Symbol → a, b, c, d ...., 0, 1, 2 .... ಕ ಖ .

On top of symbol they define Alphabet – {a, b} → finite set of symbols

Alphabets are represented by 'Sigma' → Σ

Using the alphabets we define Strings.

String → Sequence of Symbols.

Ex. $\Sigma = \{a, b\}$ ∴ $S_1 = ab$, $S_2 = abba$, $S_3 = a$, $S_4 = abbab$

In this example a string can be anything made by these two Alphabets.

Language → Language is a set of strings over an alphabet. It can be finite or infinite.

Ex: $\Sigma = \{a, b\}$ $L_1 = \{aa, ab\}$ → finite language.

$L_2 = \{a, aa, ab, aaa, aab....\}$ → infinite language.

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

$\Sigma^1 = \{a, b\}$

↓

Set of all strings of length '1'

Concat

$\Sigma^2 = \{a, b\} ! \{a, b\}$

$= \{aa, ab, ba, bb\}$

↓

Set of all strings of length '2'.

Σ^n = Set of all strings of length 'n'.

Σ^0 = Set of all strings of length '0'.

$= \{\epsilon\}$ → null string, length of the string = 0.

$\{\}$ → Empty set. Sometimes it's written as '$\{\phi\}$'.

$\{\} \neq \{\epsilon\}$

↓ Empty

$\{\epsilon\}$ → Not empty.

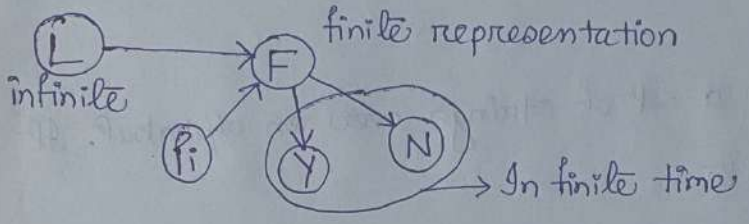
$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots\ldots$

= Set of all strings possible

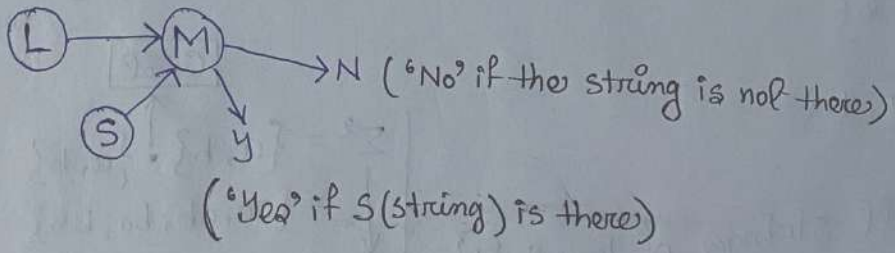
= **Universal language**

Every language over Σ is a subset of Σ^*.

Whenever the language is infinite, we want a finite representation. To this finite representation if I give a string, let us say Pi is the string then you should say → Yes or No in finite time.



∴ for a language we need a machine.

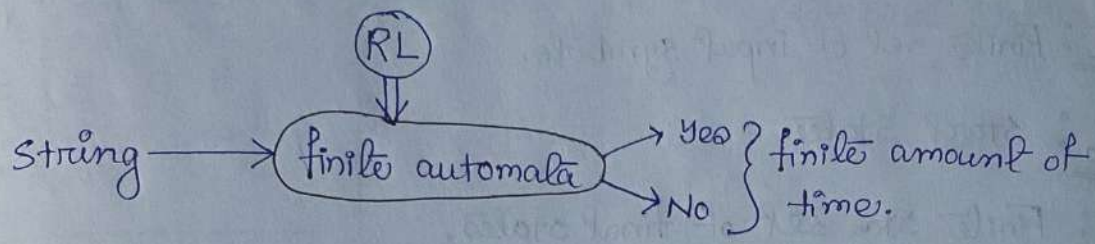


There are many types of languages. Each type has a different machine. We will see **Regular languages** first.

finite

For every **Regular language** there will be a machine called **Automata**.

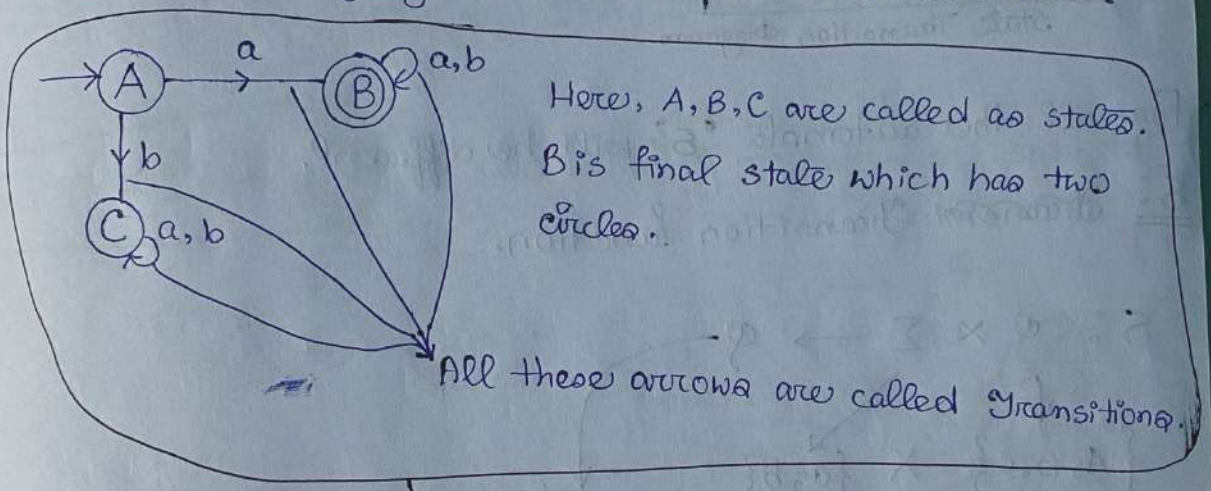
To this machine if we give a string, it will say Yes or No in finite amount of time.



Ex: L_1 is set of all strings starting with a. $\Sigma = \{a, b\}$.

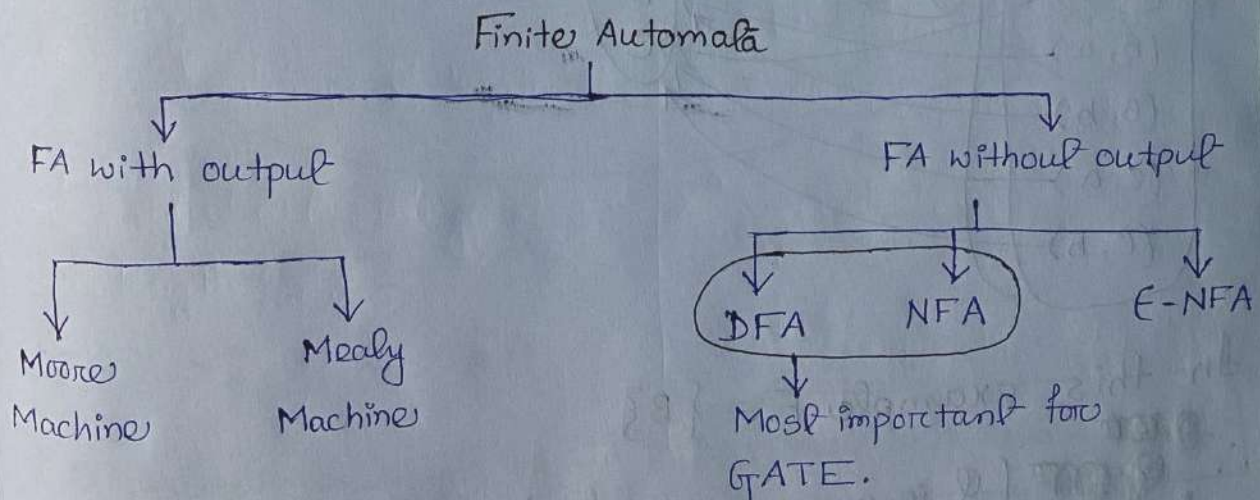
$\Rightarrow L_1 = \{a, aa, ab, \ldots\}$

→ for this language L_1, there is a Finite Automata.



↓

This is a Finite Automata



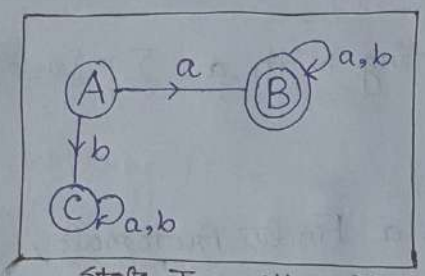
DFA: $(Q, \Sigma, \delta, q_0, F)$ — Deterministic Finite Automata

Q: Finite set of states.

Σ: Finite set of input symbols.

q_0: Start state.

F: Finite set of final states.



State Transition Diagram

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

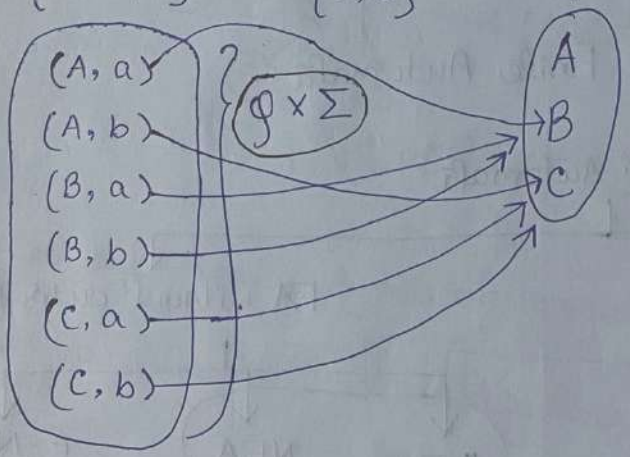
Q cross Σ to Q

For all the automata 'δ' will be different.

δ: Transition function.

$\delta : Q \times \Sigma \longrightarrow Q$

$\{A, B, C\} \times \{a, b\}$



In this example $F = \{B\}$

$\therefore$ Q (Q is a superset of F) $\longrightarrow Q \supseteq F$

We can also write like this:

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

	a	b
→A	B	C
*B	B	B
C	C	C

State Transition Table

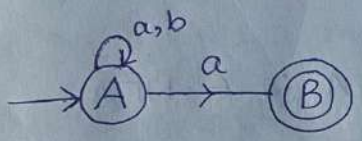
→ = Start State

* = Final State

NFA (Non Deterministic Automata): $(Q\ \Sigma\ \delta\ q_0\ F)$

Q, Σ, q_0 & F is same as DFA. Only δ is different for NFA.

$\delta : Q \times \Sigma \to 2^Q$ → Power set of Q → Set of all subsets.



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

$(A,B) \times (a,b)$

(A,a)
(A,b)
(B,a)
(B,b)

2^Q

{ }
{A}
{B}
{A,B}

$|Q| = 2$

$\therefore |2^Q| = 2^2$

$= 4$

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

DFA → $\delta : Q \times \Sigma \to Q$

Q is already present in 2^Q. So, every DFA is an NFA.

Every NFA need not be DFA.

In terms of Power:

Pow(DFA) = Pow(NFA)

→ If there is an NFA for a language, there will be a DFA for that language.

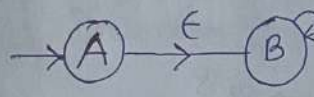
NFA —Convert→ DFA

DFA —is already→ NFA

∈-NFA: (NFA with epsilon moves) → Not imp

Here also everything is same except δ.

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$



∴ Every DFA is NFA, Every NFA is ∈-NFA

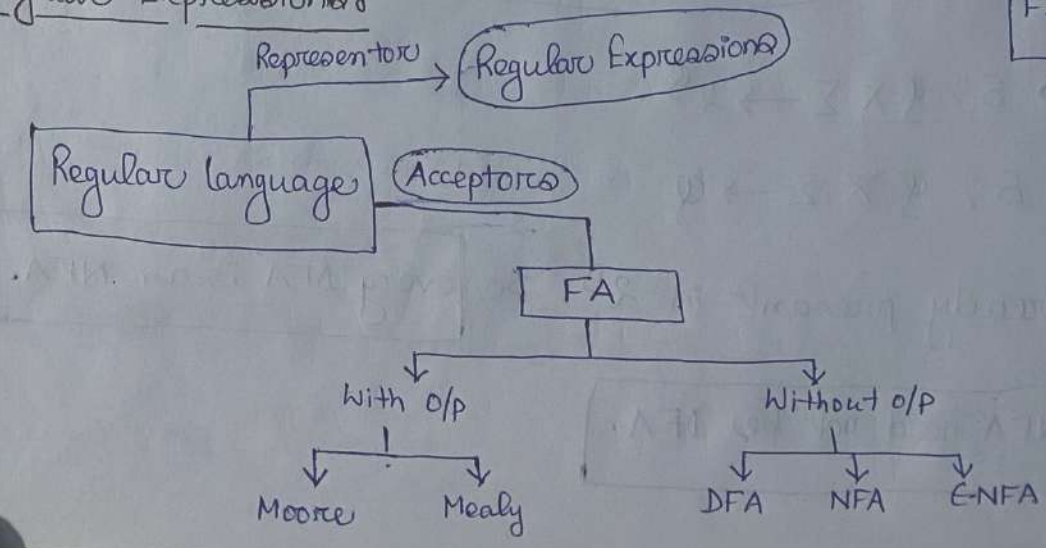
Pow(DFA) = Pow(NFA) = Pow(∈-NFA)

∈ NFA —Convert→ NFA

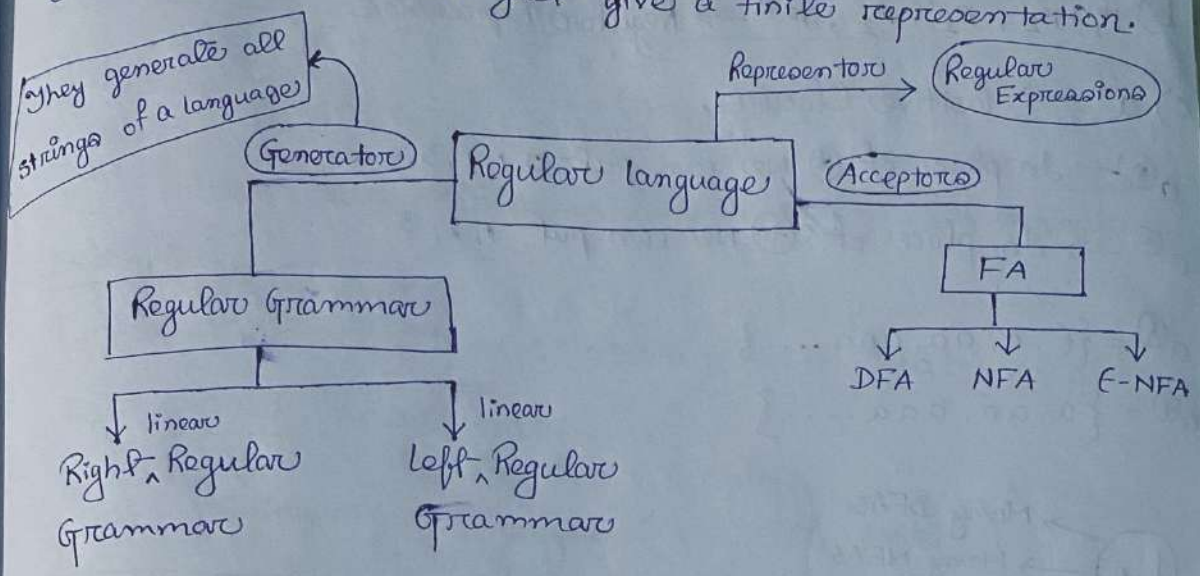
NFA —Convert→ DFA

Regular Expressions:

FA = Finite Automata



Regular expressions will just give a finite representation.



Types of Questions I'll be facing soon:

→ They will give a language and I'll have to give the machine (FA)

→ They will give a machine, then they will ask → What is the language?

→ They'll give a language and ask what is the Regular Expression for the language.

→ They will give a Regular Expression & ask What is the language.

NOTE:

1) + → Union

2) . → Concat

3) * → Kleene closure

Primitive Regular Expressions

a) ϕ, ϵ, $a \in \Sigma$ → It means, every symbol that belongs to Sigma (Σ) is also regular expression. They actually represent one string.

ϕ → { }

ϵ → {ϵ}

b) $r_1 + r_2$, $r_1 \cdot r_2$, r^* → Regular Expression

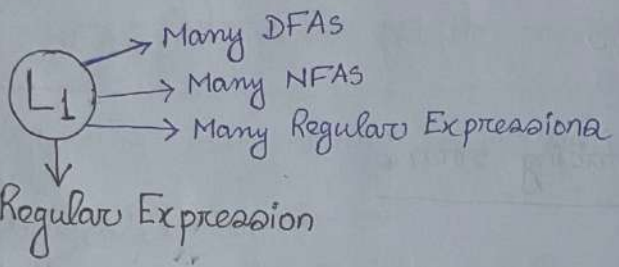
c) a^+ → Positive Closure.

$a^{\circledast}$ → In place of $\circledast$ we can put 0, 1, 2, 3 ....

$a^{\oplus}$ → In place of $\oplus$ we can put 1, 2, 3 ....

$a^{\circledast} = \{\epsilon, a, aa, aaa....\}$

$a^{\oplus} = \{a, aa, aaa....\}$



Out of all DFAs, minimal DFA will be unique (only one).

" " " NFAs, " NFA is not unique (there may be many)

But All minimal NFAs will have same no. of states.

So, questions will be mainly on how many states are there in min DFA or min NFA.

There is nothing called min 'RE' (Regular Expression).

Sometimes we have to design minimal DFA.

But minimal NFA is not unique.

Number of States in min NFA $\leq$ no. of states in min DFA.

Best way is to design min DFA, than convert to minimal NFA.

min DFA $\xrightarrow{\text{Convert}}$ Min NFA is easy.

min NFA $\xrightarrow{\text{Convert}}$ min DFA is difficult.

Shortcut to designing min NFA or min DFA:

$L = \{ \_ \_ \_ \_ \_ \}$ → Language

↓

First find minimum length string. (If language is infinite.)

Example:

L_1 = Set of all strings starting with 'aab'.

$L_1 = \{ \underbrace{aab}_{\text{minimum string}}, aaba, aabb.... \}$

length of $min_\wedge = 3$ $[|aab| = 3]$

Always min DFA or min NFA $\geq$ 4 states

$|w_{min}| = n$

∴ num of states (min dfa / min nfa) $\geq n+1$

*** Always start your design with minimum string.

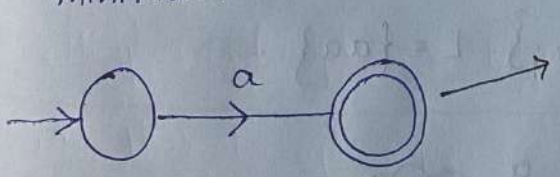
Example:

$\Sigma = \{a, b\}$ L = Set of all strings starting with 'a'.

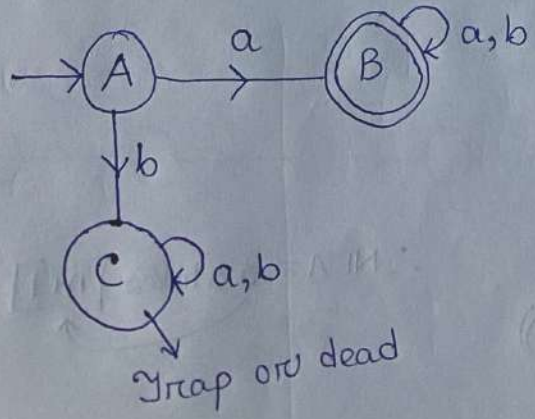
$L = \{ a, aa, ab, aaa, aab .... \}$

↓

minimum



min DFA:

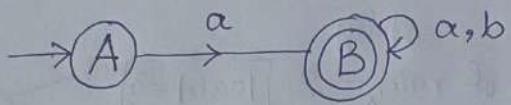


Initially we accept strings starting with 'a'.
Whenever a string comes which has 'b' at the starting, we take that string to Trap or dead and kill it.

Shortcut for min NFA:

If there's a Trap state in DFA, we simply remove it.

min NFA>



Regular Expression of the question:

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

↓ → Any number of a & b can be there.

String starting with 'a'.

$(anything)^0 = \epsilon$ [empty string].

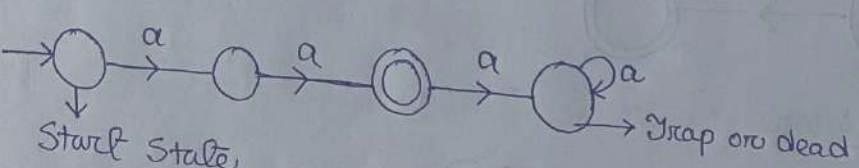
anything $\times \epsilon$ = anything

↓

$a \times \epsilon = a$

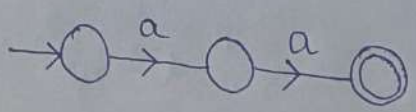
$\Sigma = \{a\}$ $\Sigma^* = \{\epsilon, a, aa, aaa, \ldots\}$ $L = \{aa\}$

⇒ DFA:



∴ DFA = 4 states [(n+1)+1]

NFA: Remove trap>



∴ NFA = 3 states [n+1]

NOTE:

$|W|$ = Length of string.

$|W| = n$ then DFA = $(n+1)+1$

NFA = $(n+1)$

Q: $L = \{aaaaa, \ldots\ldots, 100\}$, What is min DFA & min NFA states?

$\Rightarrow n = 100$. $\therefore |\text{min DFA}| = (n+1)+1$

$= (100+1)+1$

$= 102$

$|\text{min NFA}| = n+1$

$= 100+1$

$= 101$

Q: $L = \{aaaa, \ldots., 10^{10}\}$ then $|\text{min DFA}|$ & $|\text{min NFA}| = ?$

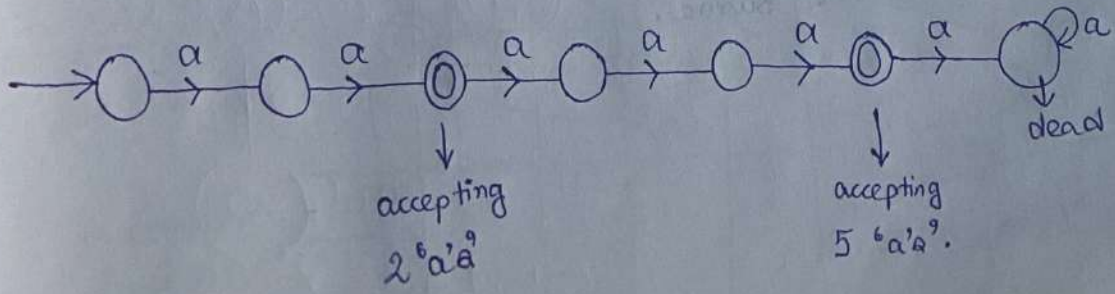
$\Rightarrow n = 10^{10}$ $\therefore |\text{min DFA}| = 10^{10}+2$

$|\text{min NFA}| = 10^{10}+1$

Q: $L = \{aa, aaaaa\}$, How many States are there in min DFA & min NFA?

$\Rightarrow$ Rule: If language is finite, do the DFA for longest string.

min DFA: Longest string here = aaaaa.



min NFA:

Remove the trap from DFA.

Rule for counting states:

Take the longest string and count states.

∴ |min DFA| = (5+1)+1 = 7 states.

|min NFA| = (5+1) = 6 states.

Rule for writing RE in case of finite language:

If the language is finite, simply put '+' in between.

∴ RE here = (aa + aaaaa)

Q: $\Sigma = \{a\}$, $L = \{w / n_a(w) = \text{even}\}$.

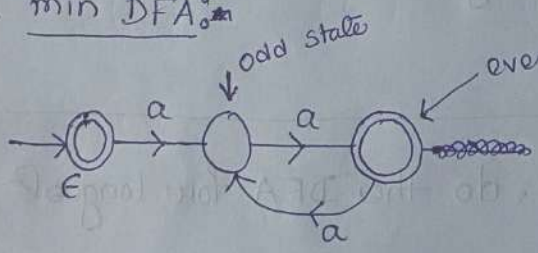
⇒ $n_a(w)$ = even ⟶ even no. of 'a'.

$L = \{aa, aaaa, aaaaaa, \ldots\ldots + \epsilon\}$

Take shortest because L = infinite.

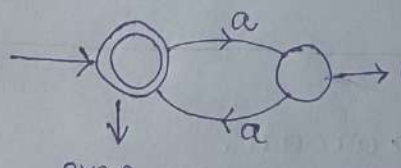
'0' a's because 0 is also even.

∴ min DFA:



X this is not min DFA.

⇒



even state

This is min DFA

In this example min DFA & min NFA are same.

$RE = a^{2n} / n \geq 0 \longrightarrow$ Not RE.

$= a^{2*} \longrightarrow$ not RE

$= (a^2)^* = \boxed{(aa)^*}$

↓ Regular Expression for the examples.

$(aa)^0 = \epsilon$
$(aa)^1 = aa$
$(aa)^2 = aaaa$
~~$(aaaaaa)^3$~~
$(aa)^3 = aaaaaa$ and so on.

Rule:

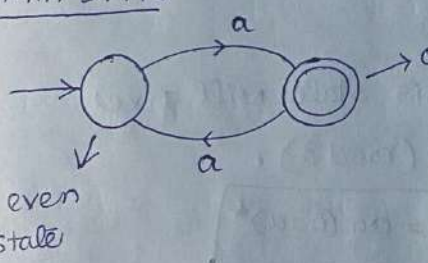
In DFA, NFA if 'ϵ' is in languages, then always initial state should be made final. There may be more final states (if needed).

Qs $\Sigma = \{a\}$, $L = \{w / n_a(w) = odd\}$

$\Rightarrow n_a(w) = odd \longrightarrow$ odd no. of 'a'.

$L = \{aaa, aaaaaa, \ldots\}$

∴ min DFA:



Here also min DFA & min NFA are same.

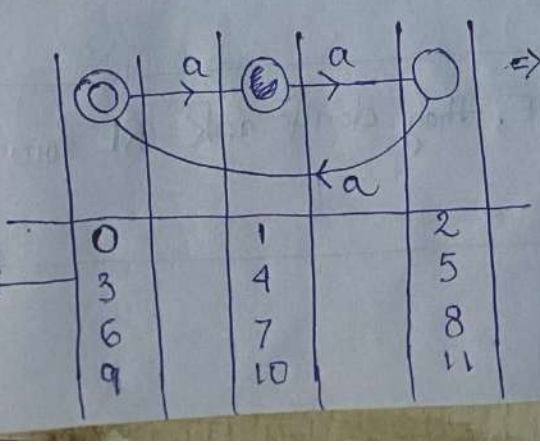
RE: $a^{2n+1} = a^{2n} . a$

$= \boxed{(aa)^* . a} \rightarrow$ RE

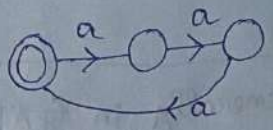
Qs $\Sigma = \{a\}$, L = no. of a's $\longrightarrow$ multiple of 3.

$\Rightarrow L = \{aaa, aaaaaa, aaaaaaaaa, \ldots\}$

min DFA:



multiple of '3' is this state



⇓ Mod machine

RE = $(aaa)^*$

In this example min DFA & min NFA is same.

NOTE:

In GATE they'll ask → no. of 'a's ⇒ Multiple of 100. What is min DFA & min NFA states?

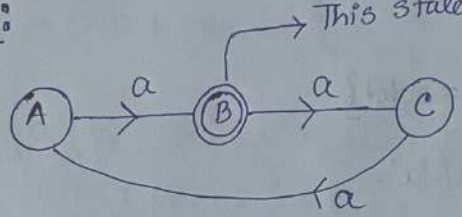
⇒ 100 min DFA & min NFA states are there.

Taking data from previous example →

no. of a's $\cong$ 1 (mod 3) >

↓ means, if we divide by 3, remainder should be 1.

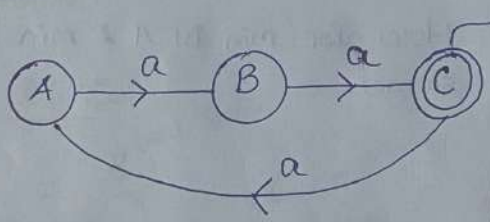
min DFA:



RE = a (aaa)*

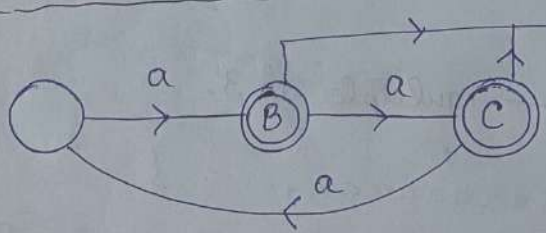
no. of 'a's $\cong$ 2 (mod 3) >

min DFA:



RE = aa (aaa)*

no. of 'a's $\ncong$ 0 (mod 3)

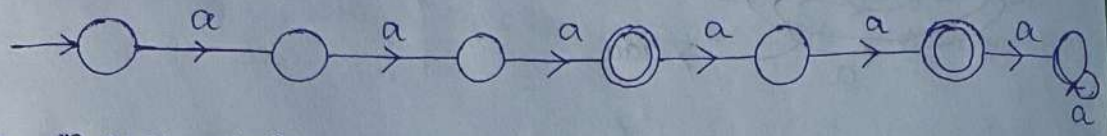


Generally in GATE, they don't ask RE for mod machines & grid machines.

Q: $\{a^n / n = 3 \text{ or } 5\}$

$L = \{aaa, aaaaa, \ldots\}$

min DFA:



DFA states $= (5+1)+1$
$= 7$

NFA states $= (5+1) = 6$

RE $= aaa + aaaaa$
$= aaa(\epsilon + aa)$

} Both are valid RE.

NOTE:

If you want to accept 'n' length string, then you need to have $(n+1)$ states.

Q: $\Sigma = \{a\}$, $L = (aa+aaa)^*$

$(aa+aaa) \rightarrow \{a^n / n = 2x + 3y, n \geq 0\}$ ✓

$(aa+aaa) \neq \{a^n / n = 2x \text{ or } n = 3y\}$ ✗

$(aa+aaa)^* \neq (aa)^* + (aaa)^*$

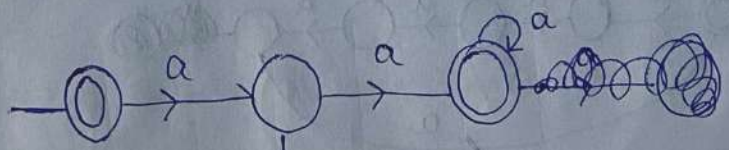
$L = \{\underset{0}{\epsilon}, \underset{2}{aa}, \underset{3}{aaa}, \underset{5}{aaaaa}, \underset{6}{aaaaaa}, \underset{8}{aaaaaaaa}, \ldots\}$

Here → 'a' is not present

NOTE:

$(r_1 + r_2)^* \neq (r_1)^* + (r_2)^*$

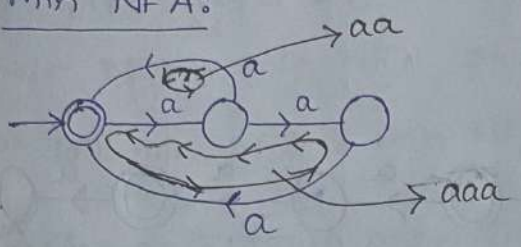
min DFA:



one 'a' is rejected

min NFA is same as min DFA but we can design one more for min NFA.

min NFA:



Q: GATE>

$\Sigma = \{a\}$, $L = (\overbrace{aaa}^{3} + \overbrace{aaaaa}^{5})^*$

$\Rightarrow L = \{\epsilon, \overset{3}{aaa}, \overset{8}{aaaaaaaa}, \overset{6}{aaaaaa}, 6+3=9, 5+5=10$

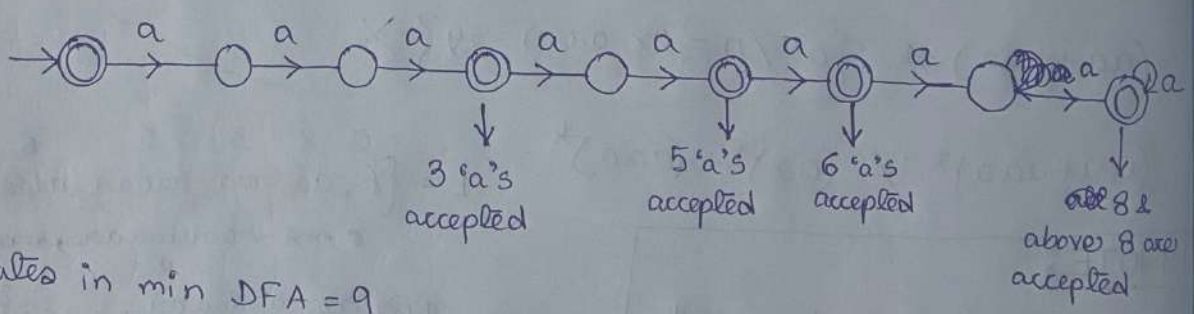
Q: GATE>

$\Sigma = \{a\}$, $L = (aaa + aaaaa)^*$

$\Rightarrow L = \{\overset{0,3}{aaa}, \overset{6}{aaaaaa}, \overset{8}{aaaaaaaa}, \overset{9}{aaaaaaaaa}, 5+5=10, 5+3+3=11$
$\ldots\}$

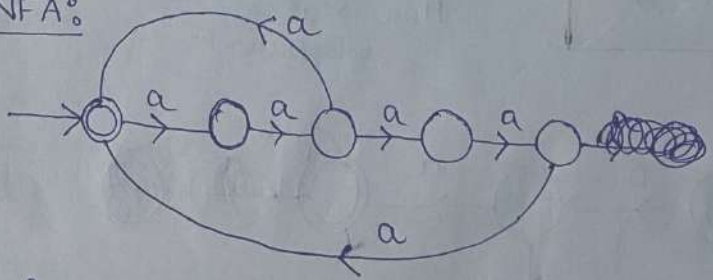
Here, 2, 1, 4, 7 are missing.

min DFA:



States in min DFA = 9

min NFA:



States in min NFA = 5

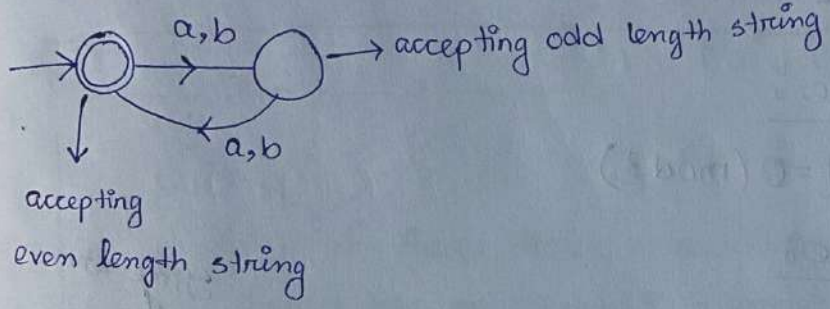
Q: $\Sigma = \{a, b\}$, $L = \{w / |w| = \text{even}\}$

⇒ $|w|$ = even means string length should be even.

$L = \{aa, ab, aaab, aaaa, bb, ba, baba \ldots + \epsilon\}$

length of ϵ is 0, which is even, we have to consider ϵ also.

min DFA:



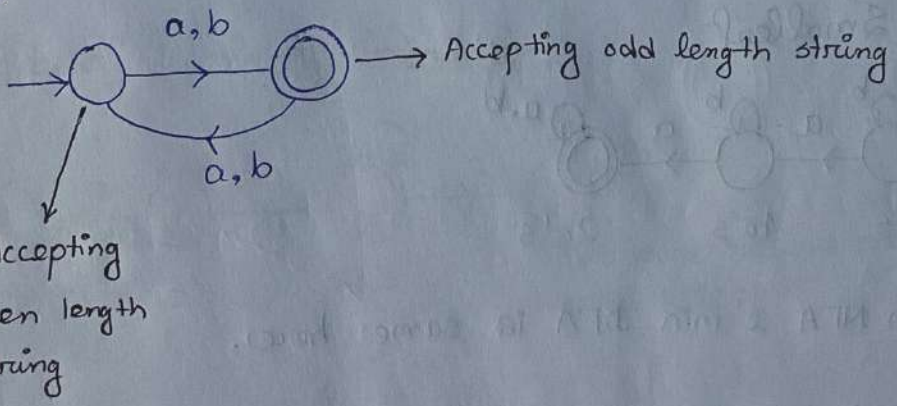
min NFA here is same as min DFA.

$RE = \{(a+b)(a+b)\}^*$

Q: $\Sigma = \{a, b\}$, $L = \{w / |w| = \text{odd}\}$

⇒ $L = \{a, aaa, b, bbb, aba, abb, bba \ldots\}$

min DFA:

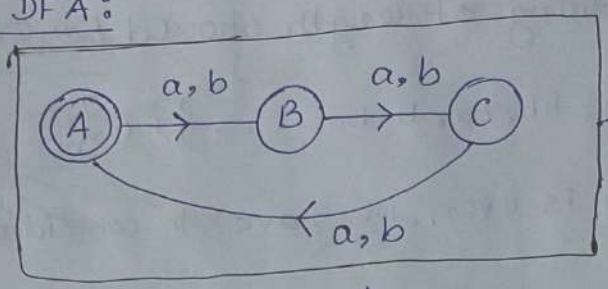


Here min DFA & min NFA are same.

$RE = (a+b)\left((a+b)(a+b)\right)^*$

Q: $\Sigma = \{a, b\}$, $|W| = 3n$ $(n \geq 0)$.

min DFA:



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

Here min DFA & min NFA is same.

State A will give:

$|W| = 3n$ or $|W| = 0 \pmod 3$

State B will give:

$|W| = 1 \pmod 3$

State C will give:

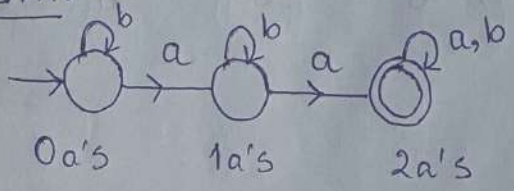
$|W| = 2 \pmod 3$

time stamp
01:41:23

Q: $\Sigma = \{a, b\}$ $L = \{W / n_a(W) \geq 2\}$
↳ Atleast.

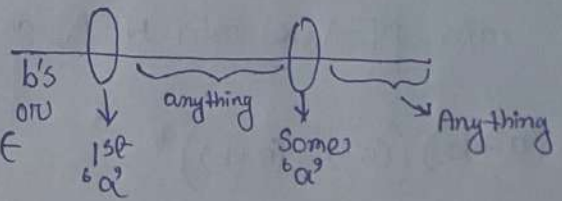
$\Rightarrow L = \{aa, aaa, aab, baa, aba, aaaa, \ldots\}$
↓
Smallest

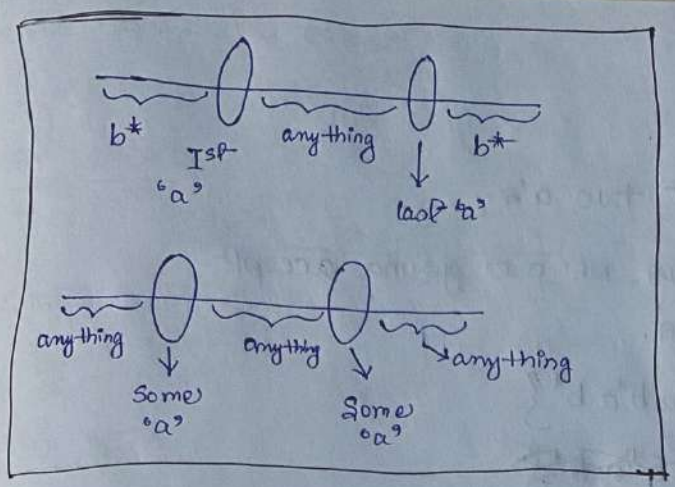
min DFA:

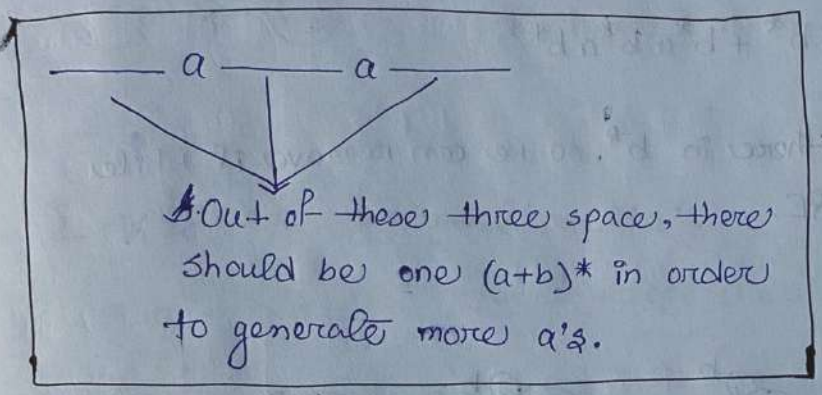


Here min NFA & min DFA is same here.

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





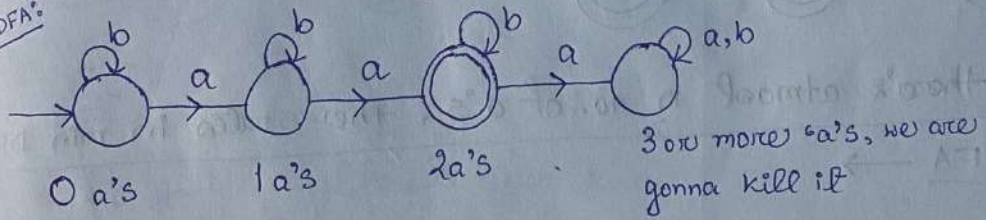


When there's atleast 'n' a's what will be the number of states in min DFA & min NFA ?

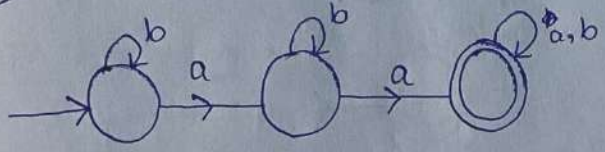
⇒ min DFA states = (n+1) } Because there's no trap or
min NFA states = (n+1) } dead state's there.

Q: $\Sigma = \{a, b\}$, exactly two a's:

⇒ min DFA:



min NFA:



When there's exactly n no. of a's, states in min DFA & min NFA:

min DFA = (n+1) + 1 → trap state.

min NFA = (n+1)

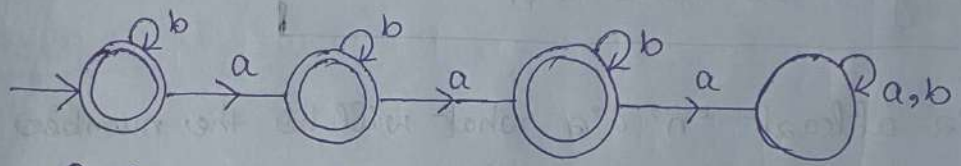
RE = $\boxed{b^*ab^*ab^*}$

Q: $\Sigma = \{a, b\}$, atmost two a's.

⇒ Atmost two a's means, we are gonna accept
→ 0 a's, 1a, & 2 a's.
$= \{\epsilon, b^*, b^*ab^*, b^*ab^*ab^*\}$
~~$\Rightarrow \{b^*, b^*ab^*, b^*ab^*ab^*\}$~~

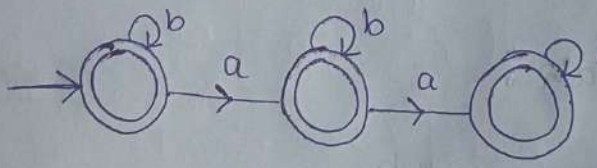
$RE = b^* + b^*ab^* + b^*ab^*ab^*$
↓
ϵ is already there in b^*, so we can remove it while writing the RE.

min DFA:



3 or more a's and any no. of b's, we kill them.

min NFA:



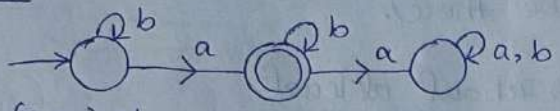
When there's atmost n no. of a's, the states in min DFA & min NFA →

min DFA = (n+1) + 1 → Trap state.
min NFA = (n+1)

Q: $\Sigma = \{a, b\}$, exactly 1 'a'.

