Alphabet: Σ - collection of symbols, Eg: $\{a,b\}, \{d,e,f,g\}$...

String: Sequence of symbols: Eg: a, abc, aabc, abbad...

Language: Set of strings, Eg:

Eg: $\Sigma = \{0,1\}$.

L_1 = Set of all strings of length 2 (Finite lang.)

$= \{00, 01, 10, 11\}$

L_2 = Set of all strings that begins with 0 (Infinite lang)

$= \{0, 00, 011, 0101, 00100, \ldots\}$

Powers of Σ: Let $\Sigma = \{0,1\}$

Σ^0 = Set of all strings of length 0 : $\Sigma^0 = \{\epsilon\}$

Σ^0 = " " " " 1 : $\Sigma^1 = \{0,1\}$.

Σ^2 = " " " 2 : $\Sigma^2 = \{01, 10, 00, 11\}$

Σ^n = " " " n :

Cardinality: no. of elements in a set.

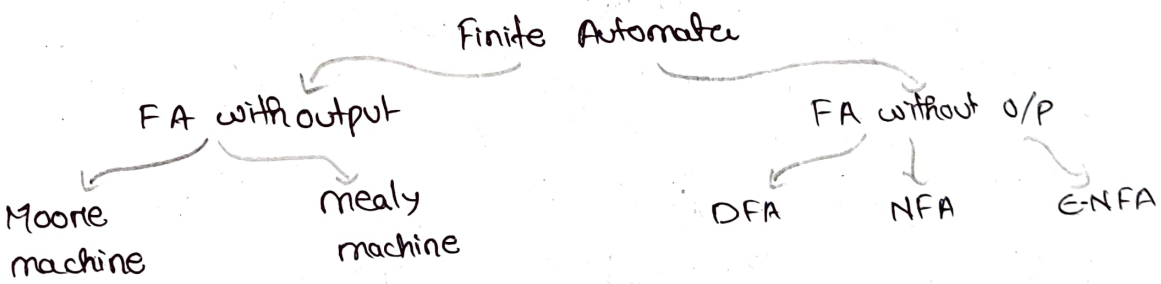
For $\in \Sigma^n$ cardinality $= 2^n$

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$

$= \{\epsilon\} \cup \{0,1\} \cup \{00, 01, 10, 11\} \cup \ldots$

= Set of all possible strings of all lengths over $\{0,1\}$.

Finite State Machine



* DFA - Deterministic Finite Automata.

→ It is simplest model of computation.

→ It has a very limited memory.

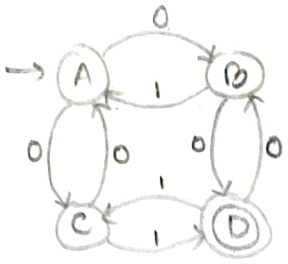
→ Every DFA can be defined using 5 tuples

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

$Q \rightarrow$ Set of all states $q_0 \rightarrow$ start/initial state

$\Sigma \rightarrow$ inputs $F \rightarrow$ Set of final state

$\delta \rightarrow$ transition function from $Q \times \Sigma \rightarrow Q$



$Q = \{A, B, C, D\}$

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

$q_0 = \{A\}$

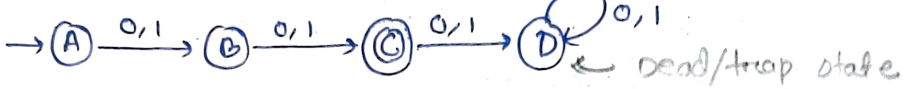
$F = \{D\}$

$\delta = Q \times \Sigma \to Q$

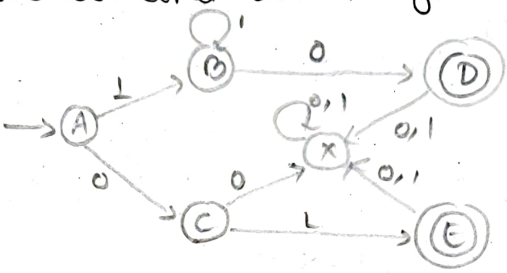
	0	1
A	C	B
B	D	A
C	A	D
D	B	C

Q construct a DFA that accepts set of all strings over $\{0,1\}$ of length 2.

$\Sigma = \{0, 1\}$. $L = \{00, 10, 01, 11\}$.



Q Figure out what DFA Recognizes.



L = { Accepts the string 01 or a string of at least one '1' followed by a '0' }.

* Regular language :

→ A lang. is said to be regular if and only if some finite state machine recognizes it.

→ Not regular languages:
- Which are not recognized by any FSM.
- Which req. memory.

→ Finite state machine has limited memory and it cannot store or count strings.

Eg: $a^N b^N$ is not regular lang., aba → regular

→ Operation on Reg. lang.

Union : $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.

compliment of Reg. lang is also Reg. lang.

Concatenation: $A.B = \{xy \mid x \in A \text{ and } y \in B\}$.

Star: $A^* = \{x_1 x_2 x_3 \cdots x_k \mid k \geq 0 \text{ and } x_i \in A\}$.

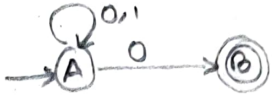
Theorem 1:
The class of Reg. lang. is closed under UNION

Theorem 2:
The class of Reg. lang. is closed under concatenation.

DFA	NFA
→ In DFA Given the current state we know what the next state will be	→ IN NFA, given the current state there could be multiple next state
→ It has only one unique state	→ The next state may be chosen at random
→ It has no choices or randomness	→ All the next state may be chosen parallel.
→ It is simple and easy to design.	

* NFA - (Non- Deterministic Finite Automata).

Q → Set of all states
Σ → Inputs
q_0 → Start/ initial state
F → Set of final state
δ → $Q \times \Sigma \to 2^Q$



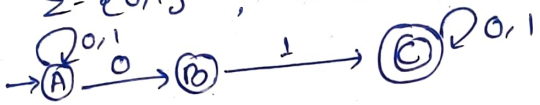
L = {Set of all strings that ends with 0}

→ If there is any way to run the machine that ends in any set of states out of which atleast one state is a final state, then NFA accepts

→ For a state, if next state is not given with particular input then take it as it leads to dead/ trap state.

Q Design NFA that contains '01' substring.

Σ = {0,1}, L = {Set of all strings that contain '01'}.



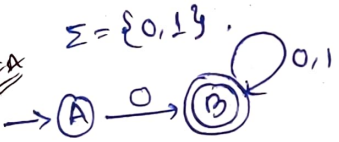
* Conversion of NFA to DFA.

→ Every DFA is an NFA, but not vice versa, but there is an equivalent DFA for every NFA.

Q L = {Set of all strings over (0,1) that starts with '0'}.

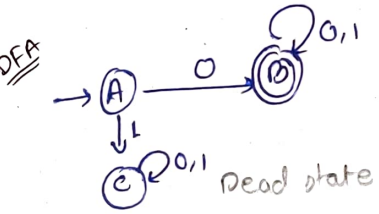
Σ = {0,1}.

NFA



	0	1
A	B	∅
B	B	B

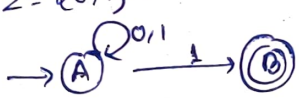
DFA



	0	1
A	B	C
B	B	B
C	C	C

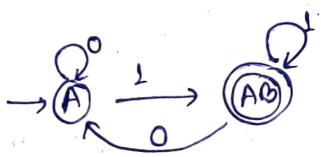
Q. L = {Set of all strings over (0,1) that ends with '1'}.

NFA Σ = {0,1}.



	0	1
→ A	{A}	{A,B}
B	∅	∅

DFA



	0	1
→ A	{A}	{AB}
AB	{A}	{AB}

AB → single state.

• Points for conversion of NFA to DFA

→ Start with initial state., if it can go to multiple state for same input then combine all them and make new state.

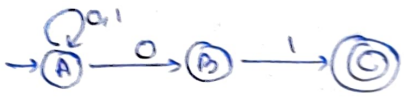
→ take only those states which are reachable from the states visited till now.

→ When new state is created, then next state from this state for a particular input is the union of next state of all individual states which are combined in this.

→ End state are the one which contains the state which is end state in NFA.

