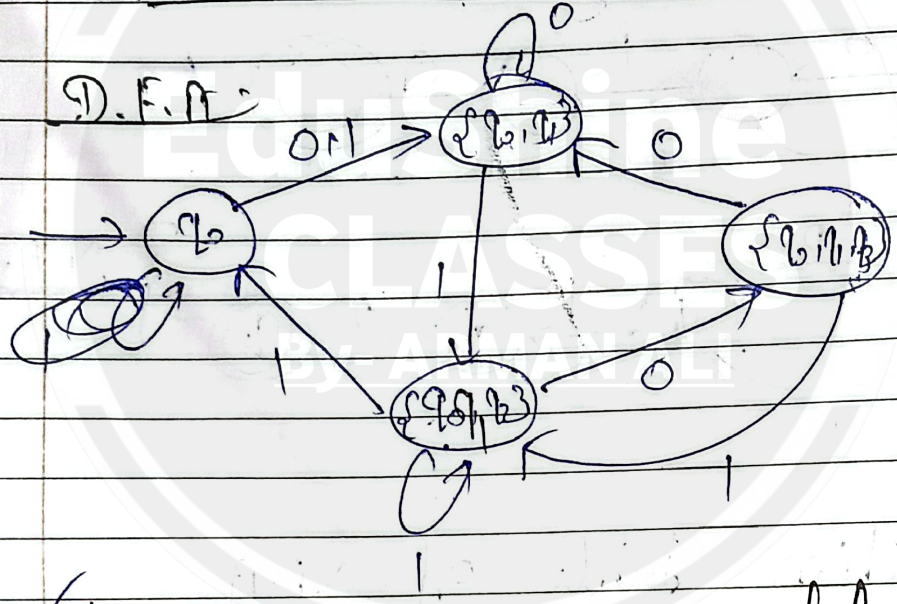
δ | Σ	0	1
→ q_0	q_0, q_1	$q_0 q_1$
q_1	–	q_2
q_2	q_3	–
* q_3	–	–

Now —

Transition table of DFA

δ/Σ	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2\}$
*$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

D.F.A:



✓ * How does F.A useful for lexical analysis

F.A are used in lexical analysis because -

→ They efficiently match patterns (like keywords, identifiers, numbers etc)

→ They can be constructed from R.E which describe the

Lexical analysis structure.

→ They operate in O(n) time, making them fast.

→ They help in Error detection by rejecting invalid tokens.

* Challenges During Compiler design -

① Correctness - The compiler must correctly understand and convert the code.

→ No Error in Conversion, O/P must match original logic.

② Efficiency of Code - The machine code generated should fast & use less memory

→ Generate fast & Optimized Code.

③ Error Handling - Detect & show useful error message to the programmer.

→ Helps user fix problems with clear error message.

④ Support for multiple Language - Be flexible & handle diff. Prog. lang.

⑤ Portable - The Compiler should work on diff. System (win, Lin. ...)

→ Compile once run anywhere.

⑥ Spud of Compilation - The Compiler should work fast.

→ Balance between spud & correctness

⑦ Security & Safety Gun, Safe Mode Code

* CFG

{ V, T, P, S }

Q.1 Construct a CFG -

i) $a^n b^n$ $n >= 0$ ab aabb aaabbb

$S \rightarrow \epsilon / aSb$ ✓

Here,

$V = \{S\}$

$T = \{\epsilon, a, b\}$

$P = \{ S \rightarrow \epsilon$
$S \rightarrow aSb \}$

$S = \{S\}$

Grammar -

* In the context of automata theory, a grammar is a set of rules that defines how strings in a language can be generated.

* It is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences.

Grammar is basically compand of two basic elements

1) Terminal Symbol
2) Non terminal Symbol

Terminal Symbols - Terminal Symbols are those which are the component of the sentences generated using a grammar & represented using small case letters like a, b, c ... etc.

* Symbols that appears in the final strings of the lang.

a, b, c

A.B.C

Non terminal Symbol -

* Symbol that can be replaced by groups of other Symbols they are called placeholders or variables.

* Non terminal Symbols are those Symbol which take part in the generation of sentence (string) but are not the component of the sentence.

* Non-terminal Symbols are also called as auxiliary Symbols and Variables. These Symbols are represented like - A, B, C ...