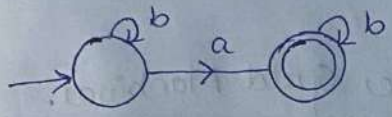
⇒ RE = $b^* a\, b^*$

min DFA:



States = (n+1)+1
= (1+1)+1
= 3

min NFA:



States = 1+1
= 2

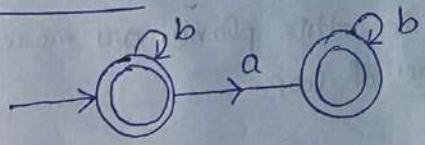
Q: $\Sigma = \{a, b\}$, atmost one 'a'.

⇒ RE = $b^* + b^* a b^*$. We'll accept 0 a's & 1 a's.

min DFA:

States = (n+1)+1
= 3

min NFA:

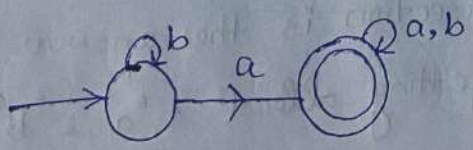


States = n+1
= 2

Q: $\Sigma = \{a, b\}$, atleast one a's.

⇒ RE = $b^* a\, (a+b)^*$

min DFA:



Here min DFA & min NFA is same

States = 2

NOTE: *****

When the question is about Exactly & Atmost

→ Trap state will be there.

When the question is about atleast

→ No Trap state will be there.

Grid Machines

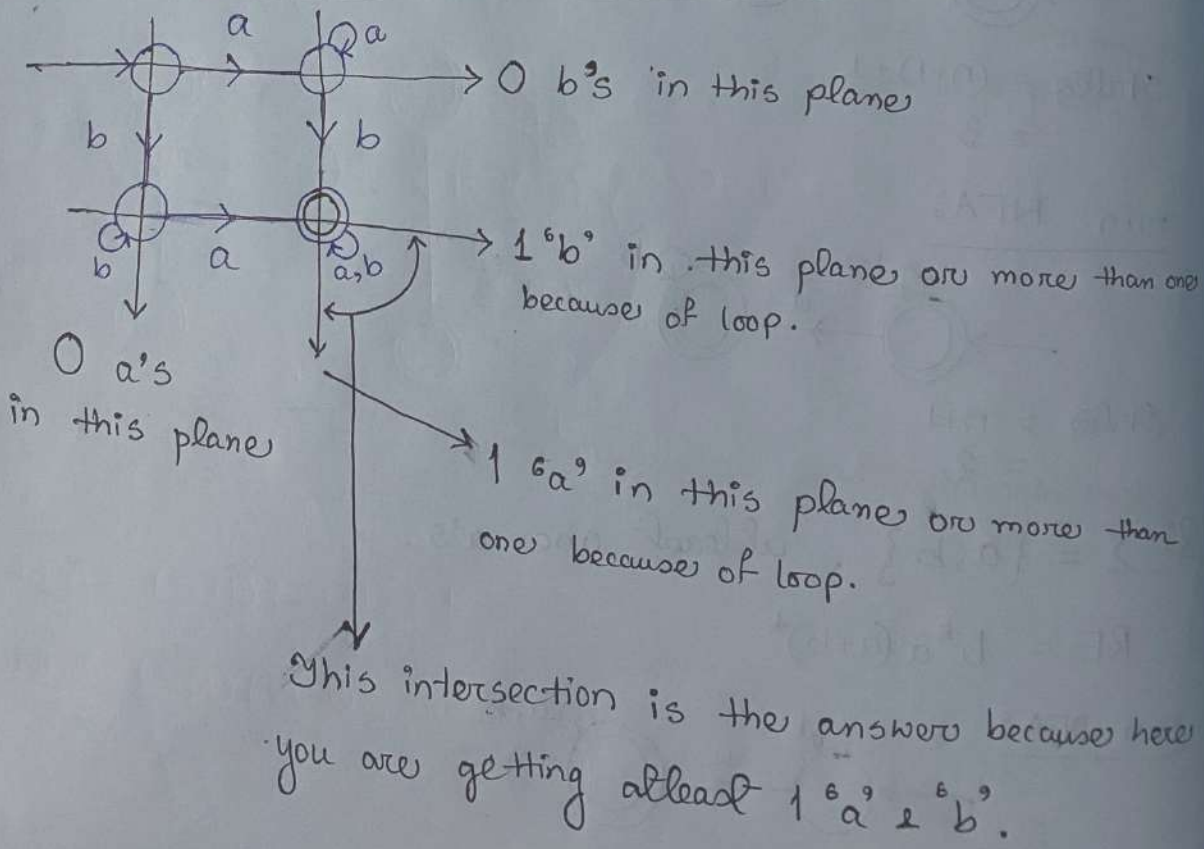
more than one input

When we have to count, we use Grid Machines.

Q: $\Sigma = \{a, b\}$, ≥ 1 'a' & ≥ 1 'b' (atleast one a's & one b's)

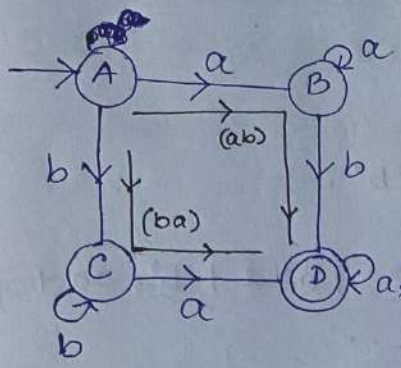
$\Rightarrow L = \{ab, ba, abb, aba, bab....\}$

We are going to count a → Horizontally.
count b → Vertically. *****



Clear diagram of the previous question:

⇒



How are we going to reach D?

⇒ either using ABD or ACD

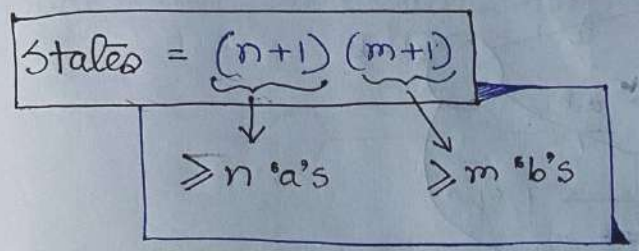
→ When we use ABD ⇒

We are seeing 1 'a' & 'b'.

→ When we use ACD ⇒

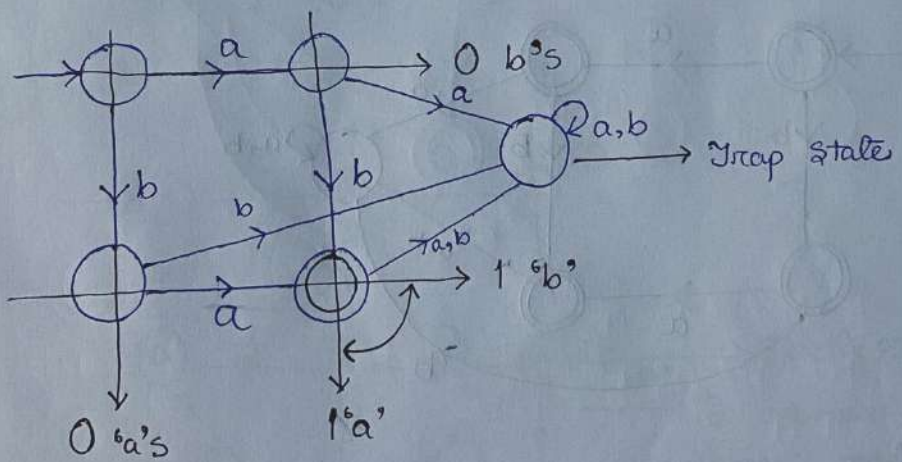
We are seeing 1 'b' & 'a'.

*****

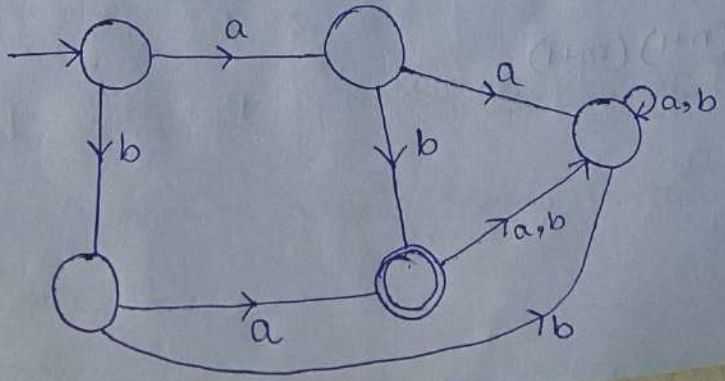


Q: $\Sigma = \{a, b\}$, = 1 'a' & = 1 'b'.

⇒ The question is about 'Exactly', so trap state will be there.



Clear Diagram:



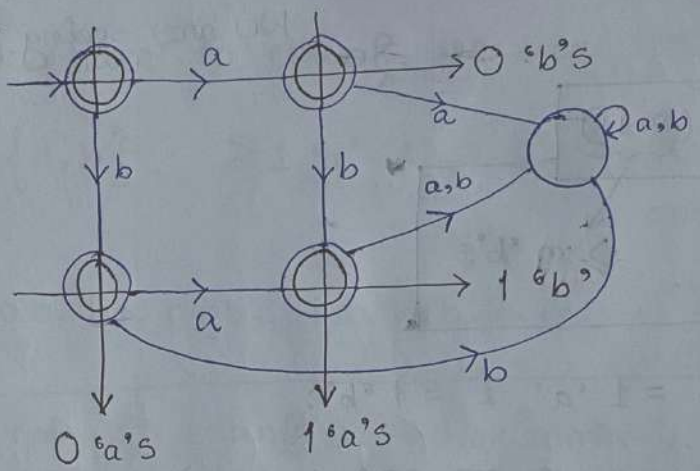
States in DFA:

DFA → $(n+1)(m+1)+1$ → For trap

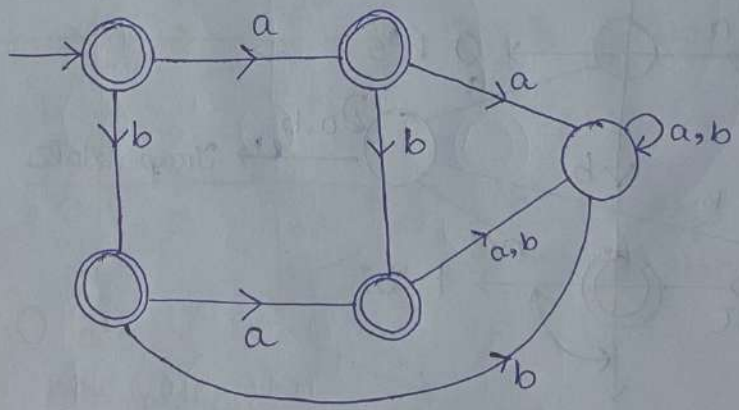
NFA → $(n+1)(m+1)$

Q: $\Sigma = \{a, b\}$, ≤ 1 'a' & ≤ 1 'b'.

⇒ The question is about 'Almost' 1 'a' & 1 'b', so trap will be there. We have to accept 0 & 1.



Clear Diagram:



States:

min DFA = $(n+1)(m+1)+1$

min NFA = $(n+1)(m+1)$

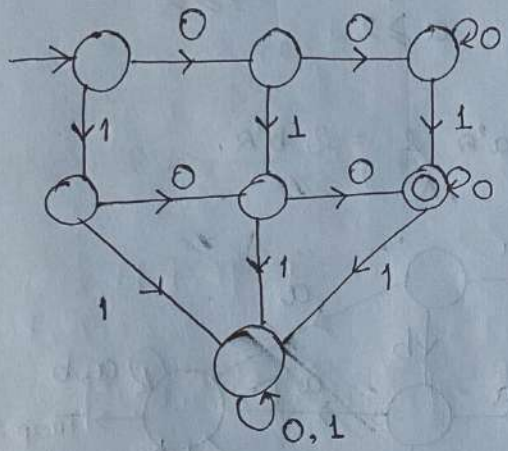
Example to use the formula → $(n+1)(m+1)+1$ & $(n+1)(m+1)$

⇒ ≤ 5 a's & ≤ 10 b's.

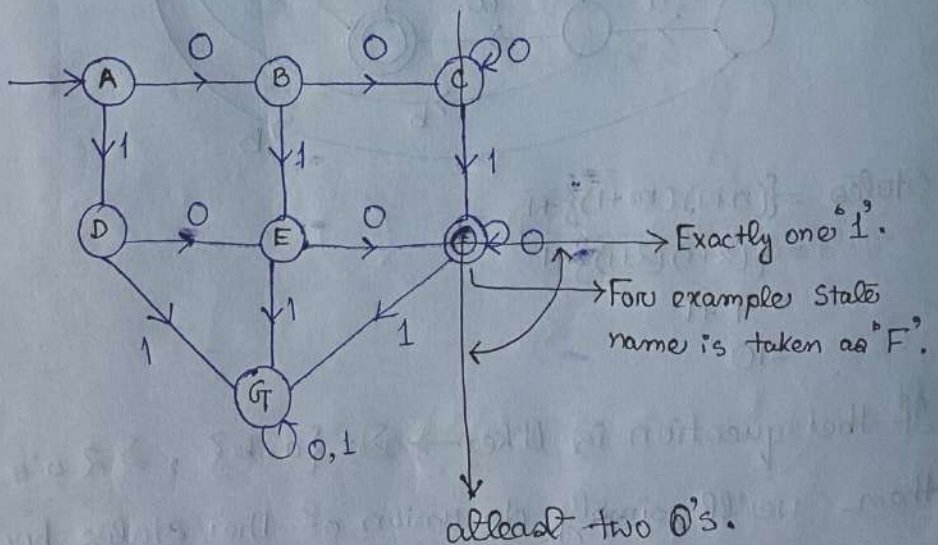
min DFA states = $(5+1)(10+1)+1$ | $(n+1)(m+1)+1$
= 67

min NFA states = $(5+1)(10+1)$ | $(n+1)(m+1)$
= 66

What does the following machine accept?



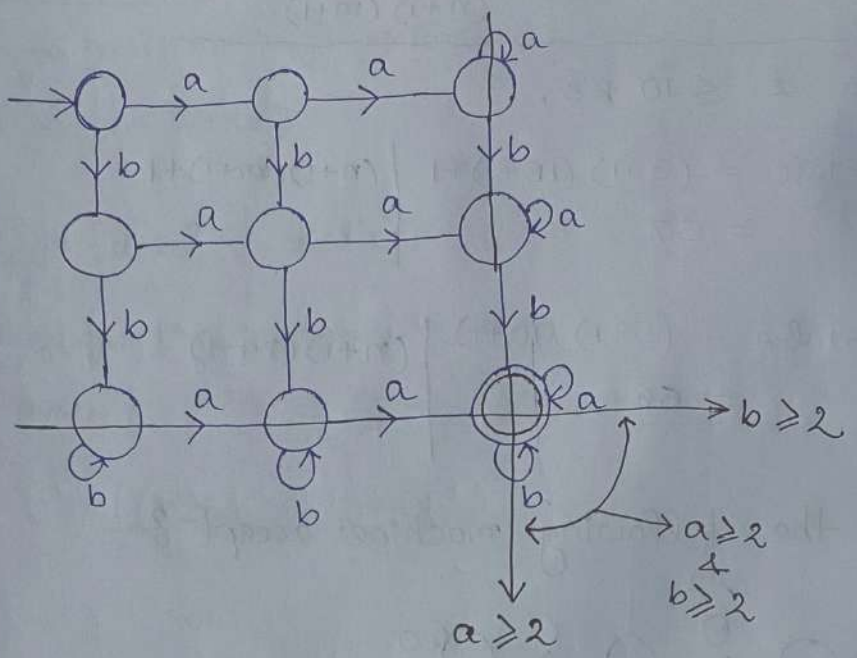
Explanation:



Ans: The following machine accepts. String containing atleast two 0's and one '1'.

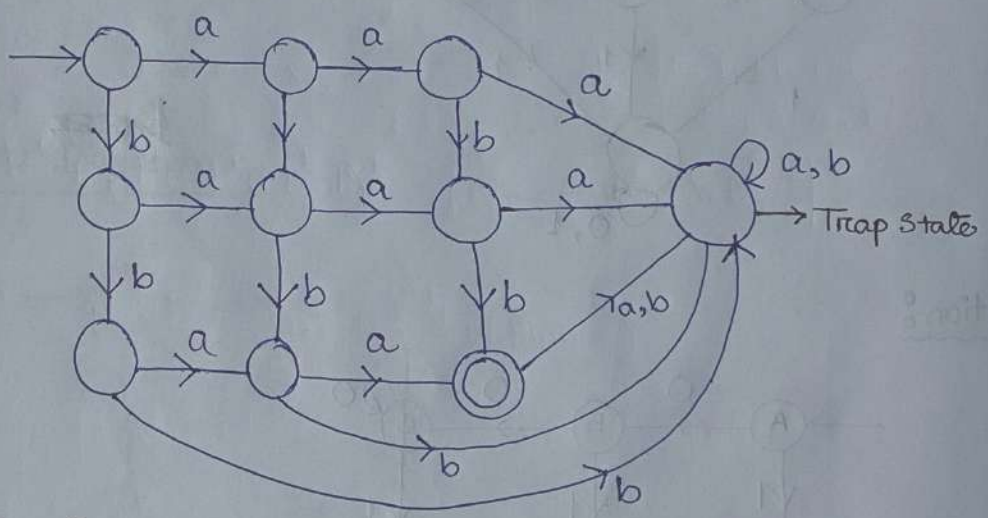
Q: $\Sigma = \{a, b\}$, ≥ 2 a's & ≥ 2 b's.

⇒



Q: $\Sigma = \{a, b\}$, $= 2$ a's & $= 2$ b's.

⇒



States $= \{(n+1)(m+1)\} + 1$

$= \{(2+1)(2+1)\} + 1$

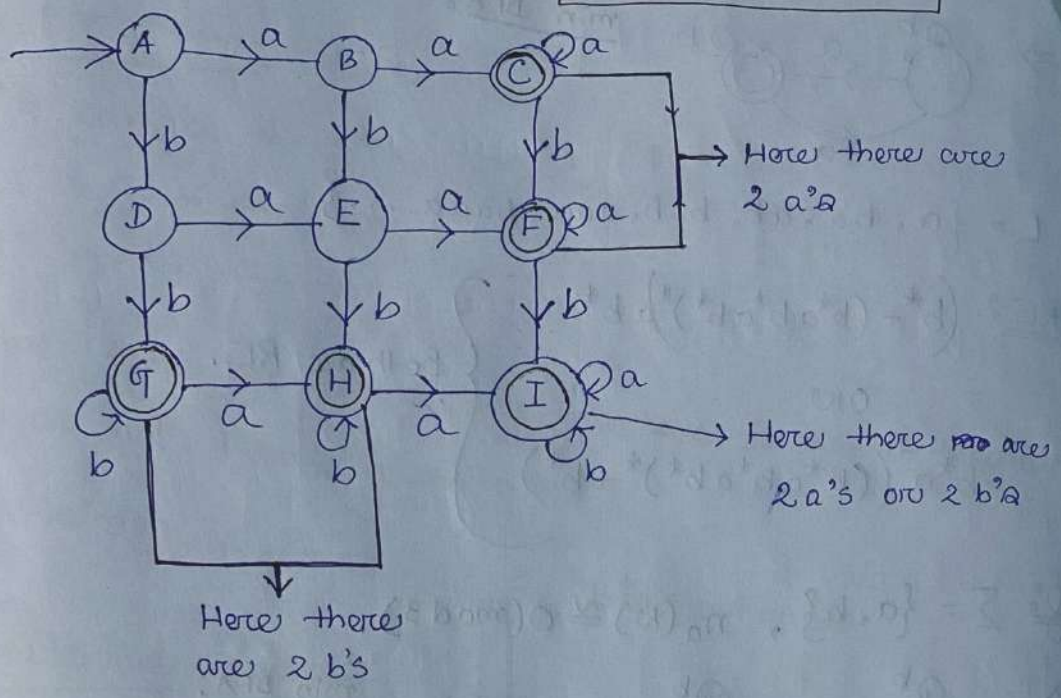
$= 9 + 1$

$= 10$

If the question is like → $\Sigma = \{a, b\}$, ≥ 2 a's or ≥ 2 b's then we'll simply do union of the states because 'OR' means Union.

Explanation with example:

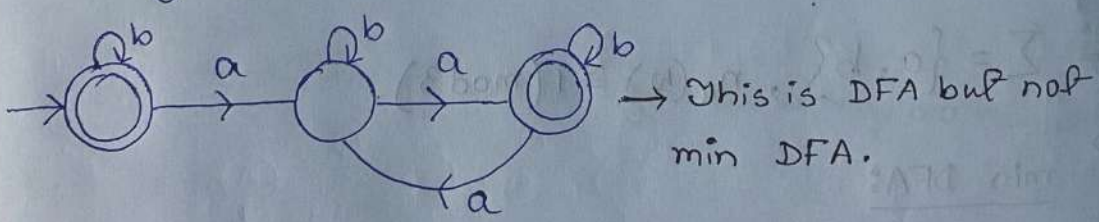
≥ 2 a's OR ≥ 2 b's



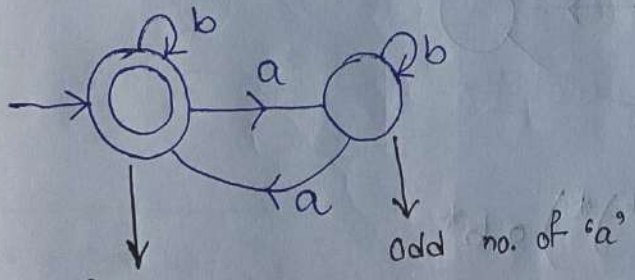
So answer will be → C, F, I, H, G

Q: $\Sigma = \{a, b\}$ even a's.

⇒ $L = \{ \epsilon, aa, baa, aba, aab, aabb, baab \ldots \}$



min DFA:

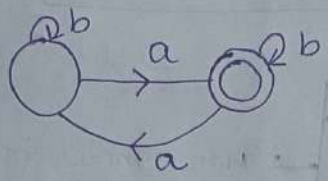


RE: $b^* + (b^*ab^*ab^*)^*$

a a

Q: $\Sigma = \{a, b\}$, odd a's.

⇒ min DFA:



$L = \{a, ba, ab, bab, aaa, baaa \ldots\}$

RE: $(b^* + (b^*ab^*ab^*)^*)ab^*$

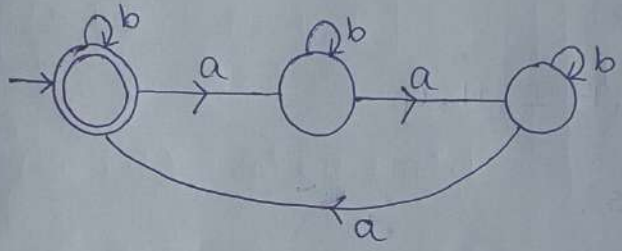
OR

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

} Both are RE.

Q: $\Sigma = \{a, b\}$, $n_a(w) \cong 0 \pmod 3$.

⇒



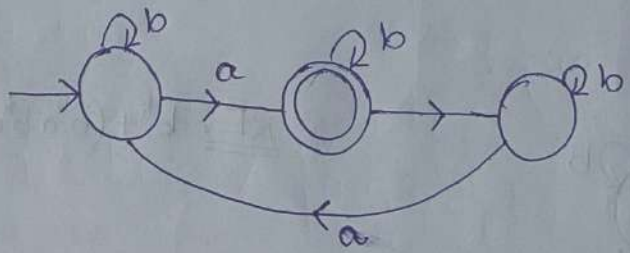
min DFA:

$L = \{\epsilon, b, bb, bbb, \ldots, aaa, baaa, abaa \ldots\}$

RE: $b^* + (b^*ab^*ab^*ab)^*$ [$b^* + (b^*ab^*ab^*ab^*)^*$]

Q: $\Sigma = \{a, b\}$, $n_a(w) \cong 1 \pmod 3$.

⇒ min DFA:



RE: $(b^* + (b^*ab^*ab^*)^*)ab^*$

Properties of Regular Expressions:

r, s, t ⟶ RE.

"r" can be ⟶ a^* or a^*b or ab^*.....

"s" " " ⟶ "

"t" " " ⟶ "

Assume there are two REs & you have to say if they are same or not. How can you do it?

⇒ 1) Intuition, 2) Properties, 3) Trial & Error.

Commutative Property:

$r + s = s + r$ ⟶ Always true

↓

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

$ab^* + a^* = a^* + ab^*$

$r.s \not\equiv s.r$

⟶ This not equal sign means, may or may not.

$a.b \neq b.a$

But there are exceptions ⟶

$a^*.a = a.a^* = \{a, aa, aaa, \ldots\}$

$= a^+$

$r^*r = rr^* = r^+$

Associative Property:

$r + (s + t) = (r + s) + t$

↓

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

$r(st) = (rs)t$

$a^*(bc^*) = (a^*b)c^*$

Both '+' & '.' are associative

Distributive Property:

$r.(s + t) = rs + st$

↓

$a^*(bc + d) = a^*bc + a^*d$

NOTE: ***
Plus (+) is not distributive over Dot (·).

$r + (s.t) \neq (r+s).(r+t)$

↓

$a + (bc) \neq (a+b).(a+c)$

⇓ This language has → $\{a, bc\}$

↳ This language has → $\{\{aa, ac, ba, bc\}$

$\therefore \{a, bc\} \neq \{aa, ac, ba, bc\}$

Identity:

$r + \phi = r$ → Phy = Empty language ***

$r.\epsilon = \epsilon.r = \epsilon.r.\epsilon = r$ ***

$r \cdot \phi = \phi$ ***

$r + \epsilon \neq r$

→ This means may or may not.

if "r" contains "ϵ" then

→ ~~r r r r r~~ $r + \epsilon = r$

Ex: $a^* + \epsilon = a^*$

↓

$\{\epsilon, a, aa, aaa \ldots\}$

"a^*" already contains ϵ so no need to write it seperately. Which means →

$a^* = a^*$

Exception:

$\phi^+ = \phi$

***

$r \cdot \phi = \phi = \phi \cdot r = \phi \cdot r \cdot \phi$

$a^+ + \epsilon = a^*$

↓

$\{a, aaa, aaaaa \ldots\} + \{\epsilon\}$

$= a^*$

$a^+ + \epsilon$

$= \{a, aa, aaa, \ldots\} + \{\epsilon\}$

$= a^*$

$\epsilon + \phi = \epsilon$

$\epsilon^* = \epsilon$

$\epsilon \cdot \phi = \phi$

$\phi^* = \epsilon$ ***

Union & Concatination:

$A = \{pq, r\}$, $B = \{t, uv\}$

AUB or A+B $= \{pq, r\} + \{t, uv\}$

$= \{pq, r, t, uv\}$

$A.B = \{pq, r\} . \{t, uv\}$

$= \{pqt, pquv, rt, ruv\}$

$r + r = r$ ⟶ $a + a = a$

$\{a\} = \{a\}$

$r \cdot r \neq r$

↓

$a \cdot a \neq a$

$\{aa\} \neq \{a\}$

$* \rightarrow 0, 1, 2, 3, \ldots$

$+ \rightarrow 1, 2, 3, \ldots$

$r^* \cdot r^* = r^*$

$r^* + r^* = r^*$

$(r^*)^* = r^*$

$(r^*)^+ = r^*$

$(r^+)^* = r^*$

$(r^+)^+ = r^+$

$r^* \cdot r^+$

$= r^* \cdot \{r, rr, rrr, \ldots\}$

$= (r^* r) + r^* rr + r^* rrr \ldots$

$= r^+ + r^* rr + r^* rrr \ldots$

$= r^+$

$r^+ \cdot r^* = r^+$

$r^+ \cdot r^+ = r^+$ ⟶ True or false?

$\{r, rr, rrr \ldots\} \cdot \{r, rr, rrr \ldots\}$

$= \{rr, rrr, rrrr \ldots\}$

$\therefore r^+ \cdot r^+ \neq r^+$ (Because you can't get a single 'r').

$r^+ \cdot r^+ = rr^+ = (r^+) r = r^* rr$

$= rr^* r$

$= r rr^*$

$r^+ = rr^*$

$= r^* r$ Using the value of r^+

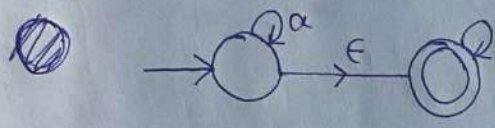
$a^*b^* + a^* = a^*b^*$

⇒ if we put 0 in the power of 'b', then we get $\rightarrow a^*$, which means

$\rightarrow (a^*b^*)$ is Superset of a^*

means : $L(a^*b^*) \supset L(a^*)$. So, we can remove 'a^*'

We want to do a Finite Automata on a^*b^*:



$\rightarrow \epsilon$-NFA

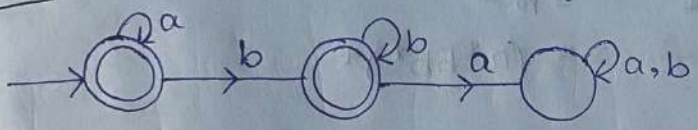
We want to eliminate 'ϵ' $\rightarrow$

$a^*(\epsilon + bb^*)$

This is b^* (We'll learn later)

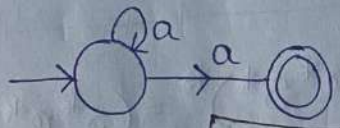
$= a^* + a^*bb^*$

min DFA:



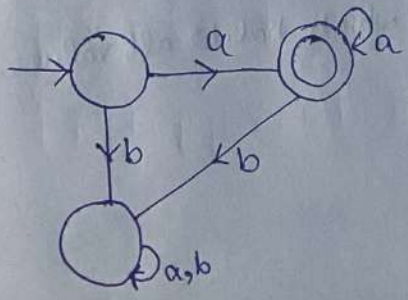
Q: $\Sigma = \{a, b\}$ a^*a.

min NFA:



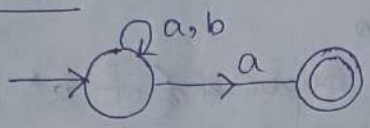
Now we can write $\rightarrow$ $a^*a = a\,a^*$ $\rightarrow$ More convenient to write min DFA.

min DFA:



Q: $(a+b)^*a$, min NFA & DFA.

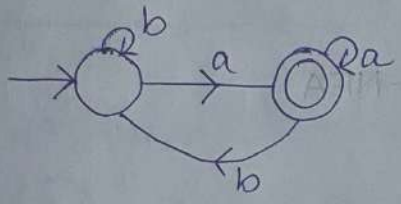
min NFA:



min DFA:

$(a+b)^*a$ → It means all strings ending with 'a'.

$L = \{a, aa, ba \ldots\}$



$$\epsilon + rr^* = r^* = r^*r + \epsilon = rr^* + \epsilon \quad ***$$

$$\epsilon + \underbrace{(a^*bb^*)}_{r}\underbrace{(a^*bb^*)^*}_{r^*} = \epsilon + rr^* = r^* = (a^*bb^*)^*$$

MOST IMPORTANT PROPERTY: Many times in GATE

$$(r^* + s^*)^* = (r+s)^* = (r^*s^*)^* = (s^*r^*)^* \quad *****$$

$$(r^* + s^*)^* = (\epsilon, r, rr, rr \ldots + \epsilon, s, ss, sss \ldots)^*$$

$$= \boxed{(r+s)^*} \rightarrow \text{Universal which holds all values}$$

$(r^* + s^*)^* = (r^* + s^* + r^+ s^*)$

$= (\epsilon, r, rr, rrr, \ldots + \epsilon, s, ss, sss, \ldots, r^+ s^*)^*$

$= (r+s)^*$

$(s^* r^*)^* = (s^0 r^*)^*$ then $(s^* r^0)^*$

$= (r^* + s^* + r^+ s^+)^*$

$= (\epsilon, r, rr, rrr, \ldots + \epsilon, s, ss, sss \ldots, r^+ s^+)^*$

$= (r+s)^*$

$(r^* s^*)^* = (r^* + s^* + r^+ s^+)^*$

$= (\epsilon, r, rr, rrr, \ldots + \epsilon, s, ss, sss \ldots, r^+ s^+)^*$

$= (r+s)^*$

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

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

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

IV) $(a+b)^*$

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

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

Which are equivalent?

⇒ I) & IV) are equivalent.

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

$= \underbrace{(a+b)^* a^*}_{(a+b)^*} a$

$= (a+b)^* a$

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

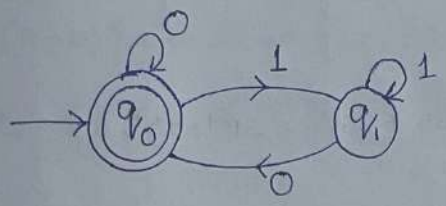
∴ II) & V) are equivalent.

III) $(a+b)^* b a^*$ = The strings may contain a or may contain b at the end.

VI) $(a+b)^* b$ = Strings will always contain b in the end.

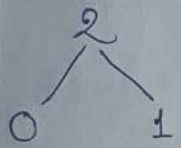
Q: $\Sigma = \{0,1\}$, $L = \{w/w$ when interpreted as a binary number, is divisible by $2\}$

NOTE:
If the question asks about divisibality by some number or $1 \pmod 2$ or $0 \pmod 3$, then it's **Mod Machines**.



Number of states in these types of questions will no. of remainders

For example, in this question no. of remainders = 2



⇒ Either you'll get 0 or 1 as remainder.

So, no. of states = 2.

Number System basics for TOC only:

Base
1 → (0) → (0 = 1, 00 = 2, 000 = 3, 0000 = 4....) ⇒ Unary
2 → (0,1) → (0 = 0, 01 = 1, 10 = 2, ...) ⇒ Binary
3 → (0,1,2) → (0 = 0, 1 = 1, 2 = 2, ...) ⇒ Ternary
4
5
6
7
8 → (0,1,2,3,4,5,6,7)
⋮
16
⋮

Base $n \rightarrow \underbrace{(0,1,2,\ldots n-1)}_{n \text{ symbols.}}$

Base 2 calculation: Binary

1 0 1 → 2^2 2^1 2^0

$1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 4+0+1 = 5$

Base 3 calculation: Ternary

1 0 1 → 3^2 3^1 3^0

$1\times 3^2 + 0\times 3^1 + 1\times 3^0 = 9+0+1 = 10$

Binary:

(___)0 — assume decimal value is 'x'; append a zero

⇒ It'll become '2x' after appending the '0'.

(___)1 — 'x'; append a '1'

$(2x)+1$

⇒ The number will be doubled & a 1 will be added. It'll become $(2x)+1$.

Ternary:

(___)0 $= 3x+0$ (x)

(___)1 $= 3x+1$ (x)

(___)2 $= 3x+2$ (x)

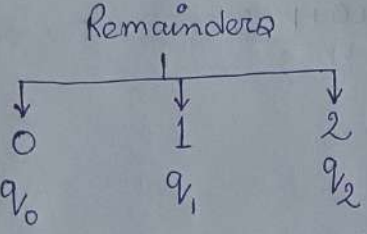
3 → base; x → The decimal value; 2 → Number Appended

Q: $\Sigma = \{0, 1\}$ $L = \{W / W$ is a binary number divisible by $3\}$

divisible → Mod machines

3 → Number of states

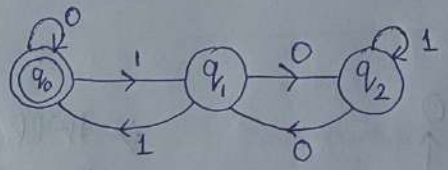
⇒



State Transition Table:

	0	1
→ * q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2

⇒



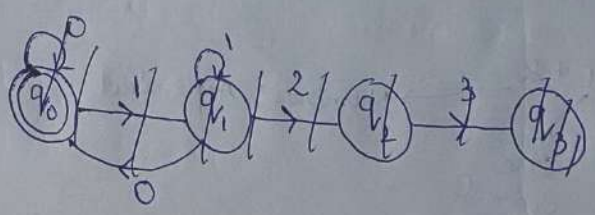
Q: $\Sigma = \{0, 1, 2, 3\}$ divisible by (4) → 4 states.

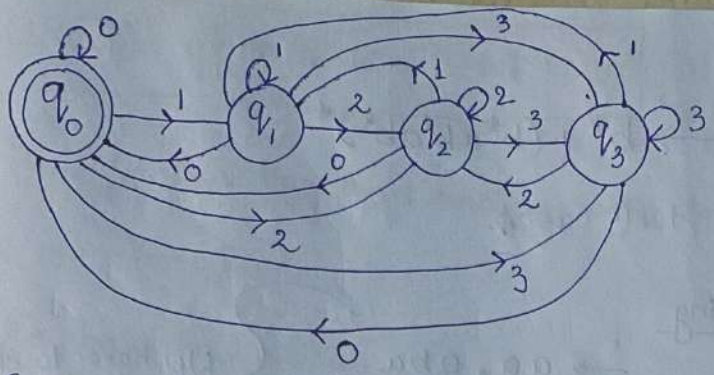
Base 4

Decimal

State Transition Table:

	0	1	2	3
→ * q_0	q_0	q_1	q_2	q_3
q_1	q_0	q_1	q_2	q_3
q_2	q_0	q_1	q_2	q_3
q_3	q_0	q_1	q_2	q_3





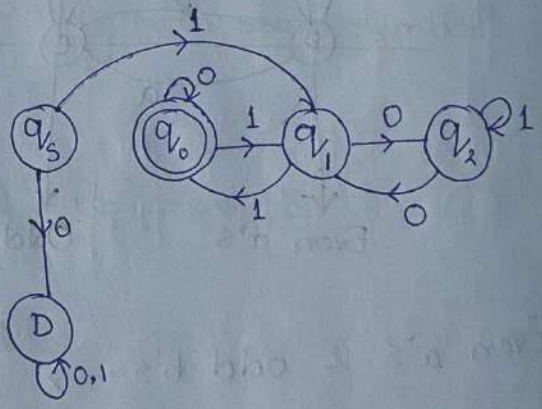
$\Sigma = \{0, 1, 2, 3\}$, Divisible by 4.

Q: $\Sigma = \{0, 1\}$. W/W is divisible by 3 & starts with '1'.

⇒ Here we can't start from the initial state because valid strings might get rejected because of the given condition. So, here we need an extra state.

State Transistion Table:

	0	1
$\rightarrow q_5$	D	q_1
$*q_0$	q_0	q_1
q_1	q_2	q_0
q_2	q_1	q_2
D	D	D



LECTURE 4:

For a given language L, there will be many REs. Most of the time we use intuition to get the RE. But there's an algorithm which is slow. Both intuition & Algo will give many RE.

Example:

Even no. of a's $\longrightarrow b^* + (b^*ab^*ab^*)^*$

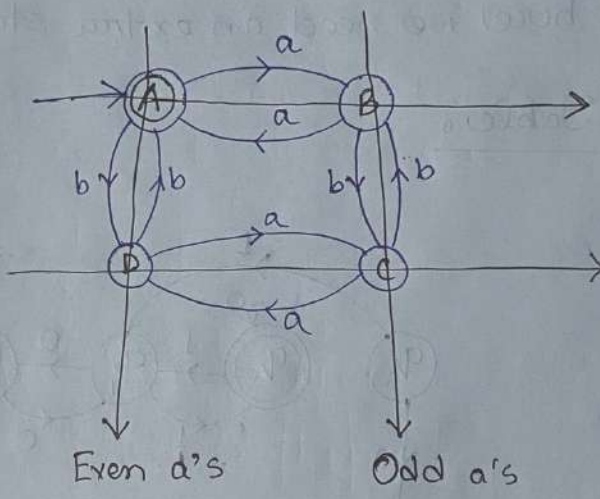
$\Rightarrow$ Generate some test cases.

Starting	Ending	
a	a	$\longrightarrow$ aa, aba
a	b	$\longrightarrow$ aab, aabb
b	a	$\longrightarrow$ baa, [illegible], baba
b	b	$\longrightarrow$ baab, bb

We have to check if all these strings are generated from the given RE.

Q: $\Sigma = \{a, b\}$, a's $\longrightarrow$ even, b's $\longrightarrow$ even.

$\Rightarrow$



Final state = A

Even a's & odd b's $\longrightarrow$	Final State: d
Odd a's & odd b's $\longrightarrow$	C
even a's OR odd b's $\longrightarrow$	ADC

For these kind of questions $\longrightarrow$
min DFA = m x n
min NFA = m x n

LECTURE 4 ENDED

LECTURE 7 of RE is PENDING

Q: $\Sigma = \{a, b\}$, starting and ending with 'a'.

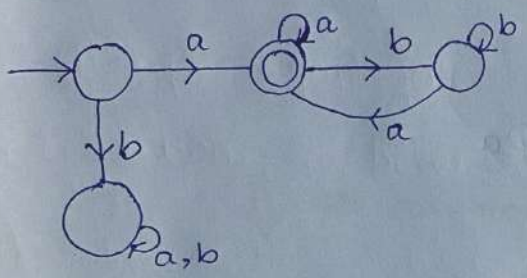
⇒ RE = $a + a(a+b)^*a$

$L = \{a, aa, aba, abba \ldots\}$

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

↳ We are putting it because using $a(b+a)^*a$, we cannot generate a single 'a'.

min DFA:

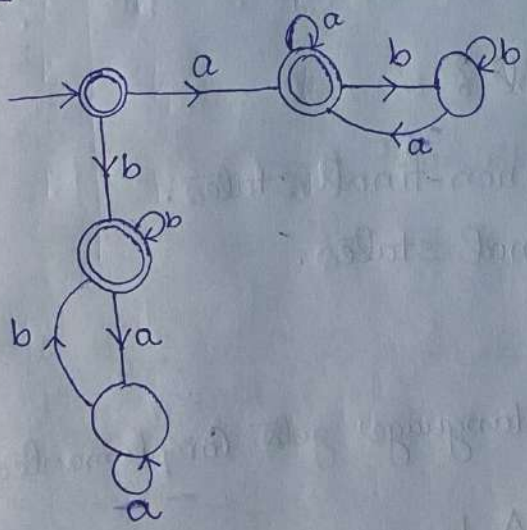


Q: $\Sigma = \{a, b\}$, starting and ending with same symbol.

⇒ $L = \{\epsilon, a, b, aa, bb, \ldots\}$

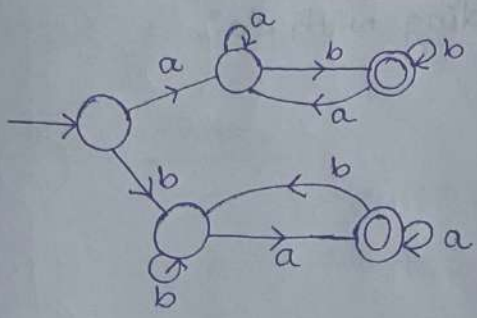
RE = $\epsilon + a + b + a(a+b)^*a + b(a+b)^*b$

min DFA:



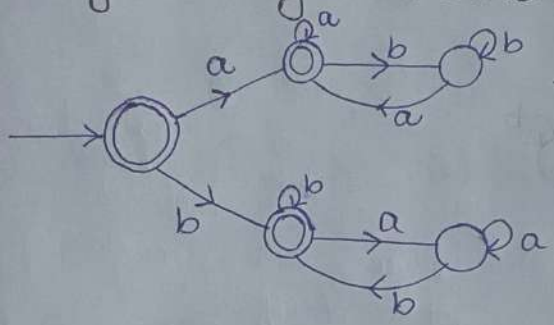
Q: $\Sigma = \{a, b\}$, Starting & Ending with different symbols. (L_1)

⇒



RE = a(a+b)*b + b(a+b)*a

Starting & Ending with same symbol. (L_2)



L_1 is complement of L_2.