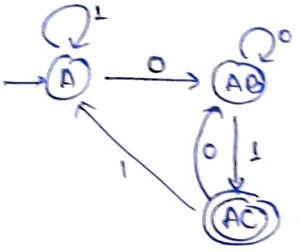
Q. L = {set of all strings over (0,1) that ends with '01'}

NFA



	0	1
→ A	A,B	A
B	ϕ	C
(C)	ϕ	ϕ

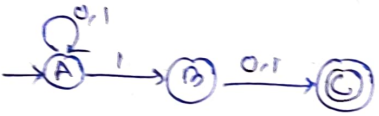
DFA



	0	1
→ A	AB	A
AB	AB	AC
(AC)	AB	A

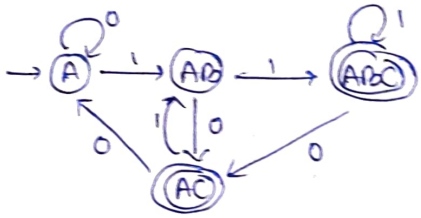
Q. Design an NFA for a lang. that accepts all strings over {0,1} in which the 2nd last symbol is always '1'. Convert it to DFA

NFA



	0	1
→ A	A	AB
B	C	C
(C)	ϕ	ϕ

DFA



	0	1
→ A	A	AB
AB	AC	ABC
(AC)	A	AB
(ABC)	AC	ABC

→ Minimization of DFA

Minimization of DFA is req. to obtain the minimal version of any DFA which consists of the minimum no. of states possible.

This can be done by combining the states. This can be only done when two states are equivalent.

• Two states 'A' and 'B' are equivalent if:

$\delta(A,x) \rightarrow F$ and $\delta(B,x) \rightarrow F$ · or · $\delta(A,x) \not\rightarrow F$ and $\delta(B,x) \not\rightarrow F$

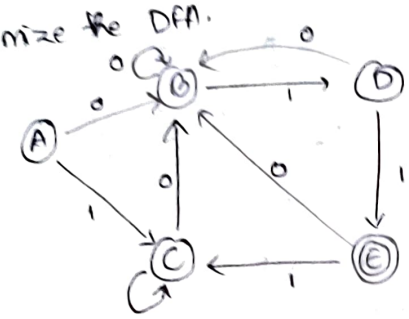
'x' → any input string.

means either A and B on getting same input both reach to final state or both not reach final state

If $|x| = 0$, A and B are said to be 0 equivalent, where $|x|$ is the length of string.

If $|x| = n$, " " " " n-equivalent., n is the length

2. Minimize the DFA.



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

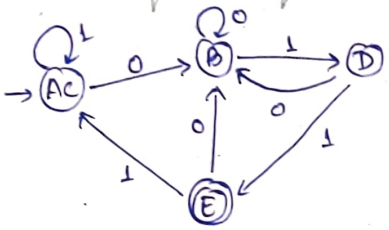
Sol 0 equivalence: {A, B, C, D} {E}
(Simply seperate the final and non-final states)

1 equivalence: {A, B, C} {D} {E}.
(Start picking states in pairs from previous equivalence, if the next states for both for a particular input lies in same block then they are equivalent else seperate them. If say A and B are equivalence then for C, check either with A or B, if its equil. then put in the same grp.)

2 equil: {A, C} {D} {E}. {B}

3 equil: {A, C} {D}, {E}, {B}

(If two consequtive equivalence are same then stop)



	0	1
→ AC	B	AC
B	B	D
D	B	E
(E)	B	AC

Q.

	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

0 equivalence: $\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}\{q_2\}$

1-equivalence:
$\{q_0, q_4, q_6\}\{q_1, q_7\}\{q_3, q_5\}\{q_2\}$

2-equivalence:
$\{q_0, q_4\}\{q_6\}\{q_1, q_7\}\{q_3, q_5\}\{q_2\}$

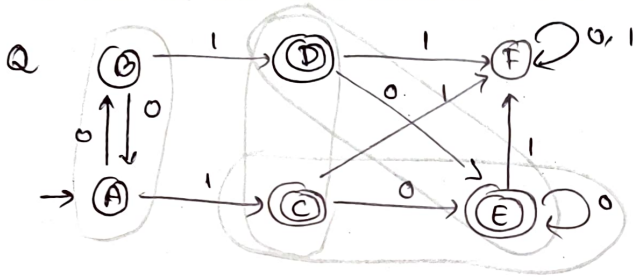
3-equivalence:
$\{q_0, q_4\}\{q_6\}\{q_1, q_7\}\{q_3, q_5\}\{q_2\}$

	0	1
→	$\{q_0, q_4\}$	$\{q_1, q_7\}$ $\{q_3, q_5\}$
	$\{q_6\}$	$\{q_6\}$ $\{q_0, q_4\}$
	$\{q_1, q_7\}$	$\{q_6\}$ $\{q_2\}$
	$\{q_3, q_5\}$	$\{q_2\}$ $\{q_6\}$
	($\{q_2\}$)	$\{q_0, q_4\}$ $\{q_2\}$

→ If there is any unreachable state in the DFA, simply remove it and proceed further.

→ My-hill - Nerode Theorem:

1. Draw table for all pairs of states (P, Q).
2. Mark all pairs where P ∈ F and Q ∉ F (by checking all inputs if any satisfy)
3. If there are any unmarked pair (P, Q) such that [δ(P, x), δ(Q, x)] is marked, then mark [P, Q], where 'x' is an input. Repeat this until no more marking can be made.
4. Combine all the unmarked pairs and make them a single state in the minimized DFA



	A	B	C	D	E	F
B	~					
C	✓	✓				
D	✓	✓				
E	✓	✓				
F	✓	✓	✓	✓	✓	

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

* Finite Automata with O/P

Mealy machine

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

where,

Q = Finite set of states

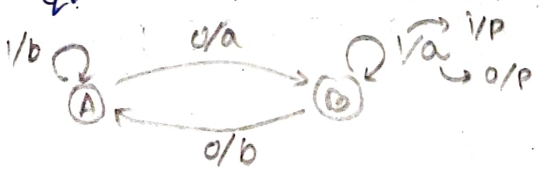
Σ = Finite non-empty set of i/p

Δ = Set of o/p alphabet.

δ = Transition function: $Q \times \Sigma \to Q$

→ λ = Output fun., $\Sigma \times Q \to \Delta$

q_0 = initial/start state,



Eg: 1010

→ A → A → B → B → A

o/p b a a b

Moore machine

$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

where,

Q = Finite set of state

Σ = Finite non-empty set of i/p

Δ = Set of o/p alphabet

δ = Transition fun., $Q \times \Sigma \to Q$

→ λ = o/p fun. $Q \to \Delta$

q_0 = initial/start state

Eg: 1010

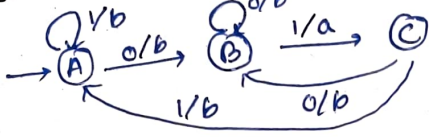
→ A → A → B → A → B

o/p a a b a b

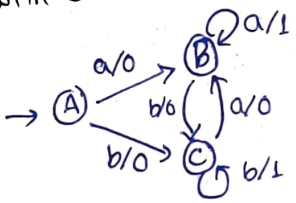
→ In mealy machine, O/P is state + input dependent.
→ In moore machine, o/p is state dependent only.
→ In mealy output length = length of I/P. , In moore , O/P length = length of I/P + 1.
→ No final state in these machines.

Q. construct a mealy machine that produces the 1's compliment

Q. construct a mealy machine that prints 'a' whenever the seq. '01' is encounted in any input binary string.

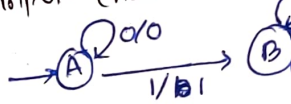


Q. Design a mealy machine accepting lang. consisting of string from Σ^*, where $\Sigma = \{a, b\}$ and the strings should ~~be~~ end with either aa or bb.



aa - 1
bb - 1

Q construct a mealy machine. that gives 2's compliment of any binary input. (Assume that the last carry bit is neglected).

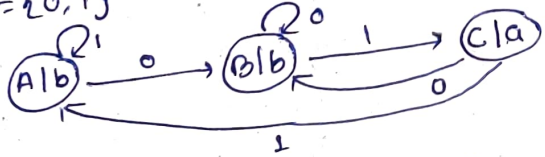


100 11 ← we are moving in this dir.
2's compliment -- 1's comp + 1

100110 → 01110
1's compliment

Q Moore machine, prints 'a' whenever the sequence '01' is encountered.