Formal Definition of Grammar -

Any Grammar can be represented by 4 tuples -

$$G = \langle N, T, P, S \rangle$$

N - Finite non empty set of non terminal Symbols

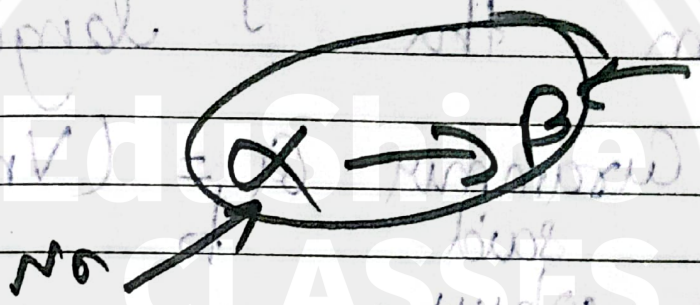
T - Finite set of terminal Symbol.

P. Finite non empty set of production rule

S.:- Starting Symbol (Symbol from where can start producing our sentences or string)

Production Rule -

Then rules



* A production or production rule in Computer Science is a rewrite rule specifying a Symbol substitution that can be perform to generate new Symbols Sequences.

→ It is of the form $\alpha \rightarrow \beta$ where α is non terminal Symbol which can be replaced by β. which is a string of terminal Symbols and Non terminal Symbols.

Context free Grammar (C.F.G)

* A Context free Grammar (CFG) is a system used to define all possibly strings in a given formal language
* It consist of a set of rules or productions that describes how to form valid strings from the language alphabets
* "A Grammar $G = (V_n, V_t, P, S)$ is said to be Context free when.

V_n: A finite set of Non terminal represented by capital letter.

V_T = A finite set of Terminal generally represented by small letter

S = Starting Non terminal is called start Symbol of Grammar

$\therefore S \in V_n$

P - Production rules in CFG

G is said to be Context free and all production in P have the form $\alpha \rightarrow \beta$

where : $\alpha \in V_n$

$\beta \in (V_t \cup V_n)^*$

A Grammar is said to be the C.F.G if every production is in the form of

$$G_1 \rightarrow (V \cup T)^*$$

where $G_1 \in V$

In the above example left-hand side of production i.e G_1 can only be a variable it can't be a terminal but on right side it can be a variable or terminal or both Combination of variable & terminal.

For example the Grammar

Exp- $A = \{S, a, b, P, S\}$ having Product

$S \rightarrow aS$
$S \rightarrow bSa$
but
$a \rightarrow bSa$
$a \rightarrow ba$

} it not a C.F.G as on the L.H.S there is terminal which doesn't follow the C.F.G Rule.

Derivation -

We now define the notations to represent the derivation.

First we define two notation and -

$\xrightarrow[G]{}$ and $\xrightarrow[a]{*}$

if $\alpha \to \beta$ is a production of P in C.F.G and a, b are strings in $(V_n \cup V_T)^*$

then, $a\alpha b \xrightarrow[G]{} a\beta b$

We say that the production $\alpha \to \beta$ is applied to the string $a\alpha b$ (to obtain) $a\beta b$

$a\alpha b \xrightarrow[G]{} a\beta b$

Language of Context free Grammar -

If $G = (V_n, V_T, P, S)$ is a C.F.G then language is denoted by $L(G)$ is the set of terminal strings that have derivation from the start Symbol.

$$L(G) = \{ \omega \text{ in } V_t \mid S \xrightarrow[a]{*} \omega \}$$

(99566) -R

if a language L is the language of some C.F.G G_1 then L is said to be Context free language or C.F.L

Ex Consider a Grammar $G_1 = (V_n, V_T, P, S)$ where

$V_n = \{S\}$

$V_T = \{a, b\}$

and set of production is given by

$P = \{S \to aSb, S \to ab\}$

find the appropriate language for the grammar.

Sol given $G_1 = (V_n, V_T, P, S)$

$V_n = \{S\}$

$V_T = \{a, b\}$

$P = \{S \to aSb, S \to ab\}$

$S \to aSb$

$S \to ab$

~~$S \to aabb$~~

$S \to aasbb$

$S \to aaabbb$

$L = \{a^n . b^n, n \geq 1, n \in N\}$

Computer N/W

L = {w | w is a palin

Here S is the only non-terminal which is the starting symbol for grammar a & b are terminals & there are two production.

$S \to aSb$

$S \to ab$

Now we have to derive string for above production.