$\therefore \boxed{L_1 = \overline{L_2}}$

DFA_1 —Complement→ DFA_2

(L_1) $\qquad$ $\boxed{\overline{L_1}}$

to

How, make the complement?

⇒ Make final states to non-final states.
Non final states to final states.

NOTE:

If you complement a DFA, language gets complemented.

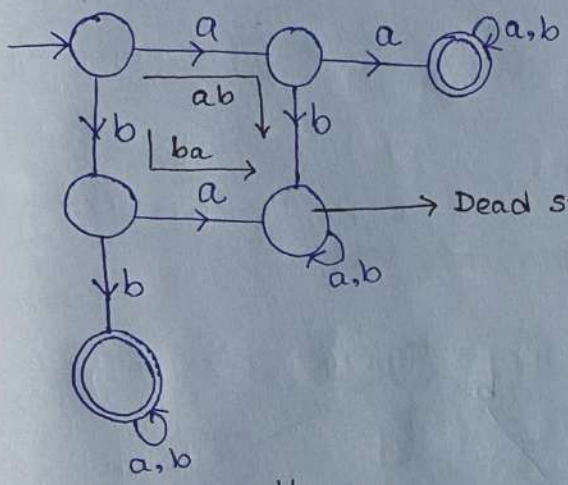
If you complement an NFA, language may or may not be complemented.

Q: $\Sigma = \{a, b\}$, First two symbols are same.

$\Rightarrow L = \{aa, bb, aab, bba \ldots\}$

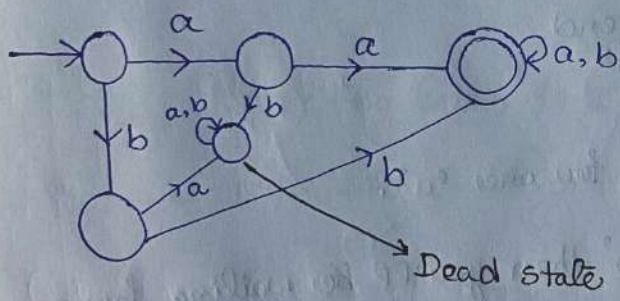
$RE = aa(a+b)^* + bb(a+b)^*$

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



⇓ We can merge final states here

min DFA:



NOTE:

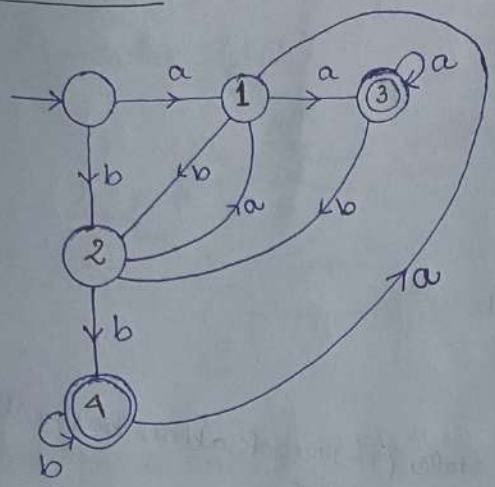
For now we'll design the DFA normally, then we'll see if we can merge the states or not.

Q: $\Sigma = \{a, b\}$, Last 2 symbols are same.

$\Rightarrow RE = (a+b)^*aa + (a+b)^*bb$

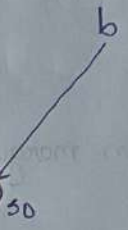
$= (a+b)^*(aa+bb)$

min DFA:



State 1: a → already seen

b → if a 'b' comes then you have to wait for a 'b' so that it'll end with a 'bb'



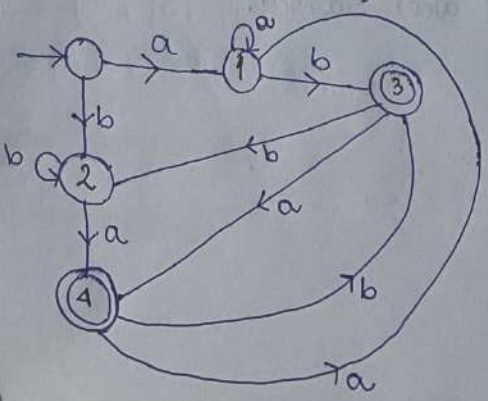
State 2: b a (a) → wailing for one 'a'.

State 3: If you get a 'b' then you'll be wailing for another 'b' so that it ends with 'bb'. Go to state 2.

State 4: If you get one 'a'. you'll be wailing for another 'a' so that it ends with 'aa'. Go to state 1.

Q: $\Sigma = \{a, b\}$ Last two symbols are different.

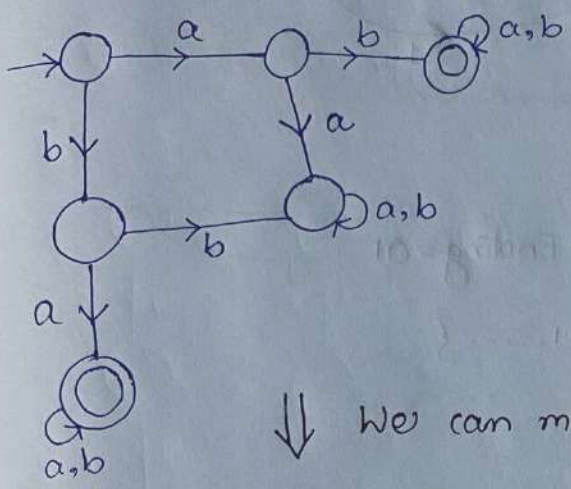
$\Rightarrow RE = (a+b)^* (ab+ba)$



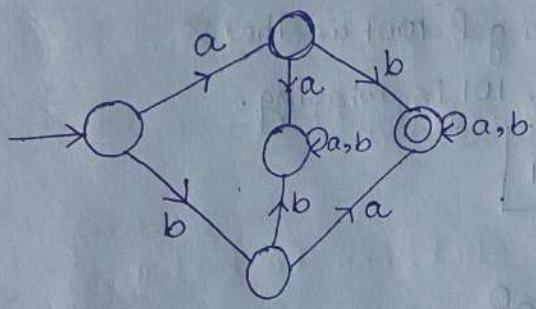
State 1: Wailing for 'b'.
State 2: Wailing for 'a'.

Q: $\Sigma = \{a, b\}$, First two symbols are different.

⇒ RE: $(ab + ba)(a+b)^*$



⇓ We can merge the final states



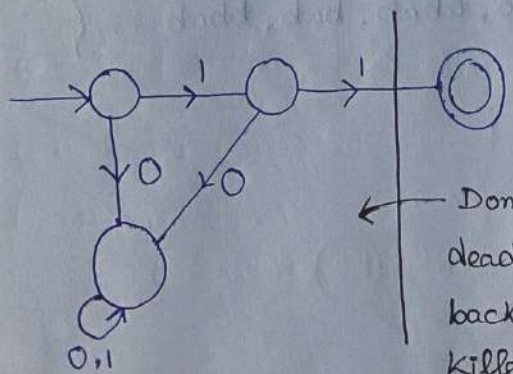
Q: $\Sigma = \{0, 1\}$, Starting with 11 & ending with 11.

⇒ $11(0+1)^*11$ → With this, the smallest string we can get is 1111

$L = \{11, 111, 1111, \ldots\}$

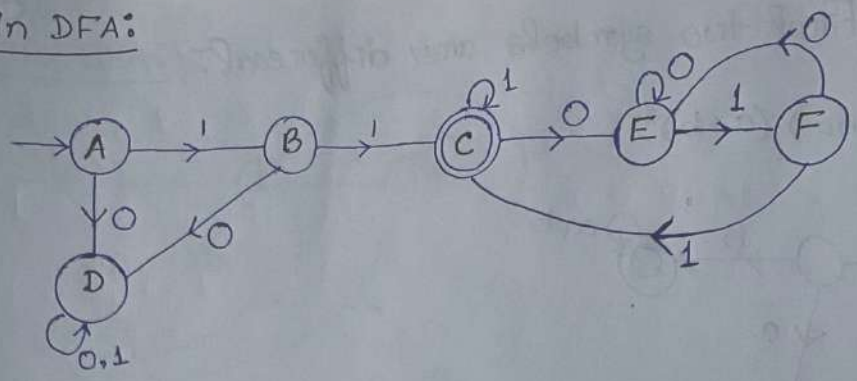
We have to get these two for RE.

∴ $\boxed{RE = 11 + 111 + 11(0+1)^*11.}$



← Don't go back after this because dead state is there, if you go back, then valid strings will be killed.

min DFA:



Q: $\Sigma = \{0,1\}$ Starting = 10 & Ending = 01

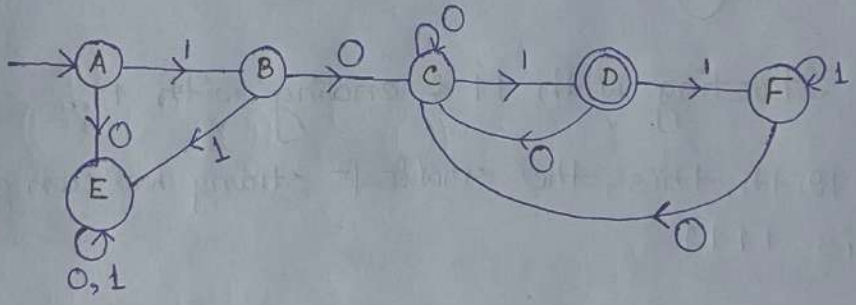
$\Rightarrow L = \{101, 1001, 100001, 101001, \ldots\}$

~~OR~~ $10(0+1)^*01$

↓

With this we can get 1001 as the smallest string. 101 is missing.

$\therefore$ **RE = 101 + 10(0+1)*01**



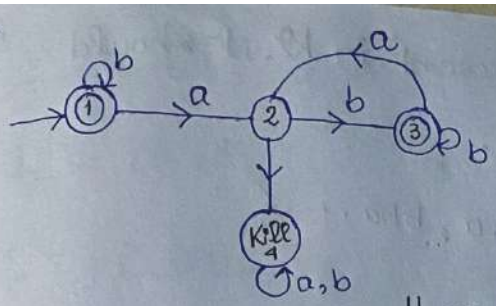
LECTURE-9: (This Model will be used in Context free Languages)

Q: $\Sigma = \{a, b\}$ L = {w / if 'a' is present in w, then it is always followed a 'b'}

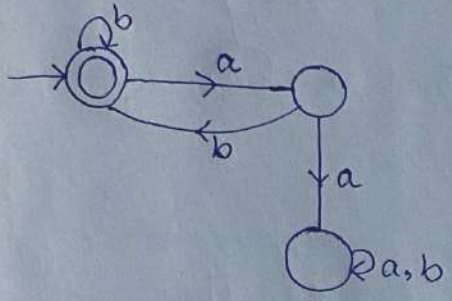
$\Rightarrow L = \{\epsilon, b, bb, bbb, \ldots, ab, abab, bab, bbab \ldots\}$

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

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



⇓ We can merge them



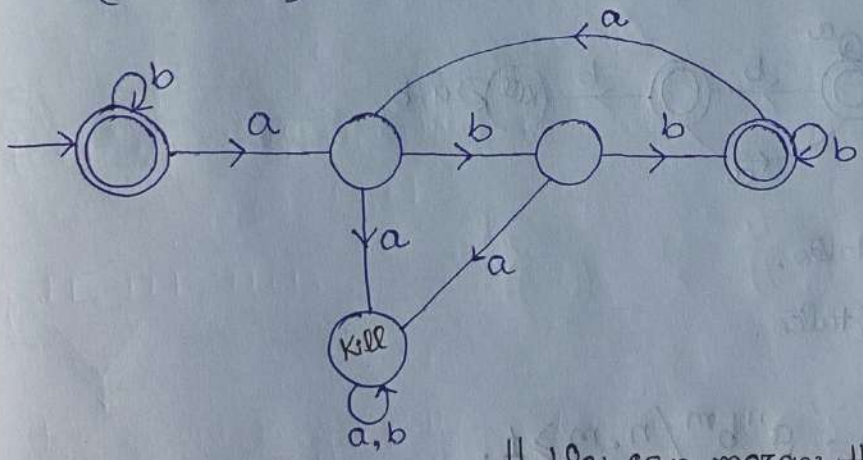
min DFA → 3 States
min NFA → 2 States

Q: $\Sigma = \{a, b\}$, Every 'a' should be followed by 'bb'.

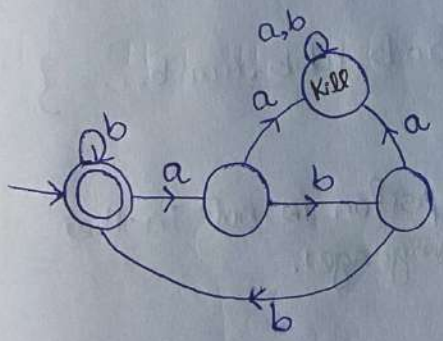
⇒ $L = \{\epsilon, b, bb, bbb, \ldots, abb, babb, abbb \ldots\}$

$RE = b^* + (abb + b)^*$

$= (b + abb)^*$



⇓ We can merge them

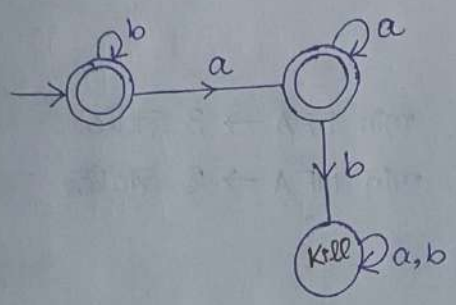


min DFA → 4 States.
min NFA → 3 States.

Q: $\Sigma = \{a, b\}$, $L = \{w$ / if 'a' is present in w, it should never be followed by a 'b'$\}$.

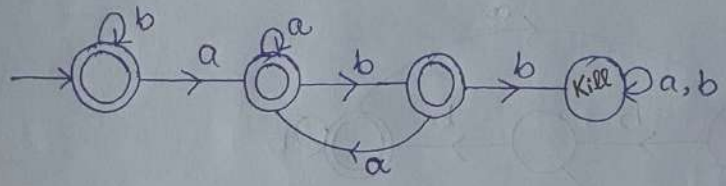
$\Rightarrow L = \{\epsilon, b, bb, bbb, \ldots, aa, aaa, a, bba \ldots\}$

$RE = b^*a^*$



Q: $\Sigma = \{a, b\}$, $L = \{w$ / if a is present in w, it should never be followed by bb$\}$.

$\Rightarrow L = \{\epsilon, b, bb, \ldots, aa, abab, baa, ba, baab, a \ldots\}$

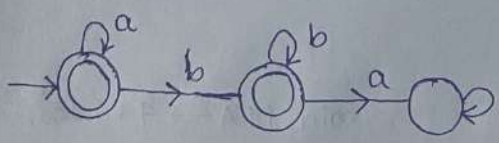


min DFA = 4 States.

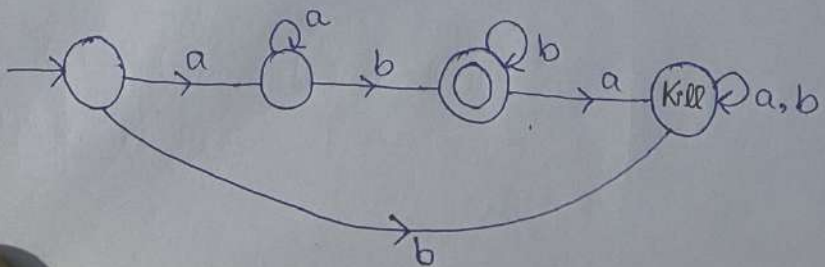
min NFA = 3 States.

Q: $\Sigma = \{a, b\}$, $a^n b^m / n, m \geq 1$.

$\Rightarrow RE = a^+b^+$ $\quad L = \{ab, aab, abb, aaab, \ldots\}$



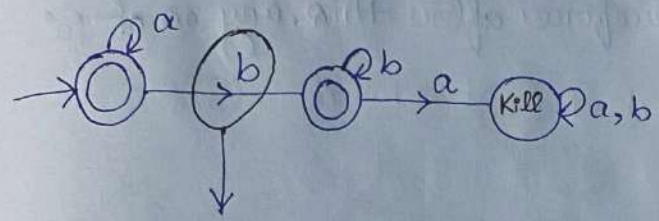
✗ Epsilon is not in the language.



Q: $\Sigma = \{a, b\}$, $a^n b^m / n, m \geq 0$

⇒ RE = $a^* b^*$

L = ~~{ϵ, aab, ...}~~ { ϵ, a^*, b^*, any no. of a's followed by any no. of b's }



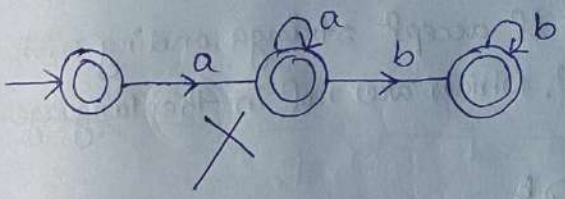
Remember that one 'b' has come and so, no 'a' should come.

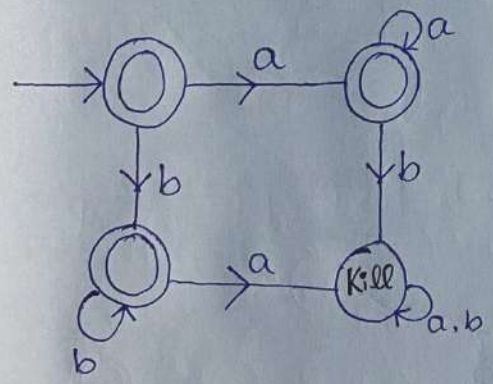
min DFA = 3 states.

min NFA = 2 states.

Q: $\Sigma = \{a, b\}$, $(a^* + b^*)$.

⇒ L = { ϵ, a, aa, aaa, ..., ϵ, b, bb, bbb, ... }

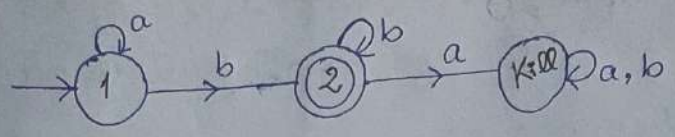




Q: $\Sigma = \{a, b\}$, $a^n b^m / n \geq 0, m \geq 1$.

⇒ RE = $a^* b^+$

L = { b, bb, bbb, ..., ab, aab, aaabb, ... }



⇒ State 1: Waits for 'b'

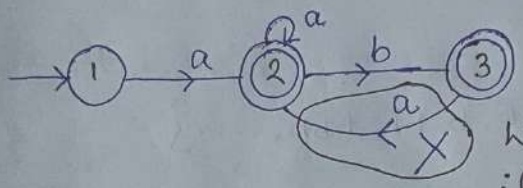
State 2: Seen a 'b'. Therefore after this, any no of 'a's will get killed.

min DFA = 3 States
min NFA = 2 States

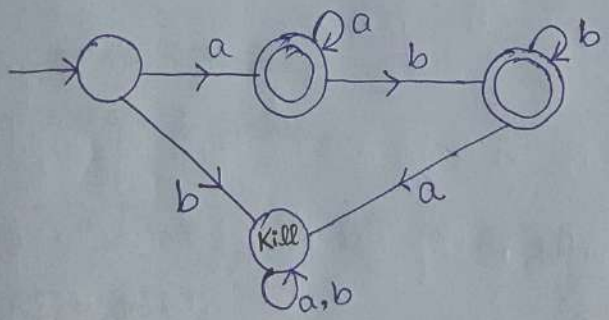
Q: $\Sigma = \{a, b\}$, $a^n b^m / n \geq 1$, $m \geq 0$.

⇒ RE = $a^+ b^*$

$L = \{a, aa, aaa, \ldots, ab, aab, aabb, \ldots\}$



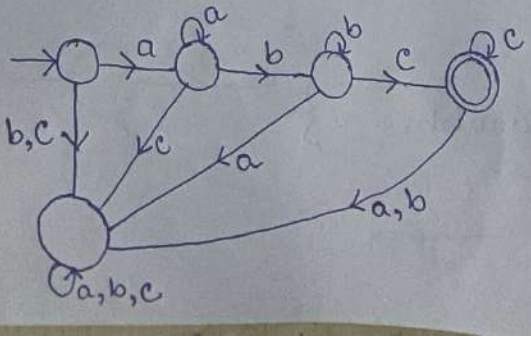
We can't do this because it'll accept strings ending with 'a', which are not in the language.



Q: $\Sigma = \{a, b, c\}$ $\{a^n b^m c^l / n, m, l \geq 1\}$

⇒ RE = $a^+ b^+ c^+$ or $aa^* bb^* cc^*$

$L = \{abc, aabc, abbc, abcc, \ldots\}$



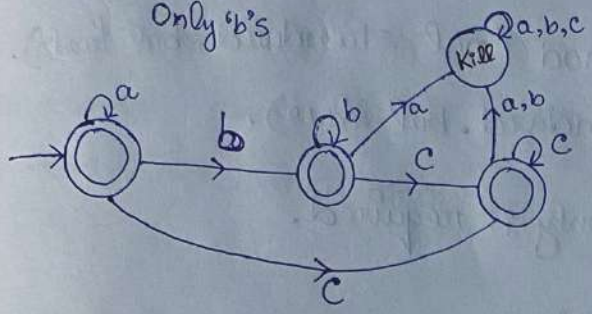
min DFA = 5 States
min NFA = 4 States

Q: $\Sigma = \{a, b, c\}$ $\{a^n b^m c^l / n, m, l \geq 0$

⇒ RE = $a^* b^* c^*$

L = { ϵ, a^*, b^*, c^* (Only 'c's), a^*b^* (no 'c'), b^*c^* (no 'a'), a^*c^* (no 'b'), $a^*b^*c^*$ }

(a^*: only 'a's; b^*: only 'b's)

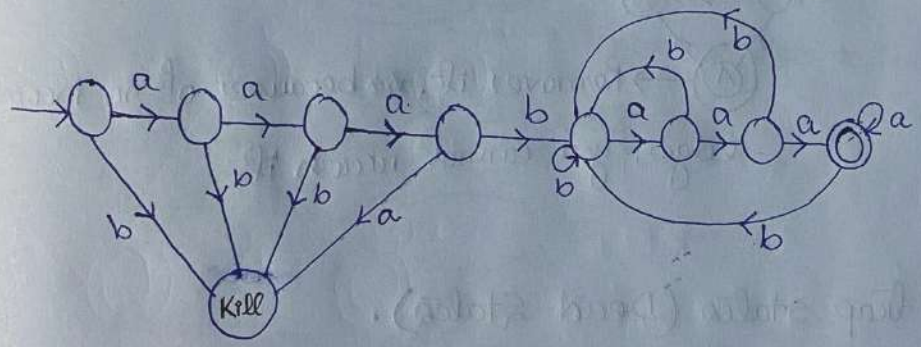


min DFA = 4 States
min NFA = 3 States

Q: $\Sigma = \{a, b\}$ $L = \{a^3 b\, w\, a^3 / w \in (a+b)^*\}$

⇒ RE = aaab $(a+b)^*$ aaa

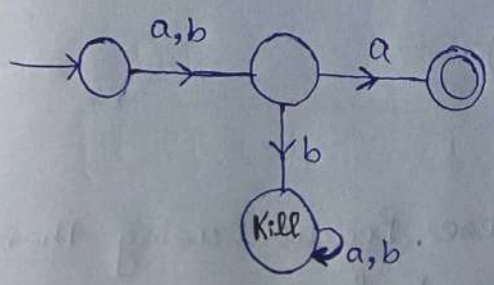
L = { aaabaaa, aaabaaaa, aaabbaaa, ... }



min DFA = 9 States
min NFA = 8 States

Q: $\Sigma = \{a, b\}$ $L = \{w /$ Second symbol from LHS is 'a' $\}$

⇒ RE = $(a+b)\,a\,(a+b)^*$



LECTURE-10:

Finite Automata to Regular Expression>

Methods>

1) Intuitions.
2) State Elimination method (not standard, but fast).
3) Arden's Theorem (Standard, but slow).

Use Arden's Theorem only if required.

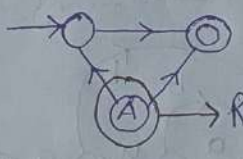
State Elimination Method:

It works for DFA, NFA, ϵ-NFA.

→ Simplify the finite automata

→ a) Remove any non-reachable state.

Example:

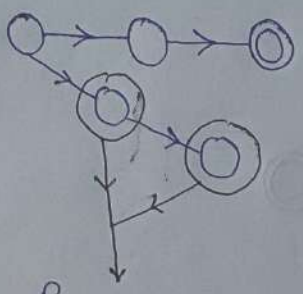


→ Remove it, because of no incoming edge, you cannot reach it.

b) Remove trap states (Dead States).

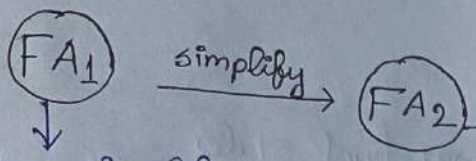
c) Remove any state from which final state is not reachable.

Example:



Remove these. Because using them you can't reach final state.

d) Merge the final states if possible.



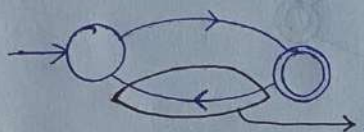
You simplify the given finite Automata and get a new Finite Automata (FA_2).

→ On this new finite Automata, there are certain steps to be followed.

Steps on the FA_2:

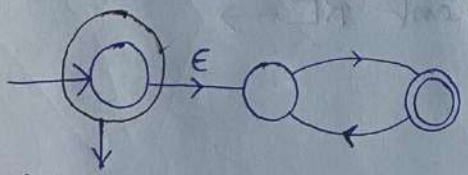
1) There should not be any incoming edge to initial state.

Example:



→ Incoming edge to initial state

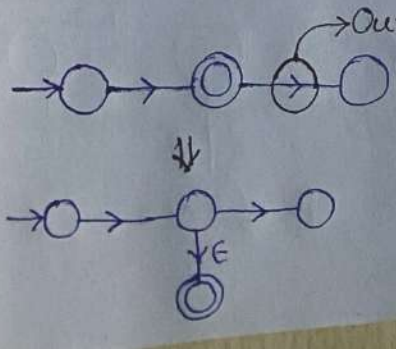
⇓ We can write like.



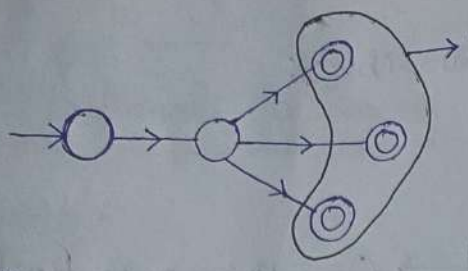
We'll simply add a new initial state which doesn't has any incoming edge, the rest will be unchanged.

Self-loop is also an incoming edge.

2) Final state should not have any outgoing edge.

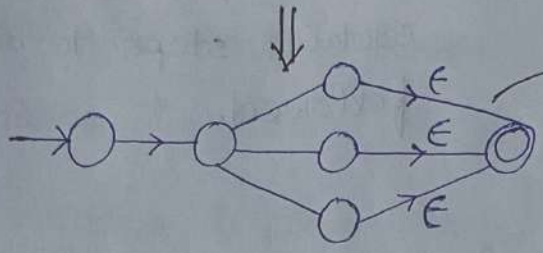


3)



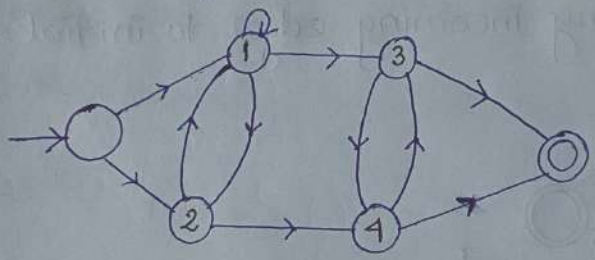
→ Try merging first.

There should be only one final state



→ Do this if merging is not possible.

4) Eliminate the states other than initial and final states in any order.



We can eliminate 1, 2, 3 & 4 in any order and for every order there'll be a different RE →

1 2 3 4 ⟶ RE_1

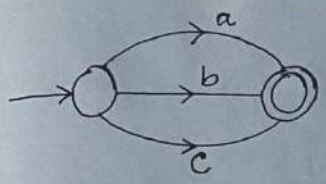
2 1 3 4 ⟶ RE_2

1 2 4 3 ⟶ RE_3

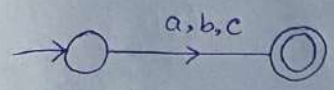
4 1 2 3 ⟶ RE_4

⋮

Q:

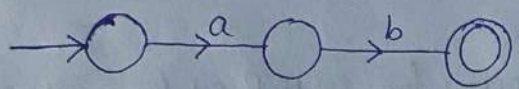


⇒

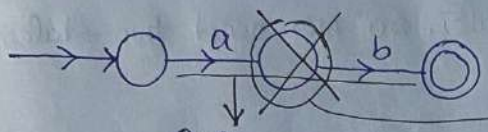


RE = (a+b+c)

Q:



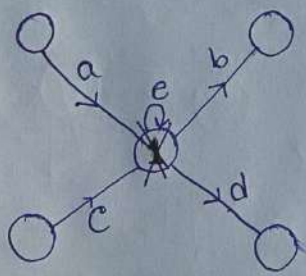
⇒



Only one path if we eliminate this.

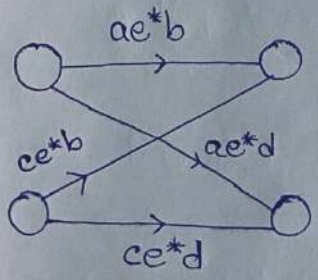
RE = (ab)

NOTE:



Here incoming path = 2 & Outgoing path = 2.
Total no. of Paths = 2 × 2
= 4

If we eliminate State 1 ⟶

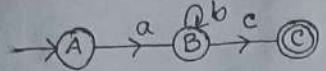


Take longest path only.

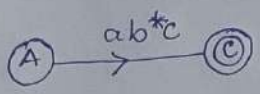
Q: If there are 3 incoming & 2 outgoing path then total no. of paths ?

⇒ 3 × 2 = 6

Q:

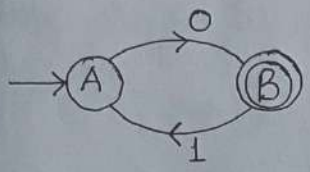


⇒ If we eliminate "B" →



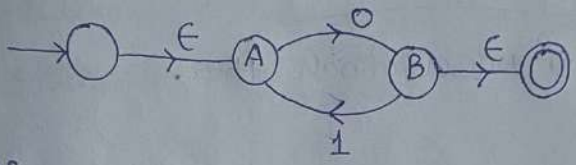
RE = ab*c

Q:

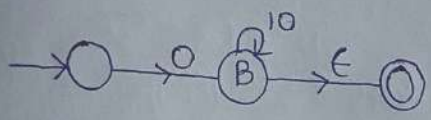


⇒

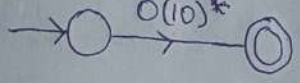
There's an incoming edge for initial state and one outgoing edge for final state. So, we need two states.



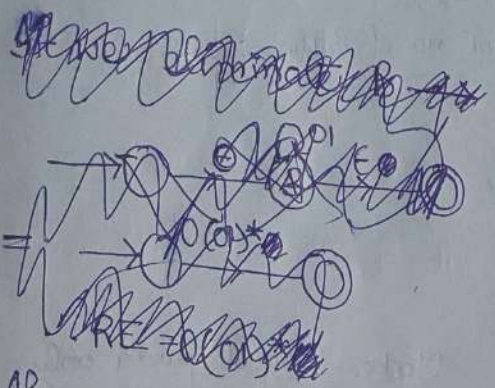
If we eliminate A →



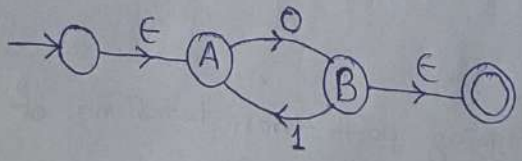
⇒

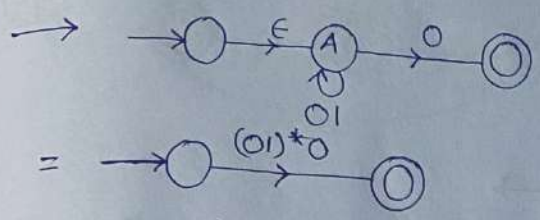


∴ RE = 0 (10)*



If we eliminate B →

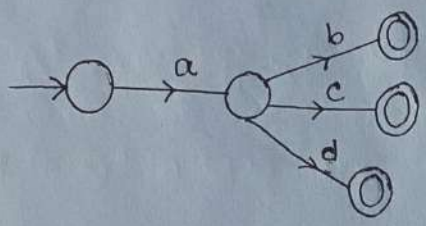




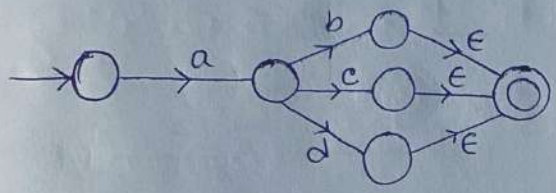
$RE = (01)^*0$

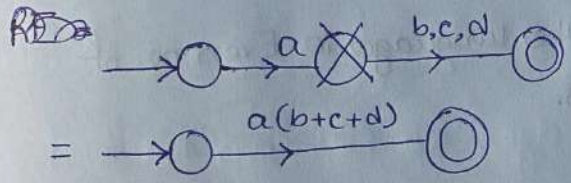
$$\boxed{(01)^*0 = 0(10)^*}$$

Q:



⇒



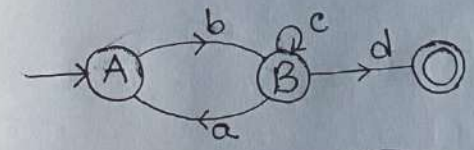


$\therefore RE = a(b+c+d)$

$= ab + ac + ad$

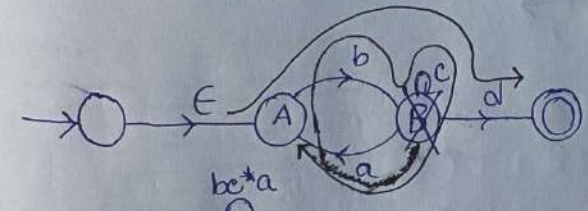
(abc*)*bc*d

Q:



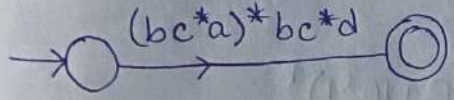
If we eliminate B →

⇒



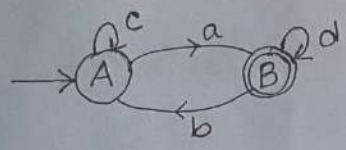
=

=

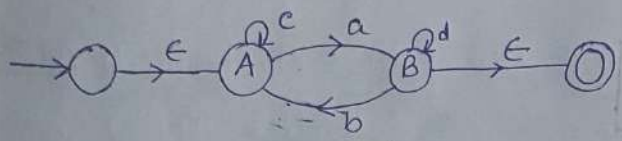


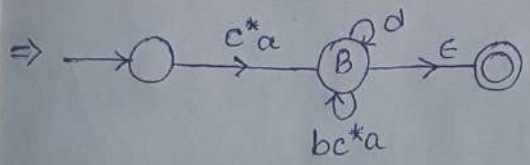
$RE = (bc^*a)^*bc^*d$

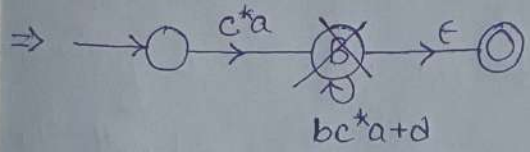
Q:

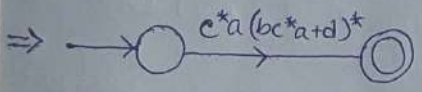


⇒ If we eliminate A then B ⟶



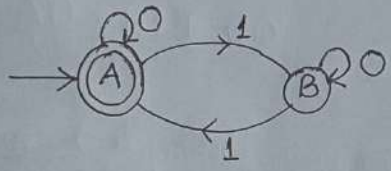




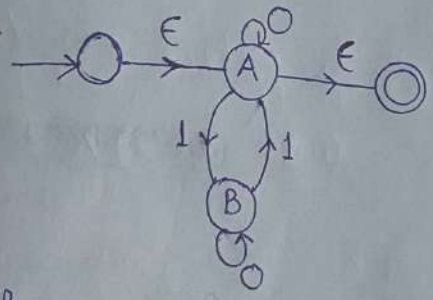


∴ RE = $c^*a\,(bc^*a+d)^*$

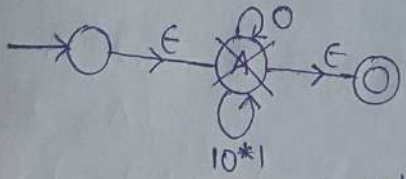
Q:

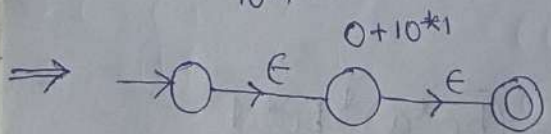


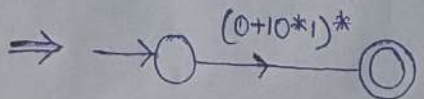
The language is Even no. of 1s.



If we eliminate B then A ⟶

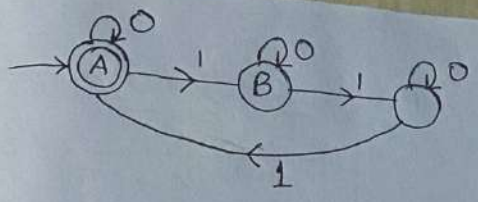




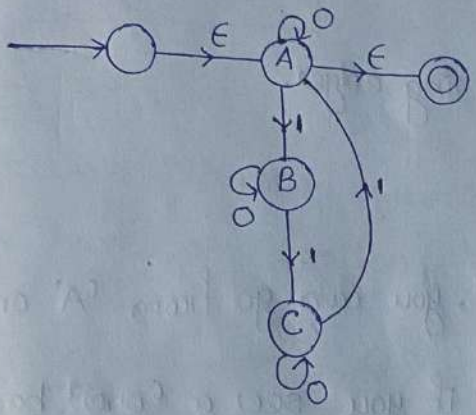


∴ RE = $(0+10^*1)^*$

Q:

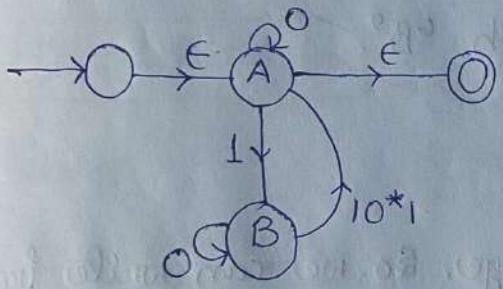


⇒

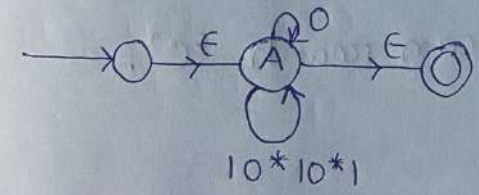


Time Stamp
↓
01:32:47

⇒ If we eliminate C →



⇒



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

Arden's Theorem:

If $\boxed{S = R + ST}$

Solution: $\boxed{S = RT^*}$

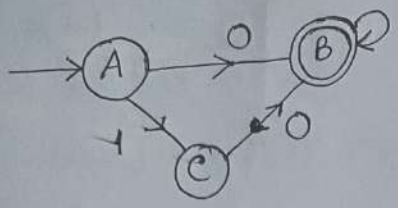
If $\boxed{S = R + TS}$

Solution: $\boxed{S = T^*R}$

Back Substitution if you forget the formula:

$$RT^* = R + RT^*T$$
$$= R(\epsilon + T^*T)$$
$$= RT^*$$

Q:



⇒ 'A' does not has any incoming edge.

So, we'll write →

$A = \epsilon$ ——— (i)

If you want to reach 'B', you can go from 'A' on seeing a '0' or on 'B' if you see a 'one', because of the loop you can reach 'B'. Then on 'C' by seeing a '0' you can reach 'B'.

$B = A0 + B1 + C0$ ——— (ii)

C has only one incoming edge. So, we can write from 'A' on seeing a 'one' we can reach 'C'.

$C = A1$ ——— (iii)

Here final state is B, so we have to solve 'B'.

→ Substitue A & C in 'B'.

$A = \epsilon$

$C = A1$

$\Rightarrow C = \epsilon 1$

$= 1$

$B = A0 + B1 + C0$

$= \epsilon 0 + B1 + (1)0$

$= 0 + B1 + 10$

$B = (0+10) + B1$

This is in the form of Arden's Theorem.

$$B = \underset{R}{(0+10)} + \underset{S}{B}\underset{T}{1}$$

$S = R + ST$

$S = RT^*$

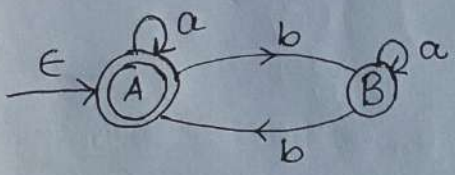
$\therefore$ $B = RT^*$

$= (0+10)1^*$ → Answer

NOTE: **

In Arden's theorem questions initial state will always have an epsilon (ϵ).

Q:



Here final state = A.
So, we'll solve A.

$\Rightarrow A = \epsilon + Aa + Bb$

$B = Ab + Ba$ → This is in form of Arden's theorem.

$$\underset{S}{B} = \underset{R}{Ab} + \underset{S}{B}\underset{T}{a}$$

$\therefore B = Aba^*$

$A = \epsilon + Aa + Aba^*b$

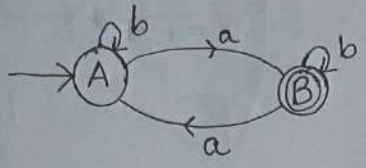
$$\underset{S}{A} = \underset{R}{\epsilon} + \underset{S}{A}\underset{T}{(a+ba^*b)}$$

$\therefore A = \epsilon(a+ba^*b)^*$

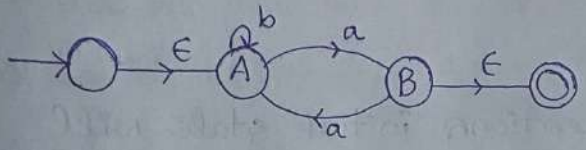
$= (a+ba^*b)^*$ ⟶ Answer

LECTURE - 11

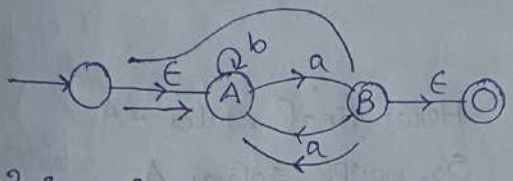
Q:



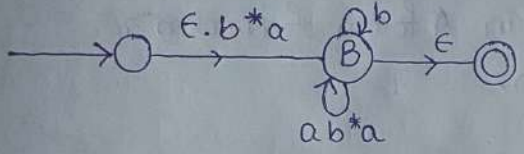
⇒

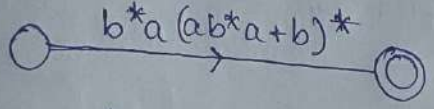


If we want to eliminate 'A', then we'll first see what are all the incoming edges.