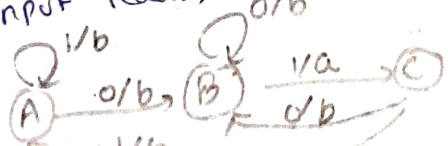
$\Sigma = \{0, 1\}$ $\Delta = \{a, b\}$



→ Mealy machine and moore machine are interconvertable.

Moore to mealy:
Put the ~~out~~ O/P associated with state to the input which are leading to this state.

Mealy to moore:
Put the O/P associated with I/P to the state where this input leads, some xtra states may form here.

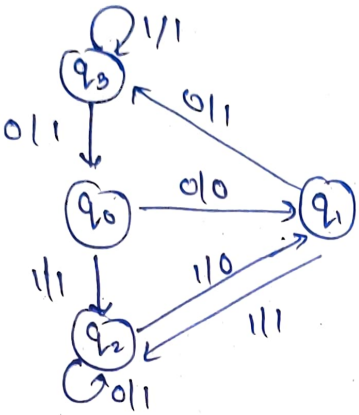


Previous question conversion.

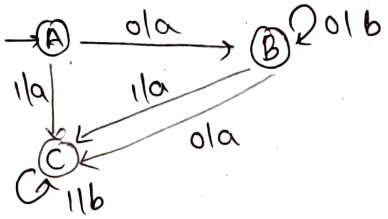
State	0	1	output
q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1

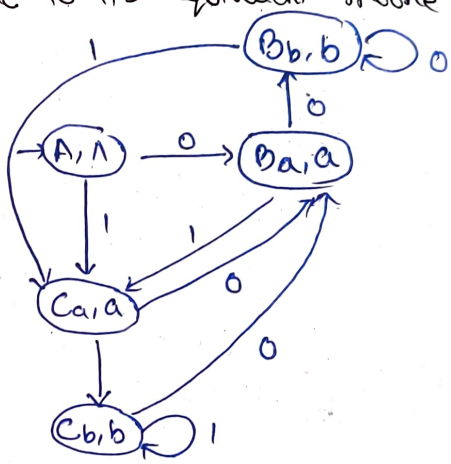
$\Sigma = \{0,1\}$, $\Delta = \{0,1\}$

	0	1
→ q_0	$q_1, 0$	$q_2, 1$
q_1	$q_3, 1$	$q_2, 1$
q_2	$q_2, 1$	$q_1, 0$
q_3	$q_0, 1$	$q_3, 1$



Q Conver the mealy machine to its equivalent moore machine.





→ More → mealy
No xtra state formed.

→ Mealy → moore
(x state, y inputs)
At max (xy) state can be present

Q convert the given mealy machine to its equivalent moore mach.

state	a	b
→ q_0	$q_3, 0$	$q_1, 1$
q_1	$q_0, 1$	$q_3, 0$
q_2	$q_2, 1$	$q_2, 0$
q_3	$q_1, 0$	$q_0, 1$

q_1 → q_{10} (0), q_{11} (1)

q_2 → q_{20} (0), q_{21} (1)

State	a	b	output
→ q_0	q_3	q_{11}	1
q_{10}	q_0	q_3	0
q_{11}	q_0	q_3	1
q_{20}	q_{21}	q_{20}	0
q_{21}	q_{21}	q_{20}	1
q_3	q_{10}	q_0	0

* Epsilon (ϵ)-NFA :

ϵ-NFA
↳ empty symbol

$\{Q, \Sigma, q_0, \delta, F\}$

$\delta : Q \times \Sigma \cup \epsilon \rightarrow 2^Q$

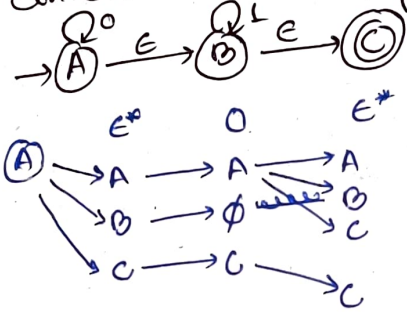
→ Every state on ϵ goes to itself even if its not given.

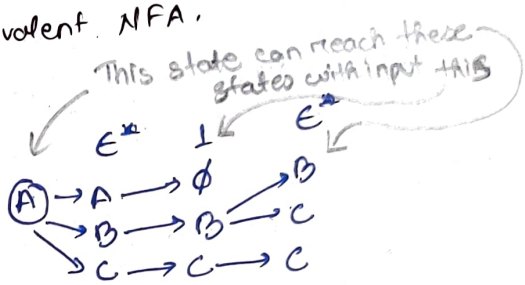
→ ∈NFA to NFA:

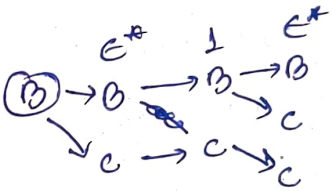
1. See where a state can go on looking ϵ^*.
2. For all state which ~~are~~ we got in previous step, check where they can go on seeing a particular input.
3. States which we got in step 2, check where they can go on looking ϵ^* ~~again~~ again.
4. States which ~~we~~ got in step3. are the ~~ste~~ states where one particular state can go with input which is taken in step 2. (All states which can reach final state with ϵ^* only are also final state).

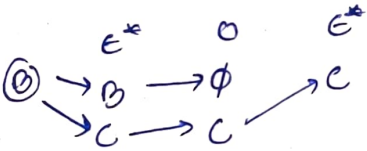
→ ϵ^* : All states that can be reached from a particular state only by seeing ϵ symbol.

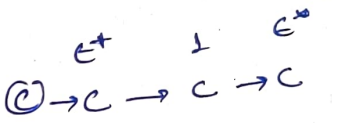
Q. Convert Given E-NFA to equivalent NFA.

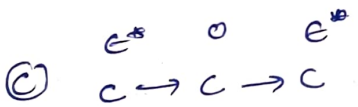




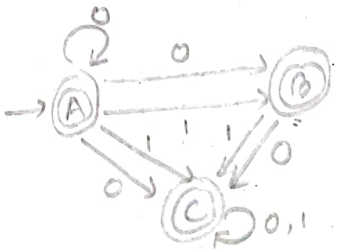




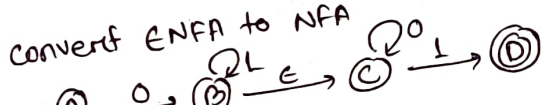


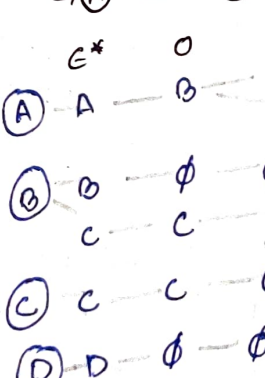


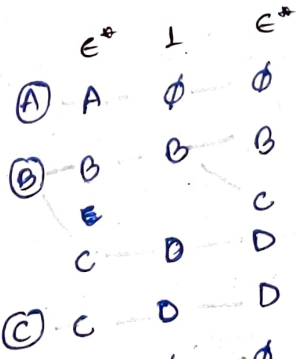
	0	1
→ A	{A,B,C}	{B,C}
B	{C}	{B,C}
C	{C}	{C}



Q Convert ∈NFA to NFA

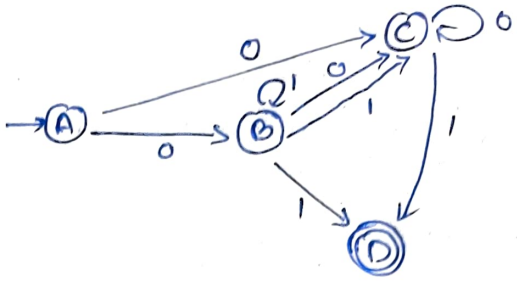






NFA

	0	1
→ A	B,C	ϕ
B	C	B,C,D
C	C	D
(D)	ϕ	ϕ



* Regular Expression:

→ Reg. expressions are used for representing certain sets of strings in an algebric fashion.

- Any terminal symbol i.e Symbol $\in \Sigma$ including $\wedge$ and ϕ are reg. Ex. ($\wedge$ → empty, ϕ → null)
- The union of RegEx is also RegEx. $R_1, R_2 \rightarrow (R_1 \cup R_2)$
- The concatenation of RegEx. is also RegEx, $R_1, R_2 \rightarrow R_1.R_2$
- The iteration (or closer of RegEx is also Regex, $R \rightarrow R^*$
- The RegEx. over Σ are precisely those obtained recursively by the application of the above rules once or serval times.