SaS^b

Ques 2 Consider a Grammar $G = (V_n, V_T, P, S)$

$V_n = \{S\}$

$V_T = \{a, b, c\}$

and production key is given by

L_2 $P = \{S \to aSa, S \to bSb, S \to c\}$

A given

$V_n = \{S\}$

$V_T = \{a, b, c\}$

$P = S \to aSa$

$S \to bSb$

$S \to c$

$\Rightarrow S \to abSba$

a+b

$S \rightarrow abbSbba$

$S \rightarrow abbcbba$

$L = \{wcw^R\}$ Ans

$L = \{wcw^R / w \in (a+b)^*\}$

Ans Write a Context free Grammar (C.F.G) which generate strings having equal no of a's & b's.

$G = (V_n, V_t, P, S)$

$V_n = \{S\}$

$V_t = \{a, b\}$

$P = S \rightarrow aSbS \mid bSaS \mid \epsilon$

$S \rightarrow aSbS$

$S \rightarrow bSaS$

$S \rightarrow \epsilon$

$S \rightarrow abSaSbS$

$S \rightarrow abbSaSabSaS$

$S \rightarrow abb\epsilon a\epsilon a b\epsilon a\epsilon b\epsilon a\epsilon$

$= abababab$

***

Derivation Tree / Parse Tree -

* Derivation tree is a graphical representation for the derivation of the given production rule for given CFG.
* There is a tree representation for derivation that has proved extremely useful.
* It is second way of showing derivation, independent of the order in which productions are used, is also called derivation tree.
* A parse tree is an ordered tree in which nodes are labelled with the left side of production & in which the children of node represents its corresponding right side.

Definition.

Let $G = (V_N, V_T, P, S)$ be a C.F.G and ordered tree of this C.F.G is the derivation tree if and only if it has following property.

① The root is labelled by the starting node terminal 'S'

② Every leaf of the order tree has a label from $V_t \cup \{\in\}$

③ Every interior node of the order tree has a label from V_n.

④ Let's assume that a vertex has label $X \in V_n$ and its children are labelled from L to R $Y_1, Y_2, Y_3 \ldots Y_n$ then production must contain of the form $X \to Y_1, Y_2, \ldots Y_n$

⑤ A leaf labelled null has no siblings i.e a vertex with a child labelled null can have no other children

* It is the simple way to show how the derivation can be done to obtain substring of given set of production rule.

* The derivation tree is also called parse tree

Derivation:

Derivation Tree - LMD RMD

LMD

Ex - $E \to E+E$
$E \to E-E$
$E \to a \mid b$

I/P - a-b+a

Sol^n $E \to E+E$ (by $E \to E-E$)

$E \to E-E+E$

$E \to a-b+a$

RMD

Ex $E \to E+E$
$E \to E-E$
$E \to a \mid b$

I/P - a-b+a

Sol^n $E \to E+E$

$E \to E-E+E$

$E \to E-E+a$

$E \to E-b+a$

$E \to a-b+a$

Q2 Derive the string 'abb' for the leftmost derivation & R.M.D using CFG

Sol^n

$S \to AB \mid \in$

$A \to aB$

$B \to Sb$

$S \to aB$

$S \to aSb$ (by ~~$S \to \in$~~)

LMD

~~$S \to aAB$~~

$S \to AB$

$S \to aBB$

$S \to aSbSb$

$S \to aBb$

$S \to aSb$

$S \to aEbEb$

$S \to abb$

RMD

$S \to AB$

$S \to ASb$ (by $B \to Sb$)

$S \to aBSb$ (by $BA \to aB$)

$S \to aSbSb$ (by $B \to Sb$)

$S \to abb$ ✓ (by $S \to \in$)

Derivation Tree

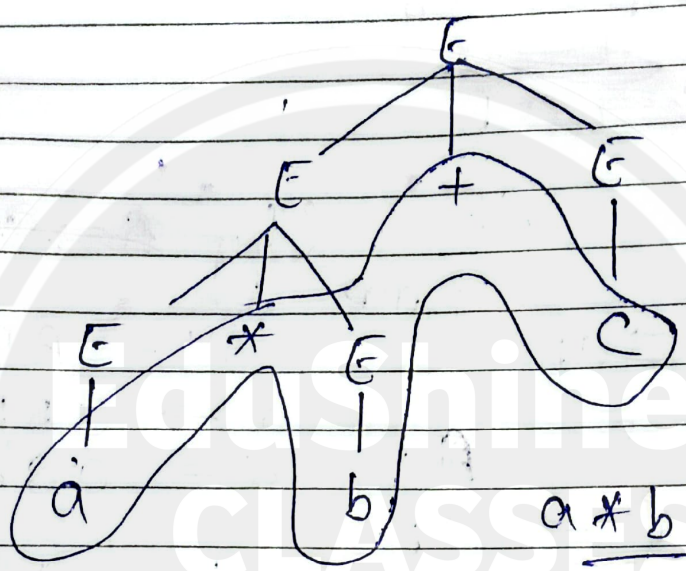
Q.1 $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow a \mid b \mid c$