2 incoming edges & 1 outgoing edge. So, total no. of paths = 2 × 1





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

Arden's Theorem Question:

Q: $A = 10 + 01A + B$
$B = 00 + CB$
$C = 11 + 0C$

$S = R + TS$
$S = T^*R$

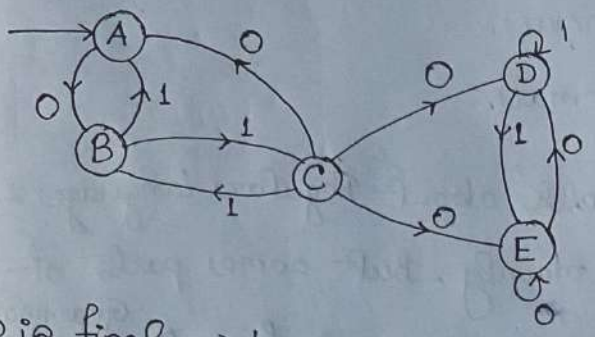
⇒ $C = 11 + 0C$
$S = \overline{R} + \overline{S}T\overline{\ }S$

$\therefore C = 0^*11$

$B = 00 + 0^*11B$
$B = (0^*11)^*00$

$A = 10 + 01A + B$
$= 10 + 01A + (0^*11)^*00$
$= (01)^*(10 + (0^*11)^*00)$

Q:



If D is final $\longrightarrow L_D$

If E is final $\longrightarrow L_E$

a) $L_D = L_E$

b) $L_D \subseteq L_E$

c) $L_E \subseteq L_D$

d) $L_D \neq L_E$

⇒ You have to use Arden's Theorem.

You have to check for the incoming edges on 'D' and 'E'.

Equation for D:

$D = C0 + D1 + E0$ ——— (I)

Equation for E:

$E = C0 + D1 + E0$ ——— (II)

$D = C0 + D1 + E0$ is same as $E = C0 + D1 + E0$.

Therefore 'option a' is the answer $(L_D = L_E)$

Regular Grammars: (RGT)

1) Right linear Grammar.

2) Left linear Grammar.

In this note we'll talk about Regular language & Regular Grammars mainly. But some parts of Context free Language & Context free Grammars will also be there.

Grammars:

(V, T, P, S)

- P → Productions ⇒ Finite & non-empty.
- S → Start Symbol ⇒ Only one symbol.
- V → Vertices ⇒ Finite and non-empty.
- T → Terminals (Σ → Sigma) ⇒ Finite and non-empty.

$\in$ is a terminal

Explanation:

$A \rightarrow aA/a$ → This actually means two productions.

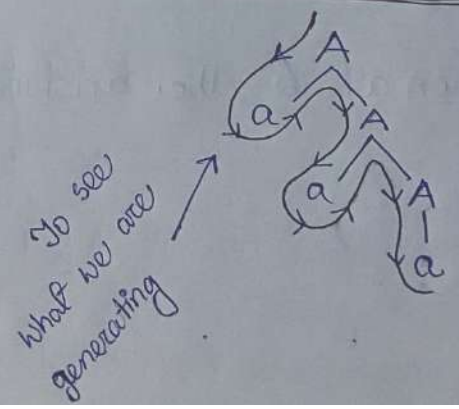
$A \rightarrow aA$
$A \rightarrow a$
→ Productions.

If we want to derive → aaa

Derivation:

$A \Rightarrow aA$
$\Rightarrow aaA$
$\Rightarrow aaa$

Derivation Tree



Here →

Vertices = {A}

Terminals = {a}

Productions = {A → aA, A → a}

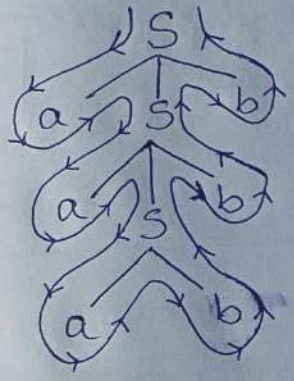
Start State/= A

Symbol

Example:

S → aSb/ab. We want to derive aaabbb

⇒



⇒ aaabbb

Here →

V = {S}

T = {a, b}

S = S

↑ Start State

Grammars will generate all the strings of a language.

THEORY POINTS:

Regular grammars generate Regular languages.

Some non-regular grammars also generate regular languages. For every Regular language there will be atleast one Regular Grammar.

Regular grammars are also known as

Type 3 Grammar.

Type 0, Type 1, Type 2 → We'll see later

Regular Grammars are of two types →

1) Right linear (RLG)

2) Left linear (LLG)

If a language is regular, there'll be atleast one RLG & one LLG.

Right linear:

$A \rightarrow \alpha B / \beta$ → The symbol should be present at right most.

Where $A, B \in V$

$\in$ → Belongs to

$\alpha, \beta \in T^*$

T → Terminals

Left linear:

$A \rightarrow B \alpha / \beta$

$A, B \in V$

$\alpha, \beta \in T^*$

For all the examples, assume small letters as Terminals & Capital letters as Vertices.

$A \rightarrow aA/a$ = RL

$A \rightarrow Aa/a$ = LL

$A \rightarrow aAA/a$ = None, because there are two symbols

$S \rightarrow aSa/a$ = None

$S \rightarrow aab/abS$ = Right linear

$S \rightarrow Sab/aa$ = Left linear

NOTE:

Regular Grammars are actually subset of Context free Grammars.

$S \rightarrow SS/a$ = None

THEORY POINT:

Every regular grammar is Context free Grammar.

Every Regular language is Context free language.

$A \rightarrow ab$ = Both RL & LL because there are no Vertices.

$A \in V$

$a, b \in T$

If a Grammar is Right or Left linear or both, then it's Regular Grammar.

Time Stamp ↓ (56:25)

The main Building Blocks which we have to learn how to write:

ϕ

$\in$

a

$a+b$

a^*

$(ab)^*$

$a^* + b^*$

$a^* b^*$

ϕ:

$S \longrightarrow A$ (no terminals)

So, S cannot generate any string.

In all examples, assume that S is the Start Symbol.

$S \longrightarrow \phi X$ → You can't write this.
'ϕ' is not there in the language.

Production → { } X → You can't write this. Atleast one production should be there.

ϵ:

CFG = Context Free Grammar

Language = $\{\epsilon\}$

$\therefore$ $S \longrightarrow \epsilon$

L = {a}:

$S \longrightarrow a$

L = {a, b}:

$S \longrightarrow a$
$S \longrightarrow b$
$\Rightarrow$ $S \longrightarrow a/b$

L = {a*}:

a* means → $\{\epsilon, a, aa, \ldots\}$

$\therefore$ $S \longrightarrow aS/\epsilon$

Right Linear

$S \longrightarrow Sa/\epsilon$

Left Linear

$L = \{a^+\}$:

$a^+ = \{a, aa, aaa, \ldots\}$

$\therefore$ $S \rightarrow aS/a$ → RLG

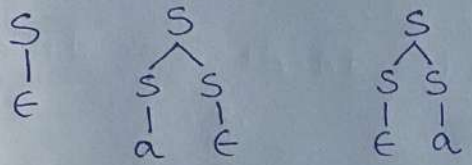
$S \rightarrow Sa/a$ → LLG.

$S \rightarrow SS/a$ → CFG

$L = \{a^*\}$:

We can also write it as $\rightarrow$

$S \rightarrow SS/a/\epsilon$ → This is not RG, this is CFG.



NOTE:

A grammar is regular only if it's either Right or Left Linear.

x^*: This 'x' can be anything

$S \rightarrow xS/\epsilon$ → RLG

$S \rightarrow Sx/\epsilon$ → LLG

If you want to generate $(1011)^* \rightarrow$

$S \rightarrow 1011S/\epsilon$ or $S \rightarrow S1011/\epsilon$

$(ab)^* \rightarrow$

$S \rightarrow abS/\epsilon$ or $S \rightarrow Sab/\epsilon$

x^+: This 'x' can be anything

$S \rightarrow xS/x$ → RLG

$S \rightarrow Sx/x$ → LLG

$(1101)^+ \rightarrow$

$S \rightarrow 1101S/1101$ or $S \rightarrow S1101/1101$

x^*y: 'x' & 'y' can be anything ?

First we'll generate 'x^*'.

$S \rightarrow xS$

This'll continue generating 'x'. We'll add a 'y' to end it.

$\therefore$ $\boxed{S \rightarrow xS/y}$

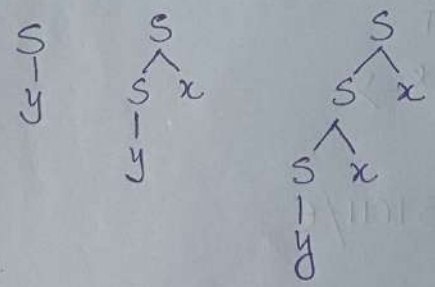
LECTURE-12:

THEORY NOTE>

For the same string there can be many derivations or derivation trees. Such grammars are called **ambiguous**.

Q: What is the language for $S \rightarrow Sx/y$.

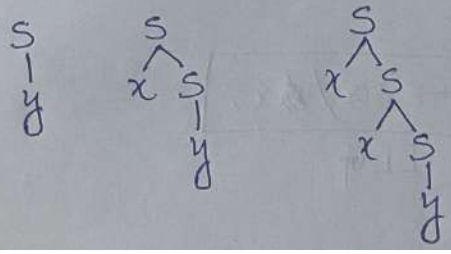
$\Rightarrow$



$\therefore$ Language = yx^*

Q: What is the language for $S \rightarrow xS/y$.

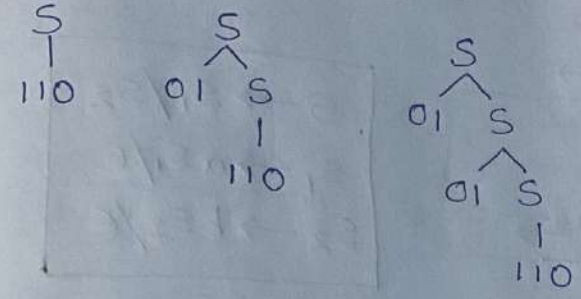
$\Rightarrow$



$\therefore$ Language = x^*y.

Q: What is the language for $S \rightarrow 01S/110$

⇒

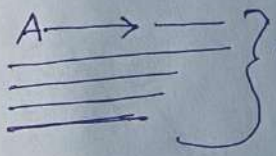


∴ Language = $(01)^*110$

Question might be like ⇒ **

$L_1 + L_2 \rightarrow$ Regular language

L_1 → Regular language



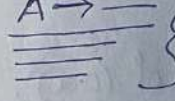
This whole part derive L_1

This whole part derive L_2.

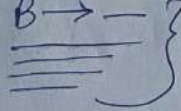
So, we can write ⇒ $\boxed{S \rightarrow A/B}$

$L_1 \cdot L_2 \rightarrow$ Regular language

L_1 → Regular language



Derive L_1



Derive L_2

So, we can write ⇒ $\boxed{S \rightarrow AB}$ → But this is not Regular.

a^*+b^* :

First we'll derive 'a^*' →

$S_1 \rightarrow aS_1/\epsilon$

Next we'll derive 'b^*' →

$S_2 \rightarrow bS_2/\epsilon$

Now, we can write →

$S \rightarrow S_1/S_2$

$S \rightarrow S_1/S_2$
$S_1 \rightarrow aS_1/\epsilon$
$S_2 \rightarrow bS_2/\epsilon$

GATE:

$S \rightarrow A/B$

$A \rightarrow aaA/\epsilon$

$B \rightarrow Bb/\epsilon$

What is the language?

$\Rightarrow$ $A \rightarrow aaA/\epsilon$

This is nothing but $(aa)^*$

$B \rightarrow Bb/\epsilon$

This is simply b^*

$\therefore L = (aa)^* b^*$

Q: $S \rightarrow A/B$

$A \rightarrow aaA/\epsilon$

$B \rightarrow Baaaa/\epsilon$

What is the language?

$\Rightarrow$ $A \rightarrow aaA/\epsilon$

This is $(aa)^*$

$B \rightarrow Baaaa/\epsilon$

This is $(aaaa)^*$

$(aaaa)^*$ is subset of $(aa)^*$.

$\therefore L = (aa)^*$

a^*b^*:

First generate a^*>

$A \rightarrow aA/\epsilon$

Then generate b^*>

$B \rightarrow bB/\epsilon$

$\therefore$ We can write>

$S \rightarrow AB$

$S \rightarrow AB$
$A \rightarrow aA/\epsilon$
$B \rightarrow bB/\epsilon$

→ But this is not Regular Grammar

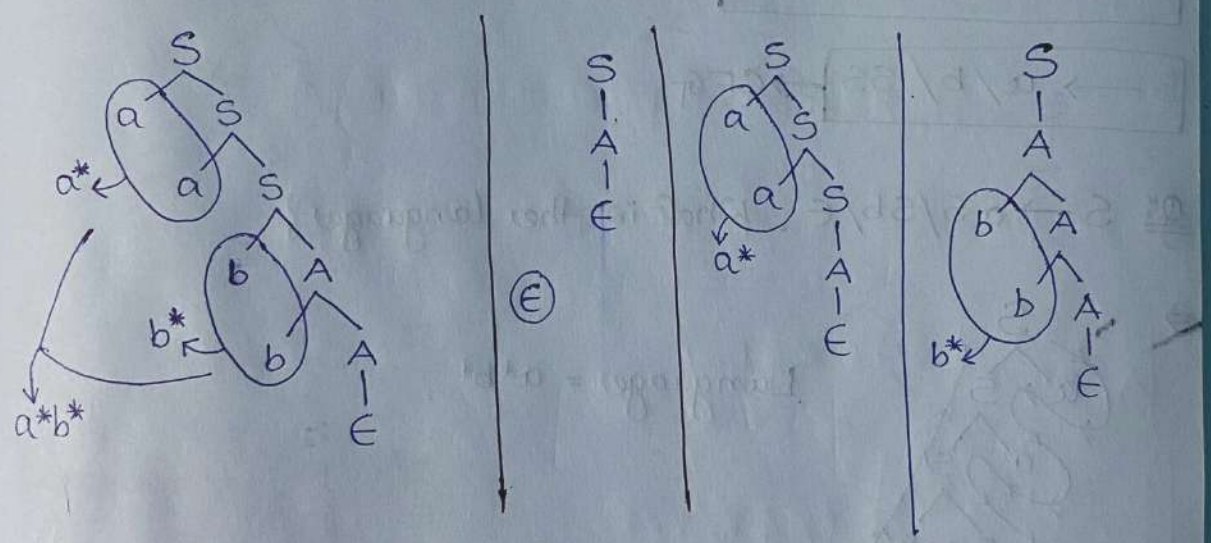
↓

This is Context free Grammar

Let's do the Regular Grammar for a^*b^*:

$S \rightarrow aS/A$
$A \rightarrow bA/\epsilon$

***** a^*b^* contains a^*, b^* also.



$(a+b)^*$:

CFG>

$S \rightarrow a/b/SS/\epsilon$

Regular Grammar>

$S \rightarrow aS/bS/\epsilon$

$(x+y)^*$: x & y can be anything >

$S \rightarrow xS/yS/\epsilon$ → RLG

$S \rightarrow Sx/Sy/\epsilon$ → LLG

$S \rightarrow x/y/SS/\epsilon$ → CFG

$(a+b)^+$: a & b can be anything >

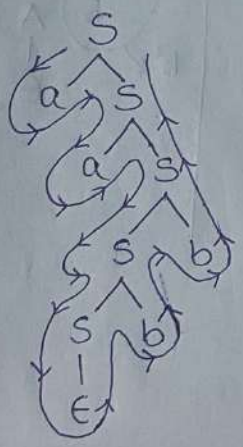
$S \rightarrow aS/bS/a/b$ → RLG

$S \rightarrow Sa/Sb/a/b$ → LLG

$S \rightarrow a/b/SS$ → CFG

Q: $S \rightarrow aS/Sb/\epsilon$. What is the language?

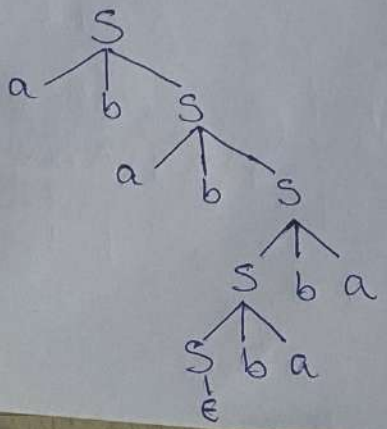
⇒



Language = a^*b^*

Q: $S \rightarrow abS/Sba/\epsilon$. Language = ?

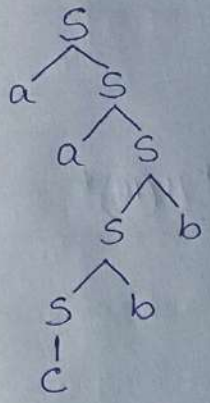
⇒



Language = $(ab)^*(ba)^*$

Q: $S \rightarrow aS/Sb/c$. Language = ? (GATE)

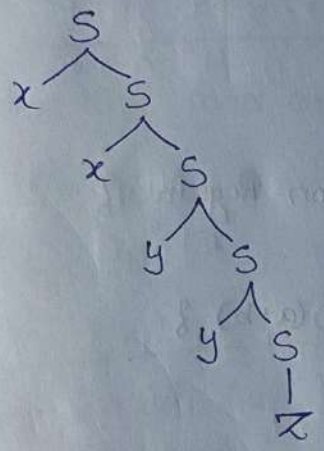
⇒

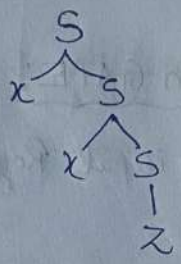


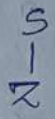
Language = a^*cb^*

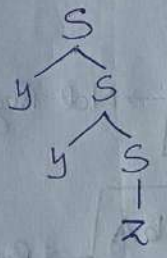
Q: $S \rightarrow xS/yS/z$. Language = ?

⇒





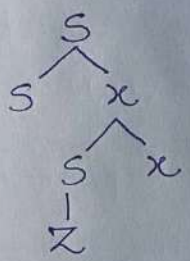


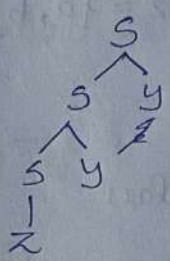


Language = $(x+y)^*z$

Q: $S \rightarrow Sx/Sy/z$. Language = ?

⇒



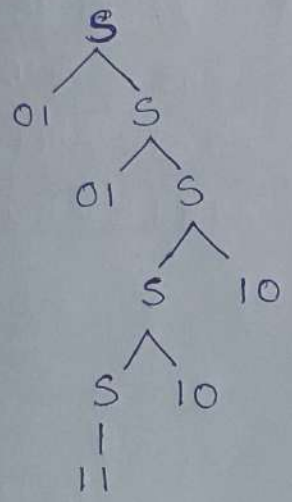


Language = $z(x+y)^*$

Q: GATE>

$S \rightarrow 01S/S10/11$. Language = ?

⇒



Language = $(01)^*11(10)^*$

Popular Question in GATE:

Given a language is it Regular or non-Regular ?

Q: $L=\{a,b\}$, $S \rightarrow a/b$ then $L=(a+b)(a+b)$?

⇒

$S \rightarrow AA$
$A \rightarrow a/b$
→ Not Regular

$S \rightarrow aA/bA$
$A \rightarrow a/b$
→ Regular

Q: String length atleast '2'. $\Sigma=\{a,b\}$.

⇒ RE = $(a+b)(a+b)(a+b)^*$

$S \rightarrow BBA$ → Non Regular
$B \rightarrow a/b$ → $(a+b)$
$A \rightarrow aA/bA/\epsilon$
↓
$(a+b)^*$

Q: String length is almost '2'. $\Sigma = \{a, b\}$.

⇒ RE = $(a+b+\epsilon)(a+b+\epsilon)$

$S \rightarrow AA$
$A \rightarrow a/b/\epsilon$ → Not Regular

Q: Starting with 'a' & ending with 'b'. $\Sigma = \{a, b\}$.

⇒ RE = $a(a+b)^*b$

$S \rightarrow aAb$
$A \rightarrow aA/bA/\epsilon$ → Not Regular

Q: Starting and ending with different symbols.

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

⇒ RE = $a(a+b)^*b + b(a+b)^*a$

$S \rightarrow aAb/bAa$
$A \rightarrow aA/bA/\epsilon$ → Not Regular

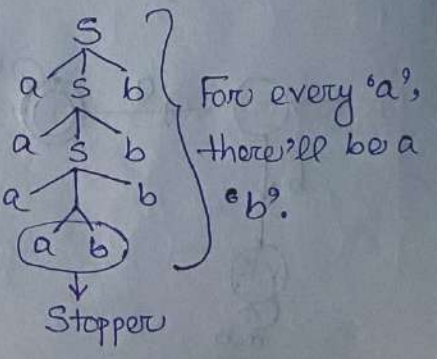
Q: $a^n b^n / n \geq 1$

⇒ Language is not Regular → Because Finite Automata cannot compare.

∴ Regular Grammar is not possible.

We can write →

$S \rightarrow aSb/ab$ → Stopper



Q: $\Sigma = \{a, b\}$, $L = WW^R \cup WaW^R \cup WbW^R$. $W \in \Sigma^*$.
'R' means Reversal. **Palindromes Question**

⇒ If $W = ab$ $\therefore W^R = ba$

$WW^R = abba$

$WaW^R = ababa$

$WbW^R = abbba$

WW^R → even length Palindromes.

WaW^R → odd length Palindrome with 'a' in middle.

WbW^R → odd length Palindromes with 'b' in middle.

All these put together means → All possible Palindromes.

Here also matching is required which Finite Automata can't do. Regular Grammar is not possible.

We can write →

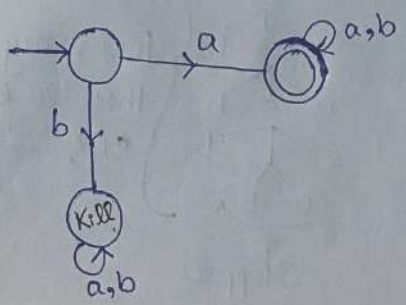
$S \rightarrow aSa / bSb / a / b / \epsilon$

LECTURE-7

Q: $\Sigma = \{a, b\}$ $L = \{W / W$ starts with 'a'$\}$.

⇒ $L = \{a, aa, ab, aba, aab, abb, \ldots\}$

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



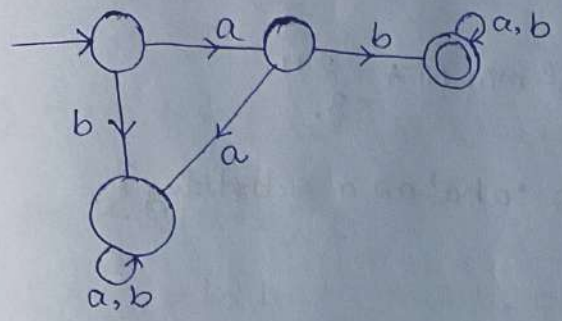
min DFA = 3 States

min NFA = 2 States

Q: $\Sigma = \{a, b\}$. $L = \{W/W$ starts with $ab\}$.

⇒ $RE = ab(a+b)^*$

$L = \{ab, aba, abba, abbb, abaa, \ldots.\}$



min DFA = 4 states
min NFA = 3 states

Q: Starting with aaaba, how many states in min DFA & min NFA?

⇒ aaaba → String length = 5.

min DFA = (5+1)+1
= 7 states

min NFA = (5+1)
= 6 States

Q: $\Sigma = \{a, b\}$, $L = \{W/W$ contains 'a'$\}$ [There can be more than one 'a'].

⇒ $RE = (a+b)^*a\,(a+b)^*$

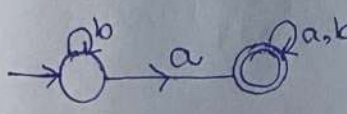
OR

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

OR

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

$L = \{a, ba, aa, ab, bab, \ldots.\}$

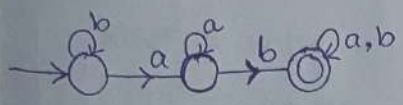


Here min DFA states = min NFA states
= 2

Q: $\Sigma = \{a, b\}$, $L = \{W / W$ contains 'ab' as a substring$\}$.

⇒ RE = $(a+b)^* ab (a+b)^*$

$L = \{ab, aab, abb, aba, \ldots\}$

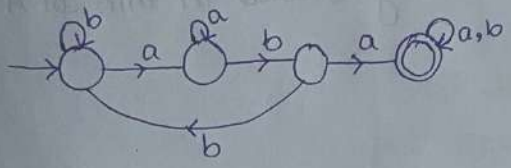


Here states of min DFA = states of min NFA = 2+1 = 3.

Q: $\Sigma = \{a, b\}$, $L = \{W / W$ contains 'aba' as a substring$\}$.

⇒ RE = $(a+b)^* aba (a+b)^*$

$L = \{aba, aaba, aba, \ldots\}$



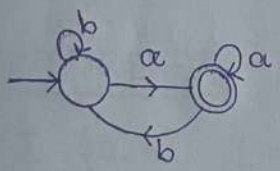
min DFA = (3+1) = 4 states

min NFA = (3+1) = 4 states

Q: $\Sigma = \{a, b\}$, $L = \{W / W$ ends with 'a'$\}$.

⇒ RE = $(a+b)^* a$

$L = \{a, aa, aba, ba, \ldots\}$

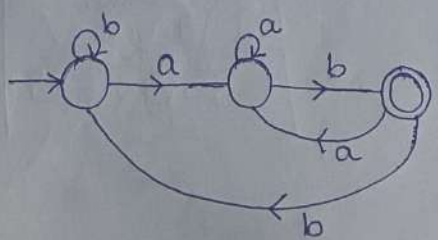


min DFA = 2 states
min NFA = 2 states

Q: $\Sigma = \{a, b\}$, $L = \{W / W$ ending with 'ab'$\}$

⇒ RE = $(a+b)^* ab$.

$L = \{ab, aab, bab, aaab, \ldots\}$



min DFA = 3 states
min NFA = 3 states

Q: ababbbab → How many states in min DFA & min NFA?

⇒ string length = 8

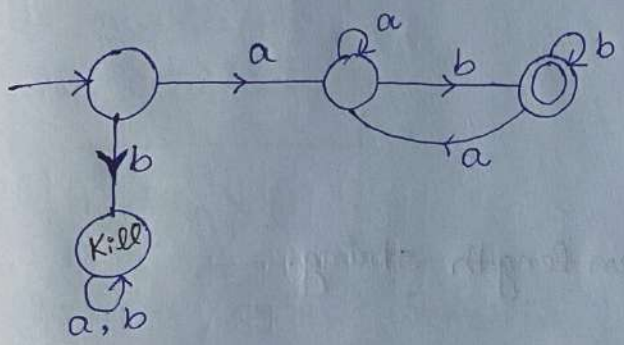
∴ min DFA = 8 + 1
= 9

min NFA = 8 + 1
= 9

Q: $\Sigma = \{a, b\}$, $L = \{w/w$ starts with 'a' & ends with 'b'$\}$.

⇒ RE = $a(a+b)^*b$

$L = \{ab, aab, aabb, abab, abbab, \ldots\}$



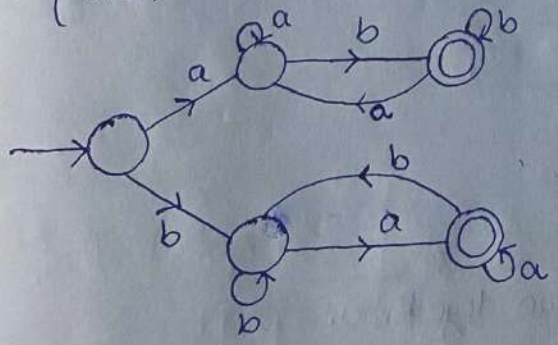
min DFA = (2+1)+1
= 4 states

min NFA = 3 states.

Q: $\Sigma = \{a, b\}$, $L = \{w/w$ starts & ends with different symbol$\}$.

⇒ RE = $a(a+b)^*b + b(a+b)^*a$
= $(a+b)^*(ab+ba)$

$L = \{ab, ba, aab, abb, baa, bba, \ldots\}$



min DFA = 5 states
min NFA = 5 states

LECTURE 7 ENDS HERE

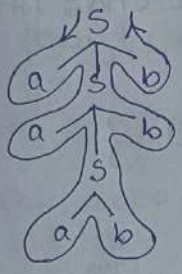
LECTURE-13:

Q: $a^n b^n / n \geq 1$. $[a^n b^n \neq a^* b^*]$

⇒ $\boxed{a^* b^* = a^n b^m}$

$a^n b^n$:

$S \longrightarrow aSb/ab.$



aaabbb → this is called yield of the tree.

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

⇒ This language is even length string.

$S \rightarrow BS/\epsilon$
$B \rightarrow AA$
$A \rightarrow a/b$
→ Not Regular

Q: $a^n b^m / n, m \geq 1$.

⇒

$S \rightarrow AB$
$B \rightarrow bB/b$
$A \rightarrow aA/a$
→ Not Regular

Q: $a^n b^n c^m / n, m \geq 1$.

⇒ $(a^n b^n) c^+$

We have to generate these two together.

$S \rightarrow AB$
$A \rightarrow aAb/ab$
$B \rightarrow cB/c$

Q: $a^n c^m b^n / n, m \geq 1$.

⇒ $a^n c^m b^n$

We have to generate these two together.

$S \rightarrow aSb / aAb$

$A \rightarrow cA / c$

Q: $a^n b^n c^m d^m / n, m \geq 1$.

⇒

$S \rightarrow AB$

$A \rightarrow aAb / ab$

$B \rightarrow cBd / cd$

Q: $a^n b^n c^n / n \geq 1$.

⇒ This is a Context Sensitive Language. → We'll see this later.

Q: $a^n b^{2n} / n \geq 1$.

⇒ $S \rightarrow aSbb / abb$.

$a^n b^{4n+1} / n \geq 1$

Q: $a^n b^m c^m d^n / n, m \geq 1$.

⇒

$S \rightarrow aSd / aAd$

$A \rightarrow bAc / bc$

~~$A \rightarrow aAd / ad$~~

~~$B \rightarrow bBd / bd$~~

Q: $a^n b^m c^n d^m / n, m \geq 1$.

⇒ This is Context Sensitive Language → We'll see this later.

Q: $a^{m+n} b^m c^n / m, n \geq 1$.

⇒ $a^n a^m b^m c^n$

$S \rightarrow aSc / aAc$
$A \rightarrow aAb / ab$

Q: $a^n b^{n+m} c^m / n, m \geq 1$.

⇒ $a^n b^n b^m c^m$

$S \rightarrow AB$
$A \rightarrow aAb / ab$
$B \rightarrow bBc / bc$

Q: $a^n b^m c^{n+m} / n, m \geq 1$.

⇒ $a^n b^m c^m c^n$

$S \rightarrow aSc / aAc$
$A \rightarrow bAc / bc$

Easy Ways:

Language $\xrightarrow{\text{Convert}}$ FA $\rightarrow$ RG

OR

RG $\xrightarrow{\text{Convert}}$ FA $\xrightarrow{\text{find}}$ Language → Important for GATE.

Q: GATE-

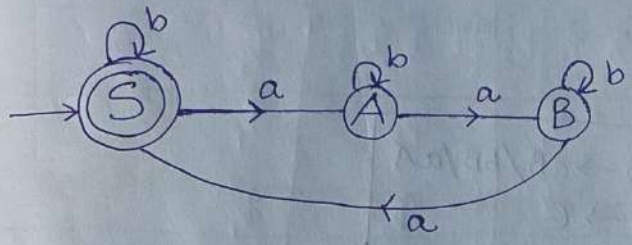
What is language generated by following grammar?

$S \rightarrow aA/bS/\epsilon$
$A \rightarrow aB/bA$
$B \rightarrow aS/bB$

Try doing the diagram as you did back in State Transition Table.

⇒ NOTE:

Always remember → 'ϵ' means final state.

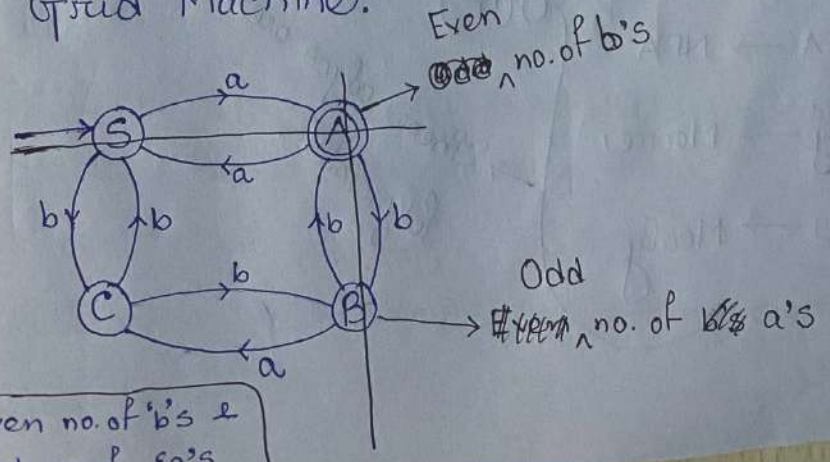


This language is → Number of a's divisible by 3.

Q: $S \rightarrow aA/bC$
$A \rightarrow aS/bB/\epsilon$
$B \rightarrow aC/bA$
$C \rightarrow aB/bS$

⇒ NOTE:

When there are more than 3-states, most probably it's a Grid Machine.



Language = Even no. of 'b's & odd no. of 'a's

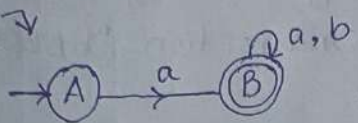
Reverse Final Automata:

Steps:

Make initial → final.
Reverse the transition.

NOTE:
While reversing, you can't reverse a self loop.

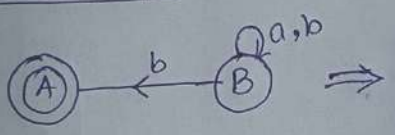
L:



RLG:

$A \rightarrow aB$
$B \rightarrow aB / bB / \in$
↓
Final state

FA → LLG:



$B \rightarrow aB / bB / aA$
$A \rightarrow \in$

$RLG(L^R)$

↓ Reverse

$B \rightarrow Ba / Bb / Aa$
$A \rightarrow \in$

$LLG(L^R)^R = L$

Algorithms in Regular Languages:

1) NFA → DFA
2) DFA → min DFA
3) $\in$-NFA → NFA
4) Mealy → Moore
5) Moore → Mealy

$O(n^k)$ → these algos are polynomial.

Important ~~ISE~~ Theory NOTE for GATE:

If for any problem, there is an algorithm, then that problem is **decideable**.

If algo is polynomial → Tractable & decidable.

If algo is ~~NOT~~ non-polynomial → Intractable & decidable.

DFA to NFA is Trivially decidable.

NFA to ϵ-NFA is Trivially decidable.

LECTURE-14

NFA to DFA Conversion using algorithm:

NOTE: Any blanks in NFA means Dead State.

NFA:

	a	b
$\to q_0$	$q_0\, q_2$	—
$*q_1$	q_2	q_1
q_2	q_0	q_1

DFA: ⇒

Write down Row 1 first

	a	b
$\to q_0$	$\{q_0, q_2\}$	D
D	D	D
$\{q_0, q_2\}$	$\{q_0, q_2\}$	q_1
$*q_1$	q_2	q_1
q_2	q_0	q_1

~~DFA~~ DFA = 5 states

$\{q_0, q_2\}$ is a new state, so we add that in the table.

To calculate 'a' on $\{q_0, q_2\}$, we will take the data of q_0 & q_1 from NFA table & we'll use union on them.

Same goes for 'b' on $\{q_0, q_2\}$.

For final state → for this example, any string containing 'q_1' is a final state.

NOTE: Any state containing final state of NFA is final state of DFA.

Q: NFA:

	a	b
$\to q_0$	q_1, q_2	—
$*q_1$	q_2	q_1
q_2	q_0	q_1

⇓ DFA:

	a	b
$\to q_0$	$\{q_1, q_2\}$	D
D	D	D
$*\{q_1, q_2\}$	$\{q_0, q_2\}$	q_1
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	q_1
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	q_1
$*q_1$	q_2	q_1
q_2	q_0	q_1

Rough work

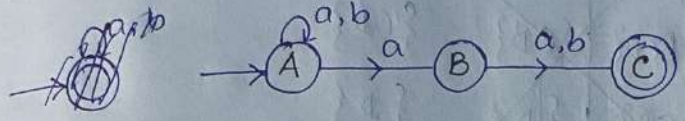
	a	b
$\to q_0$	$\{q_1, q_2\}$	D
D	D	D
$\{q_1, q_2\}$	$\{q_0, q_2\}$	q_1
$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	q_1
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	q_1
q_1	q_2	q_1
q_2	q_0	q_1

Q: Construct a DFA over $\Sigma = \{a, b\}$. Such that 2nd symbol from RHS is 'a'.

$\Rightarrow$ RE = $(a+b)^* a (a+b)$

Constructing the DFA is difficult, so we will construct the NFA first.

NFA:



	a	b
$\rightarrow$ A	AB	A
B	C	C
* C	–	–

DFA $\Rightarrow$

DFA:

	a	b
$\rightarrow$ A	{AB}	A
{AB}	{ABC}	{AC}
* {ABC}	{ABC}	{AC}
* {AC}	{AB}	{A}
~~ϕ~~	~~AB~~	~~ϕ~~

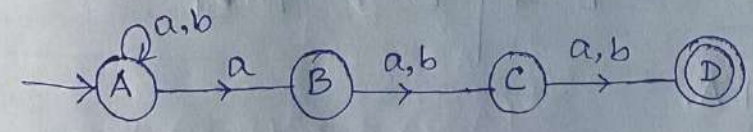
4 States in DFA.

3 States in NFA.

Q: Construct a DFA over $\Sigma = \{a, b\}$. Such that 3rd symbol from RHS is 'a'.

$\Rightarrow$ RE: $(a+b)^* a (a+b) (a+b)$

NFA:



	a	b
A	AB	A
B	C	C
C	D	D
D	–	–

	a	b
→A	A,B	B
B	C	C
C	D	D
* D	—	—

⇒

	a	b
→A	{A,B} ✓	{B}
{AB}	{ABC} ✓	{BC} ✓
{ABC}	{ABCD} ✓	{BCD} ✓
{ABCD}	{ABCD}	{BCD}
{BC}	{CD}	{CD}
{BCD}	{CD}	{CD}
{CD}	{D}	{D}

!

	a	b
→A	AB	A
B	C	C
C	D	D
* D	—	—

	a	b
→A	[AB]	[A]
⇒ [AB]	[ABC] ✓	[AC] ✓
[ABC]	[ABCD] ✓	[ACD]
* [ABCD]	[ABCD]	[ACD]
[AC]	[ABCD]	
	[ABD]	[AD]
* [ABD]	[ABC]	[AC]
* [ACD]	[ABD]	[AD]
* [AD]	[AB]	[A]

8 States in DFA.
4 States in NFA.

Short Notes:

2^{nd} Symbol → min DFA — 4 States.

3^{rd} Symbol → min DFA — 8 States.

4^{th} Symbol → min DFA — 16 States.

2^{nd} Symbol → NFA — 3 States. (2+1)

3^{rd} Symbol → NFA — 4 States. (3+1)

4^{th} Symbol → NFA — 5 States. (4+1)

We'll use the above, when the questions are like →

Construct a DFA, such that → 2^{nd} Symbol from RHS is 'a'.

→ 3^{rd} Symbol from RHS is 'a'.

→ 4^{th} Symbol from RHS is 'a'.

So, NFA is having (n+1) States.

DFA is having 2^n states.

If NFA has 'n' States, then DFA can have 2^n States in worst case.

Explanation & Example of the above NOTE:

Assume an NFA has 3 states = {A, B, C}. Then power set = $|2^Q| = 8$ States.

$\therefore$ Q of DFA $\subseteq 2^Q$.

The max we can go for DFA is 2^Q.

The Algorithm we've used for the last two questions is **Subset Construction**.

If $NFA = (Q, \Sigma, \delta, q_0, F)$ $\xrightarrow{\text{Converted}}$ DFA $= (Q', \Sigma', \delta', q_0', F')$

IMPORTANT:

$Q' \subseteq 2^Q$ $\quad$ $\Sigma' = \Sigma$ $\quad$ $q_0' = q_0$

There is no relation between F & F'.

$F \subseteq Q$ $\quad$ $F' \subseteq Q' \subseteq 2^Q$
$\Rightarrow F' \subseteq 2^Q$

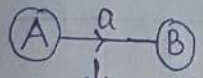
There is no relation between δ & δ'.

DFA to min DFA:

If a DFA has 'n' states, then min DFA $\leq n$.

Two states are equivalent if they both either go to a final state or a non final state on all strings. Both states should behave same.

$\delta(A, a) = B$



Single Transition

$\delta^*(A, w) = B$

String

Many Transition

A, B are equivalent if $\forall w \in \Sigma^*$

All strings possible.

$\delta^*(A, w)$ & $\delta^*(B, w)$ both go to either final or non final state.

We can't use this method.

So, we use K-equivalent.

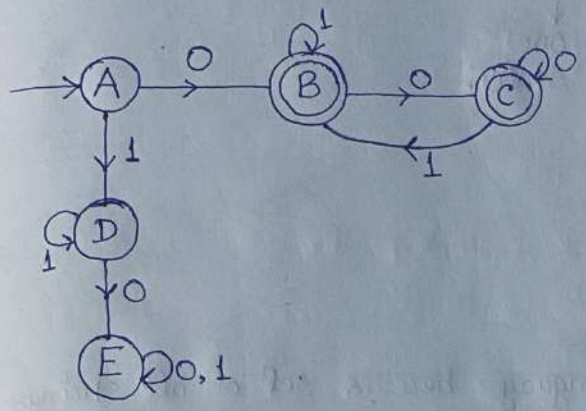
K-Equivalent:

A, B are K-equivalent, if $\forall w \in \Sigma^*$ (Only strings upto K length.) $|w| \leq K$.

For this, $\delta^*(A,w)$ & $\delta^*(B,w)$ both go to either final or non final state.

Explanation & example for K-equivalent >

01:32:47



First we'll do 0-equivalence → π_0 → Put final states in one partition & non-final states in other partition.

$$\pi_0 = \{\overbrace{\{B,C\}}^{1^{st}\text{ Partition}}\ \overbrace{\{A,D,E\}}^{2^{nd}\text{ Partition}}\}$$

$B \xrightarrow{0} C$, $B \xrightarrow{1} B$
$C \xrightarrow{0} C$, $C \xrightarrow{1} B$

From π_0 we'll find out π_1.

$\therefore \pi_1 = \{\{B,C\}\ \{A\}\ \{D,E\}\}$

2) If both states are going to same partition then they are equivalent.

$A \xrightarrow{0} B$
$D \xrightarrow{0} E$

A & E are going to different partition, so they are not equivalent.

$A \xrightarrow{0} B$
$E \xrightarrow{0} E$
$D \xrightarrow{0} E$
$E \xrightarrow{0} E$

$D \xrightarrow{1} D$
$E \xrightarrow{1} E$

E & D are equivalent because they are going to same partition.

Now we have to do $\pi_2 \rightarrow$

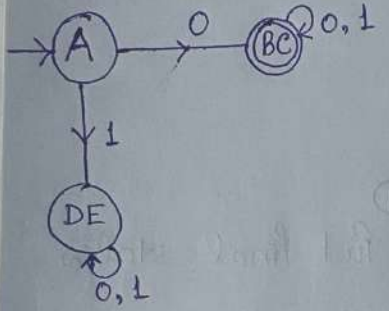
$\pi_2 = \{\{B,C\}, \{A\}, \{D,E\}\}$

Now if you see $\rightarrow$

$\pi_1 = \pi_2$, we stop and this is the min DFA.

$\pi_1 = \{\underbrace{\{B,C\}}_{\text{One State}}, \underbrace{\{A\}}_{\text{One State}}, \underbrace{\{D,E\}}_{\text{One State.}}\}$

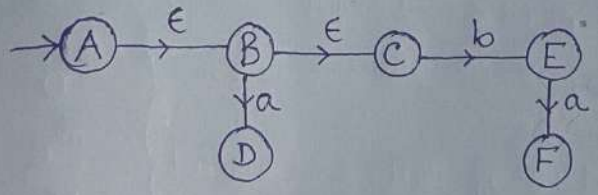
min DFA Digram:



The language here is set of all strings starting with 0.

LECTURE-15:

ϵ-NFA to NFA>



The only question which is possible $\rightarrow$

ϵ - closure. Represented as $(\delta^*(A,E))$

This means what are all states that A can go on 'ϵ'.