Q. Represt as RegEx:

1. $\{0,1,2\} \Rightarrow R = 0+1+2$
2. $\{\wedge, ab\} \rightarrow R = \wedge ab$ (when only one str. is present with λ, no need to use anything)
3. $\{abb, a, b, bba\} \rightarrow R = abb + a + b + bba$
5. $\{\wedge, 0, 00, 000, \ldots\}$ → $R \rightarrow 0^*$
6. $\{1, 11, 111, 1111, \ldots\}$ → $R \Rightarrow 1^+$

→ Identities:

1. $\phi + R = R$
2. $\phi R + R\phi = \phi$
3. $\epsilon R = R$ ($\wedge = \epsilon$)
4. $\epsilon^* = \epsilon$ $\phi^* = \epsilon$
5. $R + R = R$
6. $R^* R^* = R^*$
7. $RR^* = R^* R$ → on multi. ϵ with R, 'ϵ' get removed and it become R^+
8. $(R^*)^* = R^*$
9. $\epsilon + RR^* = R^*$
10. $(PQ)^* P = P(QP)^*$
11. $(P+Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$
12. $(P+Q)R = PR + QR$
    $R(P+Q) = RP + RQ$

→ ARDEN'S Theorem:

If P and Q are two reg.ex. over Σ, and if P doesnot contain ϵ, then the following eq. in R given by $R = Q + RP$ has a unique soln, $R = QP^*$

$R = Q + RP$
$= Q + QP^* P$ (replace $R = QP^*$)
$= Q(\epsilon + P^* P)$
$= QP^*$

$R = Q + RP$
$= Q + [Q+RP]P = Q + QP + RP^2$
$= Q + QP + [Q+RP]P^2 = Q + QP + QP^2 + RP^3$
$\vdots$
$= Q + QP + QP^2 + \cdots QP^n + RP^{n+1}$

Q Find RegEx for the NFA.

Represent final state in the form of RegEx having inputs only.

$q_3 = q_2a$ —①

$q_2 = q_1a + q_2b + q_3b$ —②

$= q_1a + q_2b + (q_2a)b$

$= q_1a + q_2(b + ab)$

$= (q_1a)(b+ab)^*$ —③ (Ardens theorem)

$q_1 = \epsilon + q_1a + q_2b$

$= \epsilon + q_1a + ((q_1a)(b+ab)^*)b$

$= \epsilon + q_1(a + (a(b+ab)^*)b)$

$= \epsilon(a + (a(b+ab)^*)b)^*$ —④ (arden's theorem)

$q_3 = q_2a = ((q_1a)(b+ab)^*)a$ (From ③)

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

When multiple final states, take union of them.

$q_1 = \epsilon + q_10$

$= \epsilon 0^* = 0^*$ —①

$q_2 = q_11 + q_21$

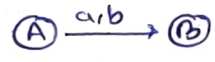
$= 0^*1 + q_21$

$= (0^*1)(1)^*$

$\therefore$ Req. RegEx $= q_1 + q_2 = 0^* + 0^*11^*$

$= 0^*(\epsilon + 11^*) = 0^*1^*$

→ Conversion of RegEx into finite automata.

$(a+b) \equiv$ 

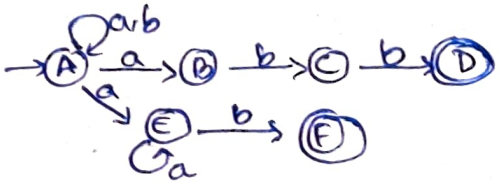
$a \cdot b \equiv$ 

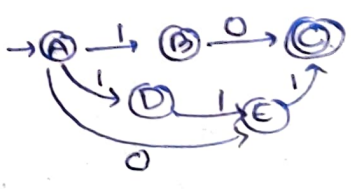
$a^* \equiv$

Q. i) $(a+b)c$ 

ii) $a(bc)^*$ 

iii) ba^*b 

iv) $(a|b)^*(abb|a^+b)$ 

v) $10+(0+11)0^*1$ 

→ Equivalence of two finite automata:

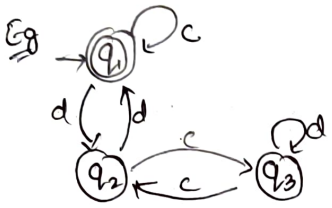
- Equi. of if they accepts the same type of language.
- Steps to identify equivalence:

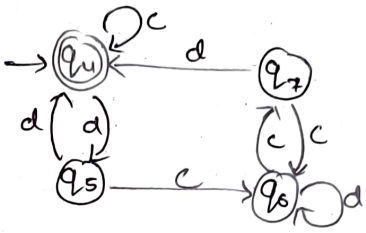
i) For any pair of states $\{q_i, q_j\}$ the transition for input $a \in \Sigma$ is defined by $\{q_a, q_b\}$ where $\delta(q_i, a) = q_a$ and $\delta(q_j, a) = q_b$.

The two automata are not equivalent if for a pair $\{q_a, q_b\}$ one is intermediate state and other is final state.

ii) Initial state of both automata must be same i.e if in one its final state then in the other one also it must be final

Eg





IS → intermediate state
FS → final state.

States	c	d
(q_1, q_4)	(q_1, q_4) FS FS	(q_2, q_5) IS IS
(q_2, q_5)	(q_3, q_6) IS IS	(q_1, q_4) FS FS
(q_3, q_6)	(q_2, q_7) IS IS	(q_3, q_6) IS IS
(q_2, q_7)	(q_3, q_6) IS IS	(q_1, q_4) FS FS

Starting from ... states, take ... pairs which gets visited from ... pairs, no need ... take any ... pair

Hence both are equivalent

* Pumping Lemma:

→ used to prove that a language is not regular.

If A is a reg. lang. then A has a pumping length 'p' such that any string 's' where $|s| \geq p$ may be divided into 3 parts $s = xyz$ such that the following conditions must be true:

i) $xy^iz \in A$ for every $i \geq 0$

ii) $|y| > 0$

iii) $|xy| \leq p$

→ Steps to prove language is not regular: (contradiction method)

1. Assume A is regular.
2. Find a string 's' in A such that $|s| \geq p$
3. divide s into xyz
4. Show that $xy^iz \notin A$ for some i
5. Then consider all ways that s can be divided into xyz.
6. Show that none of them can satisfy all the 3 pumping conditions at same time.

∴ s cannot be pumped ⇒ ⇐ contradiction.

Q Proof Lang. $A=\{a^n b^n \mid n \geq 0\}$ is not reg. Lang.

Assume that A is reg.

Pumping length = P.

Let S be a string in A such that $S = a^P b^P$, $P = 7$.

Case 1: Y is in 'a' part

$S = \underbrace{aa}_{x}\underbrace{aaaaa}_{y}\underbrace{abbbbbbb}_{z}$

$xy^i z$, $i=2$

$\Rightarrow xy^2z = aaaaaaaaaaaabbbbbbbb \notin A$

Case 2: Y is in 'b' part

$S = \underbrace{aaaaaaa}_{x}\underbrace{b}_{y}\underbrace{bbbbbbb}_{z}$

$xy^i z$, $i=2$

$\Rightarrow xy^2z = aaaaaaabbbbbbbbbbbb \notin A$

Case 3: Y is in 'a' and 'b'

$S = \underbrace{aaaaaaa}_{x}\underbrace{ab}_{y}\underbrace{bbbbbbb}_{z}$

$xy^i z$, $i=2$

$\Rightarrow xy^2z = aaaaaaabbaabbbbbbb \notin A$

Since for all possible cases, none of them able to satisfy the pumping lemma condition, it contradicts our assumption.

$\therefore$ A is not reg.

* Grammer:

→ A grammer 'G' can be formally described using 4 tuples as

$G = (V, T, S, P)$ where,

V = Set of variable or non-terminal symbols.

T = Terminal symbols.

S = start symbol.

P = Production rules for terminals and non-terminals.

A production rules has the form $\alpha \to \beta$ where α and β are strings on $V \cup T$ (union) and atleast one symbol of α belongs to V.

Eg: $G = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

$V = \{S, A, B\}$, $T = \{a, b\}$, $S = S$, $P = S \to AB, A \to a, B \to b$.