I/P → $a*b+c$

Sol^n

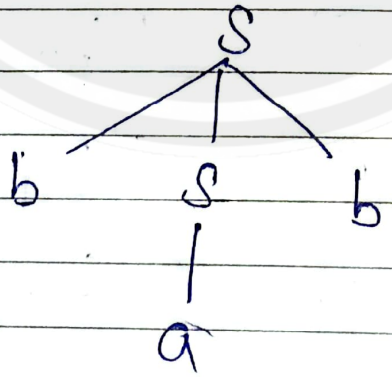


$a*b+c$

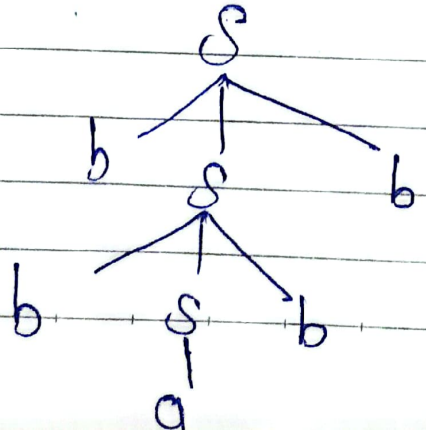
Q2 $S \rightarrow bSb \mid a \mid b$

I/P - 'bab', 'bbabb'

Sol^n



'bab' ✓



'bbabb' ✓

Ambiguity Grammar -

Q. $E \to E+E$
$E \to E*E$
$E \to id$
$w = id + id * id$

check ambiguity?

Sol^n

LMD

i) $E \to E+E$

(1) $E \to E + E*E$ (by $E \to E*E$)

$E \to id + id * id$ (by $E \to id$)

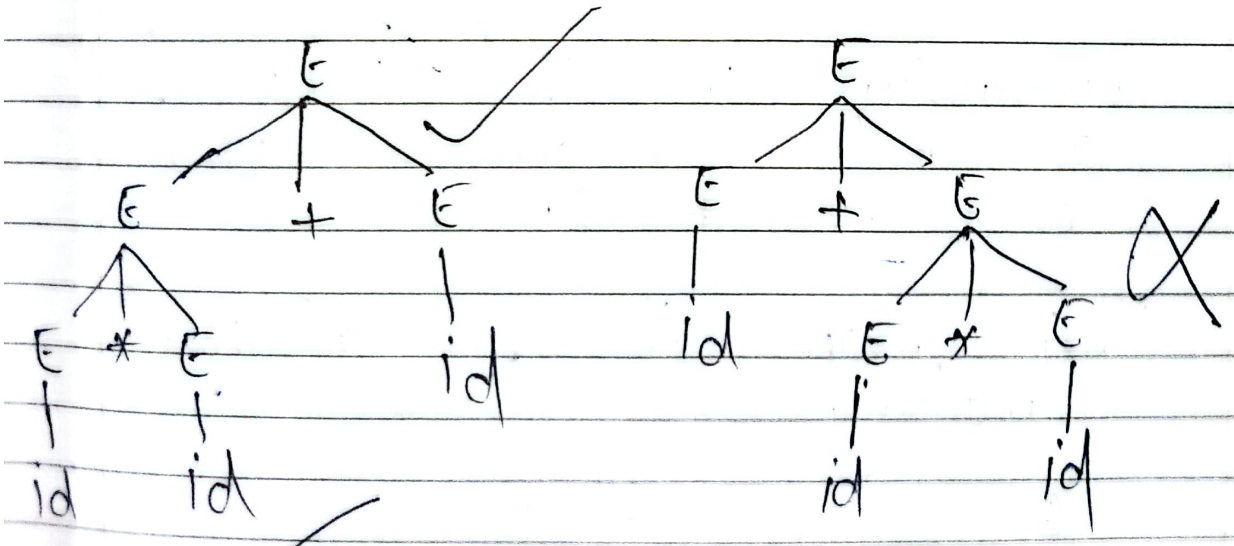
ii) $E \to E*E$

(2) $E \to E+E*E$ (by $E \to E+E$)

$E \to id + id * id$ (by $E \to id$)

Here two LMD exist for the given grammar. So it is ambiguous.