$\delta^*(A,\epsilon) = \{A,B,C\}$

$\delta^*(B,\epsilon) = \{B,C\}$

$\delta^*(C,\epsilon) = \{C\}$

$\delta^*(E,\epsilon) = \{E\}$

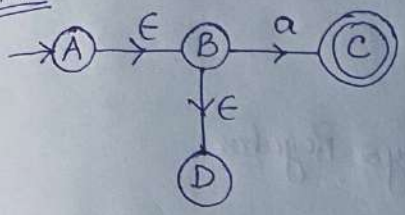
$\delta^*(F,\epsilon) = \{F\}$

$\delta^*(D,\epsilon) = \{D\}$

If a Finite Automata contains ϵ-moves, then it is ϵ-NFA.

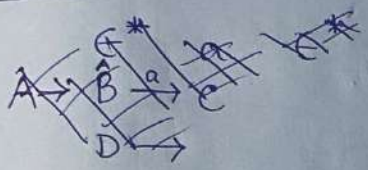
LECTURE-16:

ϵ-NFA:



→ We want to convert it into NFA.

For A:



$A \xrightarrow{\epsilon^*} A \xrightarrow{a} \phi$

$A \xrightarrow{\epsilon^*} B \xrightarrow{a} C \xrightarrow{\epsilon^*} C$

$A \xrightarrow{\epsilon^*} D \xrightarrow{a} \phi$

$A \xrightarrow{\epsilon^*} A \xrightarrow{b} \phi$

$A \xrightarrow{\epsilon^*} B \xrightarrow{b} \phi$

$A \xrightarrow{\epsilon^*} D \xrightarrow{b} \phi$

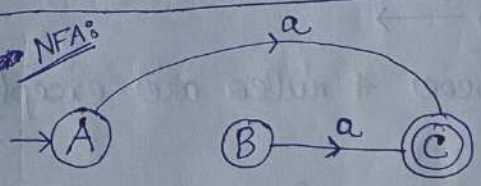
$B \xrightarrow{\epsilon^*} B \xrightarrow{a} C \xrightarrow{\epsilon^*} C$

$B \xrightarrow{\epsilon^*} D \xrightarrow{a} \phi$

$B \xrightarrow{\epsilon^*} B \xrightarrow{b} \phi$

$B \xrightarrow{\epsilon^*} D \xrightarrow{b} \phi$

NFA:



$C \xrightarrow{\epsilon^*} C \xrightarrow{a} \phi$

$C \xrightarrow{\epsilon^*} C \xrightarrow{b} \phi$

$D \xrightarrow{\epsilon^*} D \xrightarrow{a} \phi$

$D \xrightarrow{\epsilon^*} D \xrightarrow{b} \phi$

Whether a language is Regular or not?

→ Guaranteed 1 mark or 2 marks.

NOTE:

Don't solve the questions.

Remember all the rules below.

Identify Regular languages → Rules

1) If a language is **finite**, it is always Regular.

Example: $a^n b^n c^n d^n / \underbrace{n \leq 100}$

Look at this part.

This is finite.

So, the language is Regular.

2) What if infinite language is given? For this, there are 4 rules →

These 4 rules are exceptions.

a) Non-Linear Power → Non Regular:

a^n
a^{2n}
a^{3n+2}
a^{Kn+l}

} Linear → Therefore Regular.

Linear Power always Regular.

$a^{m+1}\ b^{3n+2}$
$a^{Km+2}\ b^{ln+3}$

} Regular because there are no comparison.

$a^{n^2} / n \geq 0$

$a^{n!} / n \geq 0$

$a^{3n^2} / n \geq 0$

$a^{2n} b^{n^2} / n \geq 0$

a^n / n is perfect square

} Non-linear Power. So, not regular.

$a^{n^2} / \underbrace{n \leq 10000}$

Now this has ~~bee~~ become finite, so it's Regular.

b) $(+ \; - \; \times \; \div) \rightarrow$ CSL : These are not Regular.

CFL

$a^m b^n c^p / m \oplus n = p$ → Here addition is there. So this is CFL.

$m, n, p \geq 0$

$a^m b^n c^p / (n \times m = p) \rightarrow$ CSL

$n, m, p \geq 0$

$a^m b^n c^p / (\frac{m}{n} = p) \rightarrow$ CSL

$n, m, p \geq 0$

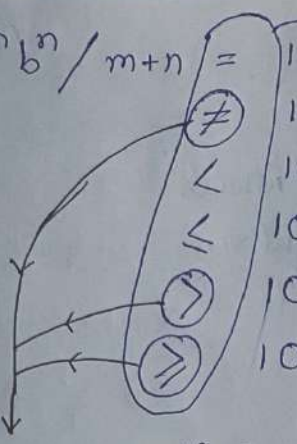
$a^m b^n / m+n = 10 \; ; \; m, n \geq 0 \longrightarrow$ This is finite, so Regular Language.

$a^m b^n / m+n = p \longrightarrow$ This language is not regular.

$m, n, p \geq 0$ This is CFL.

*** If 'p' ~~bee~~ becomes a constant then the language will become Regular.

$a^m b^n / m+n$ = 10, ≠ 10, < 10, ≤ 10, > 10, ≥ 10 → All these operations are Regular.



These operations are Infinite.
Still Regular, these are exception.

$a^n b^m / m+n = 10$, $m, n \geq 0$ ⟶ Finite & Regular.

$a^m b^n / \underbrace{n+m < 10}$; $m, n \geq 0$
Finite, so Regular

$a^m b^n / \underbrace{m+n \leq 10}$; $m, n \geq 0$
Finite & Regular

$a^m b^n / m+n \neq 10$; $m, n \geq 0$ ***

→ $m+n = 0, 1, 2, 3, 4, 5, \ldots, 9, 11, 12, \ldots, \infty$ } We may think that this is not Regular, but this is Regular.

$L^c = \{\underbrace{a^m b^n / m+n = 10}_{\text{Regular}}\} \cup \{\underbrace{(a+b)^* ba (a+b)^*}_{\text{Regular}}\}$ ← Because L does not contain ba.

Complement:
If language = L
Then $L^c = \Sigma^* - L$.

NOTE:

Union of two Regular languages is Regular.

If L is Regular then L^c is Regular.

If L^c is Regular then L is Regular.

Similarly >

$a^m b^n / m+n > 10; m,n \geq 0 \longrightarrow$ Regular & Infinite.

$a^m b^n / m+n \geq 10; m,n \geq 0 \longrightarrow$ Infinite & Regular.

$a^m b^n /$ $\left.\begin{matrix} 2m+3n=10 \\ 5m+6n=10 \\ \vdots \end{matrix}\right\}$ Regular & finite.

$a^m b^n / \underbrace{m-n=10}_{\downarrow}; m,n \geq 0$

$m = n+10$

It's an infinite comparison. So this is Non Regular.

NOTE:

$a^m b^n / m-n$ $\left.\begin{matrix} = \\ \neq \\ < \\ > \\ \leq \\ \geq \end{matrix}\right\}$ $\begin{matrix} 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{matrix}$ $\longrightarrow$ Non-Regular.

If minus is there, no matter which sign, there will be comparison, so it'll always be Non-Regular.

$a^m b^n / m \times n = p ; m,n,p \geq 0 \longrightarrow$ CSL

In case of mulliplication if p becomes a constant, then the Language will become Regular.

$a^m b^n / \frac{m}{n} = p \longrightarrow$ CSL

$a^m b^n / \frac{m}{n} = 10, m, n \geq 0$

$\rightarrow \frac{m}{n} = 10$

$\Rightarrow m = 10n$

Comparison

Here one comparison is there, which means it's a CFL.

NOTE:

CFL can do one comparison at a time.

CSL can do more than one comparison at a time.

NOTE:

In case of division no matter which sign is there, it'll be either CFL or CSL. Regular not possible.

c) Comparison:

CFL → 1 comparison at a time.

CSL → 2 or more comparison at a time.

RL → 0 comparison.

$a^m b^n / m, n \geq 0 \longrightarrow$ This is nothing but $a^* b^*$, so it's Regular.

$a^{2m+2} b^{3m+1} / m \geq 0 \longrightarrow$ CFL

One comparison

$a^{2m+2} b^{3n+1} / m, n \geq 0 \longrightarrow$ Regular, because it's Linear & no comparison is there.

$a^{n^2} b^{3n+2} / m, n \geq 0$ → Not regular because of non linear.

$a^n b^n / n \geq 0$ → CFL

↓↑

$a^m b^n / m = n, n, m \geq 0$ → CFL

Same language

$a^n b^n c^n / n \geq 0$ → CSL

↑↓

$a^m b^n c^p / m = n = p, m, n, p \geq 0$ → CSL

↑↓

$a^m b^n c^p / m = n$ & $n = p$ → CSL

Same language

Same language

$a^m b^n c^p / m = n$ (or) $n = p$ → CFL

NOTE:

Whenever there is 'or' → it's CFL.

" " " 'and' → it's CSL.

$a^m b^m c^n d^n / m, n \geq 0$ → Even though there are two comparison, we are doing one at a time. So, it's CFL.

One Comparison

One Comparison

d) String Matching:

$W W^R$ → Reversal, $W \in \Sigma^*$ → Infinite language.

When you have this, it's CFL not Regular.

$WW / W \in \Sigma^*$ → Not Regular. CSL

$WW^R / W \in \Sigma^*, |W| \leq 10$ → Regular

Finite

$WW / |W| \leq 10$ → Regular

Finite

$|W| = 10$, < 10, ≤ 10 → Finite → Regular

$|W| \neq 10$, > 10, ≥ 10 → Infinite → Not Regular

$WW^RW / W \in \Sigma^* \longrightarrow$ CSL

$W_1 W_1^R W_2 W_2^R / \longrightarrow$ CFL

$W \in \Sigma^*$

$W_1 W_2 W_3 W_4 / W_1, W_2, W_3, W_4 \in \Sigma^*$, $W_2 = W_1$ or $W_3 = W_1^R$

$\longrightarrow$ CFL

---

for RL

Pumping Lemma: It's a -ve test.

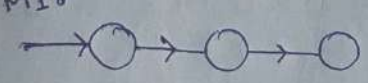
Let L be an infinite Regular language $\exists$ some positive integer m such that any $W \in L$, $|W| > m$ can be decomposed as $W = xyz$ with $|wy| \leq m$ and $|Y| \geqslant 2$ such that $W_i = xy^iz$ is also in L for all $i = 0, 1, 2, \ldots$

Example & Explanation:

Assume there are two FA or machine with 3 & 4 states

$\longrightarrow$

M1:

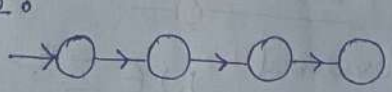


Max strings it can accept = 2

↓

If it's accepting 3 length then there's loop somewhere for sure.

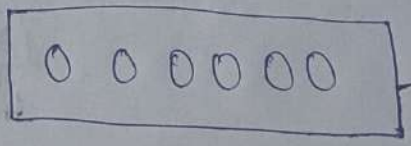
M2:



Max no. of strings it can accept = 3

↓

If it's accepting 4 length then loop is there somewhere for sure.



$\therefore |W| \geq K \longrightarrow$ When there's loop.

↓

We can write W as $\longrightarrow$ x (y) z

↓

Assume this is the loop part.

We can pump 'y' means $\longrightarrow$

Pump

$xy^{*}z \longrightarrow xyz, xy^2z, xy^3z \ldots \infty$

Pumping lemma says $\longrightarrow$ Whenever there's a loop, you can surely break a string in such a way that some part is present in the loop. Now you pump that part, you'll get other strings which are also in the language.

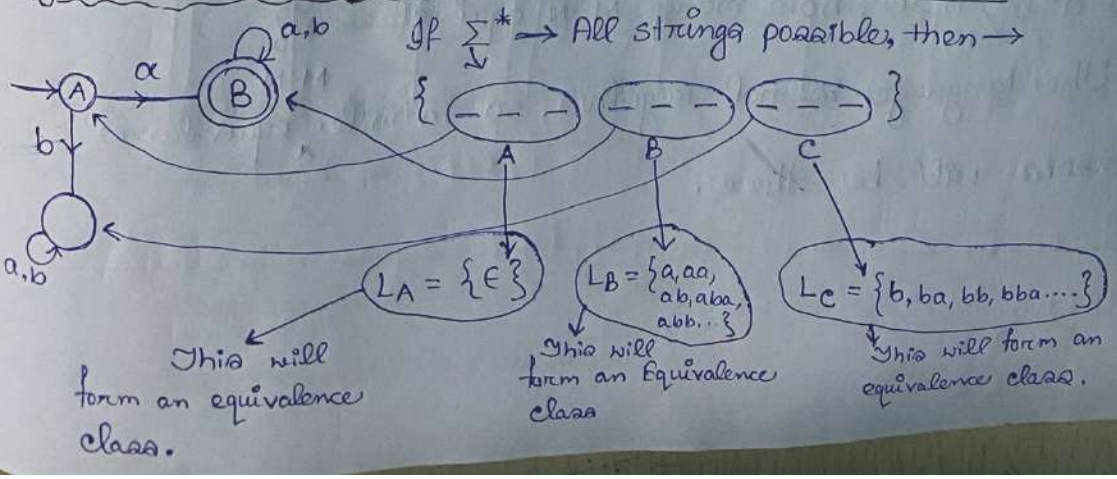
Pumping $\longrightarrow$ Repeating the string again and again.

NOTE:

Pumping Lemma cannot say whether a language is Regular or not.

LECTURE-17:

Myhill-Nerode Theorem > Based on minimal DFA:



Therefore →
Number of equivalance classes = Number of states in min DFA.

NOTE:
If equivalance classes are finite, then we can build Finite Automata. Which means the language is Regular.

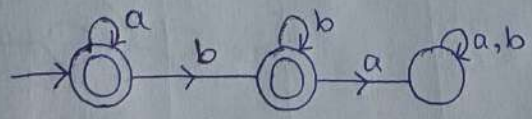
No question from this Theorem till now, but Questions we might face can be like →

If a language is Regular <=> No. of Myhill-Nerode equivalance classes is finite.

NOTE:
Two strings x, y are equivalent if $\delta^*(q_0, x) = \delta^*(q_0, y)$.
(Same State)

For non-regular languages, M-N classes are infinite.

Q: How many M-N equivalance classes are in a^*b^*?
⇒ We have to build the min DFA.



3 States are there so, the answer is 3.

Q: $a^n b^n / n \geq 0$, how many M-N equi classes?
⇒ The language is not regular, so infinite (M-N) equivalance classes will be there.

If $E_1, E_2, \ldots E_n$ are M-N equi classes ⟶

1) $|E_i| \geq 1$

⟶ It means, atleast one string will be present in equivalance class.

2) $i \neq j \Rightarrow E_i \cap E_j = \phi$

This means two equivalance classes will not have anything in common.

3) $\bigcup E_i = \Sigma^*$

If you take union of all the equivalance classes, then it'll be Σ^*.

RE = Recursive Enumerable
REC = Recursive language.

Chomsky Hierarchy

If L is Regular, then it is DCFL, CFL, CSL, REC & RE also.

If L is CSL then it is REC & RE also.

Q: If a language is CSL, is it Regular?

→ May or may not be regular.

NOTE:

Always push up	Never Push Down
RE ↑ Push	↓ Don't Push down.
REC ↑ Push	↓ " " "
CSL ↑ Push	↓ " " "
CFL ↑ Push	↓ " " "
DCFL ↑ Push	↓ " " "
Reg ↑ Push	↓ " " "

If the question is from bottom to top → for example if Reg is given ~~any~~ and you don't know whether Reg then push up and make it DCFL & So, on.

If the question is about RE then don't push down. No matter what, don't push down.

Table: Closure Properties

	Reg	DCFL	CFL	CSL	REC	RE
$\cap$	✓	×	×	✓	✓	✓
$\cup$	✓	×	✓	✓	✓	✓
L^C	✓	✓	×	✓	✓	×
$*$	✓	×	✓	✓	✓	✓
$\cdot$	✓	×	✓	✓	✓	✓
L^R	✓	×	✓			
$H(L)$	✓	×	✓			
$H^{-1}(L)$	✓	✓	✓			

If L_1, L_2 are regular then $L_1 \cup L_2$ is Regular.

If L_1, L_2 are DCFL then $L_1 \cup L_2$ may or may not be DCFL.

Which means → DCFL ∪ DCFL is not closed, so push up.

→ DCFL↑ ∪ DCFL↑
= CFL ∪ CFL
= CFL

∴ You can say that DCFL ∪ DCFL is surely CFL.

Q: $a^n b^n c^m \cap a^n b^m c^m$ [n is the superset].

⇒ $\underbrace{a^n b^n c^m}_{\text{CFL}} \cap \underbrace{a^n b^m c^m}_{\text{CFL}} = \underbrace{a^n b^n c^n}_{\text{CSL}}$

∴ CFL ∩ CFL may or may not be CFL.

Q: $L = \{a^m b^n c^p / m=n \text{ or } m=p\}$

⇒ $L^c = \{a^m b^n c^p / m \neq n \underbrace{\text{ and }}_{\text{'and' means CSL}} m \neq p\}$

∴ If L is CFL, then L^c may or may not be CFL.

NOTE:

H(L) → Homomorphism, it means substitution.

If L = {01, 10, 11}

H(0) = a
H(1) = bb

H(L) = {abb, bba, bbbb} → If you do it in the reverse order, then you'll get the strings

↓

$H^{-1}(L)$

= {01, 10, 11}

Horizontal Question from the table:

Q: Which of the following language is not closed under union?

a) Reg b) DCFL c) REC d) RE.

⇒ b) DCFL

Vertical Question from the table:

Q: Under what operation DCFL is closed?

a) ∪ b) ∩ c) Complement d) *

⇒ c) Complement.

Other type of Questions:

Q: $L_1 \rightarrow DCFL$, $L_2 \rightarrow DCFL$; $L_1 \cap L_2 = ?$

→ $L_1 \cap L_2 = DCFL\uparrow \cap DCFL\uparrow$

$= CFL\uparrow \cap CFL\uparrow$

$= CSL \cap CSL$

$= CSL$

Rule of Order:

"Reversal" then do "intersection" then do "union".

Q: Regular ∪ $(DCFL)^R$ ∩ CFL = ?

→ Regular ∪ $(DCFL)^R$ ∩ CFL

= Regular ∪ $(DCFL\uparrow)^R$ ∩ CFL

= Regular ∪ $(CFL)^R$ ∩ CFL

= Regular ∪ CFL↑ ∩ CFL↑

= Regular↑ ∪ CSL

= CSL ∪ CSL

= CSL

Theory NOTES:

No language is closed under following operations

⊆ → Subset
⊇ → Superset
infinite union
infinite intersection
infinite − ⇒ Infinite set difference

We'll prove for Regular.

$L_1 = a^*b^*$, $L_2 = a^nb^n$

→ $L_2 \subseteq L_1$ → Regular ⇒ Subset Proof

↓

May or may not be Regular

Proof for Superset:

~~$L_1 = \phi$~~ $L_2 = a^nb^n$

$a^nb^n \supseteq \{\}$

Proof for infinite union:

$\inf\cup = L_1 \cup L_2 \cup L_3 \cup L_4 \ldots$ inf union

$\{\epsilon\} \cup \{a\} \cup \{aa\} \cup \{aaa\} \longrightarrow a^* \rightarrow$ Regular

$\{ab\} \cup \{aabb\} \cup \{aaabbb\} \cup \ldots \longrightarrow a^nb^n \rightarrow$ Not Regular.

∴ Infinite Union of Regular Languages may or may not be Regular.

Proof for infinite intersection:

$\inf \cap = L_1 \cap L_2 \cap L_3 \ldots$. $\inf \cap \rightarrow$ May or may not be Regular.

↓

Because we can write it like →

$\overline{L_1} \cup \overline{L_2} \cup \overline{L_3} \cup \ldots$

We already know that union is not closed.

Proof for infinite set difference:

$L_1 - L_2 - L_3 \ldots = L_1 \cap \overline{L_2} \cap \overline{L_3} \ldots$

Intersection is not closed so it's not closed.

IMPORTANT: Secondary Operations

$L_1 - L_2 = L_1 \cap \overline{L_2}$

$L_1 \oplus L_2 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$

$L_1 \Rightarrow L_2 = \overline{L_1} \cup L_2$

$L_1 \Leftrightarrow L_2 = (L_1 \cap L_2) \cup (\overline{L_1} \cap \overline{L_2})$

$L_1 \uparrow L_2 = \overline{L_1 \cap L_2}$

$L_1 \downarrow L_2 = \overline{L_1 \cup L_2}$

NAND NOR

Expand the Secondary Operations.

$L^c = \overline{L}$ → Complement

↳ Same meaning

$\oplus$ → Symmetric Difference

Q: RE - REC

→ RE - REC → Secondary Operation, so expand

RE - REC

= RE ∩ $\overline{REC}$

= RE ∩ REC ↑

= RE ∩ RE

= RE

Table:

	Reg	DCFL	CFL	CSL	REC	RE
LUR	✓	✓	✓	✓	✓	✓
L∩R	✓	✓	✓	✓	✓	✓
L-R	✓	✓	✓	✓	✓	✓

"R" in the table means Regular.

In the above operation (∪, ∩, -) when Regular is involved, don't push.

Example:

Reg ∩ RE = RE

Reg ∪ CFL = CFL

Reg ∪ CSL = CSL

Reg - DCFL

= Reg ∩ $\overline{DCFL}$

= DCFL

DCFL - Reg

= DCFL (Check the above table)

Reg - CFL

= Reg ∩ $\overline{CFL}$ ↑ → As complement of CFL is not closed, push

= Reg ∩ CSL

Reg ∩ CSL = CSL

Q: CFL ∩ Reg

⇒ This is in the form of (L-R). So, don't push Reg. Directly write → CFL ∩ Reg
= CFL

Steps:

1) Normally push.

2) Don't push when secondary operations are there.

3) Don't push when Regular is involved with →
(∪, ∩, -).

Q: Reg - REC

→ Reg - REC
= Reg ∩ $\overline{REC}$ → Complement of REC is closed so no need to push.
= REC

Q: Reg - CFL

→ Reg - CFL
= Reg ∩ $\overline{CFL}$
= Reg ∩ $\overline{CFL}$ ↑ → Complement of CFL is not closed, so we have to push.
= Reg ∩ CSL
= CSL

LECTURE-18 :

Table :

	Regular	CFL
Prefix or init(L)	✓	✓
Right Quotient L_1/L_2	✓	✓
Not Required: Cycle(L)	✓	✓
Not Required: Min(L)	✓	×
Not Required: Max(L)	✓	×
Not Required: Half(L)	✓	×
Not Required: Alt(L)	✓	×

Init(L) : or Prefix(L)

$L = \{10, 001\} = \underline{\in 10} \quad \underline{\in 001}$

Prefix (L) = $\{\in, 1, 10, \in, 0, 00, 001\}$

NOTE :

→ $\in$ always belongs to init(L). Except $L = \phi$.

→ L is always subset of Prefix of L (Prefix(L)).

If L is finite then, Prefix (L) is finite. If L is infinite then Prefix (L) is infinite.

Ex: $L = a^*$ init(L) = ?

Sol: $L = \{\in, a, aa, aaa, \ldots\}$

init(L) = $\{\in, a, aa, aaa, \ldots\}$

= a^*

Q: $L = a^+$ init(L) = ?

→ $L = \{a, aa, aaa, \ldots\}$

init(L) = $\{\epsilon, a, \epsilon, a, aa, \epsilon, a, aa, aaa, \ldots\}$

already covered, so we can remove them.

$= \{\epsilon, a, aa, aaa, \ldots\}$

$= a^*$

Q: $L = (ab)^*$ init(L) = ?

→ $L = (ab)^*$

$= (abababab\ldots)$

init(L) = $\{\epsilon, a, ab, aba, abab, ababa \ldots\}$

$= (ab)^* + (ab)^*a$ ⟶ Answer.

Q: $L = a^*b^*$ init(L) = ?

→ $L = a^*b^*$

$= \{a, b, aa, bb, ab, abb, aab, \ldots\}$

init(L) = $\{\epsilon, a, aa, aaa, \ldots b, bb, bbb, \ldots ab, aab, aabb \ldots\}$

$= a^*b^*$ ⟶ Answer.

Q: $L = (a+b)^*$ init(L) = ?

→ $L = (a+b)^*$

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

We know, $L \subseteq$ init(L). $(a+b)^*$ is universal, so it can only be subset of $(a+b)^*$ itself.

Q: $L = a^n b^n / n \geq 1$ init(L) = ?

→

$L = a^n b^n$

$= aaabbb$

aaabbb

ϵ
a
aa
aaa
aaab
aaabb
aaabbb → no. of a = no. of b

Here no. of 'a' is always ≥ of no. of b's.

We need this part

Already cover in $a^n b^n$

$\therefore$ init(L) = $\{\epsilon\} \cup a^n b^n \cup a^m b^l / m \geq l$

Q: $L = \{w / n_a(w) = n_b(w)\}$ init(L) = ?

→ L = { ab, ba, aabb, bbaa, abab, baba, ... }

ab → ϵ, a, ab ; ba → ϵ, b, ba ; aabb → ϵ, a, aa, aab, aabb .....

This is (a+b)*

$\therefore$ init(L) = (a+b)*

Right Quotient:

$L_1 = \{1101, 01\}$ $\quad L_1/L_2 = \{u / uv = L_1 \ \& \ v \in L_2\}$

$L_2 = \{01, 1, 0\}$.

Divide from Right side:

$\frac{1101}{01} = 11$ $\quad \frac{1101}{1} = 110$ $\quad \frac{1101}{0} = \phi$ (We can't divide)

$\frac{01}{01} = \epsilon$ $\quad \frac{01}{1} = 0$ $\quad \frac{01}{0} = \phi$

$\therefore$ Strings we got = (11, 110, ϵ, 0)

This is L_1/L_2

Left Quotient:

'\' → Left Quotient
'/' → Right Quotient

$L_1 = \{1101, 01\}$, $L_2 = \{01, 1, 0\}$

Divide from Left side>

$L_1 \backslash L_2 = \frac{1101}{01} \quad \frac{1101}{1} \quad \frac{1101}{0}$

$\downarrow \quad \downarrow \quad \downarrow$

$\phi \quad 101 \quad \phi$

$\frac{01}{01} \quad \frac{01}{1} \quad \frac{01}{0}$

$\downarrow \quad \downarrow \quad \downarrow$

$\epsilon \quad \phi \quad 1$

$\therefore$ Strings = {ϵ, 1, 101}

Q: $L_1 = 10^*$ $\quad L_1/L_2 = ?$
$L_2 = 0$

$\rightarrow$ $\frac{10^*}{0}$ = infinite or 10^*

$\frac{1}{0}, \frac{10}{0}, \frac{100}{0}, \frac{1000}{0}$
$\downarrow \; \phi, 1, 10, 100$

$\therefore L_1/L_2 = 10^*$

Easy Questions:

$a^*/a \rightarrow \frac{a}{a}, \frac{aa}{a}, \frac{aaa}{a} \ldots$
$\downarrow \; \epsilon, a, aa \ldots$

$= a^*$

$a^+/a \rightarrow \frac{a}{a}, \frac{aa}{a}, \frac{aaa}{a}, \frac{aaaa}{a}, \ldots$
$= \epsilon \quad a \quad aa \quad aaa \ldots$

$= a^*$

Q: aa^+/a ?

$\rightarrow$ $\left\{\frac{aa}{a}, \frac{aaa}{a}, \frac{aaaa}{a}\right\} \ldots$
$\downarrow \; a, aa, aaa$

$\therefore aa^+/a = a^+$

$$\boxed{a^+a/a = a^+}$$

Q: $L_1 = 10^*$ $\quad L_1/L_2 = ?$
$L_2 = 01^*$

$\rightarrow$ $\frac{10^*}{0} \rightarrow 10^*$ $\quad \frac{10^*}{1} \rightarrow \phi$

$\therefore L_1/L_2 = 10^*$ (Infinite)

Explanation:

$\frac{10^*}{01^*} = \frac{10^*}{0}, \frac{10^*}{01}, \frac{10^*}{011}, \frac{10^*}{0111}, \ldots$
$\downarrow$ F 10^*, ϕ, ϕ, ϕ

F 10^*

Q: $10^+ / 01^+ = ?$

→ $\frac{10^+}{01^+} = \frac{10^+}{01}, \frac{10^+}{011}, \frac{10^+}{0111}, \ldots$

$= \phi$ → finite

Theory NOTE:

→ Infinite language / Infinite language → Divide, may be finite, or infinite.

→ Finite language / Finite Language → Divide is always Finite.

LECTURE-19:

Q: $L_1 = a^n b^n$ $L_2 = (a+b)^*$, $L_1 \cup L_2 = ?$

a) Regular ✓, b) CFL, c) CSL, D) RE

→ NOTE:

Use the tables only when the given languages are complex & you can't guess it.

For this question we can easily see that $L_1 \cup L_2 = (a+b)^*$ Which is Regular.

IMPORTANT:

If $L_1 \cup L_2$ is Regular then L_1, L_2 may or may not be Regular.
If $L_1 \cap L_2$ is Regular then L_1, L_2 may or may not be Regular.
If $L_1 . L_2$ is Regular then L_1, L_2 may or may not be Regular.
If $\bar{L}$ is Regular then L_1 is Regular.
If L_1^R is Regular then L_1 is Regular.

If L_1 is Regular $\Leftrightarrow \bar{L_1}$ is Regular.

If L_1 is not Regular $\Leftrightarrow \bar{L_1}$ is not Regular.

If L_1 is Regular & L_2 is Regular $\Rightarrow L_1 \cup L_2$ is Regular

Contrapositive $\rightarrow p \rightarrow q$

$\sim q \rightarrow \sim p$.

We'll see in Discrete Maths

If $L_1 \cup L_2$ is non Regular then $\Rightarrow L_1$ is non-Regular or L_2 is non-Regular.

Table: Short notes from the big table in Page no. 106.

	Regular	DCFL	CSL	REC
$\bar{L}$	✓	✓	✓	✓

From the above table:

L & $\bar{L}$ are both Regular & L and $\bar{L}$ are both non-Regular.

L & $\bar{L}$ are both DCFL & L and $\bar{L}$ are both non-DCFL.

L & $\bar{L}$ are both CSL & L and $\bar{L}$ are both non-CSL.

L & $\bar{L}$ are both REC & L and $\bar{L}$ are both non-REC.

If L^* is Regular $\Rightarrow$ L may or may not be Regular.

PROOF>1)

$L = \{a^{n^2} / n \geq 1\}$ $\rightarrow$ non-Regular.

$\rightarrow L^* = \{a, aaaa, aaa\ldots 9 \text{ times}, aaa\ldots 16 \text{ times}, \ldots\}$

$= (a^*)$

This is Regular.

PROOF) 2)

$L = \underbrace{\{a^{2^n} / n \geq 1\}}$ → non Linear, so this is non-Regular.

$L^* = \{a^2, a^4, a^6, a^8, a^{10}, \ldots\}$

$= (aa)^*$ → This is Regular.

Table for Decidability: D → Decidable ; UD → Undecidable

	Reg	CFL	CSL	REC	RE
Membership	D	D	D	D	UD
Emptyness	D	D	UD	UD	UD
Finiteness	D	UD	UD	UD	UD
Equivalence	D	UD	UD	UD	UD
Regularity	D	UD	UD	UD	UD
Ambiguity	D	UD	UD	UD	UD
Completeness	D	UD	UD	UD	UD
Disjointedness	D	UD	UD	UD	UD

Same table will be used for complement

Membership → Given a string 'w' whether it belongs to the language 'L' or not is Membership.

$w \in L$ — Table will be same for $w \notin L$

Emptyness → Is $L = \phi$?

$L \stackrel{?}{=} \phi$ — Table will be same for $L \neq \phi$

Finiteness → Is L a finite ?

Table will be same for L is not finite.

Equivalence → Is $L_1 = L_2$?

Regularity → Is L a Regular Language ?

Ambiguity → Is L an ambiguous language ?

Completeness → Is $L = \Sigma^*$?

Disjointness → Is $L_1 \cap L_2 = \phi$?

$L_1 \neq L_2$
L is not Regular
L is not Ambiguous
$L \neq \Sigma^*$
$L_1 \cap L_2 \neq \phi$

→ Complements of the above. Tables will be same for these as well.

Applications of Finite Automata :

1) Lexical analysis → Finite Automata is used.
(Syntax analysis → PDA is used)

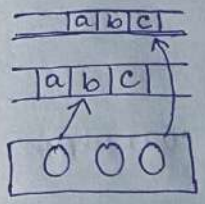
2) Text editors → In Text editors when you use Search → Regular Expression is used.
Auto Suggestions in Text Editors use → Regular Expression.
GREP → Unix Command which uses Regular Expressions.

3) Sequential Circuit Diagram → Mealy & Moore machines are used.

Validations of Finite Automata:

1) Multitape Finite Automata>

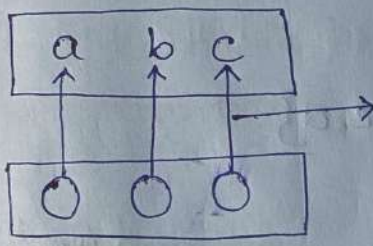
The input can be on multiple tapes.



It has same power as Finite Automata.

2) Multihead Finite Automata>

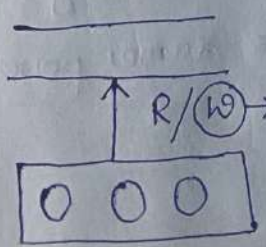
If you have a Finite Automata, then Read-Write heads can be many.



Same power as finite Automata.

3) Finite Automata + R/W Head>

Generally FA has Read head only. But here Write head will also be there.
So, FA can also write on Input, but moves only to Right.
Same power as Finite Automata.



4) Finite Automata + R ⟷ L>

Right — Left

In Finite Automata whenever you can move Right-Left, then it's equivalent to Finite Automata only.

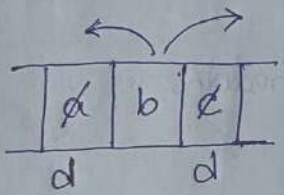
5) FA + R/W + R ⟷ L>

Here Finite Automata can Read/Write and can also move Right & Left.

This is actually Turing Machine.

Turing Machine is more powerful than FA.

TM > FA



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

Time Stamp
1:02:06

6) Finite Automata + 1 Stack >

FA + 1 Stack = PDA

PDA is more powerful than FA.

7) FA + 2 Stacks >

FA + 2 Stacks = Turing Machine

Not exact definition of Turing Machine but it has same power.

8) FA + 1 counter → This machine has no name.

Counter: Counter means Stack with one symbol.

Example →

FA + 1 counter means Stack with only one symbol.

(FA + 1 counter) → Let us call this machine → 'X'

X is more powerful than Finite Automata.

X is less powerful than PDA.

FA < X < PDA

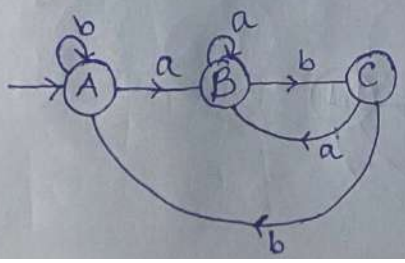
9) FA + 2 Counters
FA + 3 Counters
⋮
n Counters
} These are Turing Machines only.

LECTURE: 20 (Mealy & Moore Machines)

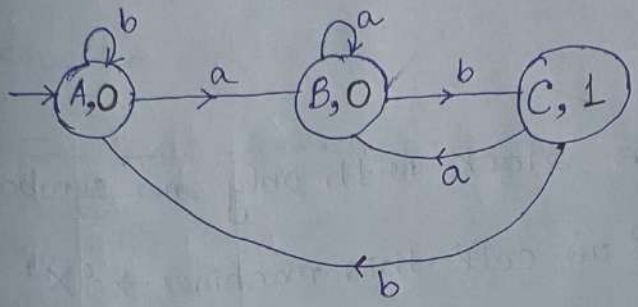
Q: Construct a moore machine that ~~construct~~ takes set of all strings over {a, b} and print '1' as o/p for every occurance of 'ab' as a substring.

→ NOTE:

Whenever you see Mealy or Moore Machines you have to construct the DFA first.



In the DFA whenever there's an 'ab', we want to print the output '1'.

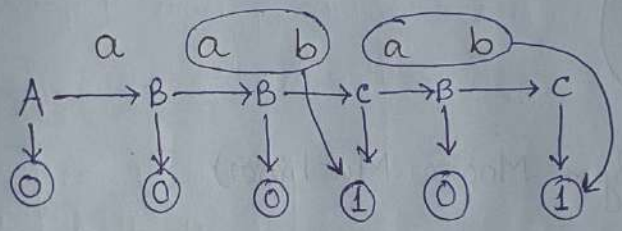


Whenever you hit an 'ab', '1' will be printed.

No need of final state in DFA

NOTE:

In Moore machine something will be printed without seeing the input.



Whenever you get ab, '1' is printed.

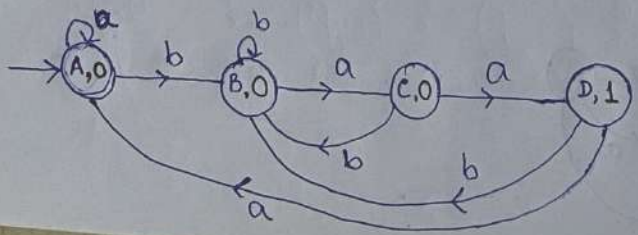
Theory NOTE:

In Moore machine if input is n bits, then output is (n+1) bits.

In Moore machine there'll be output per state.

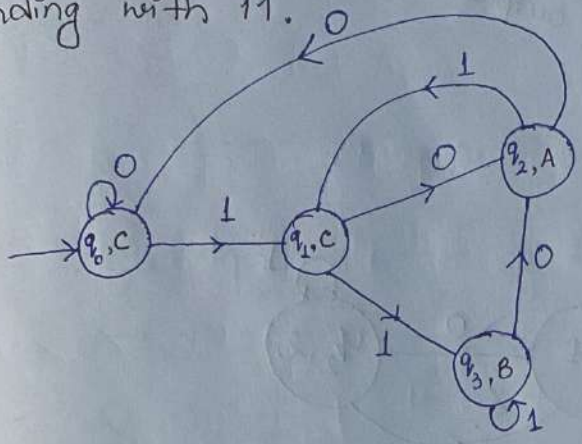
Q: Construct a moore machine that prints '1' for every occurance of 'baa'.

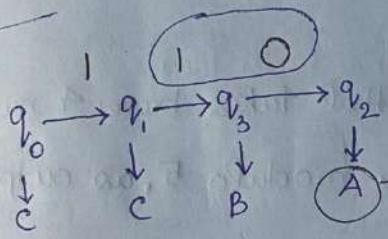
→



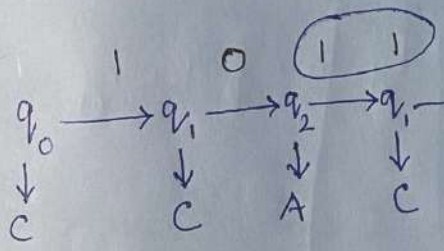
Q: Construct a moore machine that takes set of all strings over {0,1} and produces 'A' as o/p if i/p ends with '10' or produces 'B' as o/p if i/p ends with '11' or produces 'C' otherwise.

→ We are going to make a DFA for ending with 10 & or ending with 11.

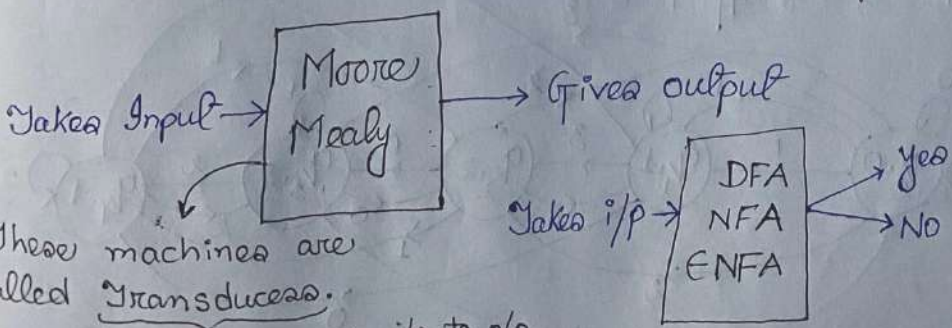




→ Even though 'B' is produced before, we have to take the last one. Which in this case is 'A'.



→ In this case we are taking 'B' because it's the last output

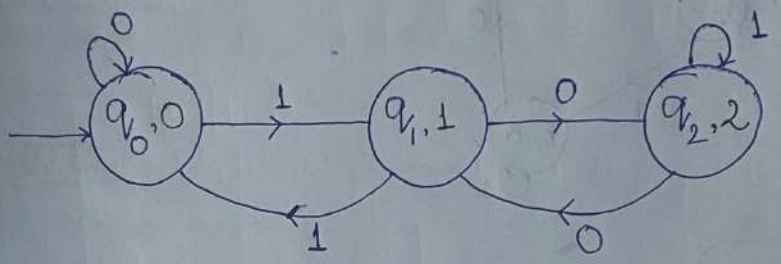


Q: Construct a moore machine that takes binary numbers as input and produces residue modulo 3 as output.

→ This is nothing but mod machine.

	0	1	Δ
→ q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_1	q_2	2

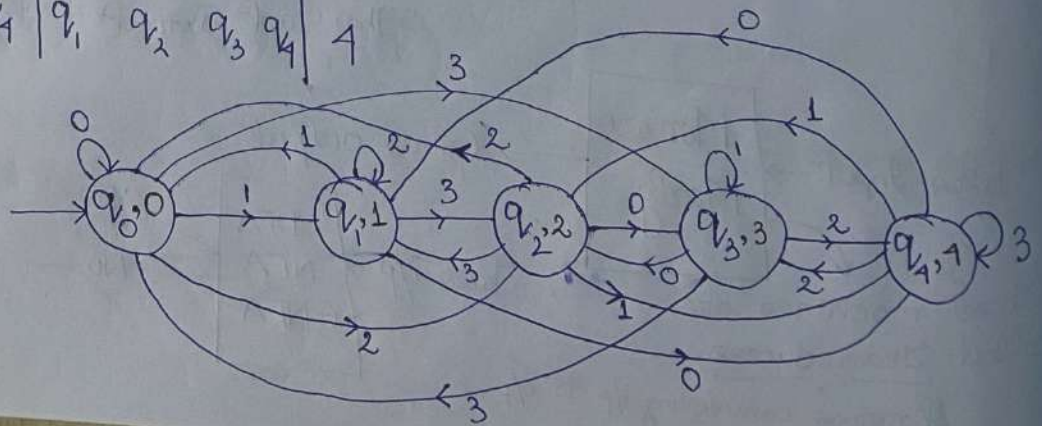
→ Big delta which means, output.



Q: Construct a moore machine that takes base 4 number as Input and produces residue modulo 5, as output.

→ This is a mod machine.

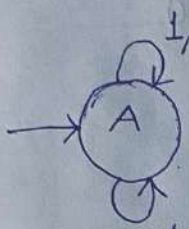
	0	1	2	3	Δ
→ q_0	q_0	q_1	q_2	q_3	0
q_1	q_4	q_0	q_1	q_2	1
q_2	q_3	q_4	q_0	q_1	2
q_3	q_2	q_3	q_4	q_0	3
q_4	q_1	q_2	q_3	q_4	4



Mealy Machine:

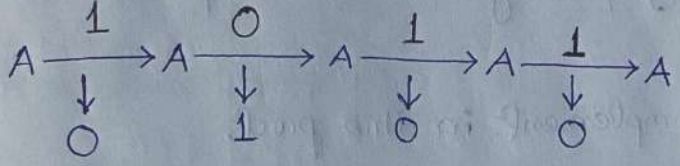
Q: Construct a mealy machine that takes binary numbers as i/p and produces 1's complement as output.

→



1/0 → Seeing a '1', output will be 0

0/1 → Seeing a '0', output will be 1.



1's Complement of 1011 = 0100.

NOTE:

In Mealy machine output is on the **transition.**

In Moore machine output is on the **states.**

In Mealy machine if input is n-bits, then output is n-bits.

Q: Construct a mealy machine for 2's complement. Assuming that i/p is read from LSB to MSB and carry is discarded.

→

1011

← Read in this way

Assume 101100 is the given string for which you have to design the machine.

Steps:

1) While reading from LSB to MSB when you see the first '1', leave that part intact.