$S \to AB \to aB \to ab$ this is what designed using this grammer.

→ Regular grammer can be divided into two parts:

Right linear grammer

A grammer is said to be right linear if all productions are of the form

$A \to xB$

$A \to x$

where $A, B \in V$ and $x \in T$

Eg: $S \to ab\underset{\text{terminal}}{S} \mid \underset{\text{non-T}}{b} \to T$ → Right linear

Left linear grammer.

A grammer is said to be left linear if all productions are of the form

$A \to Bx$

$A \to x$

where, $A, B \in V$ and $x \in T$.

$S \to \underset{\text{non-Terminal}}{S}bb \mid \underset{\text{Terminal}}{b}$ → Left linear

→ Derivation from a Grammer:

The set of all strings that can be derived from a grammer is said to be the Language generated from that grammer.

Eg: $G1 = (\{S, A\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \epsilon\})$.

$S \to aAB$ (by $S \to aAb$)
$\to aaAbb$ (by $aA \to aAb$)
$\to aaaAbbb$
$\to aaabbb$ ($A \to \epsilon$).

$\therefore L(G_1) = \{aaabbb, \ldots\}$
we can make more.

$G_2 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

$S \to AB$
$\to ab$

$\therefore L(G_2) = \{ab\}$

$G_3 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to aA|a, B \to bB|b\})$

$S \to AB$ $S \to AB$ $S \to AB$
$\to ab$ $\to aAb$ $\to aAbB$
$\to aab$ $\to aabb$ ...

$L(G_3) = \{ab, abaabb, aabb \ldots\}$.

$L(G_3) = \{a^m b^n \mid m \geq 1 \text{ and } n \geq 1\}$

* Context free Language:

In formal language theory, a context free lang. is a lang. generated by some context free grammer.

The set of all CFL is identical to the set of lang. accepted by pushdown automata.

→ Context free grammer is defined by 4 tuples as $G = (V, \Sigma, S, P)$ where,
V = set of variables or non-terminal symbols.
Σ = set of terminal symbols.
S = start symbol
P = Production Rule.

Context free grammer has production rule of the form $A \to \alpha$, where $\alpha = \{V \cup \Sigma\}^*$ and $A \in V$

Q. For generating a lang. that generates equal no. of a's and b's in the form $a^n b^n$, the context free grammer will defined as
$G = \{(S, A), (a, b), S, (S \to aAb, A \to aAb \mid \epsilon)\}$

→ Every Reg. Lang is context free, but not vica-versa

→ Method to find whether a string belongs to a Grammer or not.
1. Start with the start symbol and choose the closest production that matches to the given string.
2. Replace the variables with its most appropriate production. Repeat the process until string is generated or until no other production are left.

Q verify whether grammer $S \to 0B|1A$, $A \to 0|0S|1AA|\wedge$, $B \to 1|1S|0BB$ generates 00110101.

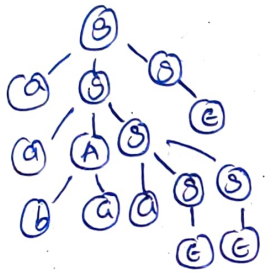
$S \to 0B$
$\to 00BB$
$\to 001B$
$\to 0011S$
$\to 00110B$
$\to 00110IS$
$\to 00110101$.

→ Ambiguous tree
An ordered rooted tree that graphically represents the semantic information of strings derived from context free grammer.
- Rooted vertex: must be labelled by start symbol.
- Vertex: Non-terminal symbol
- Leaves: Terminal symbol.

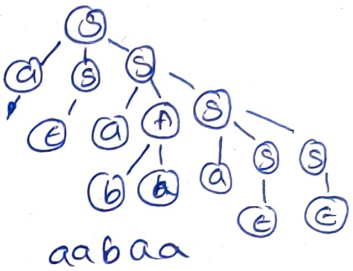
Left derivation
The left derivation tree is obtained by applying production to the leftmost variable in each step.

Right derivation.
obtained by applying production to the rightmost variable in each step.

$S \to aAS \mid aSS \mid \epsilon$, $A \to SbA \mid ba$



aabaa



aabaa

* Ambiguous Grammer:
A grammer is said to be ambiguous if there exists two or more derivation tree for a string w (that means two or more left derivation trees).

Q. $G = (\{S\}, \{a+b, +, *\}, P, S)$ where $P = S \to S+S \mid S*S \mid a \mid b$.
The string $a+a*b$ can be generated as:

$S \to S+S$
$\to a+S$
$\to a+S*S$
$\to a+a*b$

$S \to S*S$
$\to S+S*S$
$\to a+a*b$

∵ two different, its ambiguous.

* Simplification of context free grammer:
→ Reduction of CFG
On CFG sometimes all the production rules and symbols are not needed for the derivation of string. Besides this, there may also be some Null & unit production. Elimination of these production & symbols is called simplification of CFG.
- Phase I: Derivation of an equivalent grammer G', from CFG G, such that each variable derives some terminal string.
- Phase II: Derivation of a equivalent grammer G'' from CFG G', such that each symbol appears in ~~seq~~ sequential form

Q Find the reduced grammer equivalent to grammer G:
P: $S \to AC \mid B$, $A \to a$, $C \to c \mid BC$, $E \to aA \mid e$

$T = \{a, e, \epsilon\}$.

Phase I: $W_1 = \{A, C, E\}$. ← which derives terminal state.

$W_2 = \{A, C, E, S\}$ ← which derives W_{i-1} state

$W_3 = \{AC, ES\}$

$G' = \{(A, C, E, S), \{a, c, e\}, P, (S)\}$

P: $S \to AC$, $A \to a$, $C \to c$, $E \to aA|e$

Phase 2:

$Y_1 = \{S\}$ ← start symbol

$Y_2 = \{S, A, C\}$ ← symbols derived from Y_{i-1}

$Y_3 = \{S, A, C, a, c\}$

$Y_4 = \{S, A, C, a, c\}$

$G'' = \{(A, C, S), \{a, c\}, P, \{S\}\}$ ← Req. reduced grammar

P: $S \to AC$, $A \to a$, $C \to c$

→ Removal of unit production:

Any production rule of the form $A \to B$ where $A, B \in$ Non-terminal is called unit production.

- Procedure:
  1. To remove $A \to B$, add $A \to x$ to the grammar rule when ever $B \to x$ occurs in the grammar [$x \in$ Terminal, x can be null].
  2. Delete $A \to B$ from grammar.

Q. Remove unit production in: P: $S \to Xy$, $X \to a$, $Y \to z|b$, $Z \to M$, $M \to N$, $N \to a$.

Unit productions: $Y \to Z$, $Z \to M$, $M \to N$.

1. Since $N \to a$, we add $M \to a$

   P: $S \to Xy$, $X \to a$, $Y \to z|b$, $Z \to M$, $M \to a$, $N \to a$

2. " $M \to a$, " " $Z \to a$

   P: " " " $Z \to a$ " "

3. " $Z \to a$, " " $Y \to a$

   P: $S \to XY$, $X \to a$, $Y \to a|b$, $Z \to a$, $M \to a$, $N \to a$.

Remove the unreachable symbols:

P: $S \to XY$, $X \to a$, $Y \to a|b$,

→ Removal of null production:

In a CFG, a non-terminal symbol 'A' is a nullable variable if there is a production $A \to \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ (like $A \to \dots \to \epsilon$).

- Procedure:
  1. To remove $A \to \epsilon$, Look for all productions whose right side contains A.
  2. Replace each occurrences of 'A' in each of these productions with ϵ
  3. Add the resultant production to the grammar.

Q. Remove Null production from: $S \to ABAC$, $A \to aA \mid \epsilon$, $B \to bB \mid \epsilon$
$C \to C$.

To eliminate $A \to \epsilon$

$S \to ABAC$

$S \to ABC \mid BAC \mid BC$

$A \to aA$

$A \to a$

New production: $S \to ABAC \mid ABC \mid BAC \mid BC$, $A \to aA \mid a$

$B \to bB \mid \epsilon$, $C \to C$.