Derivation tree -



Q.2 check whether Gr. is ambiguous or not.

$S \rightarrow aSb$
$S \rightarrow SS / \epsilon$
I/P - 'aabb'

Sol^n i) $S \rightarrow aSb$

$S \rightarrow aaSbb$ (by $S \rightarrow aSb$)

$S \rightarrow \underline{aabb}$ (by $S \rightarrow \epsilon$)

ii) $S \rightarrow SS$

$S \rightarrow aSbS$ (by $S \rightarrow aSb$)

$S \rightarrow aaSbbS$ (by $S \rightarrow aSb$)

$S \rightarrow \underline{aabb}$ (by $S \rightarrow \epsilon$)

Types of Grammar -

1. Recursive Grammar -

* A Grammar is recursive if the L.H.S of a Rule appears again on the R.H.S

* This mean the rule calls itself making it recursive rule.

Ex - $A \rightarrow A + B$

Here A appears again on the R.H.S making it recursive.

② Non-Recursive Grammar -

* A Grammar is non-recursive if the L.H.S doesn't appear again on the R.H.S

* This Rule doesn't call itself.

Ex $A \rightarrow B+C$

When a Grammar is recursive, it can be of two types -

i) Left Recursion -

if a non-terminal (A) appears first on the R.H.S before anything else, it is left recursive.

Ex- $A \rightarrow A+B$

Here A is first on the R.H.S, making it left Recursive.

Problems -

* Left Recursion causes infinite loops in top down parser.

* It needs to be removed to avoid error

Solution. Convert it to Right Recursion

ii) Right Recursion - * if the non-terminal appears later on the R.H.S (not at the start) it is right recursion.

Ex - $A \rightarrow B+A$

* It doesn't cause infinite loops in parsing.

Steps to Remove Left Recursion —

Q. 1 $A \rightarrow A\alpha / \beta$ (Left Recursion)

Sol i) Create a new non-terminal

ii) Rewrite Rule —

$A \rightarrow \beta A'$ (R)

$A' \rightarrow \alpha A' / \epsilon$ (Right Recursion)

2) $S \rightarrow Sa / b$

$S \rightarrow bS'$

$S' \rightarrow aS' / \epsilon$ (Right Recursion)

3) $A \rightarrow A+B \mid C$

$A \rightarrow CA'$

$A' \rightarrow +BA' \mid \epsilon$ (Right Recursion)

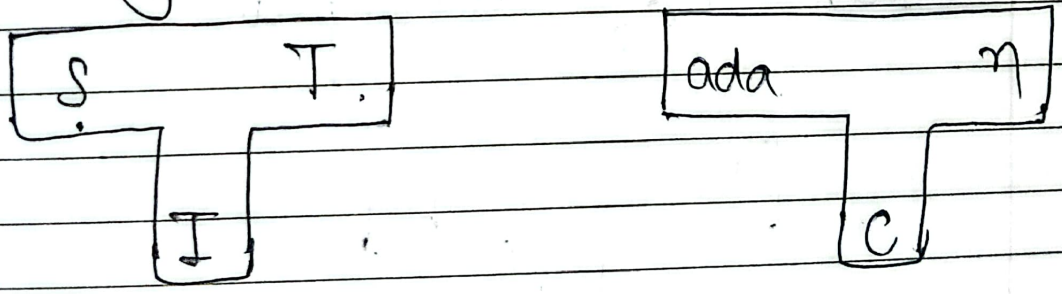
* Bootstrapping -

* Bootstrapping is a process of writting a compiler in the source programming language that it intends to Compile.

* This technique leads to self hosting compiler, the language basic C, Java, LISP, Python are bootstrapped

* A Compiler is characterized three languages its source language its target language & its implimented language.

T - Diagram -



④.

BNF Notation -

* BNF (Backus Naur Form) is a way to write rule for progr. lang. or grammar of any lan. in structured and formal way.

⇒ discrib how language is made.

BNF Defin -

⇒ Syntax of progr lang.
⇒ Rule to form Statmt & exp.
⇒ Structure of Compiler or interpre

BNF Structure

- Angle bracket <rule>
- ::= which means is "defined as"
- | (OR)

Eg. <expr> ::= <term> | <term> + <expr>

<term> ::= id | num

* YACC (Yet Another Compiler Compiler)

* YACC is a tool that makes parser using grammar rules automatically

→ YACC is the part of Compiler construction.

Subscribe

(Edushine Classes)