In this case →

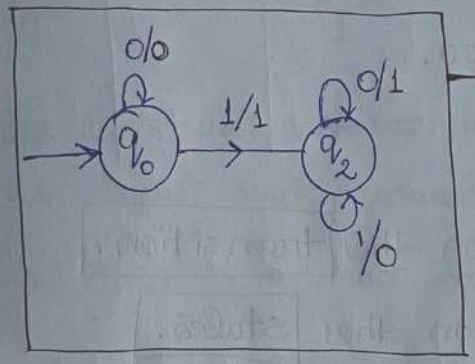
101 | 1 00

↳ Don't touch this part.

2) The remaining part you do 1's complement.

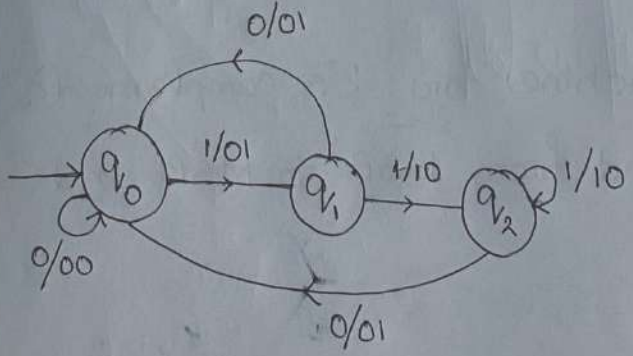
| 101 | 100

↳ Do 1's complement in this part.



→ Mealy machine for 2's complement.

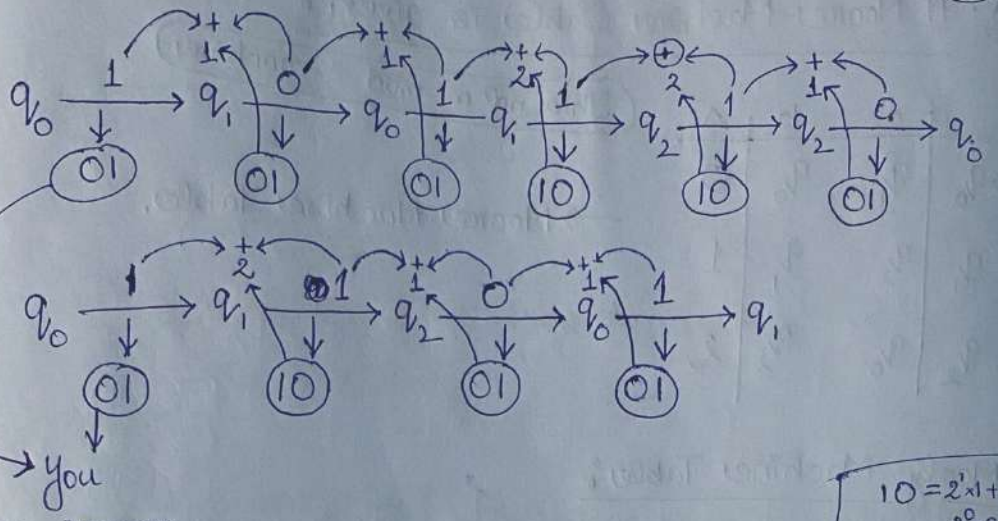
Q: What is the output produced by the following machine:



a) 11 → 01.

b) Sum of present & previous bit. ✓

c) 10 → 100. d) none.



You can ignore this, because there's no previous bit.

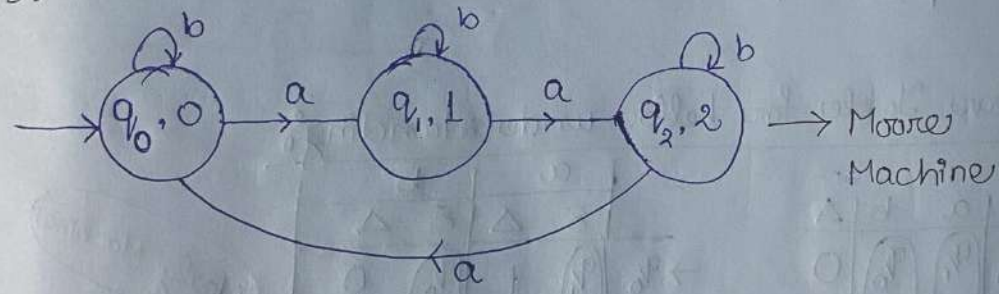
$10 = 2^1 \times 1 + 2^0 \times 0 = 2$

$01 = 2^1 \times 0 + 2^0 \times 1 = 1$

So, the answer is → b)

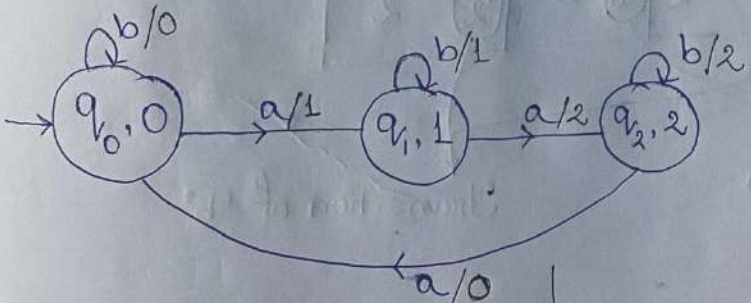
Q: Construct a moore machine for counting no. of a's % 3.

→

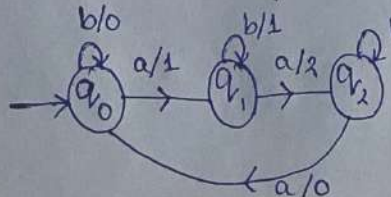


⇓ Convert it into Mealy machine

↓



↓



→ Mealy Machine

Q: A Moore Machine Table is given:

	a	b	Δ
→q_0	q_1	q_0	0
q_1	q_2	q_1	1
q_2	q_0	q_2	2

This not a mod 3 machine

→ Moore Machine Table.

Mealy Machine Table:

In Mealy machine output should be associated with transition. So we have to give the output in the transistion.

	δ a	Δ	δ b	Δ
→q_0	q_1	1	q_0	0
q_1	q_2	2	q_1	1
q_2	q_0	0	q_2	2

→ Small delta means transistion. Which means next state.

→ Big delta means output, associated with transition.

Clear Table for better understanding:

	a	b	Δ
→q_0	q_1	q_0	0
q_1	q_2	q_1	1
q_2	q_0	q_2	2

Moore Machine

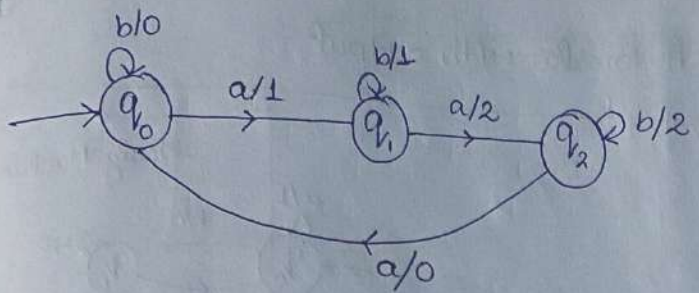
→

	a		b	
	δ	Δ	δ	Δ
→q_0	q_1	1	q_0	0
q_1	q_2	2	q_1	1
q_2	q_0	0	q_2	2

Mealy Machine

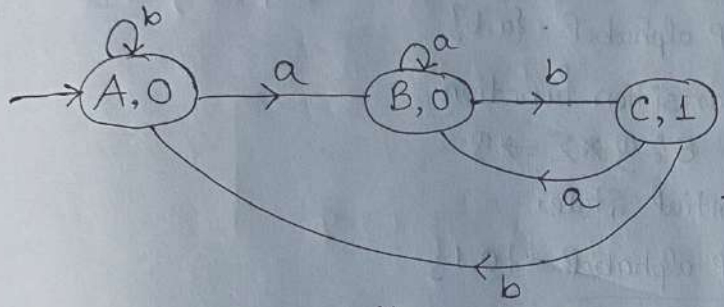
Info/ of 'a' transition

Transition of 'b'

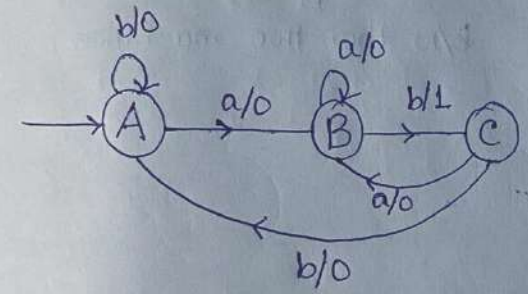


Mealy Machine Diagram from the Table.

Q: Convert the given Moore Machine to Mealy Machine:

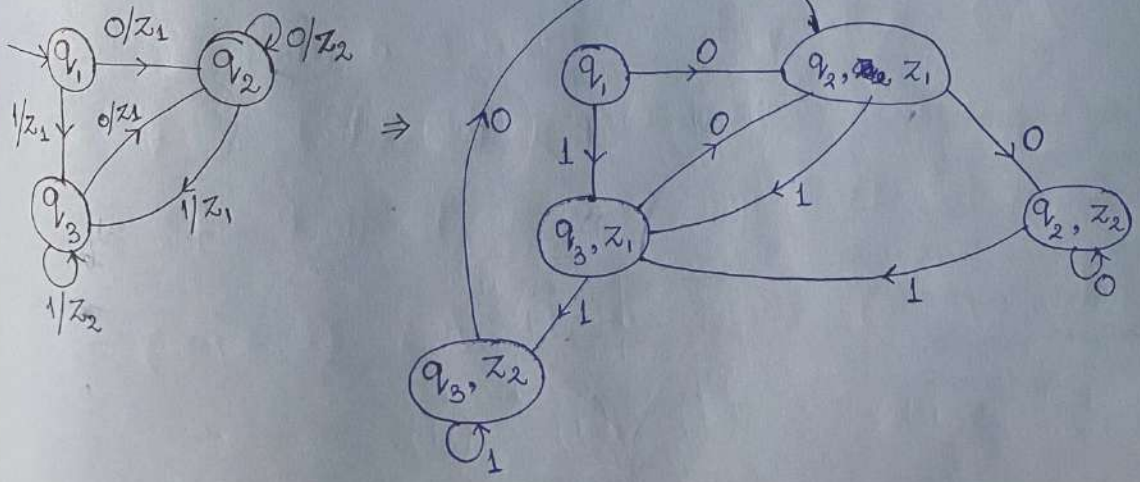


Moore Machine.



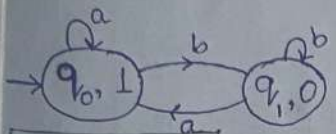
Mealy Machine.

Mealy to Moore Conversion:



Finite Automata with output

Moore Machine

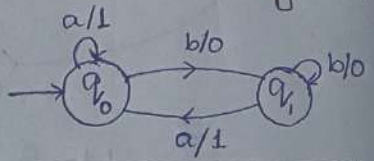


$\lambda : Q \rightarrow \Delta$

$q_0 \rightarrow 1$

$q_1 \rightarrow 0$

Mealy Machine



$\lambda : Q \times \Sigma \rightarrow \Delta$

$(q_0, a) \rightarrow 1$

$(q_0, b) \rightarrow 0$

$(q_1, b) \rightarrow 0$

$(q_1, a) \rightarrow 1$

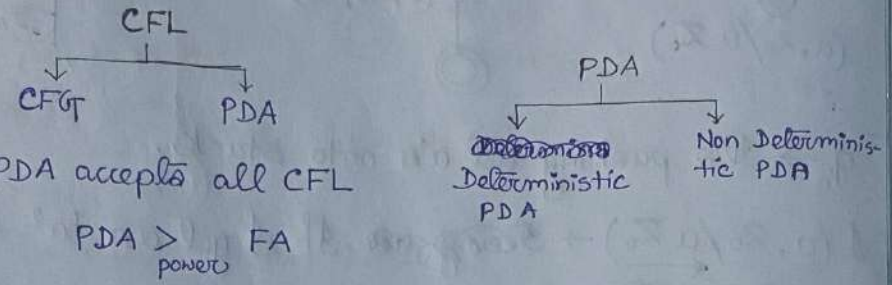
$(Q, \Sigma, \delta, q_0, \Delta, \lambda)$

All same for both machines

- Q = Finite set of States
- Σ = I/P alphabet = $\{a, b\}$
- δ = Transition function
  $\delta : Q \times \Sigma \rightarrow Q$
- q_0 = Initial State
- Δ = O/P alphabet = $\{0, 1\}$
- λ = O/P function → The only difference b/w the two machines.

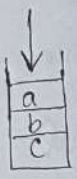
Context Free Languages

LECTURE : 1

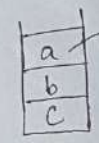


PDA accepts all CFL

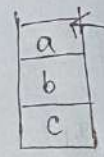
PDA $\underset{\text{power}}{>}$ FA



Assume you have a stack & you're asked to tell the top of the stack.



You'll pull out 'a' & tell that 'a' is in the top of the stack



After telling the top, you have to put 'a' back in the stack.

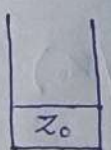
NOTE:

There is no pop operation in PDA. If you want to do POP operation, you simply don't put the element back in the stack after doing the top operation.

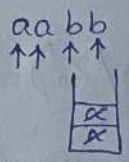
Q: $a^n b^n / n \geq 1$. Make a PDA.

→ NOTE:

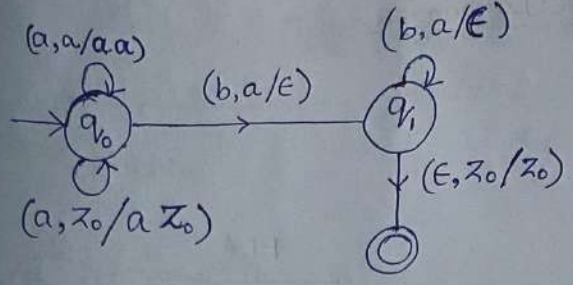
Always at the bottom of the stack there's a special character. Most of the textbooks using Z_0.



This important to know whether the stack has been emptied or not.



Whenever you see an 'a' you push it in stack. And when a 'b' comes you pop an 'a', now they'll be matched.



aa bb

q_0 → It's pushing all a's onto stack.

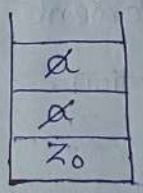
(a, Z_0/a Z_0) → If I get an 'a' & the top of the stack is Z_0, I'm pushing 'a' but before that we have to push Z_0 because it has come out of the stack.

For popping we'll use a different state q_1.

q_1 → It'll be popping a's for every "b"

(b, a /ϵ) → After getting a 'b' we already know that 'a' is in the top of the stack, so pop it.

a a b b ϵ

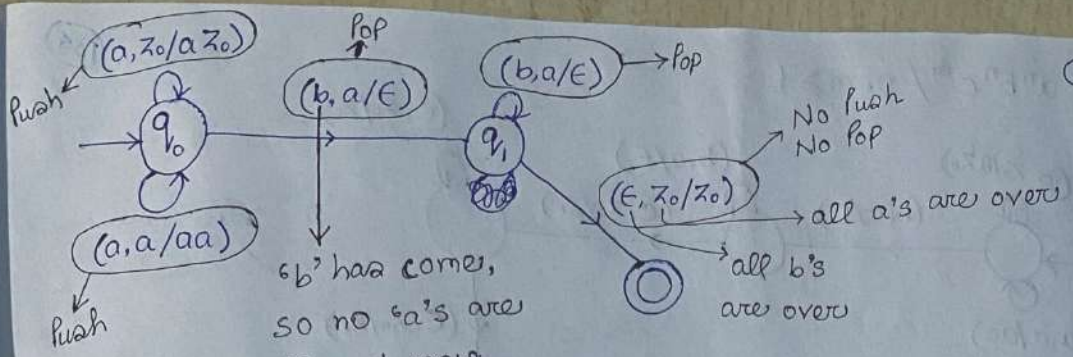


At the end there's epsilon. When we come to Z_0 & ϵ it means a's & b's are equal.

(ϵ, Z_0/Z_0) → If inputs are over and stack is empty. Then you can either empty the stack or leave it as it is. And you can go to final state.

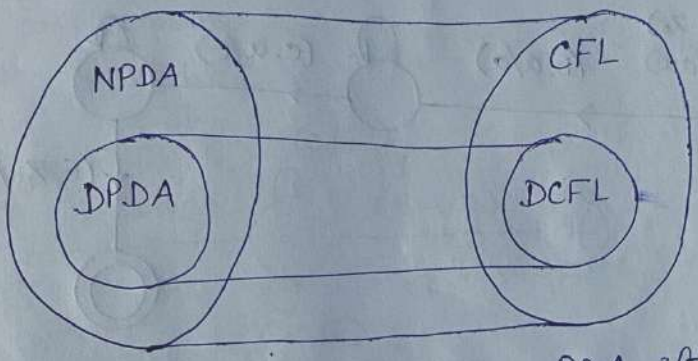
(a, Z_0/a Z_0) (a, a/aa) (b, a/ϵ) (ϵ, Z_0/Z_0)

These all are called configuration



NOTE:

DPDA is less in power than NPDA.



Whenever in question you see PDA, it's NPDA.
They'll mention if it's DPDA.
When they say FA, it means NFA only.

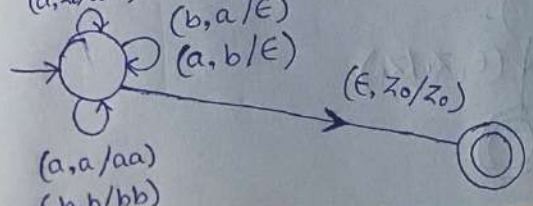
DFA NFA
↓
Based on transition

DPDA
↓
One Copy
Dead Configuration can be there.

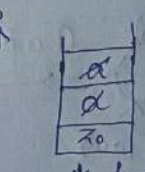
NPDA
↓
Mulliple Copies
Dead Configuration can be there.

This is not $a^n b^n / n \geq 1$

Q: $w / n_a(w) = n_b(w)$.



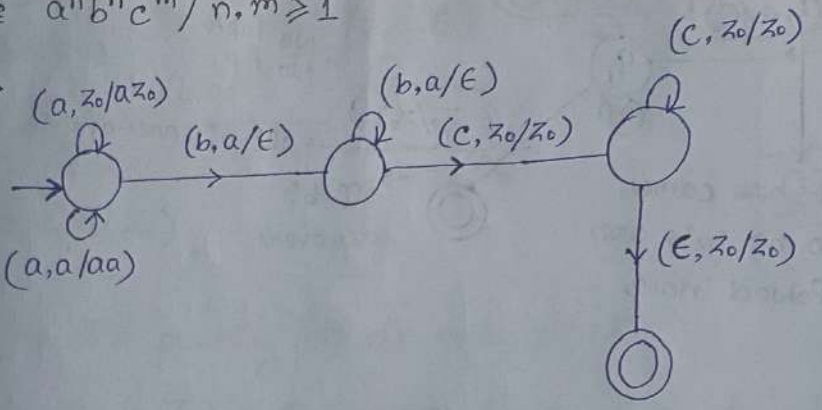
abab



Same for bbaa, baba

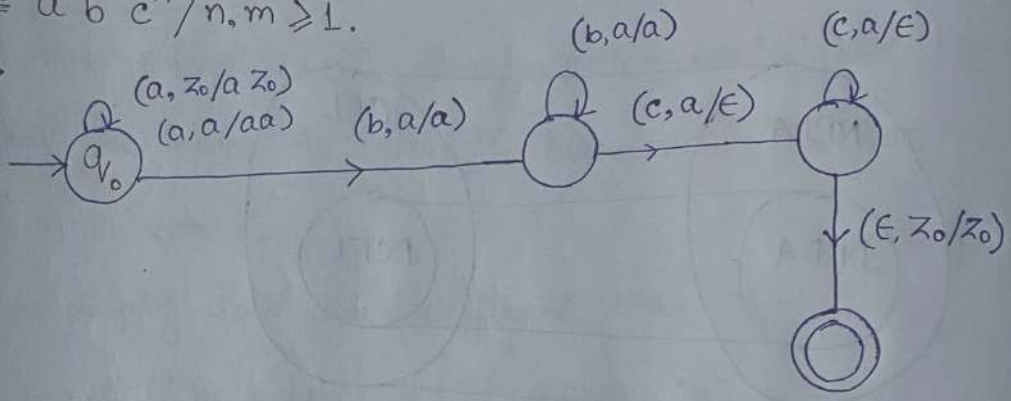
Q: $a^n b^n c^m / n, m \geq 1$

→



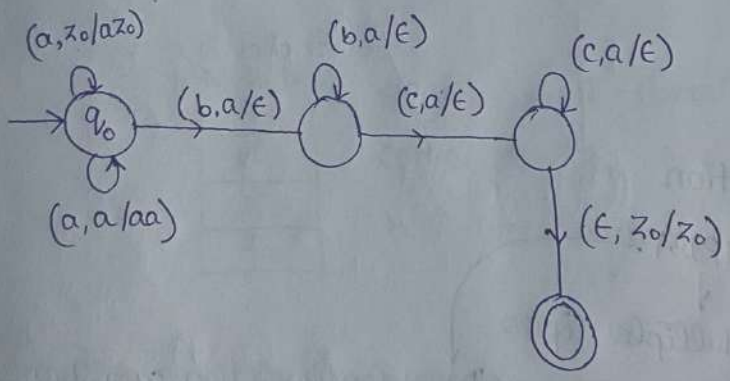
Q: $a^n b^m c^n / n, m \geq 1.$

→



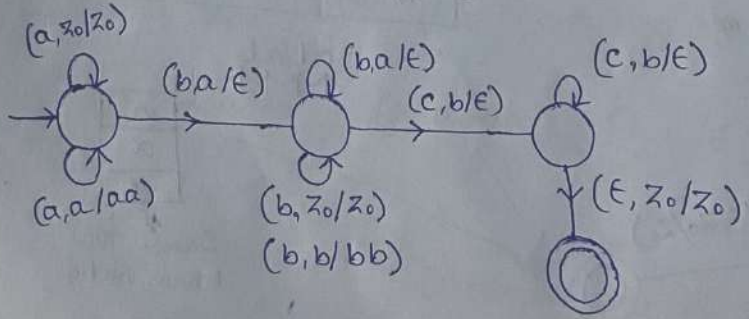
Q: $a^{m+n} b^m c^n / n \geq 1$

→ $a^n a^m b^m c^n = a^{m+n} b^m c^n$



Q: $a^n b^{m+n} c^m / n, m \geq 1.$

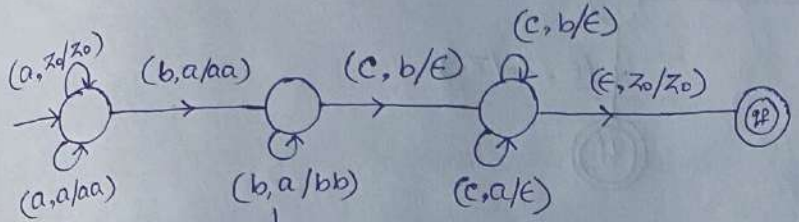
→ $a^n b^{m+n} c^m = a^n b^n b^m c^m$



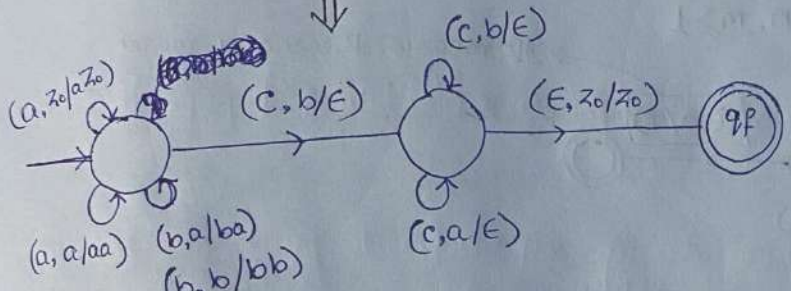
LECTURE: 2

Q: $a^n b^m c^{n+m} / n, m \geq 1$

→ $\underbrace{a^n b^m}_{\text{Push}} \underbrace{c^m c^n}_{\text{Pop}}$



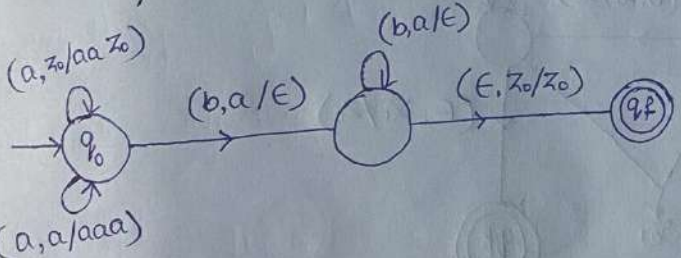
We can combine these two states



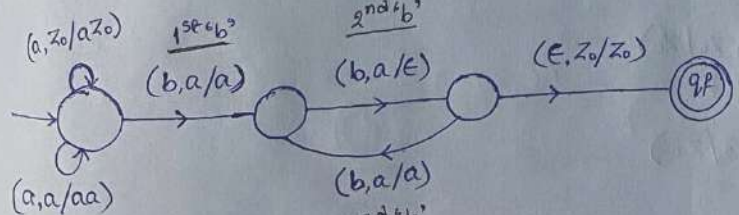
Q: $a^n b^{2n} / n \geq 1$

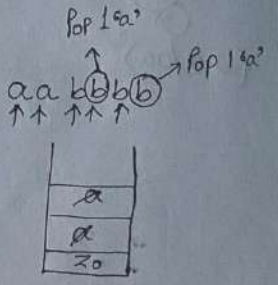
1st posibility: For every 'a' → Push 2 a's.
For every 'b' → Pop 1 'a'.

→



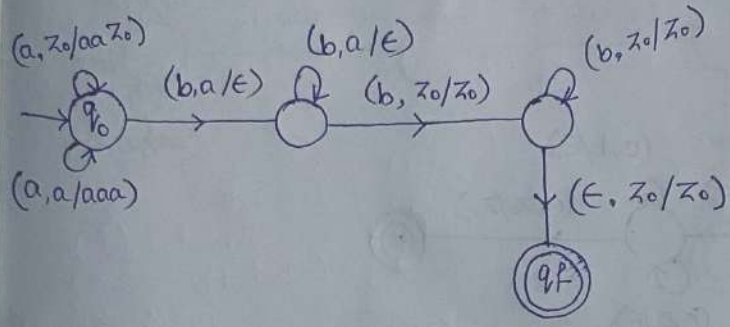
2nd posibility: For every 'a' → push 1 'a'.
For every 2 'b' → pop 1 'a'.





Q: $a^n b^{2n+1} / n \geq 1$

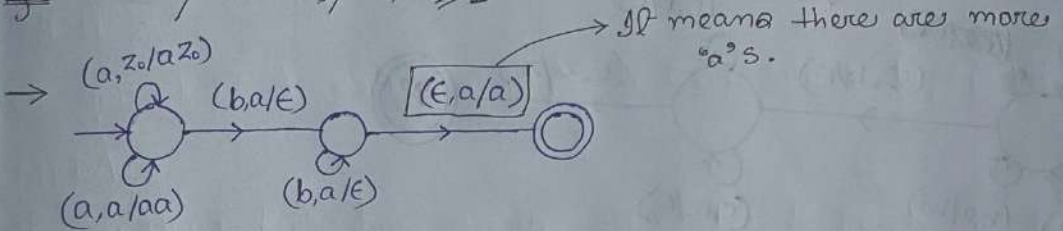
→ $a^n b^{2n} b$



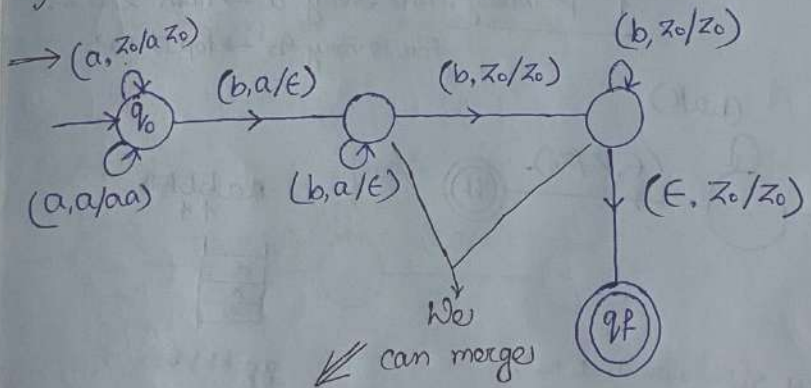
Q: $a^n b^n c^n / n \geq 1$

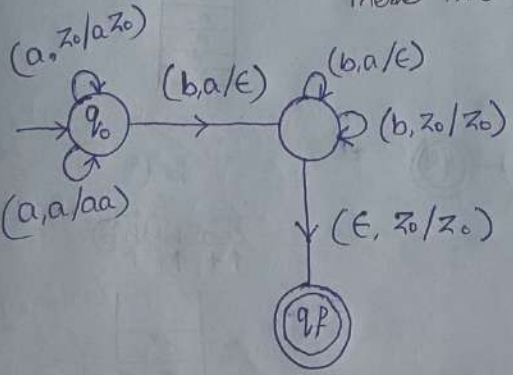
→ This is CSL, we'll see later.

Q: $a^n b^m / n > m, / n, m \geq 1$



Q: $a^n b^m /$ ~~$n \leq m$~~ $n < m$





Q: $WcW^R / W \in (a,b)^+$

→ WcW^R = abcba
↓ abbcbba
Special Character.

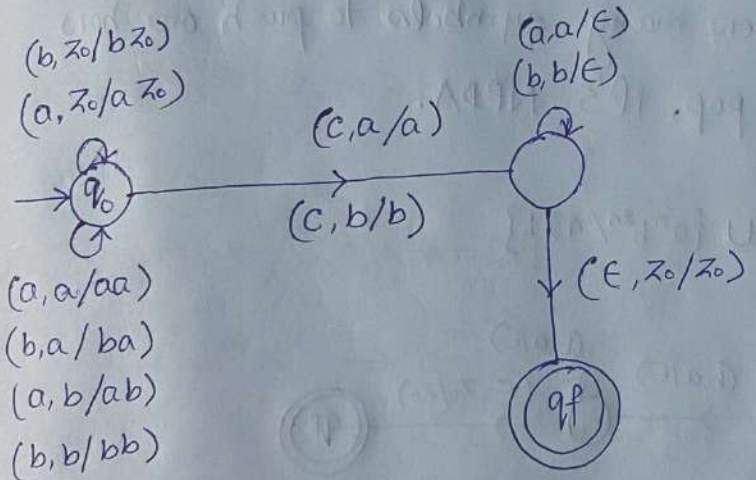
NOTE:
If 'c' becomes Regular the whole WcW^R will become Regular.

Assume: $c = (a+b)^*$

$W = abb$

a bbc bb a
↓
This part will get consumed by 'c'.

So we'll get → $a(a+b)^*a + b(a+b)^*b$.



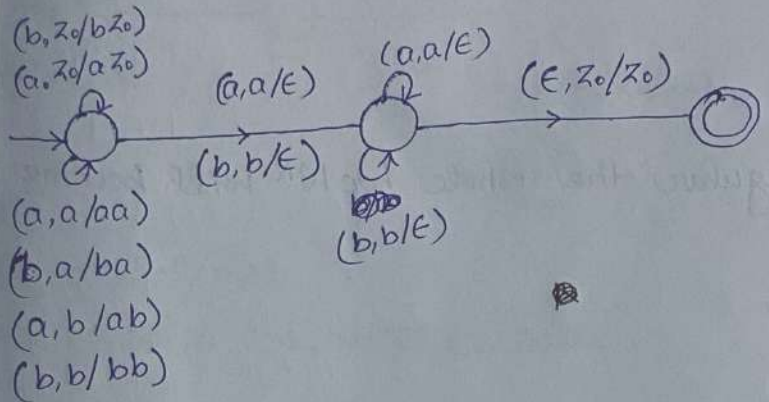
Till Now whatever we've seen is called Deterministic Push Down Automata (DPDA).

The languages which are accepted by the Machine is called Deterministic Context Free Languages (DCFL).

NPDA (Non Deterministic PDA):

Q: $WW^R / W \in (a,b)^+$

→ These are even length palindromes except epsilon.



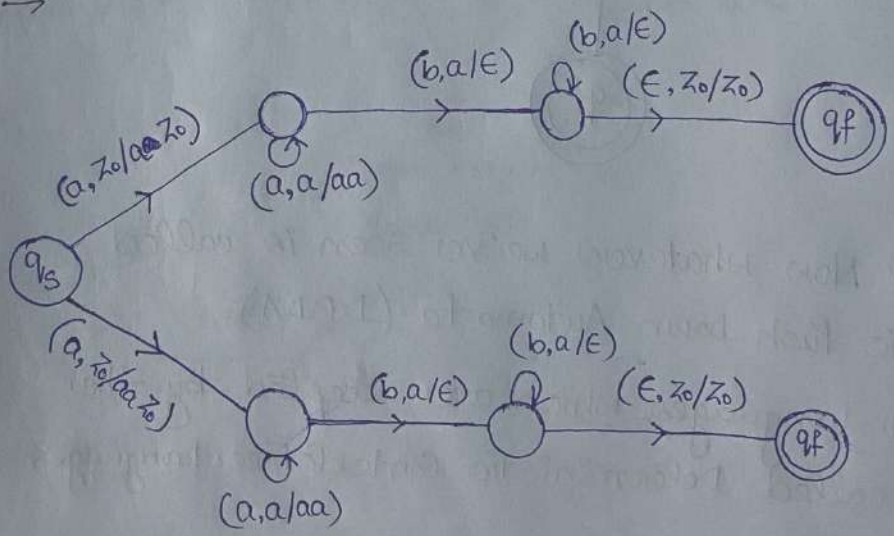
NOTE:

If you don't know when to push & or when to pop. It's NPDA.

If you don't know how many symbols to push or how many symbols to pop, it's NPDA.

Q: $L = \{a^n b^n / n \geq 1\} \cup \{a^n b^{2n} / n \geq 1\}$

→



LECTURE: 03

1) $a^{m+n} b^n c^m / n, m \geq 1$

→ DCFL

2) $a^m b^{m+n} c^n / n, m \geq 1$

→ DCFL

3) $a^m b^n c^{m+n} / n, m \geq 1$

→ DCFL

4) $a^m b^m c^n d^n / m, n \geq 1$

→ DCFL

5) $a^m b^n c^m d^n / m, n \geq 1$

→ CSL

6) $a^m b^n c^n d^m / n, m \geq 1$

→ DCFL

7) $a^m a^i c^m d^k / m, i, k, n \geq 1$

→ $a^m a^i c^m d^k \rightarrow a^{m+i} c^m d^k / m, i, k \geq 1$

$\rightarrow a^x c^y d^z / x > y$

$\rightarrow a^m b^n / m > n$

DCFL

8) $a^m b^n / m < n$

→ DCFL

9) $a^n b^{2n} / n \geq 1.$

→ DCFL

10) $a^n b^{n^2}$

→ CSL

11) $a^n b^{2^n} / n \geq 1$

→ CSL

12) $WW^R / W \in (a,b)^*$

→ CFL

13) $W c W^R / W \in (a,b)^*$

→ DCFL

14) $WW / W \in (a,b)^*$

→ CSL

15) $a^n b^n c^m / n > m$

→ $a^n b^n c^m$ → Two comparisons are there.

Once you're done with $a^n b^n$, the stack will be empty, so you can't compare C.

CSL

16) $a^n b^n c^n d^n / n \leq 10^{10}$

→ Regular.

17) $a^n b^{2n} c^{3n} / n \geq 1$

→ CSL. Just like Q. No 15.

18) $x c y / y, x \in (0,1)^*$

→ Regular.

19) $x x^R / x \in (a,b)^*, |x| = l.$

→ Regular.

20) $WWW^R / W \in (a,b)^*$

→ CSL.

21) $a^n b^{3^n} / n \geq 1$

→ CSL

22) $WW^RW / W \in (a,b)^*$

→ CSL

23) $a^m b^n / m \neq n ; m,n \geq 1$

→ DCFL.

OR = CFL
AND = CSL

24) $a^i b^j / i \neq 2j+1$

→ DCFL

25) $a^m b^n / m = 2n+1$

→ DCFL

26) $a^{n^2} / n \geq 1$

→ CSL

27) $a^{2^n} / n \geq 1$

→ CSL

28) $a^{n!} / n \geq 1$

→ CSL

29) a^m / m is Prime.

→ CSL

30) a^k / k is even.

→ Regular.

31) $a^i b^j c^k / i > j > k$

→ CSL

32) $a^i b^j c^k / j = i+k$

→ DCFL

33) $a^i b^j c^k / i = k$ OR $j = l$.

→ CFL

34) $a^i b^j c^k d^l / i = k$ and $j = l$.

→ CSL

35) $xwwR / x, w \in (a+b)^+$

→ CFL

36) $xwwR / x, w \in (a+b)^*$

→ Regular

37) $a^m b^l c^k d^n / m, l, k, n \geq 1$

→ Regular

38) $a^n b^{4m} / n, m \geq 1$

→ This is $a^+(bbbb)^+$.
∴ Regular.

39) $\{a^3, a^8, a^{13}, a^{18}, \ldots\}$

→ This is Arithmatic Progression.
∴ Regular.

RE = aaa(aaaaa)*

40) $\{a^{2n+1} / n \geq 1\}$

→ Regular

41) $\{a^{n^n} / n \geq 1\}$

→ CSL

42) $\{w / w \in (a,b)^*, |w| \geq 100$

→ Regular.

43) $\{w / w \in (a,b)^*; n_a(w) = n_b(w)\}$

→ DCFL

44) $\{w / w \in (a,b,c)^*, n_a(w) = n_b(w) = n_c(w)\}$

→ CSL

45) $\{w / w \in \{a,b,c\}^* / n_a = n_b$ or $n_b = n_c\}$

→ CFL

A6) $\{w / w \in \{a,b\}^*, n_a(w) \geq n_b(w)+1\}$

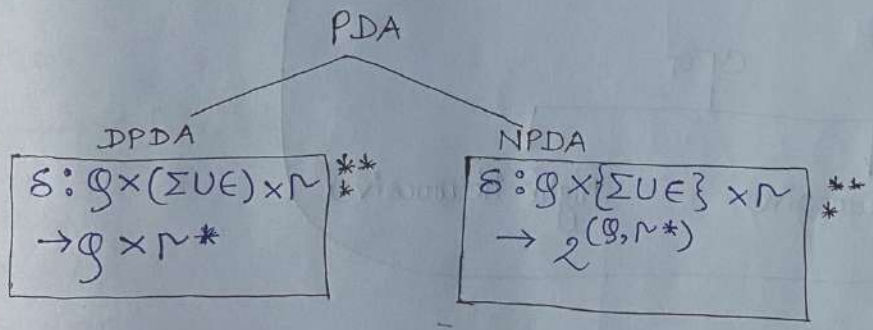
→ DCFL.

LECTURE: 09

PDA

$(Q, \Sigma, \delta, q_0, Z_0, F, \Gamma)$

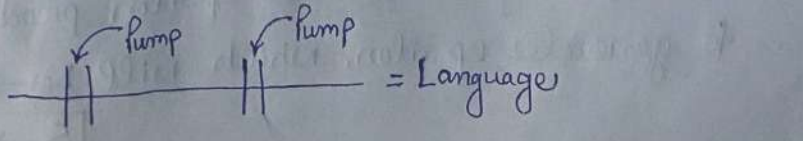
$Q \rightarrow$ Finite set of states
$\Sigma \rightarrow$ i/p symbols
$\delta \rightarrow$ Transition Function
$q_0 \rightarrow$ initial state
$Z_0 \rightarrow$ Bottom of the stack
$F \rightarrow$ set of Final states
$\Gamma \rightarrow$ Stack alphabet: Symbols we push onto stack.



Pumping Lemma for Context Free Languages:

If L be a CFL and $w \in L$ such that $|w| \geq n$ for some positive integer 'n', then 'w' can be decomposed as $w = abcde$ such that →

1) $bd \neq \epsilon$
2) $|bcd| \leq n$
3) For all $i \geq 0$, string ab^icd^ie is also in 'L'.



Context Free Grammars : Type 2

$A \rightarrow \alpha$, $A \in V$

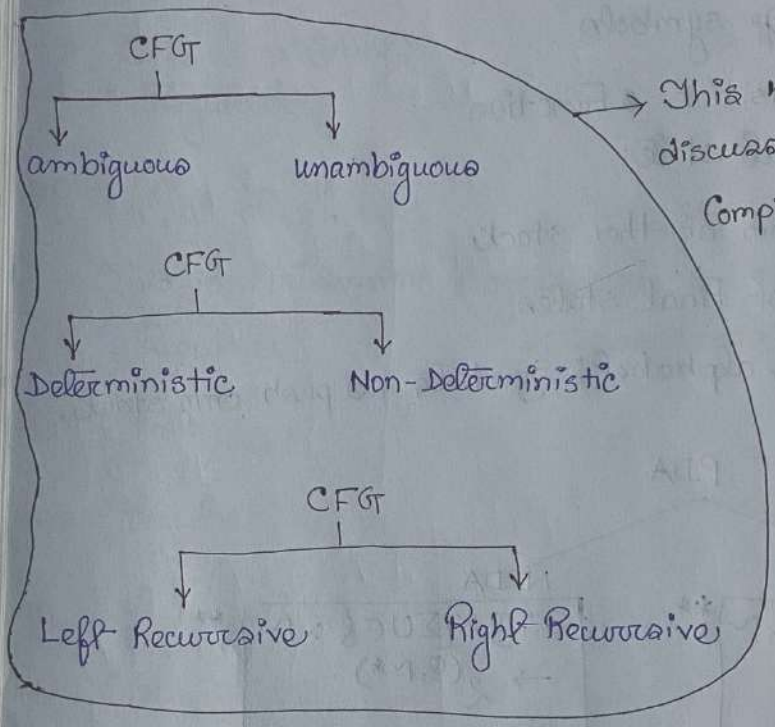
$\alpha \in (V \cup T)^*$

Example :

$A \longrightarrow aAb/a$

$A \longrightarrow BaBA/ab$

There are many ways in which we can classify CFGs :



There can be many Grammars for a given language. Some Grammars are more suitable for Derivation purposes.

$A \rightarrow \epsilon$ → These are called epsilon production which can be eliminated.

But if language contains 'ϵ', then there should be definitely atleast one epsilon production to generate epsilon, which will be →

$S \rightarrow \epsilon$

If the language doesn't contain 'ϵ' then there is no need of epsilon production.

Elimination of Epsilon productions:

$S \longrightarrow aSb/aAb$

$A \longrightarrow \epsilon$

> Wherever 'ϵ' is there in a variable. That is nullables.

1st Step:

We have to see the nullable variables = {A}

2nd Step:

You go to every part which contains 'A'. You write with it & write without it.

$\therefore S \longrightarrow aSb/aAb/ab$

We can eliminate $A \longrightarrow \epsilon$, so in 'S', aAb can also be removed.

$\therefore S \longrightarrow aSb/ab$

Q: $S \longrightarrow AB$

$A \longrightarrow aAA/\epsilon$

$B \longrightarrow bBB/\epsilon$

→ nullable = {B, A, S}

↓

Whenever starting symbol is nullable then epsilon is surely present in the language. Therefore we can't eliminate all null productions. (ϵ productions)

$B \longrightarrow bBB/\epsilon$

$= B \longrightarrow bBB/bB/b$

$A \longrightarrow aAA/\epsilon$

$= A \longrightarrow aAA/aA/a$

$S \longrightarrow AB$

$= S \longrightarrow AB/A$

$S \longrightarrow AB/A$

$= S \longrightarrow AB/A/B/\epsilon$

NOTE:

When 'S' is nullable, you have to add 'ϵ' in 'S'.

∴ Reault =

$S \longrightarrow AB/A/B/\epsilon$

$A \longrightarrow aAA/aA/a$

$B \longrightarrow bBB/bB/b$

Q: $S \longrightarrow AabC$

$A \rightarrow BC$

$B \rightarrow b/\epsilon$

$C \rightarrow D/\epsilon$

$D \rightarrow d$

Here in 'S' because of 'ab', S is not nullable.