To eliminate $B \to \epsilon$

$S \to AAC \mid AC \mid C$, $B \to b$

New production: $S \to ABAC \mid ABC \mid BAC \mid BC \mid AAC \mid AC \mid C$

$A \to aA \mid a$

$B \to bB \mid b$

$C \to C$

* Chomsky Normal form:

In this we have restriction on the length of RHS; which is: elements in RHS should either be two variables or a Terminal.

i.e. Production should be of the form:

$A \to a$

$A \to BC$

where A, B, C are non-terminal and a is terminal.

• Procedure:

1. If the start symbol occurs on some right side, create a new start symbol S' and a new production $S' \to S$.
2. Remove null and unit productions.
3. Replace each production $A \to B_1 B_2 \cdots B_n$ where $n > 2$ with $A \to B_1 C$, where $C = B_2 \cdots B_n$ (Repeat for C then)
4. If $A \to aB$ type exist then replace it with $A \to XB$, $X \to a$ where a is terminal, A, B, X are non-Terminal.

Q. Convert CFG to CNF: P: $S \to ASA \mid aB \mid a$, $A \to B \mid S$, $B \to b \mid \epsilon$

↳ Since S appears in RHS, we add a new state S' and $S' \to S$ is added to the production.

P: $S' \to S$, $S \to ASA \mid aB \mid a$, $A \to B \mid S$, $B \to b \mid \epsilon$

2. Remove the null production:

After Removing $B \to \epsilon$: P: $S' \to S$, $S \to ASA \mid aB \mid a$, $A \to B \mid S \mid \epsilon$, $B \to b$

After removing $A \to \epsilon$: P: $S' \to S$, $S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$, $A \to B \mid S$, $B \to b$

3. Remove the unit production:

$S \to S$, $S' \to S$, $A \to B$ and $A \to S$:

After Removing $S' \to S$: P: $S' \to ASA \mid aB \mid a \mid AS \mid SA$.

$S \to ASA \mid aB \mid a \mid AS \mid SA$

$A \to B \mid S$, $B \to b$.

After removing $A \to B$: P: $S' \to ASA \mid aB \mid a \mid AS \mid SA$
$S \to ASA \mid aB \mid a \mid AS \mid SA$,
$A \to b \mid S$, $B \to b$.

After removing $A \to S$: P: $S' \to ASA \mid aB \mid a \mid aAS \mid SA$
$S \to ASA \mid aB \mid a \mid AS \mid SA$
$A \to b \mid ASA \mid aB \mid a \mid AS \mid SA$
$B \to b$

4. Now find out the productions that has more than 2 variables in RHS
$S' \to ASA$, $S \to ASA$, and ASA
After removing these we get: P: $S' \to AX \mid aB \mid a \mid AS \mid SA$
$S \to AX \mid aB \mid a \mid AS \mid SA$
$A \to b \mid AX \mid aB \mid a \mid AS \mid SA$
$B \to b$
$X \to SA$

5. Now change the productions $S' \to aB$, $S \to aB$, and $A \to aB$.
Finally we get: P: $S' \to AX \mid YB \mid a \mid AS \mid SA$
$S \to AX \mid YB \mid a \mid AS \mid SA$
$A \to b \mid AX \mid YB \mid a \mid AS \mid SA$
$B \to b$
$X \to SA$
$Y \to a$

which is the req. CNF of given CFG

* Greibach normal form:
$A \to b$ Production in this form
$A \to bC_1C_2 \cdots C_n$.
$A, C_1, C_2, C_3, \cdots C_n \to$ non-Terminal, $b \to$ terminal.

- Procedure:
1. Convert given CFG into CNF.
3. Change the names of non-terminal symbols into some A_i in ascending order of i.

Q. $S \to CA \mid BB$
$B \to b \mid SB$
$C \to b$
$A \to a$
Its already in the CNF.

Replace: S with A_1
C with A_2
A with A_3
B with A_4.

Take care of order. Jo pehle dikha usko pehle assign karo.

we get: $A_1 \to A_2A_3 \mid A_4A_4$
$A_4 \to b \mid A_1A_4$
$A_2 \to b$
$A_3 \to a$

3. After the rules so that the non-terminals are in ascending order, such that, if the production is of the form $A_i \to A_jX$, the $i < j$ and not $i \geq j$

$A_4 \rightarrow b \mid A_1 A_4$ (only the 1st element smaller in $A_i \rightarrow A_j x$)

$A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$ (Replace A_1 with value)

$A_4 \rightarrow b \mid b A_3 A_4 \mid A_4 A_4 A_4$

same — Left recursion

4. Remove left recursion.

Introduce a new variable to remove the left recursion.

$A_4 \rightarrow b \mid b A_3 A_4 \mid A_4 A_4 A_4$

$Z \rightarrow A_4 A_4 Z \mid A_4 A_4$ (once with Z and once without Z)

$A_4 \rightarrow b \mid b A_3 A_4 \mid bZ \mid b A_3 A_4 Z$ (Remove left recursion and write the previous values one time with Z and one with out Z)

without Z | with Z

Now the grammar is:

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid b A_3 A_4 \mid bZ \mid b A_3 A_4 Z$

$Z \rightarrow A_4 A_4 \mid A_4 A_4 Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3 A_4 A_4 \mid bZA_4 \mid bA_3 A_4 ZA_4$ (Replacing the first A_4 with its value)

$A_4 \rightarrow b \mid bA_3 A_4 \mid bZ \mid bA_3 A_4 Z$

$Z \rightarrow bA_4 \mid bA_3 A_4 A_4 \mid bZ A_4 \mid bA_3 A_4 ZA_4 \mid$
$bA_4 Z \mid bA_3 A_4 A_4 Z \mid bZA_4 Z \mid bA_3 A_4 ZA_4 Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

* Pumping lemma for context free lang.

Used to prove lang. is not context free.

Same as pumping lemma of Reg. lang.

S can be divided into 5 parts $uvxyz$,

i) $uv^i xy^i z \in A$ for every $i \geq 0$

ii) $|uv| > 0$

iii) $|vxy| \leq p$

- prove process is same.

Q Show that $L = \{a^N b^N c^N \mid N \geq 0\}$ is not context free

Assume L is context free.

L must have pumping length (say P).

Now we take a string S such that $S = a^p b^p c^p$

We divide S into 5 parts S $uvxyz$.

Eg $P = 4$, so $S = a^4 b^4 c^4$.

Case I: v and y each contain only one type of symbol

aaaabbbbcccc

u v x y z

For $i = 2$, $uv^i xy^i z = uv^2 xy^2 z$

$= aaaaaabbbbccccz$

$= a^6 b^4 c^5 \notin L$.

Case 1: Either v or y has more than one kind of symbols.

aaaabbbbccccc uv^{i}xy^{i}z, for i=2,

u v x y z aaaabbaabbbbbbccccc ∉ L

∴ It is not context free.

* Push down Automata (PDA):

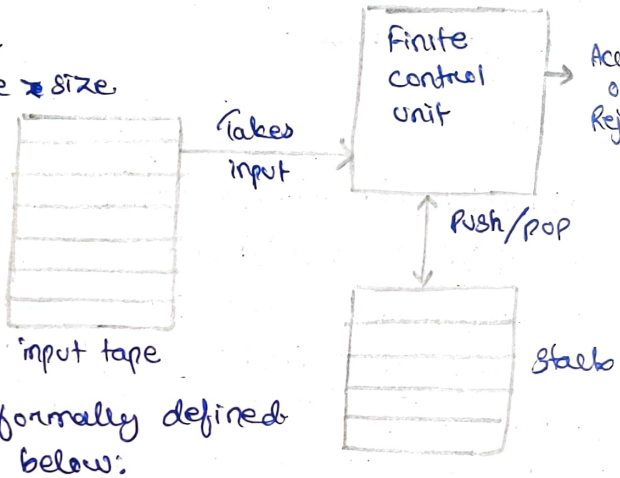
A pushdown automata is a way to implement context free Grammar in a similar way we design finite Automata for reg. grammer.

→ More powerful than FSM.

→ FSM has a ~~memory~~ very limited memory but PDA has more memory

→ PDA = FSM + A stack.

- Components:
  - An input tape (string)
  - A finite control unit
  - A stack with infinite size



→ Pushdown automata is formally defined by 7 tuples as shown below:

$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

- Q → A finite set of state
- Σ → A finite set of input symbols.
- Γ → A finite stack ~~symbols~~ Alphabet
- δ → Transition function.
- q_0 → Start state
- z_0 → The start stack symbol.
- F → Set of final/accepting symbols.