→ nullable = {B, C, A}

$S \rightarrow AabC/Aab$

$A \rightarrow BC/B$

$B \rightarrow b/\epsilon$

$C \rightarrow D$

$D \rightarrow d$

→ Eliminating ∈ from C.

$S \rightarrow AabC/Aab$

$A \rightarrow BC/B/C$

$B \rightarrow b$

$C \rightarrow D$

$D \rightarrow d$

→ Eliminating ∈ from B.

$S \longrightarrow AabC/Aab/ab/abC$

$A \longrightarrow BC/B/C$

$B \longrightarrow b$

$C \longrightarrow D$

$D \longrightarrow d$

→ Eliminating ∈ from A for final reault.

These are called Unit production, we can eliminate them.

Elimination of Unit Productions:

For example →

$A \rightarrow B$

↓

We can eliminate them

$A \rightarrow B$
$B \rightarrow a$

→ We can write: $A \rightarrow a$

$S \longrightarrow Aa/B$
$B \longrightarrow A/bb$
$A \longrightarrow a/bc/B$

We first have to find all chains, after that we can break them.

$S \rightarrow \cancel{B} \longrightarrow bb$

$\therefore S \longrightarrow Aa/B/bb.$

$S \rightarrow \cancel{B} \rightarrow A \longrightarrow a$
$\qquad\qquad\qquad\quad \rightarrow bc$

$\therefore S \longrightarrow Aa/bb/a/bc$

$B \rightarrow \cancel{A} \longrightarrow a$
$\qquad\quad \rightarrow bc$
$\qquad\quad \rightarrow B \longrightarrow bb$

$\therefore B \longrightarrow bb/a/bc$

$A \rightarrow \cancel{A} \rightarrow bb$
$\qquad\quad \rightarrow a$
$\qquad\quad \rightarrow bc$

$\therefore A \longrightarrow a/bc/bb$

NOTE:

From "S" we can reach A but we can't go to B using anything. So, "B" is useless.

Result:

$S \longrightarrow Aa/bb/a/bc$
$A \longrightarrow a/bc/bb$

Q:

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow \cancel{C}/b/a$

$C \rightarrow \cancel{D}/a$

$D \rightarrow \cancel{E}/a$

$E \rightarrow a$

→ Vertices = {S, A, B, C, D, E}

Terminals = {a, b}

$B \rightarrow \cancel{C} \rightarrow D \rightarrow E \rightarrow a$

$C \rightarrow \cancel{D} \rightarrow E \rightarrow a$

$D \rightarrow \cancel{E} \rightarrow a$

∴ $S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow a/b$

$C \rightarrow a$

$D \rightarrow a$

$E \rightarrow a$ → Useless, not needed.

From 'S' only A & B are reachable. Therefore other parts are useless.

Elimination of Useless Productions:

$S \rightarrow Aa$

$B \rightarrow aAb$

$A \rightarrow cd$

Sometimes 'B' cannot generate any string (Terminals). Means 'B' is not deriving anything. ∴ B is useless.

Sometimes 'B' can't be reached.

Example:

$S \rightarrow AB/a$

$A \rightarrow BC/b$

$B \rightarrow aB/C$

$C \rightarrow aC/b$

Terminals = {a, b}

Vertices = {S, A, B, C}

First we have to check Derivability = {a, b}

↓

Deriving a String.

$S \rightarrow a$ → S is deriving a string

$A \rightarrow b$ → A is deriving a string

∴ Now it'll look like = {a, b, S, A}

B & C are not generating any string. Therefore useless.

In simple words → {a, b, S, A} any combination from this is not present in B & C.

NOTE:
Terminals are always useful.

∴ Result: $S \rightarrow AB/a$
$A \rightarrow BC/b$

Now if you see, B is in 'S', so we can remove it

∴ $S \rightarrow a$

B & C both are in 'A', so we can remove it.

∴ $A \rightarrow b$

We get: $S \rightarrow a$
$A \rightarrow b$

Here from S, we can't reach 'A'.

∴ Final Result: $\boxed{S \rightarrow a}$

Q:

$S \rightarrow AB/AC$
$A \rightarrow aAb/bAa/a$
$B \rightarrow bbA/aaB/AB$
$C \rightarrow abCA/aDb$
$D \rightarrow bD/aC$

Terminals: {a, b}
Vertices: {S, A, B, C, D}

Ans: Useful Symbol: {a, b, A, B, S}

C & D are not useful, so we can remove them.

$S \rightarrow AB$
$A \rightarrow aAb/bAa/a$
$B \rightarrow bbA/aaB/AB$
} Answer.

Q:

$S \rightarrow ABC/BaB$
$A \rightarrow aA/BaC/aaa$
$B \rightarrow bBb/a$
$C \rightarrow CA/AC$

Vertices: {A, B, C, S}
Terminals: {a, b}

→ Useful Symbol: {a, A, B, b, S}

C is not useful, we can remove it. Delete wherever C is.

$S \rightarrow BaB$
$A \rightarrow aA/aaa$
$B \rightarrow bBb/a$
} If you observe, from 'S' we can't reach A. So we can remove 'A'.

∴ Result: $S \rightarrow BaB$
$B \rightarrow bBb/a$

You can represent a grammar in many ways.
Two popular are →

1) CNF: Chomsky Normal Form.

2) GTNF: Griebach Normal Form.

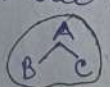
CNF:

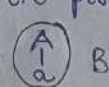
If all the productions are of form $A \rightarrow BC$ or $A \rightarrow a$.
where A, B, C are variables and 'a' is a terminal.

Advantages:

(i) Length of each production is restricted, so we know exactly how long the production is.

ii) Derivation tree or parse tree of CNF is always Binary.





Binary, either 1 or 2.

iii) The number of steps required to derive a string of length $|w|$ is $(2|w|-1)$.

iv) It is easy to apply CYK membership algorithm.

LECTURE: 5

CNF: $A \longrightarrow BC \longrightarrow$ Either two vertices
$A \longrightarrow a \longrightarrow$ One terminal
(OR) } Structures of CNF

In order to derive a string $|w|$ how many steps will be there?

→ for example $|abab| = 4 \rightarrow$ String length.

So, steps required to derive the string = $\boxed{2|w|-1}$

Steps $= 2 \times 4 - 1$
$= 7$ (for $|abab|$)

Example:

$S \longrightarrow AB$
$A \longrightarrow a$
$B \longrightarrow b$

$S \xrightarrow{\text{Step 1}} AB \xrightarrow{\text{Step 2}} aB \xrightarrow{\text{Step 3}} ab$

Replaced with 'a' (Step 2)
Replaced with 'b' (Step 3)

ab is generated.

$\therefore |ab| = 2$

So, steps required $= 2 \times 2 - 1$
$= 3$

CNF is suited for CYK algorithm.

Cocke, Younger, Kasami

CYK is membership algorithm.

Membership:

Given a string 'w', whether it belongs to L(G) or not.

$w \overset{?}{\in} L(G)$ → It means, whether a string can be generated by a given Grammar or not.

We can also write → $S \overset{*}{\Rightarrow} ab$. This star means many steps.

Given Grammar (G)

$S \longrightarrow AB/BC$
$A \longrightarrow BC/a$
$B \longrightarrow AC/b$
$C \longrightarrow AB/c$

} This is CNF.

Now if we are asked whether abab $\in L(G)$ or not, we can do it because it's decidable, we have CYK algo for this. This is fast.

Time Complexity of CYK Algo: $O(n^3)$

Convert the following grammar into CNF:

$S \longrightarrow bA/aB$
$A \longrightarrow bAA/aS/a$
$B \longrightarrow aBB/bS/b$

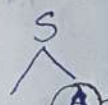
} This is a standard text book grammar. IMPORTANT.

The above grammar generates equal no. of a's and b's.

Step by step observation:

We have to generate equal no. of a's & b's.

1) $S \longrightarrow bA$
↓
Here S is generating a 'b'. So, we need one 'a' here.



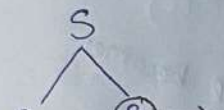
b (A) → The purpose of A is to generate strings with one extra 'a'. So that we get equal no. of a's & b's.

A → 'A' is supposed to generate one extra 'a'.

ii) S → aB

↓

Here S is generating one 'a'. So, B should be generating one extra 'b' to match the no. of a's & b's.



a (B) → The purpose of 'B' is to generate strings with one extra 'b', so that we get equal no. of a's & b's.

B → 'B' is supposed to generate an extra 'b'.

iii)

A → (bAA)/aS/a

↓ Explanation of this part.

'A' is supposed to generate an extra 'a', but here it has generated a 'b', for that we need two A's. One 'a' which was already needed & one extra for the 'b'.

A → bAA/(aS)/a

↓ Explanation of this part.

'A' is generating one extra 'a' and then equal no. of a's & b's which is why S is there.

'S' means equal no. of a's & b's.

A → bAA/aS/(a)

↓ Explanation of this part.

Here 'a' means simply one extra 'a'.

A → (bAA)/(aS)/(a) → In simple words, we can get an extra 'a' using any of the three.

iv) $B \rightarrow aBB/bS/b$

↓

You can apply the same logic as step(iii) here. In place of 'a' here it'll be 'b'.

Next Step of Converting the Grammar to CNF:

We'll introduce two new variables for a & b respectively →

N_a, N_b

$S \rightarrow N_bA/N_aB$

$N_b \rightarrow b$

$N_a \rightarrow a$

$A \rightarrow N_bAA/N_aS/a$ → Don't write N_a here as it'll become unit production.

$B \rightarrow N_aBB/N_bS/b$ → Don't use N_b here as it'll become unit production.

Everything is fine except these two parts.

For AA & BB we'll again introduce two new variables X & Y respectively.

$AA \rightarrow X$

$BB \rightarrow Y$

$S \rightarrow N_bA/N_aB$

$N_b \rightarrow b$

$N_a \rightarrow a$

$A \rightarrow N_bX/N_aS/a$

$B \rightarrow N_aY/N_bS/b$

Now the Grammar is in CNF.

So, we can apply CYK Algorithm.

CYK Algorithm:

Grammar (G):

$S \rightarrow AB/BC$

$A \rightarrow BA/a$

$B \rightarrow CC/b$

$C \rightarrow AB/a$

We can use the derivation tree to answer it.

Q: baaba $\in$ L(G)?

Q:1) What are all the "Substrings of baaba which belong to L(G)?

↓

For this Q: we need the algo.

→ b a a b a

1 2 3 4 5

	5	4	3	2	1
1	XV	XIII	X	VI	i
2	XIV	XI	VII	ii	
3	XII	VIII	iii		
4	IX	iv			
5	V				

From (i) to (xv) you have to fill in the given Roman number sequence.

	5	4	3	2	1
1	S, A, C	ϕ	ϕ	A, S	B
2	S, C, A	B	B	A, C	
3	B	S, C	A, C		
4	A, S	B			
5	A, C				

How many cells are there?

→ $1+2+3+4+\cdots+\boxed{n}$ cells

↓

n = length of the string

$\therefore \frac{n(n+1)}{2} = O(n^2)$

Time taken for each cell $= (n-1)$

$= O(n)$

∴ Total time complexity $= O(n^2)\,O(n)$

$= O(n^3)$

b a a b a
1 2 3 4 5

i) Cell no. 11

11 → (i) = b

In order to generate 'b' we have to check which productions are generating 'b'.

B → CC/b

So, we are going to write B in the box of (i)

ii) 22 → (ii) = a Cell no. 22

22 means → starting at 2 & ending at 2.

b a a b a
1 2 3 4 5

In order to generate 'a', we have to see the productions which are generating 'a'.

A → BA/a

C → AB/a

So, we are going to write A, C in the box of (ii)

iii) b a a b a
1 2 3 4 5 Cell no. 33

33 → a (iii) = a

We already know the productions which generates 'a' from point no. (ii)

iv) b a a b a
1 2 3 4 5 Cell no. 44

44 → (iv) = b

We already know the production which generates 'b' from point (i)

v) b a a b (a)
1 2 3 4 5

Cell no. 55

55 → (V) = a

We already know which productions are generating 'a' from point (ii)

Now ~~bre~~ breaking of cells will start, be careful

vi) Cell no. (VI):

Row 1, Col 2 = 12 → We can write it like ⇒ 11 22

{B} {A, C}

We have to concat them

{B} . {A, C}

= BA, BC

Now we have to check which productions in the Grammar are generating BA & BC

NOTE:
AB ≠ BA
BC ≠ CB

S → AB / BC
A → BA / a

So, in cell no. (VI) we are going to write A, S.

vii) Cell no. (VII):

Row 2, Col 3 → 23 → We can write it like ⇒ 22 33

{A, C} {A, C}

{A,C} . {A, C}

= AA, AC, CA, CC → Present in the Grammar

× × × ✓

We don't have these in the Grammar

B → CC / b

So, we write 'B' in cell no. (VII)

VIII) Cell no. (VIII):

Row 3, Col 4 = 34 → We can write it like ⇒ 33 44 → {A, C} {B}

{A,C} · {B}

= AB, CB → This is not present in the Grammar
✓ ×

This is present in the Grammar.

S ⟶ AB/BC

C ⟶ AB/a

So, we are going to write S, C in cell no. (VIII)

IX) Cell no. (IX):

Row 4, Col 5 = 45 → We can write it like ⇒ 44 55 → {B} {A, C}

{B} · {A,C}

= BA, BC

Both are present in the Grammar.

S ⟶ AB/BC

A ⟶ BA/a

So, we will write S, A in cell no. (IX).

X) Cell no. (X):

Row 1, Col 3 = 13 → We can write it like ⇒ 1 | 2 | 3

11 | 23 12 | 33

11 | 23 → B B

12 | 33 → {A, C}
↓
{A, S}

{B} · {B}

= BB → Not present in the Grammar.
X

{A, S} · {A, C}

= AA, AC, SA, SC
X X X X

None of them are there in the Grammar.

So, we will write 'ϕ' in cell no. (X)

x1) Cell no. (X1):

Row 2, Col 4 = 24 → We can write ⇒ 2 | 3 | 4

22 | 34 → {A, C} {S, C}

23 | 44 → {B} {B}

{A, C} · {S, C}

= AS, AC, CS, CC ✓ Present in the Grammar
X X X

None of them are there in the Grammar

So, we write "B" in cell no. (X1).

{B} · {B}

= BB → Not present in the Grammar.
X

B → CC/b

x11) Cell no. (X11):

Row 3, Col 5 = 35 → We can write ⇒ 3 | 4 | 5

33 | 45 → {A, C} {A, S}

34 | 55 → {S, C} {A, C}

{A, C} · {A, S}

= AA, AS, CA, CS
X X X X

None of them are there in the Grammar.

$\{S,C\} \cdot \{A,C\}$

$= SA, SC, CA, \boxed{CC}$ → Present in the Grammar.
$\quad\ \times\ \times\ \times\ \checkmark$

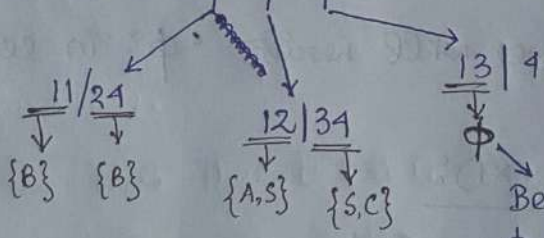
$B \longrightarrow CC/b$

So, we will write 'B' in cell no. (XII).

~~XIII~~ XIII) Cell no. (~~XIII~~):

Row 1, Col 4 = 14

↳ We can write ⇒ 1 | 2 | 3 | 4



Because of Φ, it'll be Φ for the 13/44 part.

$\{B\} \cdot \{B\}$

$= BB$
$\quad\ \times$

$\{A,S\} \cdot \{S,C\}$

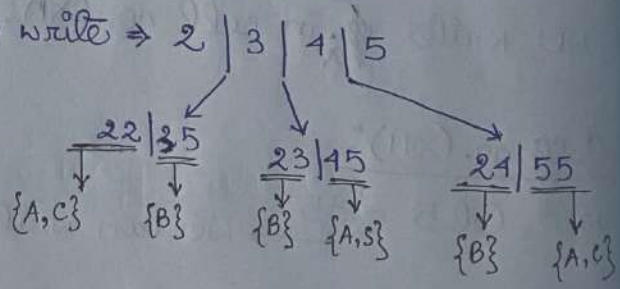
$= AS, AC, SS, SC$
$\quad\ \times\ \times\ \times\ \times$

None of them are there in the Grammar.
So, we will write Φ in cell no. (~~XIV~~), (XIII).

XIV) Cell no. (XIV):

Row 2, Col 5 = 25

↳ We can write ⇒ 2 | 3 | 4 | 5



$\{AC\} \cdot \{B\}$

$= AB, CB$
$\quad\ \checkmark\ \ \times$

$S \longrightarrow AB/BC$

$C \longrightarrow AB/a$

$\{B\} \cdot \{A, S\}$

$= BA, BS$

✓ ✗

$A \rightarrow BA/a$

$\{B\} \cdot \{A, C\}$

$= BA, BC$

✓ ✓

$S \rightarrow AB/BC$

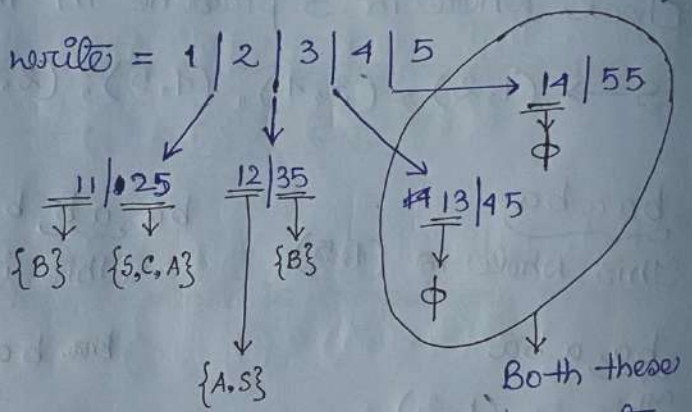
$A \rightarrow BA/a$

So, we will write S, C, A in cell no. (XIV).

xv) Cell no. (XV):

Row 1, Col 5 = 15

↳ We can write = 1 | 2 | 3 | 4 | 5



$\{B\} \cdot \{S, C, A\}$

$= BS, BC, BA$

✗ ✓ ✓

$S \rightarrow AB/BC$

$A \rightarrow BA/a$

$\{A, S\} \cdot \{B\}$

$= AB, SB$

✓ ✗

$S \rightarrow AB/BC$

$C \rightarrow AB/a$

So, we will write S, A, C in cell no. (XV).

NOTE: ***

If 'S' is present in Row 1, Col 5 cell which means cell no. (XV), then $w \in L(G)$.

Possible GATE Questions from CYK Algorithm:

1) Does $w \overset{?}{\in} L(G)$.

Solution:

Check whether 'S' is present in the final cell.

[See the last NOTE of previous page for explanation].

2) Fill in the blank in the table.

3) What are the substrings of 'w' which are in the language?

→ Answer from last question.

Check where is S present in the table.

$S \rightarrow (1,2), (3,1), (4,5), (2,5), (1,5)$

baaba
This whole is (1,5)

ba ab ba
This is (3,1)

baaba
This is (1,2)

baaba
This is (4,5)

baaba
This is (2,5)

So, the substrings are → baaba, ba, aaba, ab

Q: $S \rightarrow AB$

$A \rightarrow BB/a$

$B \rightarrow Ab/b$

String = aabbb.

→

	5	4	3	2	1
1	S, B	A	S, B	Φ	A
2	S, B	A	S, B	A	
3	S, B	A	B		
4	A	B			
5	B				

Total cells $= \frac{n(n+1)}{2}$

$= \frac{5(5+1)}{2}$

$= \frac{30}{2}$

$= 15$

Time complexity of CYK Algo $= O(n^3)$

GNF: (Griebach Normal Form)

→ If all the productions are of the form $A \rightarrow a\alpha$, where $\alpha \in V^*$ then it is called GNF.

Examples:

$A \rightarrow aABCD$ OR $B \rightarrow a$ } One Terminal is compulsory in GNF

↓

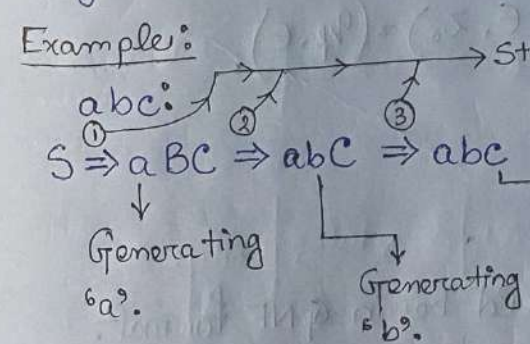
Production should be like this.

Advantages:

→ Number of steps required to generate string of length $|w|$ is $|w|$.

Examples:

abc:



If string length is $|w|$ then steps required $= |w|$

In this example $|W| = |abc| = 3$.

∴ Steps = 3.

→ Most Important Advantage:

GNF is ~~used~~ useful to convert a CFG to PDA.

Algo to convert any grammar to GNF.

Algo to convert GNF to PDA.

∴ Any Grammar can be converted to PDA.

So, it's Decidable.

LECTURE: 6

Conversion of CFG to PDA: (Algo)

→ the CFG should be in the form of GNF in order to use the algo.

1) Push that start symbol 'A' on the stack.

$$\delta(q_0, \epsilon, Z_0) = (q_1, AZ_0)$$

2) Push RHS of A, as follows →

$$\delta(q_1, a, A) = (q_1, \alpha)$$

if $A \to a\alpha$ is in 'G'.

$$\delta(q_1, b, A) = (q_1, \beta)$$

if $A \to b\beta$ is in 'G'.

3) Add final state with $\delta(q_1, \epsilon, Z_0) = (q_f, \epsilon)$.

NOTE:

CFG to PDA is decidable.

↓

Always remember: The CFG should be in GNF format.

Convert following Grammar to PDA:

$S \rightarrow aSb/ab$

$B \rightarrow b$

This Grammar is not in GNF. Because in GNF it should be like: $S \rightarrow aABC/a$

⇓

We'll convert it into GNF.

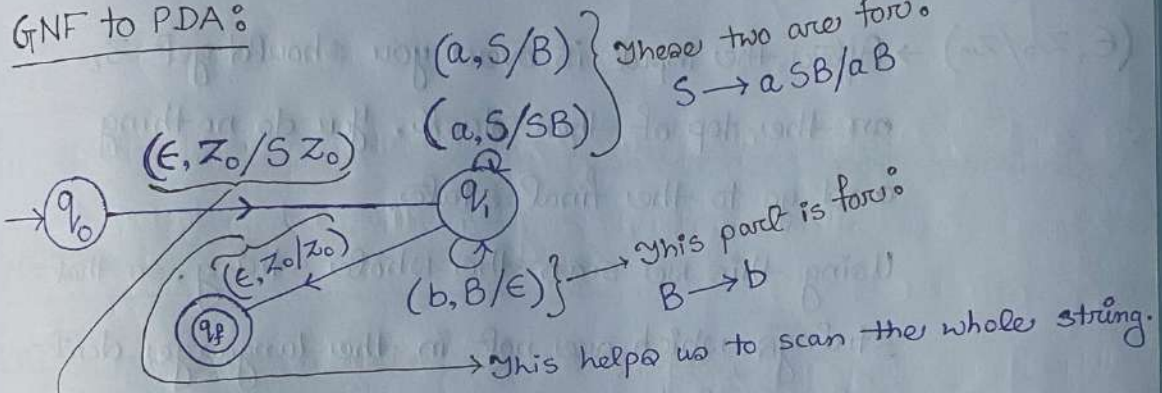
$S \rightarrow aSB/aB$

$B \rightarrow b$

→ This is now in GNF

→ Here we replaced 'b' with 'B', because $B \rightarrow b$ already.

GNF to PDA:



Meaning of this move:

Without seeing any input, with Z_0 on the top of the stack, push the start symbol 'S', of GNF.

Explanation of q_1 state:

$S \rightarrow \underset{\text{(i)}}{aSB} / \underset{\text{(ii)}}{aB}$

(i) If 'a' is in the input & top of the stack is 'S'. Then you are going to push SB in place of S. After following the steps, we are getting: $a, S/SB$

(ii) If 'a' is in the input & 'S' is in the top of the Stack. Then you replace it with right hand side, in this case which is 'B'. So, we are getting $a, S/B$

$S \rightarrow a$ [B] → Right hand side.

$B \longrightarrow b$

→ Whenever you see this type of production, you simply add 'ϵ' on the RHS to make it easy for calculation.

Which means →

$B \longrightarrow b\epsilon$

When small b is in the input & 'B' is on the top of the stack, then replace 'B' by 'ϵ'.

So, we are getting: $\boxed{b, B/\epsilon}$

$(\epsilon, Z_0/Z_0)$ → After the input is over you should get Z_0 on the top of the stack. You do nothing and go to the final state.

Using this we scan the whole string, so that strings, which are not in the language don't get accepted.

For example, the string = abab

So, ϵ abab ϵ

↑ Initial move. From q_0 to q_1

↑ Final move. From q_1 to q_f

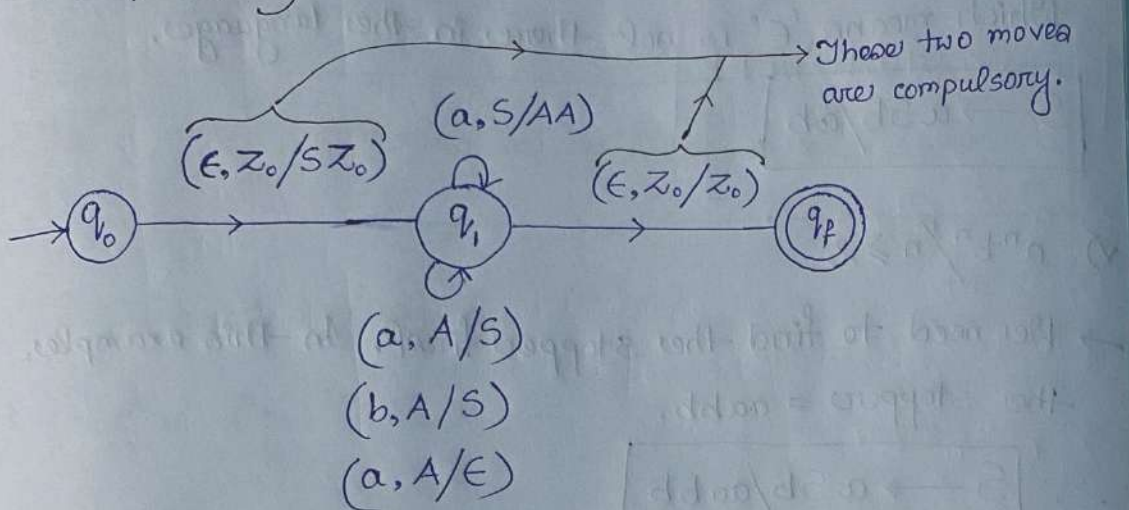
NOTE:

GNF to PDA it'll always have $\boxed{3 \text{ states}}$ only.

Q:

$S \rightarrow aAA$
$A \rightarrow aS/bS/a$ } GNF

Make the PDA.



NOTE:

In GNF to PDA, we always push vertices, not terminals.

Not needed: Just for knowledge>

Why we push vertices, not terminals?

→ Because we are trying to build a derivation tree by mimicing with PDA.

Context Free Grammars: (Basic to advance)

i) $L = \{a^n b^n / n \geq 0\}$

→ $S \rightarrow aSb/\epsilon$

ii) $b^n a^n / n \geq 0$

→ $S \rightarrow bSa/\epsilon$

iii) $a^m b^m c^n d^n / m, n \geq 0$

→ $a^m b^m$ $c^n d^n$

$a^m b^m$ ↓ First we'll generate this

$c^n d^n$ → Then we'll generate this.

$S \rightarrow AB$

$A \rightarrow aAb/\epsilon$

$B \rightarrow cBd/\epsilon$ } Putting them side by side in 'S'

iv) $a^n b^n / n \geq 1$.

→ So, here the language is : ab, aabb, aaabbb.....

Which means, 'ϵ' is not there in the languages.

$$\boxed{S \longrightarrow aSb/ab}$$

v) $a^n b^n / n \geq 2$

→ We need to find the stopper first. In this example, the stopper = aabb.

$$\therefore \boxed{S \longrightarrow aSb/aabb}$$

vi) $a^n b^{2n} / n \geq 0$ or $a^m b^n / n = 2m$

→ Here if we observe, for every 'a' there has to be two 'b's or 'ϵ'.

$$\therefore \boxed{S \longrightarrow aSbb/\epsilon}$$

vii) GATE:

$S \longrightarrow aSb/abb$. $L(G) = ?$

→ $S \Rightarrow a^n S b^n$

$\Downarrow$

abb

$$\therefore \boxed{L(G) = a^n abbb^n} \Rightarrow \boxed{a^{n+1} b^{n+2} / n \geq 0}$$

Why $n \geq 0$, why not $n \geq 1$?

→ Here the smallest string = abb.

If we put $n \geq 1$, then the smallest string we are going to get = $a^2 b^3$ = aabbb, which is not matching with the given smallest string of the Language.

If we use $n \geq 0$, then we are going to get: $a^1 b^2 = abb$, which matches the smallest string of the given Language.

$a^{n+1} b^{n+2} / n \geq 0$ can also be written as $a^m b^n / n = m+1$

VIII) $S \longrightarrow aSbb/aab$; $L(G_T) = ?$

→ $S \Rightarrow a^n S b^{2n}$

$\Downarrow$

aab

∴ We are getting: $a^n aab\, b^{2n} = a^{n+2} b^{2n+1} / n \geq 0$ → Ans.

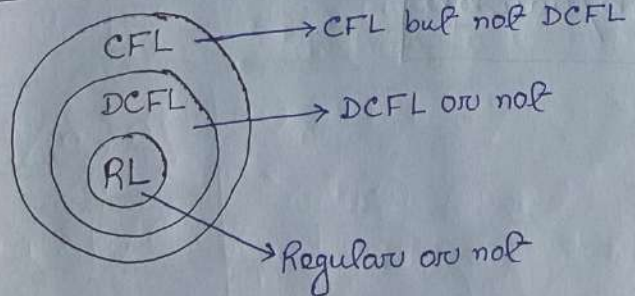
IX) $a^{2n} b^n / n \geq 0$

OR

$a^m b^n / m = 2n$

→ $S \longrightarrow aaSb/\epsilon$

GATE Questions we'll be facing:



New Model:

1) $a^n b^{2n+3} / n \geq 0$ OR $a^m b^n / n = 2m+3$

→ $a^n b^3 b^{2n}$ → Stoppers

→ First generate this part.

$S \longrightarrow aSbb/bbb$

2) $a^n b^{3n+4} / n \geq 0$

$\rightarrow a^n b^4 b^{3n}$

$S \longrightarrow aSbbb / bbbb$

3) $a^{3n+2} b^n / n \geq 0$

OR

$a^m b^n / m = 3n+2$

$\longrightarrow a^{3n} a^2 b^n$

$S \longrightarrow aaaSb / aa$

4) $a^{2n} b^{3n} / n \geq 0$

$\rightarrow S \longrightarrow aaSbbb / \epsilon$

5) $a^{2n+3} b^{3n+2} / n \geq 0$

$\rightarrow a^{2n} a^3 b^2 b^{3n}$

$S \longrightarrow aaSbbb / aaabb$

Next Model:

$\neq$

$>$

$\geq$

$<$

$\leq$

1) $a^m b^n / m \geq n \; ; m, n \geq 0$

→ i) First generate equal no. of a's and b's.

$A \rightarrow aAb/\epsilon$

ii) Now add extra a's in the beginning

$B \rightarrow aA/\epsilon$

∴ We get:

$S \rightarrow BA$
$A \rightarrow aAb/\epsilon$
$B \rightarrow aA/\epsilon$

$S \rightarrow [B][A] \rightarrow a^n b^n$

B → Extra 'a's $[a^*]$

NOTE:
In the above question, it's $m \geq n$ which means, a & b can be of same length.
For this we are getting help from: $B \rightarrow aA/(\epsilon)$ → Because it can generate '0' a's

So, $S \rightarrow [B][A] \rightarrow$ Generates $a^n b^n$

B → Here if B generates '0' a's, we'll get equal no. of a & b's from A

2) $a^m b^n / m > n$

→ $m > n$, so we'll do $a^+ a^n b^n$

$S \rightarrow AS_1$
$S_1 \rightarrow aS_1 b/\epsilon$
$A \rightarrow aA/a$

3) $a^m b^n / m \leq n$

→ $a^n b^n b^*$

$S \rightarrow S_1 B$
$S_1 \rightarrow a S_1 b / \epsilon$
$B \rightarrow bA \; bB / \epsilon$

4) $a^m b^n / m < n$

→ $a^n b^n b^+$

$S \rightarrow S_1 B$
$S_1 \rightarrow a S_1 b / \epsilon$
$B \rightarrow bB / b$

5) $a^m b^n / m \neq n$

→ $m \neq n$ means $m > n$ or $m < n$

This is union of two languages.

$\{a^+ a^n b^n\}$ or $\{a^n b^n b^+\}$

Now we'll do union of the above two languages.

First we'll generate 'a^+' and 'b^+'.

$S \rightarrow AS_1 / S_1 B$
$S_1 \rightarrow aS_1 b / \epsilon$ → Generates $a^n b^n$
$A \rightarrow aA / a$ → Generates a^+
$B \rightarrow bB / b$ → Generates b^+

6) What is the language of following Grammar?

$S \longrightarrow AS_1 / S_1B$

$S_1 \longrightarrow aS_1b / \epsilon \longrightarrow a^n b^n$ ∴ This language is Regular.

$A \longrightarrow aA / \epsilon \longrightarrow a^*$

$B \longrightarrow bB / b \longrightarrow b^+$

$AS_1 = a^* a^n b^n$

$S_1B = a^n b^n b^+$

$\therefore \underbrace{(a^* a^n b^n)}_{\text{Here we can say} \rightarrow m \geq n} \cup \underbrace{(a^n b^n b^+)}_{\text{Here we can say } m < n}$

So, all combinations are covered, which means it's $a^* b^*$

$m = n$
$m < n$ → All covered
$m > n$

7) $a^m b^n / m \geq 2n$

→ i) First generate $a^{2n} b^n$.

$S_1 \longrightarrow aaS_1b / \epsilon$

ii) Now generate extra 'a's.

$A \longrightarrow aA / \epsilon$

iii) Now put them together.

$S \longrightarrow AS_1$

Final Answer:

$S \longrightarrow AS_1$
$A \longrightarrow aA / \epsilon$
$S_1 \longrightarrow aaS_1b / \epsilon$

8) $a^m b^n / m \geq 2n+3$

→ $a^{2n+3} b^n = a^{2n} a^3 b^n$

$S \rightarrow AS_1$
$S_1 \rightarrow aaS_1b / aaa$
$A \rightarrow aA / \epsilon$

9) $a^m b^n / m \leq 2n+3$

→ Just remember the language is CFL.
Not asked in Exam.

10) $a^n b^n \cup a^n b^{2n}$

OR

$a^m b^n / n = m$ or $2m$

→

$S \rightarrow S_1 / S_2$
$S_1 \rightarrow aS_1b / \epsilon$
$S_2 \rightarrow aS_2bb / \epsilon$

11) $a^m b^n / m \leq n \leq 2m$ *****

→ For example: Put $m=2$

$2 \leq n \leq 4$

Which means 'n' can be → 2, 3, 4

aabb, aabbb, aabbbb

Strings can be these.

$S \rightarrow aSb / aSbb / \epsilon$ ***

For every 'a' you are generating a 'b'.

For every 'a', you are generating two 'b's.

∴ Combining them, now we can generate valid strings which belong to the language.

Next Model:

1) $\Sigma = \{a, b, c\}$

$a^m b^m c^n / m, n \geq 0$

OR

$a^m b^n c^p / m = n$

→

$S \rightarrow S_1 C .$
$S_1 \rightarrow a S_1 b / \epsilon .$
$C \rightarrow cC / \epsilon .$

2) $a^m b^n c^n / n, m \geq 0$

→

$S \rightarrow AS_1$
$S_1 \rightarrow bS_1 c / \epsilon$
$A \rightarrow aA / \epsilon$

3) $a^n b^m c^n / m, n \geq 0$

OR

$a^m b^n c^p / m = p$

→ $a^n b^m c^n$

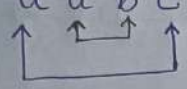
Generate a's & c's together.

$S \rightarrow aSc / B$
$B \rightarrow bB / \epsilon$

The language = $a^n b^* c^n$

4) $a^m b^n c^p / m = n+p \; ; \; m,n,p \geq 0$

→ $a^p a^n b^n c^p$

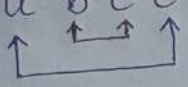


$S \longrightarrow aSc / A$

$A \longrightarrow aAb / \epsilon$

5) $a^m b^n c^p / p = m+n \; ; \; m,n,p \geq 0$

→ $a^m b^n c^n c^m$



$S \longrightarrow aSc / A$

$A \longrightarrow bAc / \epsilon$

6) $a^m b^n c^p / n = m+p \; ; \; m,n,p \geq 0$

→ $a^m b^m b^p c^p$

$S \longrightarrow S_1 A$

$S_1 \longrightarrow a S_1 b / \epsilon$

$A \longrightarrow bAc / \epsilon$

LECTURE : 7

7) $\{a^m b^n c^p / m = n \text{ or } m = p\}$

OR

$\{a^m b^n c^p / m = n\} \cup \{a^m b^n c^p / m = p\}$

→ $a^m b^n c^p / m = n$

$= a^n b^n c^p$

$X \rightarrow S_1 C$
$S_1 \rightarrow a S_1 b / \epsilon$
$C \rightarrow cC / \epsilon$

$a^m b^n c^p / m=p$

$= a^p b^n c^p$

Generate these two together & put "b" in the middle.

$Y \rightarrow a Yc / B$
$B \rightarrow bB / \epsilon$

Final Result:

$S \rightarrow X / Y$
$X \rightarrow S_1 C$
$S_1 \rightarrow a S_1 b / \epsilon$
$C \rightarrow cC / \epsilon$
$Y \rightarrow a Yc / B$
$B \rightarrow bB / \epsilon$

8) $a^m b^n c^p d^q / m=n$ and $p=q$.

$\rightarrow$

$S \rightarrow XY$
$X \rightarrow aXb / \epsilon$
$Y \rightarrow cYd / \epsilon$

$a^m b^n c^p d^q / m=n$ & $p=q$

$= [a^n b^n] [c^q d^q]$

'X' is generating this part

'Y' is generating this part.

Next Model:

1) $a^m b^n c^p d^q$ / ~~$m=n=p=q$~~ $m+n=p+q$, $m+q=n+p$, $m+p=n+q$ } Difficult to write CFG but these are CFLs.

→ i) $m+n=p+q$: CFL

$a^m b^n c^p d^q$

Push this part → Start popping from this part.

ii) $m+q=n+p$: CFL

$a^m b^n c^p d^q$

Here $m+q$ should be equal to $n+p$.

So, first ~~all~~ push all a's, then you push all b's. Then for every 'c' you pop a 'b'. Then for every 'd' you pop one 'a'.

iii) $m+p=n+q$: CFL

$m+p=n+q$

$\Rightarrow m-n+p-q=0$

IMPORTANT

$\{a^m b^n c^m d^n / m,n \geq 0\}$ → This is not CFL

$\{a^m b^n c^p d^q / m+p=n+q\}$ → This is CFL, because PDA is possible for this one.

NOTE:

Alternate comparison is not possible but alternate addition like → $\boxed{m+n=p+q}$ is possible in CFL.

Next Model:

1) $W / n_a(w) = n_b(w)$

→ This is not $a^n b^n$.

$S \rightarrow bSa/\epsilon$ → Here we can't generate "ab".

$S \rightarrow aSb/bSa/\epsilon$ → Here we can't generate "abba".

∴ Final Result:

$S \rightarrow aSb/bSa/SS/\epsilon$

⇓

There are two more ways to write it

i) $S \rightarrow aSbS/bSaS/\epsilon$

ii) $S \rightarrow SaSb/SbSa/\epsilon$

2) Which of the following are equivalent?

i) $S \rightarrow aSb/\epsilon$

ii) $S \rightarrow aSb/ab/\epsilon$

iii) $S \rightarrow aSb/ab$

→ (i) & (ii) are same. The language is $a^n b^n / n \geq 0$

(iii) the language is $a^n b^n / n \geq 1$.

3) $W / n_a(w) = 2 n_b(w)$

→ Trick for these types of questions:

i) First find out smallest strings.

In this example, the smallest strings are

→ aab, aba, baa

ii) $S \rightarrow aab/aba/baa$

Now put 'S' in the middle.

So, we get:

$S \rightarrow aSaSb/aSbSa/bSaS/SS/\epsilon$

If you want to remove this, you can write it in two more ways.

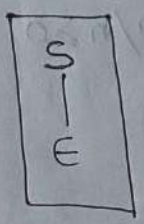
a) $S \rightarrow aSaSbS/aSbSaS/bSaSaS/\epsilon$

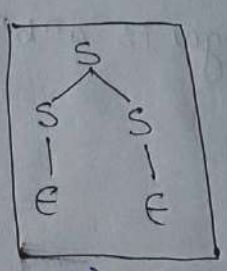
b) $S \rightarrow SaSaSb/SaSbSa/SbSaSa/\epsilon$

NOTE:

In any Grammar whenever you see 'SS/ϵ', which means, $S \rightarrow SS/\epsilon$. The Grammar is ambiguous. Because here, for the same string there are two derivation trees.

i)





ii)

A) $W / n_a(w) = 3 n_b(w)$.

→ Smallest strings are:

aaab, abaaa, baaa, aaba

Final Result:

$S \rightarrow aSaSaSb / aSbSaSa / bSaSaSa / aSaSbSa / SS / \epsilon$

Perfectly balanced paranthesis or valid paranthesis:

'(' ')' → First brackets.

> $W / n_($ (w) = $n_)$ (w), & v is any prefix of w, then
> $n_($ (w) ≥ $n_)$ (w)

It means number of open braces should always be greater or equal to no. of close braces.

Example & explanation:

For example: Check if '((()))' is perfectly balanced.

Take any prefix & check if no. of open braces are greater than or equal to no. of close braces.

Prefix: Prefix means starting part.

((()))

Prefix:

no. of open braces = 1

no. of close " = 0

Open braces > close braces here ✓

((()))

Prefix:

Open braces = 2

Close " = 0

∴ No. of open braces > no. of close braces. ✓

Open Braces = OB
Close Braces = CB
} Taken for simplicity

(((())))	(((()))	((()()))
Prefix:	Prefix:	Prefix:
OB = 3	OB = 3	OB = 3
CB = 0	CB = 1	CB = 2
∴ OB > CB ✓	∴ OB > CB ✓	∴ OB > CB ✓

((()))

Prefix:
OB = 3
CB = 3
∴ OB = CB ✓

∴ The string is perfectly balanced. — Ans:

Grammar for Balanced Paranthesis:

$S \rightarrow (S)/SS/\epsilon$

Next Model: String Matching >

i) Even Length Palindrome: $\{WW^R / W \in (a,b)^*$

ii) Odd Length Palindrome: $\{WxW^R / W \in (a,b)^*, x \in \{a,b\}\}$

iii) All Palindromes: $\{WW^R \cup WxW^R\}$ or $\{WxW^R / W \in \{a,b\}^*, x \in \{\epsilon, a, b\}$

iv) Hash Palindrome: $\{W\#W^R / W \in \{a,b\}^* \ \# \in \{a,b\}\}$

i) WW^R : Even Length Palindromes

Assume, $W = ab$ | $WW^R = abba$

$\therefore W^R = ba$ |

Grammar:

$S \longrightarrow aSa / bSb / \epsilon$

ii) Odd Length Palindromes

$WaW^R \cup WbW^R$

OR

WxW^R / $W \in (a,b)^*$, $x \in \{a,b\}$.

Grammar:

$S \longrightarrow aSa / bSb / a / b$

NOTE:
Don't use epsilon, otherwise it'll become even length.

iii) All Palindromes

All palindromes mean, union of even length palindromes & odd length palindromes.