δ takes as argument a triple $\delta(q, a, x)$ where:

i) q is a state in Q

ii) a is either an input symbol in Σ or $a = \epsilon$

iii) x is a stack symbol, that is member of Γ.

The output of δ is finite set of pairs (p, γ) where:

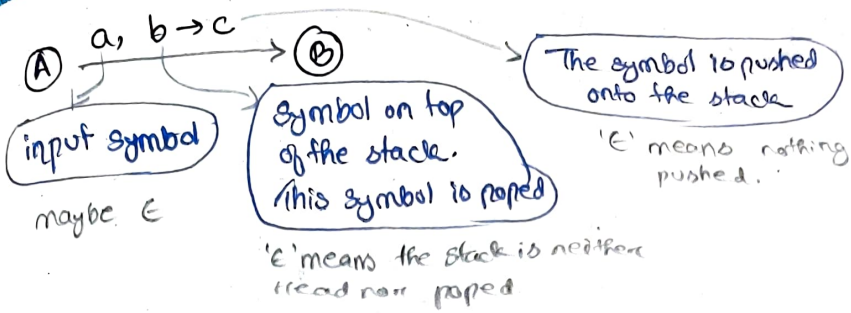
p is a new state.

γ is a string of stack symbols that represent x at the top of the stack

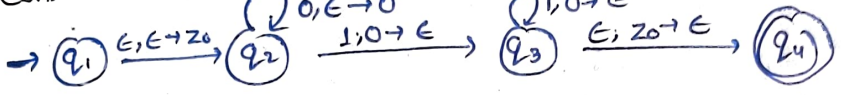
Eg: If $\gamma = \epsilon$ then stack is poped.

$\gamma = x$ stack is unchanged

$\gamma = yz$ x is replaced by z and y is pushed onto stack.



Q. Construct a PDA that accepts $L=\{0^n 1^n \mid n \geq 0\}$

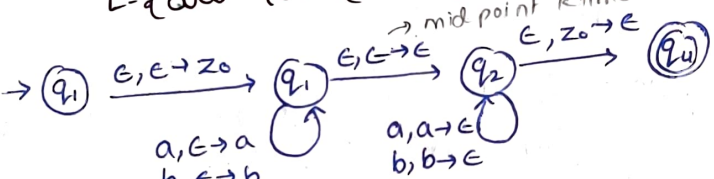


Z_0 = $ is just used to denote, its the last element in stack

→ String accepted when:
i) Stack is empty. (Z_0 vichi hona chahiye)
ii) We reached final state.

→ Its a kind of NFA (no need to show all input ~~token~~ transition).

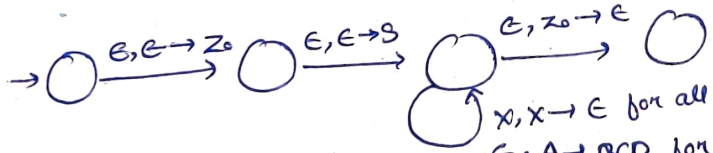
Q. Construct a PDA that accepts even palindromes of the form
$L=\{ww^R \mid w=(a+b)^+\}$



To know correct mid-point, it follows MCM by inserting ∈ after each symbol & proceeds and checks if it gives ~~correct~~ right output.

* Equivalence of CFG and PDA.
→ A lang. is context free iff some pushdown Automata recognizes it.
• Proof:
i) Given a CFG, show how to construct a PDA that recog. it.
ii) Given a PDA, " " " " CFG " " "

Q. Given a grammer: $S \to BS \mid A$
$A \to 0A \mid \in$
$B \to BB1 \mid 2$
Find or build a PDA.

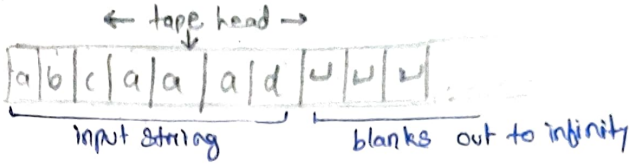


1. Push the start symbol on stack
2. When you see non-terminal symbol, Pop it and push right side for that symbol ($A \to BCD$), '∈' because we don't have to move to next input now.
3. When terminal on the top, just pop and move to next input.

* Turing machine:
→ "Tape" data structure is used.
→ Tape is an infinite sequence., tape head is present where the current control is present.
→ ~~move~~ move both left and right

→ Empty cells of tape are filled with "␣" blank.

→ Blank ~~is used~~ ␣ ∉ Σ (input symbols)

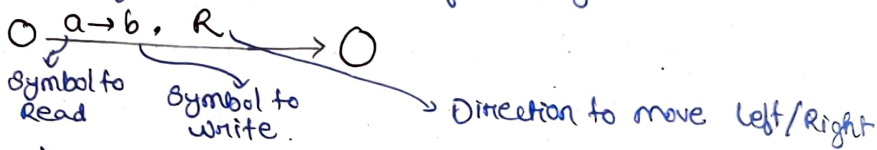


→ control portion is similar to FSM or PDA.

⇒ control portion is deterministic. (Need to show every possible input. In some questions, they may not give reject state path, but you have to consider that.)

* Rules of operation:

1: At each step of computation
- Read the current symbol.
- Update (i.e write) ~~to~~ the same cell.
- Move exactly one cell either Left or Right.



2: is
- Control with a sort of FSM
- Initial state
- Final states: (there are two final states)
  i) Accept state ii) Reject state.
- Computation can either
  1. Halt and accept 2. Halt and reject
  3. Loop (fails to halt)

* A turing machine can be defined as ~~the~~ a set of 7 tuples.

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

$Q \rightarrow$ non empty set of states.
$\Sigma \rightarrow$ " " " " symbols
$\Gamma \rightarrow$ " " " " Tape "
$\delta \rightarrow$ Transition function: $Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$
$q_0 \rightarrow$ Initial state $b \rightarrow$ blank symbol
$F \rightarrow$ set of final states (Accept state & reject state).

→ Production rule of Turing machine:

$\delta(q_0, a) \longrightarrow (q_1, Y, R)$

q_0: initial state; a: input; q_1: final state; Y: output; R: direction of movement (Left/Right)

→ Turing Thesis:

Turing's thesis states that any computation that can be carried out by mechanical means can be performed by some T.M.

→ Few arguments for accepting this thesis are:

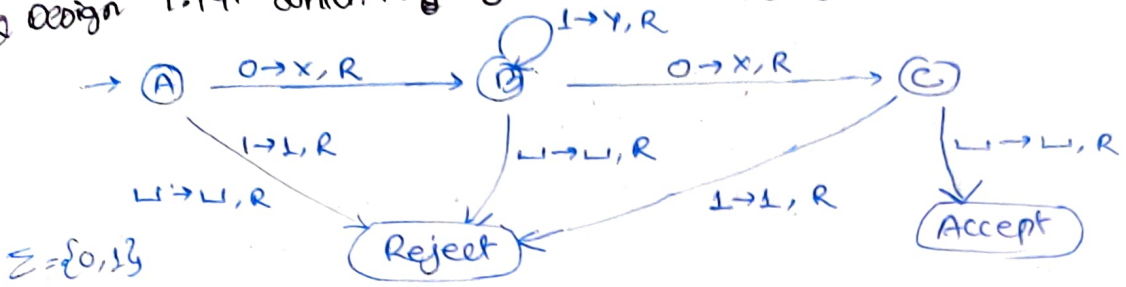
(i) Anything that can be done on existing digital computer can also be ~~done~~ done by T.M.

(ii) No one has yet been able to suggest a problem solvable by what we consider an algo, for which a Turing

Machine program cannot be written.

Lang. which L and Σ is said to be Recursively Enumerable if there exists a T.M that accepts it.

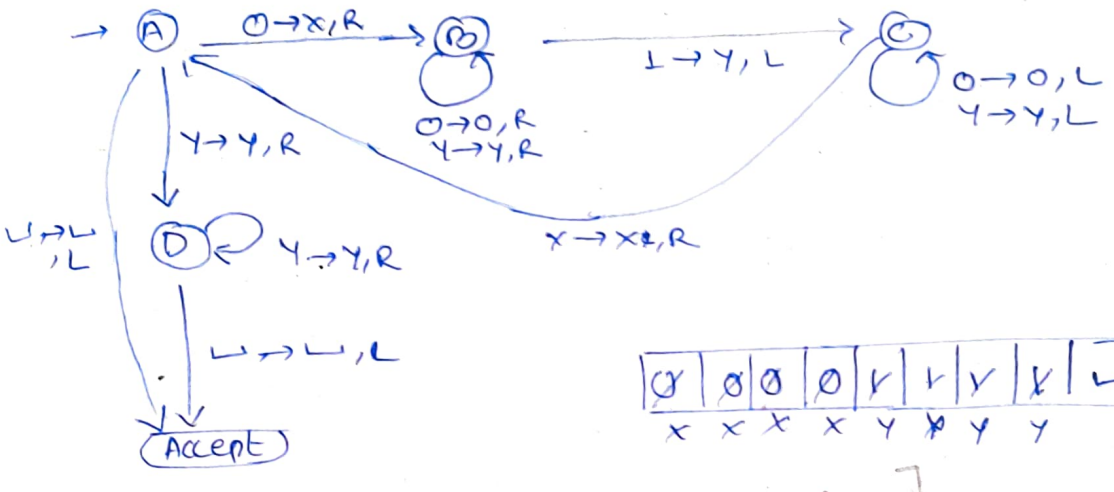
Q Design T.M. which recognizes the language: $L = 01^*0$.

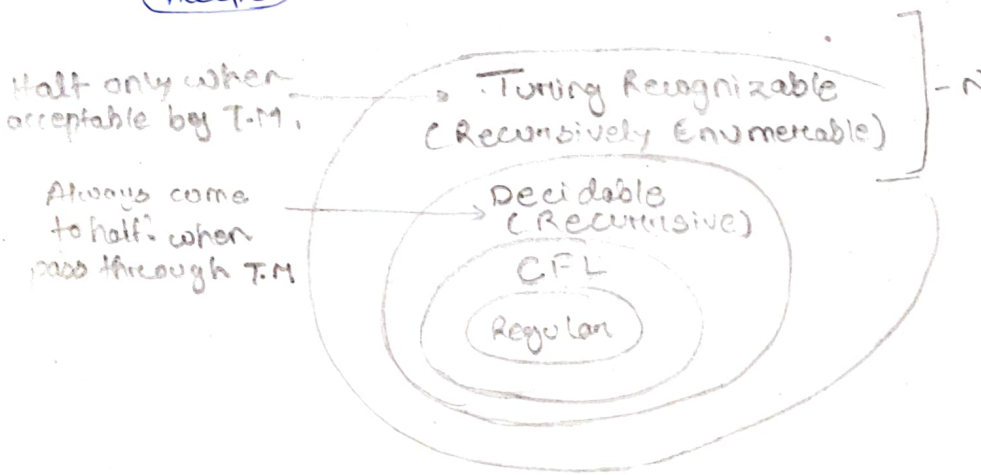


Σ = {0, 1}

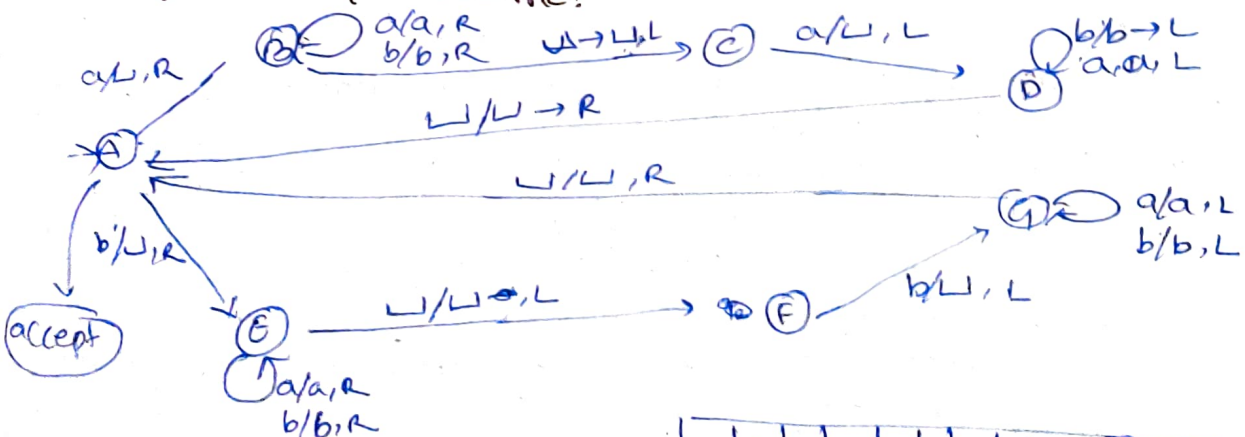
b = ␣

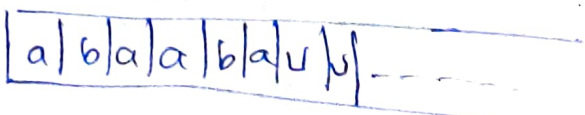
Q Design T.M for $L = 0^n 1^n$.





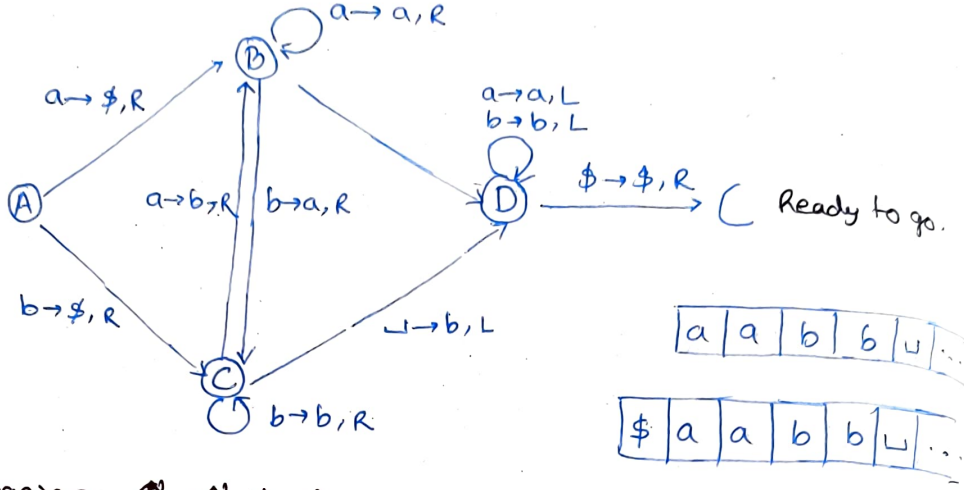
Q T.M. for even palindrome:





Q How to recognize the left end of Tape of a TM.

Put a special $ on the left end of the tape and shift the input over one cell to the right.



Q Recognise $0^N 1^N 0^N$.

Build a turing machine to recognize $0^N 1^N$ (done already) once done, it will change $0^N 1^N$ to $X^N Y^N$, Now use another TM for $Y^N 0^N$.

→ When you have to match certain string, without loosing the original one, replace the characters with unique symbols instead of replacing all with same (x).

→ Theorem: Every Multitape TM has a equivalent single tape TM.

* Non-deterministic Turing Machine:

$$\delta \rightarrow Q \times \Sigma \longrightarrow P\{\Gamma \times (R/L) \times Q\}$$

P is the power

- **Configuration:**

A way to represent the entire state of a TM at a moment during computation.

A string which captures:

- → Current state.
- → The current position of the Head.
- → The entire tape contents.

- **Outcomes of Non-deterministic computation.**

→ Accept: If any branch of the computation accepts, then the TM will accepts.

→ Reject: If all branches of the computation HALT and REJECT (i.e. no branches accept, but all computations HALT) then the ND TM rejects.

→ LOOP: computation continues but Accept is never encountered some branches in the computation history are infinite.

→ Theorem: Every Non-deterministic TM has an equivalent deterministic TM.

IMP

Recursive lang.	TM will always halt.
Recursive Enumerable lang.	TM sometimes halt. & sometime may not
Decidable lang.	Recursive lang.
partially decidable. lang.	Recursively Enumerable lang.
UNDECIDABLE	No. TM for this lang.

* Universal TM:

Input: M = Description of some TM.
w = an input string for M.

Action: Simulate M.
Behave just like M would (may accept, reject or loop.

The UTM is a recognizer (but not a decider) for

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$.

Q. ~~Given~~ Can we design a TM that will never go in loop.
NO., we need to run the program to know if it halts or not.