We can write it in TWO ways:

i) $\{ WW^R \cup WxW^R / W \in (a,b)^*, x \in (a,b) \}$

(WW^R: Even Length Palindromes; WxW^R: Odd Length Palindromes)

ii) $\{ WxW^R / W \in (a,b)^*, x \in (\epsilon, a, b) \}$

For the two ways, we can write two grammars which fits the individual structures:

Grammars for previous page's (i) no. point >

$S \rightarrow S_1 / S_2$

$S_1 \rightarrow$ even palindrome

$S_2 \rightarrow$ odd palindrome

$\therefore$

$S \rightarrow S_1 / S_2$

$S_1 \rightarrow aSa/bSb/\epsilon$

$S_2 \rightarrow aSa/bSb/a/b$

Structures following:

$\{ WW^R \cup WxW^R / W \in (a,b)^*, x \in (a,b) \}$

Grammars for previous page's (ii) point >

$S \rightarrow aSa/bSb/a/b/\epsilon$

Structures following:

$\{ WxW^R / W \in (a,b)^*, x \in (a,b,\epsilon) \}$

iv) Hash Palindrome: $W \# W^R$ >

$W \# W^R$

↓

Special symbol, single element.

Assume: $W = ab$

$\therefore W \# W^R = ab \# ba$

Grammar:

$S \rightarrow aSa/bSb/\#$

New Model based on →

a) Given a language is it a DCFL?

b) Given a language is it a CFL but not DCFL.

To answer the above mentioned questions, first we have to understand power & limitations of DPDA.

Limitations of DPDA:

1) If, when to pop or when to push is not clear → then DPDA fails. Which means it's not DCFL.

Ex: Palindrome → i) Even length
ii) Odd length
iii) All palindromes

} These are not DCFL because we don't know about the center. So, we don't know when to push or when to pop.

iv) Hash Palindrome: DCFL

Because '#' is a special single element, we can easily get the centre using #. So, we know when to push or when to pop.
Which means it's DCFL.

2) When how much to push is not clear → Then it's not DCFL.

Example & Explanation: Example (i)

$\{(a^n b^n) \cup (a^n b^{2n}) / n \geq 0\}$

Here PDA is possible, but not DPDA.

Because if we follow → $a^n b^n$, then for every 'b' we have to push one 'a'.

But if we follow → $a^n b^{2n}$, then for every 'b' we have to push two a's.

DPDA can't do this, here we'll have to use NPDA.

Which means the language is CFL, but not DCFL.

Example (ii)

$W / n_b(w) = n_a(w)$ or $n_b(w) = 2 \times n_a(w)$

Here also, if you observe, we are pushing one 'a' for one 'b', and sometimes we are pushing two 'a's for one 'b'.

Which means the language is not DCFL. This is CFL.

3) Double Comparison with OR:

$a^m b^n c^p / m=n$ or $n=p$

Here we have to do two comparisons together which DPDA can't do. Which means the language is not DCFL, but it's CFL.

Some more questions:

i) $a^m b^n c^p / m=n$ & $m=p \rightarrow$ CSL.

ii) $a^m b^n / n \leq m \leq 2n \rightarrow$ CFL, but not DCFL.

iii) $W / n_a = n_b$ or $n_a = n_c \rightarrow$ CFL, but not DCFL.

LECTURE: 8

Model for $WW^R \rightarrow$ Palindromes:

$WW^R \rightarrow$ Most of the times they are non-regular, but sometimes they can be Regular.

1) $\{WW^R \cup (0,1)^* / W \in (0,1)^*\}$

→ This is Regular. The language is $(0,1)^*$.

Number of states in DFA = 1.

Since this is Regular, this is also DCFL, & CFL, CSL, REC, RE.

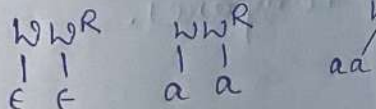
2) $WW^R / W \in (0,1)^*$

→ This not Regular, because string comparison is needed here, which Finite automata can't do.

This is CFL. But not DCFL, because we don't know when to push or when to pop.

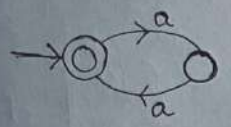
3) $WW^R / W \in \{a\}^*$

→



$\{\epsilon, \quad aa, \quad aaaa, \ldots\}$

∴ The language is $(aa)^*$, which is Regular.



→ Two states in min DFA.

4) $\{W(W^R)^* / W \in (a)^*\}$

→ $W(W^R)^*$ is in the form: $W.(W^R)^*$

$W.(W^R)^*$

$= W.\{\epsilon, W^R, (W^R)^2, (W^R)^3, \ldots\}$

$= W, WW^R, W(W^R)^2, \ldots$

↓

This is 'a^*'. Therefore the language is a^*, which is Regular.

5) $\{W\#W^R / W \in (a)^*\}$

→ This is not Regular because there can be any no of a's before & after '#'. Finite Automata can't do these types of counting.

We need DPDA here, which means the language is DCFL. So it's CFL also.

Example:

aaaa#aaaa

4 'a's 4 'a's → FA can't count, so it cannot compare.

6) $\{WW^R / W \in \{a,b\}^*, |W| \leq 10\}$

→ The language is finite, which is why it's Regular.

7) $\{WWW / W \in \{a,b\}\}$

→ The language is finite because $W \in (a,b)$, not $(a,b)^*$.
Therefore the language is Regular.

8) $\{W(W^R)^* / W \in (a,b)^*\}$

→ $W(W^R)^* = W(\in, W^R, (W^R)^2, (W^R)^3, \ldots\}$

$= W, WW^R, W(W^R)^2, \ldots\ldots$

$\downarrow$

$(a,b)^*$

Therefore the language is Regular.

9) $\{W^* W^R / W \in (a,b)\}$ $W \in (a,b)$.

→ The language is Regular because it's [redacted]

10) $\{W^* W^R / W \in (a,b)^*\}$

→ We can write $W^* W^R$

$\in$ (W^R)

$W \in (a,b)^*$

$\therefore W^R \in (a,b)^*$

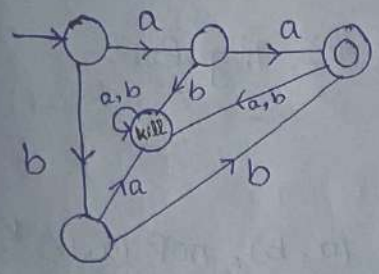
→ The reversal of universal language is universal itself.

Therefore the language is Regular.

11) $\{WW^R / W \in (a,b)\}$, how many states are there in minimal DFA?

→ W can be 'a' → aa

W " " 'b' → bb



Therefore 5 states for min DFA.
4 " " min NFA.

12) $\{W(W^R)^* / W \in (a,b)^+\}$

→ $W.(W^R)^* = W.\{\epsilon, W^R, (W^R)^2, (W^R)^3, \ldots\}$

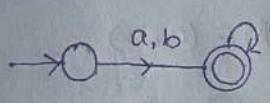
$= W, WW^R, WW^{R^2}, \ldots$

↓

$(a,b)^+$

∴ The language is $(a+b)^+$, which is Regular.

DFA:



2 states in DFA.

13) $\{WxW^R / x, W \in (a,b)^*\}$

→

$W \quad x \quad W^R$

$\epsilon \quad x \quad \epsilon = \boxed{x}$ → This can be $(a,b)^*$.

∴ The language is $(a,b)^*$.

NOTE:

If in a language, one element is $(a,b)^*$, then it'll eat up all the elements, so the language will become $(a,b)^*$.

14) $\{WxW^R / W \in (a,b)^*, x \in (a,b)^+\}$

→

W	x	W^R
$\in$	x	$\in$

$= \boxed{x}$ → In this example $x = (a,b)^+$

∴ The language = $(a+b)^+$.

∴ Language is Regular.

NOTE:

'x' is not the entire language. There can be many strings possible. But if 'x' becomes $(a,b)^+$ or $(a,b)^*$, we don't have to compare more strings.

The 'x' may be the language or may be a subset of the language.

15) $\{WxW^R / W \in (a,b)^*, x \in (a,b)\}$

→ The language is odd length palindromes. Which means, the language is CFL but not DCFL.

16) $\{WxW^R / W \in (a,b), x \in (a,b)^*\}$

→

W	x	W^R
a	x	a

axa

W	x	W^R
b	x	b

bxb

NOTE:

'W' can only be 'a' or 'b'.

If $W = a$ $\quad W = b$

$W^R = b$ $\quad W^R = b$

∴ The language is $= a(a+b)^*a + b(a+b)^*b$

Therefore the language is Regular.

17) $\{wxw^R / w \in (a,b), x \in (a,b)\}$

→

w	x	w^R
a	x	a

$\Rightarrow a(a+b)a$

$\Rightarrow aaa+aba$

w	x	w^R
b	x	b

$\Rightarrow b(a+b)b$

$= bab+bbb$

∴ The language = aaa+aba+bab+bbb, which is Regular.

18) $\{wxw^R / w \in (a,b)^+, x \in (a,b)^*\} = L_1$

→

w	x	w^R
a	x	a

$L_2 =$

NOTE:
We can't put 'ϵ' in 'w' because $w \in (a,b)^+$

w	x	w^R
b	x	b

$L_3 =$

$L_2 = a(a+b)^*a$

$L_3 = b(a+b)^*b$

∴ The language is: $a(a+b)^*a + b(a+b)^*b$. Which is Regular.

Which of the following is Regular?

1) $\{xww^R / x, w \in (a,b)^+\}$

2) $\{wxw^R / x, w \in (a,b)^+\}$

3) $\{ww^Rx / x, w \in (a,b)^+\}$

Answers with Explanations:

1)

Assume, 'ab' is a string.

∴ xWWR of 'ab' = $\boxed{(a+b)^+ abba}$

↓

This can't be generated by L_2 or L_3.

Therefore the Language is not Regular but CFL.

---

2)

⇒ $\underbrace{a\,(a+b)^+\,a}_{L_2}$ ⇒ $\underbrace{b(a+b)^+\,b}_{L_3}$

Assume, 'ab' is a string.

∴ WxWR of 'ab' = $a\boxed{b\,(a+b)^+\,b}a$

↓

This part can be generated by $(a+b)^+$.

Therefore L_2 can generate this.

Therefore the language is Regular.

NOTE:

By using simpler Regular expressions, if a part of the language is generating the entire language, then I can say that the language is Regular.

Which is why we are checking whether L_2 or L_3 can generate the assumed strings.

---

3)

W	W^R	x
a	a	$(a+b)^+$

L_2

W	W^R	x
b	b	$(a+b)^+$

L_3

Assume, 'ab' is a string.

∴ WW^Rx of 'ab' = $abba(a+b)^+$

↓

This can't be generated by L_2 or L_3.

Therefore the language is not Regular, but CFL.

NOTE:

Whenever you see 'WW^R', it's not DCFL, because you don't know, when to push or when to pop.

New Topic: Derivation >

The process of deriving a string from a Grammar is called Derivation.

$S \longrightarrow AB$
$A \longrightarrow a/\epsilon$
$B \longrightarrow b/\epsilon$

Assume, we want to derive 'ab'.

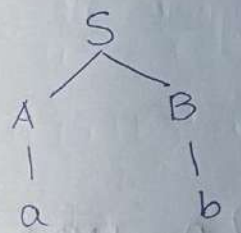
$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$ → Derivation

'A' will be substituted by 'a'.

'B' will be substituted by 'b'.

NOTE:
While doing derivation, only one variable should be expanded at a time.

Derivation Tree:



Here 'ab' is the yield of Derivation Tree.

Theory Point:

$S \longrightarrow AB$
$A \longrightarrow a/\epsilon$
$B \longrightarrow b/\epsilon$

LMD: ab
$S \Rightarrow AB$
$\Rightarrow aB$
$\Rightarrow ab$

RMD: ab
$S \Rightarrow AB$
$\Rightarrow Ab$
$\Rightarrow ab$

Left Most Derivation: In every derivation if there is a choice in variables substitution, and left most variable is sustituted, then it is called LMD.

Right Most Derivation: In every derivation if there is a choice in variables sustitution, and right most variable is substituted, then it is called RMD.

$S \rightarrow AB$

$A \rightarrow aA/\epsilon$

$B \rightarrow bB/\epsilon$

LMD:

$S \rightarrow AB \rightarrow aAB \rightarrow aaAB \rightarrow aaaB \rightarrow aaab$

RMD:

$S \rightarrow AB \rightarrow AbB \rightarrow Ab\epsilon \rightarrow b$

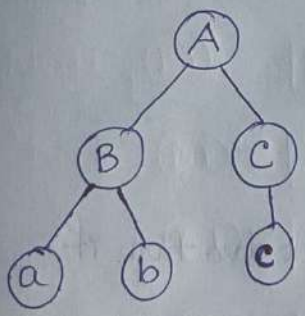
The another way which is not LMD or RMD:

$S \rightarrow AB \rightarrow aAB \rightarrow aA\epsilon \rightarrow aa$

Theory Point:

From one derivation tree there is only one LMD and one RMD possible.

Example with Explanation>



LMD =

$A \Rightarrow BC \Rightarrow abC \Rightarrow abc$

RMD =

$A \Rightarrow BC \Rightarrow Bc \Rightarrow abc$

} One LMD & One RMD.

Explanation:

If 'L' is the language and 'w' is a string which belongs to language 'L', then there'll be atleast one derivation tree for 'w' and one LMD & one RMD.

LECTURE: 9

Ambiguity >

From one derivation tree there is only one LMD or only one RMD are possible.

$S \rightarrow AB$ $A \rightarrow aA/\epsilon$ $B \rightarrow bB/\epsilon$	I want to generate "aab": LMD> $S \Rightarrow AB \Rightarrow aAB \Rightarrow aaB \Rightarrow aab$ RMD> $S \Rightarrow AB \Rightarrow Ab \Rightarrow aAb \Rightarrow aab$

Unambiguous Grammar:

If the grammar can generate exactly one derivation tree for any string in the language, then that Grammar is unambiguous.

Example:

$S \rightarrow AB$
$A \rightarrow aA/\epsilon$
$B \rightarrow bB/\epsilon$

→ Unambiguous Grammar

→ Language = a^*b^*

NOTE:

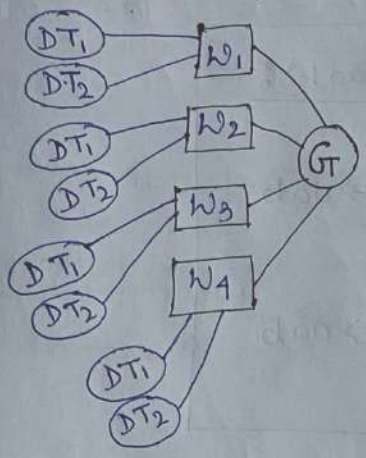
For every string there will be unique LMD & RMD.

If the Grammar is unambiguous, then it has one derivation tree exactly for any string. So, only 1 LMD & 1 RMD are possible.

Ambiguous Grammar:

"$\exists$" → There exists. "$\forall$" → For all.

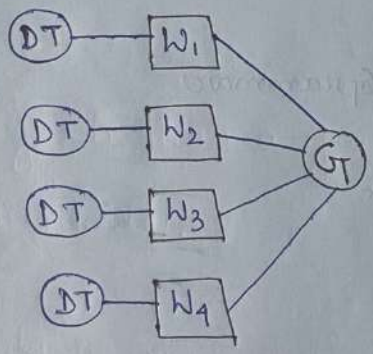
A grammar "G" is said to be ambiguous if $\exists (w) \in L$, such that ≥ 2 derivation trees are possible.



DT: Derivation tree.

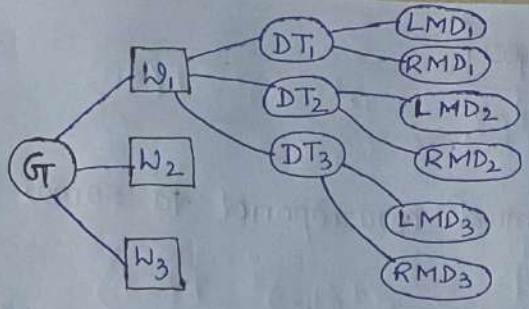
Unambiguous Grammar:

A grammar 'G' is said to be unambiguous if $\forall w \in L(G)$ $\exists$ exactly 1 Derivation tree.



Theory Points:

In an ambiguous grammar, we may have more than one derivation tree for a string. So there may be more than one LMD or RMD.



Ambiguous Grammar

THEORY POINTS:

→ In unambiguous Grammar ⇒ Every string has exactly one RMD.
⇒ Every string has exactly one LMD.
⇒ Every string has exactly one derivation tree.

→ If a string has 2 derivation trees, then there will be 2 LMD's, 2 RMD's and language is ambiguous.

→ i) If a string 'w' has 1 Derivation tree ⇔ 'w' has 1 RMD ⇔ 'w' has 1 LMD.

ii) If a string 'w' has atleast 1 DT ⇔ 'w' has atleast 1 RMD ⇔ 'w' has atleast 1 LMD.

Derivation Tree: Informal definition

1) Root must be 'S'.
2) Every expansion must correspond to some production of 'G'.
3) The yield must be a sentence. (↓ means a word.)

Partial Derivation Tree:

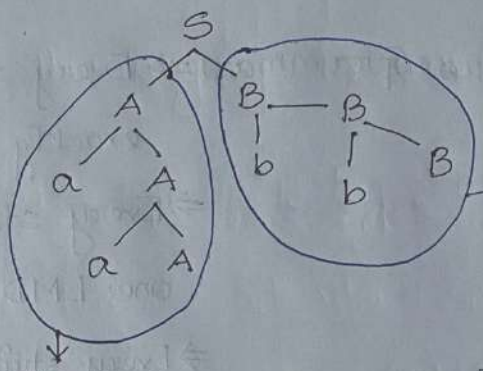
→ Root need not to be 'S'.

→ Every expansion must correspond to some production of 'G'.

→ The yield is a sentential form.

Example:

$S \rightarrow AB$
$A \rightarrow aA/\epsilon$
$B \rightarrow bB/\epsilon$



→ This is a partial derivation tree. Because 'S' is not the root. Yield of the PDT = bbB. Root = B.

This is a partial derivation tree. Because 'S' is not the root. Yield of the PDT = aaA. Root = A.

NOTE:

Checking whether a grammar is unambiguous or not is undecidable.

Ambiguity of Grammar:

Rules > 1st Rule >

1) If there are productions like $S \rightarrow S_1/S_2$, then there are two paths i) $S \rightarrow S_1$ ii) $S \rightarrow S_2$ and now, if $L(S_1) \cap L(S_2) \neq \phi$, then it is ambiguous.

Which means if there's something common b/w $L(S_1)$ & $L(S_2)$, then the grammar is ambiguous.

Q: $S \longrightarrow AB/CD$
$A \longrightarrow aA/\epsilon$
$B \longrightarrow bbB/\epsilon$
$C \longrightarrow cC/\epsilon$
$D \longrightarrow ddD/\epsilon$

Is it ambiguous or unambiguous?

$\Rightarrow S \longrightarrow \underbrace{AB/CD}$
Two paths are there.

So, we can take AB and CD path. If there's common b/w AB & CD, then we can say it's ambiguous.

$L(AB) = A\ B$
$\quad\quad\ \ |\ \ \diagdown$
$\quad\quad\ a^*\ \ bb^*$ $\quad \therefore$ The language $= (a^*(bb)^*)$

$L(CD) = C\ \ D$
$\quad\quad\ \ |\ \ |$
$\quad\quad\ c^*\ \ dd^*$ $\quad \therefore$ The language $= (c^*(dd)^*)$

So, we get epsilon as common b/w AB & CD.

Therefore, the Grammar is ambiguous.

Q: $S \longrightarrow AB/CD$
$A \longrightarrow aA/\epsilon$
$B \longrightarrow bbB/\epsilon$
$C \longrightarrow cC/c$
$D \longrightarrow ddD/\epsilon$

Is it ambiguous or unambiguous?

→ $L(AB) = A \; B$

$a^* \; bb^*$

$a^*(bb)^* = L(AB)$

$L(CD) = C \; D$

$c^+ \; dd^*$

$c^+(dd)^* = L(CD)$

$\therefore L(AB) \cap L(CD) = \phi$

Which means the language is unambiguous.

2) Left Recursion & left factoring : Rule no. 2

Examples & Explanation of a Standard Grammar which is asked many times in GATE :

$S \longrightarrow S+S/S*S/a/b/c$

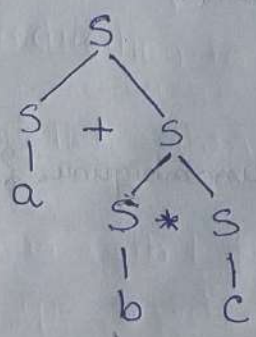
Assume a string : $a+b*c$

NOTE:

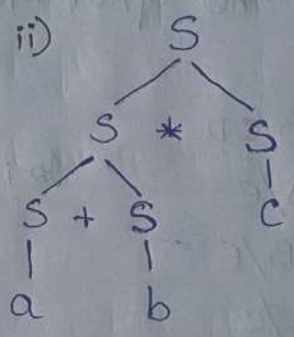
Left recursion & left factoring may lead to ambiguity. But not always.

$a+b*c$>

i)



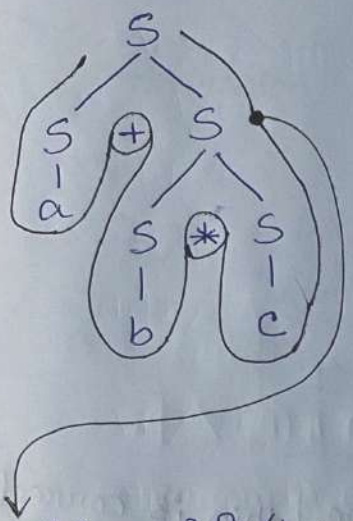
ii)



We are getting a+b*c in two ways.

Explanation of (i) & (ii) DTs:

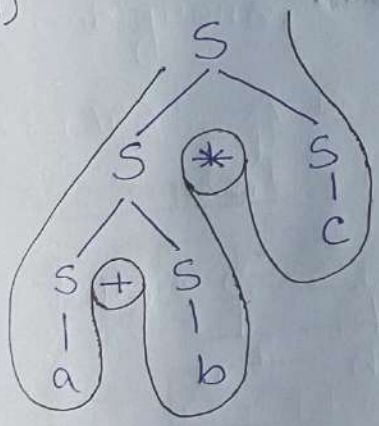
i)



Here, first 'a' is seen, then '+' is seen, so we get → a+.
Then 'b' is seen, next '*' is seen, then 'c' is seen. So, we get b*c.

At this point (b*c) is calculated. Then on the root 'S', the whole (a+(b*c)) is calculated.

ii)

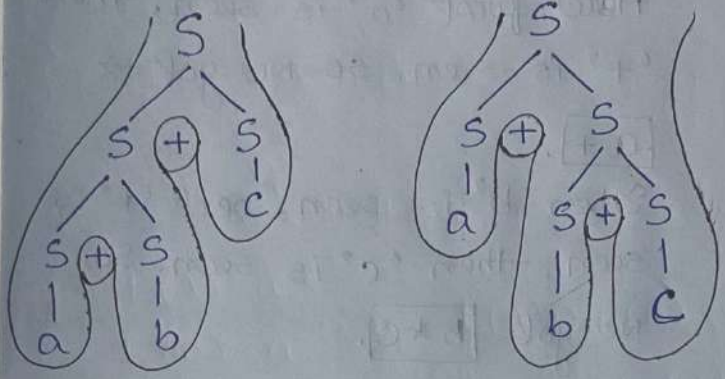


Here, if you observe, first (a+b) is calculated then '*c' is calculated. So, we get → (a+b)*c.

Therefore, no.(i) Derivation tree is correct.
Hence, we can say that, the Grammar is ambiguous.
Because, for the string (a+b*c), we get two derivation trees.

Q: $S \longrightarrow S+S/a/b/c$.

→ Assume a string: a+b+c



(i) (ii)

(i) We got: (a+b)+c → Symmantically this is correct.

(ii) We got a+(b+c)

Therefore, the Grammar is ambiguous.

NOTE:

'+' sign is Left associative.

What Left recursion means?

→ $A \rightarrow A\alpha/\beta$

When the left most symbol on the right hand side of a production is equal to the Left side of the production, then it's called Left Recursion

Problem with Left Recursion:

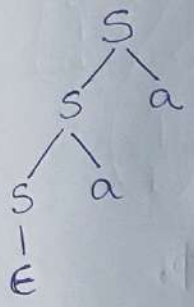
The Grammar may be ambiguous. But it's not ambiguous always.

Example of Left Recursion where it's unambiguous:

$S \rightarrow Sa / \epsilon$

↓

Left Recursion.



For 'aa' the only possibility is this.
There's no other posibility.
Which means, even though it's LR, the Grammar is unambiguous.

Left Factoring:

If you have something like: $S \rightarrow \alpha x / \alpha y$.

↓

This alpha is the common part.

For example:

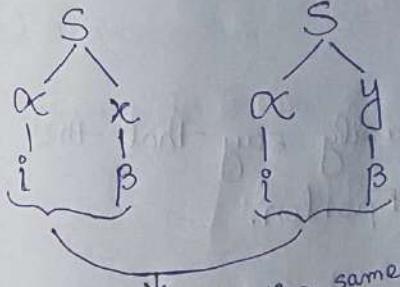
$S \rightarrow ABC / ABD$ → Common part

$S \rightarrow AaAax / Aa Aay$ → Common part

This is called left factoring.

Problem with Left Factoring:

$S \rightarrow \alpha x / \alpha y$



Both generating same language. So, nothing common.

Alpha is deriving some string 'i'.
But if x & y both deriving a same strings, then we are generating same language with different approach. Which means Grammar

$= L(x) \cap L(y) \neq \phi$ ∴ Language

= Unambiguous

Q: $S \rightarrow aA/aB$

$A \rightarrow a$

$B \rightarrow a$

→ $S \rightarrow \underline{a}A/\underline{a}B$

Remove this part

$\therefore L(A) = a$

$L(B) = a$

$L(A) \cap L(B) = a$

Therefore, $L(A) \cap L(B) \neq \phi$, which means the Grammar is ambiguous.

Q: $S \rightarrow aaA/aaB$

$A \rightarrow aA/\epsilon$

$B \rightarrow a$

→ $S \rightarrow \boxed{aa}A/\boxed{aa}B$

Remove this part.

$\therefore L(A) = a^*$

$L(B) = a$

$\therefore L(A) \cap L(B) = a$

Therefore, $L(A) \cap L(B) \neq \phi$, which means, the Grammar is ambiguous.

Q: $S \rightarrow aA/aB$

$A \rightarrow aA/\epsilon$

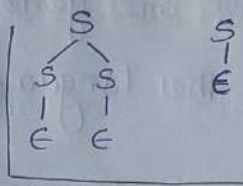
$B \rightarrow bB/\epsilon$

→ By seeing the Grammar you can easily say that the common part b/w $L(A)$ & $L(B)$ = Epsilon.

∴ Grammar is ambiguous.

3) Presence of SS: 3rd Rule

$S \rightarrow ab/ba/SS/\epsilon$ → Ambiguous



Shortcut to know the language when 'SS/ϵ' is there:

First remove the 'SS/ϵ':

$S \rightarrow ab/ba/\boxed{SS/\epsilon}$ → Remove it

$S \rightarrow ab/ba$

↳ make it (ab+ba) and put a '*' on it.

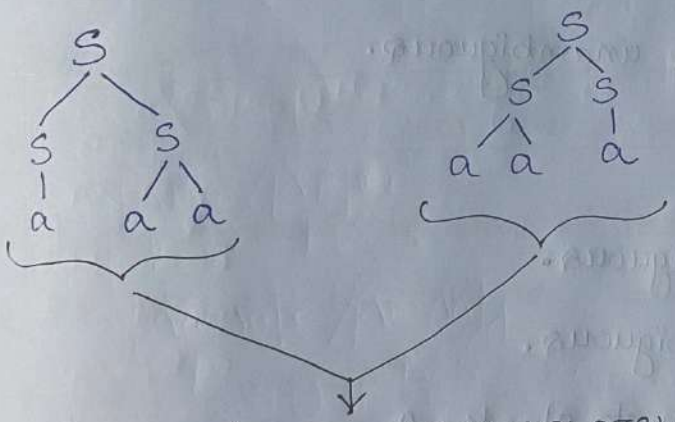
So, we get → $(ab+ba)^*$

↓

This is the language.

Q: $S \rightarrow ab/ba/SS/a$.

⇒ Assume we want to generate: aaa



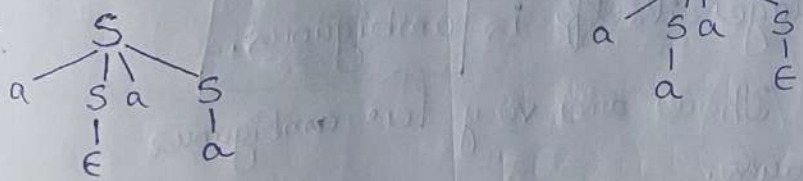
In two ways we are able to derive 'aaa'.

∴ Grammar = Ambiguous.

Q: Example with SS & ϵ not side by side or together:

$S \rightarrow aSaS/a/\epsilon$

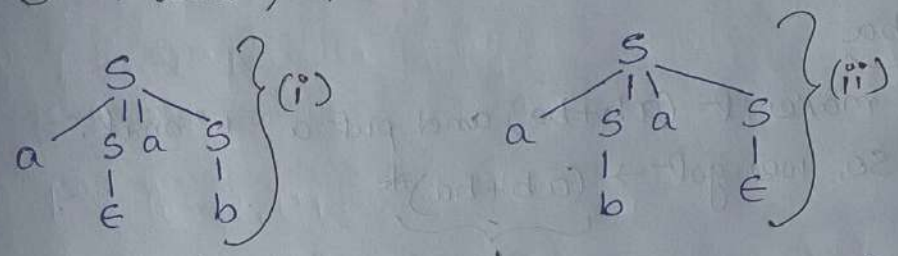
⇒



Here also we are generating 'aaa' in two ways.
Therefore the language is ambiguous.

Example where SS & ϵ is there, but the Grammar is unambiguous:

$S \longrightarrow aSaS/b/\epsilon$



We can't derive 'aab' in two different ways.
In (i), we are getting → aab.
In (ii), we are getting → aba.

∴ The language is unambiguous.

NOTE:

SS/ϵ = surely ambiguous.
SS/a = surely ambiguous.
aSaS/a = you have to check, it may or may not be ambiguous.

Ambiguity Of Languages:

For a language if all the Grammars are ambiguous, then the language itself is **ambiguous.**

There are very few ambiguous languages.

Ambiguous languages are called inherently ambiguous languages.

Which means their nature itself is ambiguous.

For Grammar Ambiguity:

If a Grammar derives more than one derivation tree for a string, then it is **ambiguous.**

If a Grammar derives exactly one derivation tree for a string in the language, then it's called **unambiguous.**

LECTURE : 10

In a language if there's atleast one unambiguous grammar, then the language is **unambiguous.**

To check whether a language is ambiguous or not, there are 3 points we have to check:

1) A Grammar should be union of finite no. of languages:

$L = L_1 \cup L_2 \cup L_3 \ldots$ finite no. of languages.

Example: $a^n b^n \cup a^* b^* \cup b^* a^*$

2) $L_1 \cup L_2 \cup L_3 \ldots$ **shouldn't collapse** into a single language.

Example:

$a^n b^n \cup a^* b^*$

Here we know, $a^n b^n \subseteq a^* b^*$, which means there are no two languages. The languages collapsed.

$\therefore$ language = unambiguous.

3) $L_1 \cap L_2 \cap L_3 \ldots \neq \phi$ [This should not happen].

$L_1 \cap L_2 \cap L_3 \ldots$ should always be '∞'.

NOTE:

A language is ambiguous if it follows all the three points mentioned above.

Q: $L = a^n b^n \cup a^n b^{2n}$. Ambiguous or Unambiguous?
$n \geq 0$.

⇒ 1st rule = $L_1 \cup L_2$

$= a^n b^n \cup a^n b^{2n}$ ✓

2nd rule = $L_1 \cup L_2$ should not collapse.

$a^n b^n \cup a^n b^{2n}$ ~~should not~~
is not collapsing. ✓

Because here we can say → $a^n b^n \cup a^n b^{2n}$ / $n = m$ (or) $m = 2n$

Because of 'or' it's not collapsing.

3rd rule = $(a^n b^n) \cap (a^n b^{2n}) = \{\epsilon\}$ → This is finite
(ϵ and ϵ)

Epsilon is common, so the Grammar is abmbiguous. But the language may not.

We can write the Grammar as →

$S \rightarrow A/B$
$A \rightarrow aAb/\epsilon$
$B \rightarrow aBbb/\epsilon$

Now we can put the epsilon directly in 'S' to make the language unambiguous.

Therefore, we get ⇒

$S \rightarrow A/B/\epsilon$

$A \rightarrow aAb/ab$

$B \rightarrow aBbb/abb$

In 3rd rule it became ⇒ $L_1 \cap L = \phi$.

∴ It fails the 3rd Rule.

Therefore, we can say that the language is unambiguous. Because we got an unambiguous grammar for the given language.

Q: $a^n b^n \cup a^{2n} b^{2n} / n \geq 0$. Ambi or Unambi?

⇒ 1st rule: $L_1 = L_2 \cup L_3$

$= a^n b^n \cup a^{2n} b^{2n}$ ✓

2nd rule: $L_2 \cup L_3$ should not collapse.

But if you observe, $a^n b^n$ is superset of $a^{2n} b^{2n}$, which means the language is unambiguous.

Q: $(a^n b^n c^m) \cup (a^n b^m c^m) / n, m \geq 0$.

⇒ 1st rule: $L_2 \cup L_3$ ✓

2nd rule: $L_2 \cup L_3$ can't collapse. ✓

3rd rule: $L_2 \cap L_3 = \infty$ ✓

In, $a^n b^n c^m \rightarrow$ no. of a's = no. of b's

$a^n b^m c^m \rightarrow$ no. of b's = no. of c's

So, we can say → $a^n b^n c^n / n \geq 0$

This is infinite

Therefore the language is ambiguous.

Important Theory Points:

→ If a language is ambiguous, then all the grammars of the language are ambiguous.

→ If all the Grammars of a language are ambiguous, then the language is ambiguous.

→ If a language is ambiguous, then it's also called inherently ambiguous language. i.e. ambiguity can't be removed from it's grammars.

→ If there exists atleast one ambiguous grammar for a language, then it's called ambiguous language.

Standard text book examples of ambiguous languages:

$a^n b^n c^m \cup a^m b^n c^n$ → This format will be followed by the other ambiguous languages.

Q: $a^n b^n c^m d^m \cup a^n b^m c^m d^n$

⇒ 1st rule: $L = L_1 \cup L_2$ ✓

2nd rule: L_1 & L_2 won't collapse. ✓

3rd rule: $a^n b^n c^m d^m \cap a^n b^m c^m d^n = \infty$ ✓ $[a^n b^n c^n d^n / n \geq 0]$

Therefore the language is ambiguous.

NOTE:

The above language is inherently ambiguous.

Q: $(a^n b^n) \cup (a^n b^{2n})$ $n \geq 0$ → Ambiguous or not?

⇒ 1st rule: $L = L_1 \cup L_2$ ✓

2nd rule: L_1 & L_2 won't collapse. ✓

3rd rule: $(a^n b^n) \cap (a^n b^{2n})$

$\{\epsilon, ab, aabb, \ldots\} \cap \{\epsilon, abb, aabbbb, \ldots\}$

$= \{\epsilon\}$

↓

This is finite.

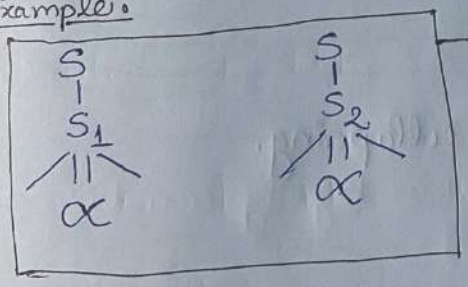
Therefore, the language is unambiguous.

Theory Notes:

→ If a Grammar has two parts like S_1, S_2 which can't be collapsed then, $S \rightarrow S_1 / S_2$ will be there.

→ If $L(S_1) \cap L(S_2) \neq \phi$, then some strings can be produced in two ways.

Example:



→ This leads to ambiguity.

→ Sometimes ambiguity is not a problem of the Grammar, the underlying language may be ambiguous, i.e. the language is inherently ambiguous language.

NOTE:

If you are trying to turn an ~~un~~ ambiguous Grammar to unambiguous, then first check whether the language generating the ambiguous grammar is following the 3 points of ambiguity of language.

If the language is following the 3 points, then you can declare the language as inherently ambiguous language. Hence, you can't generate ~~unam~~ unambiguous grammar for the given language.

Theory Point:

If a language is unambiguous, then there will be atleast one grammar for the language that is unambiguous.

Q: $L = \{a^m b^n / m > n\} \cup \{a^m b^n / m < n\}$

⇒ 1st rule: $L = L_1 \cup L_2$ ✓

2nd rule: The L_1 & L_2 are collapsing.

Because you can write,

$L = \{a^m b^n / m \neq n\}$

Therefore, the given language fails in Rule no. 2.

So, the language is **unambiguous.**

Q: $\{a^m b^n / m>n\} \cup \{a^m b^n / m<n\} \cup \{a^m b^n / m=n\}$

⇒ 1st rule: $L = L_1 \cup L_2 \cup L_3$ ✓

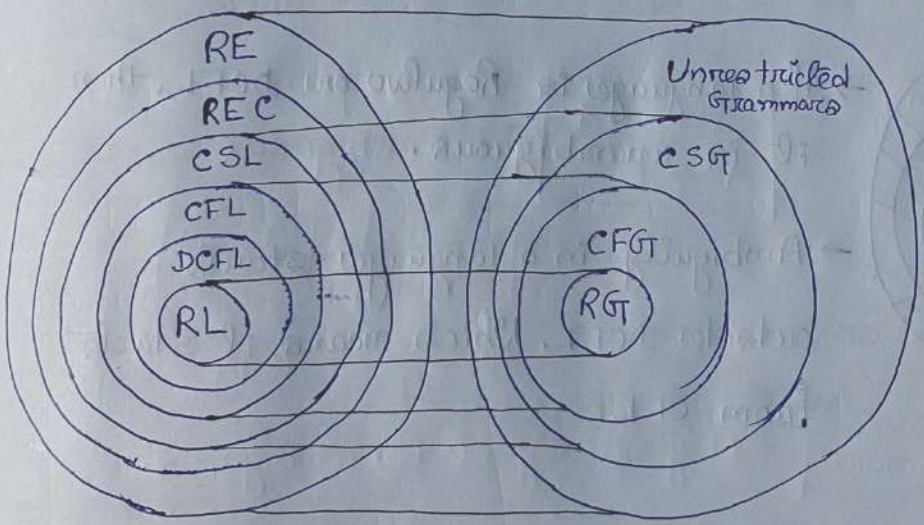
2nd rule: If you observe, the language is generating a^*b^*, which is unambiguous.

The given language is collapsed.

Therefore, the given language is **unambiguous.**

LECTURE: 11

Which Grammar belongs to which language?



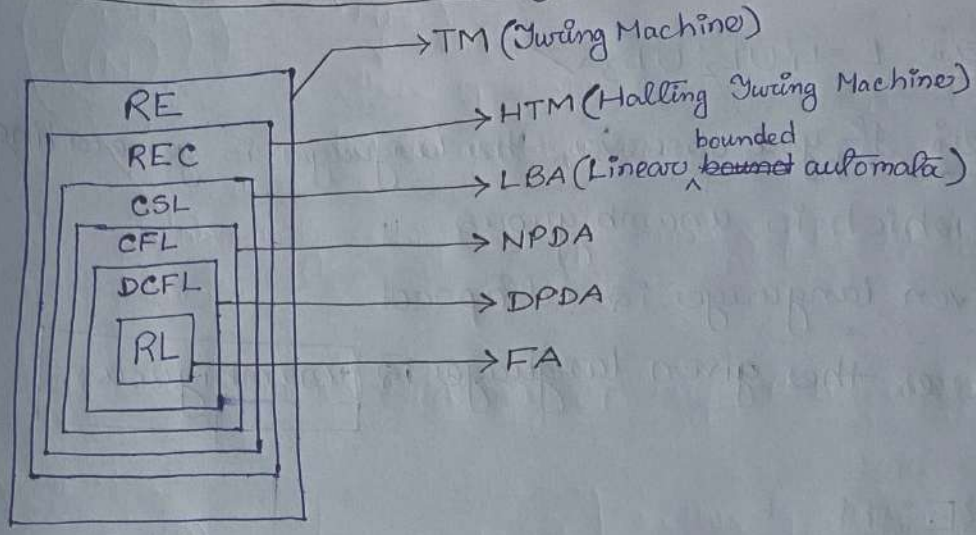
RL → RG.

CFL → CFG.

CSL → CSG.

RE → Unrestricted Grammars.

Machines used for each languages:



Theory Notes:

→ Certain types of languages are exempted from being ambiguous.

→ If a language is Regular or DCFL, then it is unambiguous.

→ Ambiguity in a language starts outside DCFL. Which means, it starts from CFL.

Language ambiguity and Grammar ambiguity can be different.

Example:

L:→ $a^* \cup b^*$

↓

The language is unambiguous. Because it's a Regular Language.

Grammar:

$S \rightarrow S_1 / S_2$
$S_1 \rightarrow aS_1 / \epsilon$
$S_2 \rightarrow bS_2 / \epsilon$

→ The Grammar is ambiguous. But we can get a grammar which is unambiguous.

$S_1 \rightarrow S_1 / S_2 / \epsilon$
$S_1 \rightarrow aS_1 / a$
$S_2 \rightarrow bS_2 / b$

→ Unambiguous Grammar for the same 'L'.

Theory Point:

There may be some ambiguous grammars for Regular languages and DCFL. (See the previous example for proof).

For Regular Languages:

If a language 'L' is given, first construct the DFA. Then get the Grammar from it.
This Grammar will always be unambiguous as DFA does not give any choice, means there won't be two moves on the same configuration.

→ This won't happen in DFA.

For DPDA: (Same as DFA)

L → DPDA → Grammar

↓

Always unambiguous, as there is no choice given by DPDA.

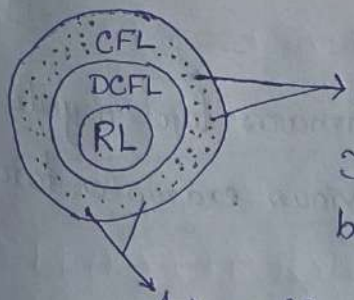
Theory Point:

Sometimes NFA and NPDA can also generate unambiguous grammars.

Explanation:

Every DFA is an NFA.
Every DPDA is an NPDA.

} Whatever rules we got for DFA & DPDA, can also be applied for NFA & NPDA.



→ NPDA

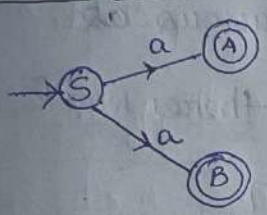
↓

These Grammars may or may not be ambiguous.

Inherently ambiguous grammars falls in this part.

A Grammar from an NFA may or may not be ambiguous.

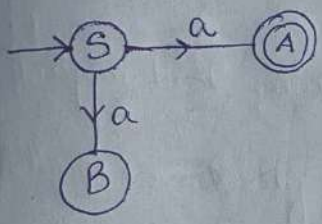
Example:



$S \rightarrow aA/aB$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

→ This Grammar is ambiguous.

Taking the same NFA:



$S \rightarrow aA$
$A \rightarrow \epsilon$

→ This Grammar is unambiguous.

Theory Points:

NFA → [Grammar] → May or may not be ambiguous.

DFA → [Grammar] → Always unambiguous.

DPDA → [Grammar] → Always unambiguous.

LECTURE: 12

If you take a DPDA and convert it into Grammar then it is going to give unambiguous grammar always.

i) → So, Regular languages & DCFL never can be ambiguous.

ii) But Regular Grammars and DCFG may be ambiguous.

Theory Point:

Even a non DCFL language can be ~~ambiguous~~ unambiguous.

Example:

$a^n b^n \cup a^n b^{2n}$ → This language is CFL but not DCFL.

$$\Downarrow$$

$S \to A/B$ $A \to aAb/\epsilon$ $B \to aBbb/\epsilon$	$\Longrightarrow$	$S \to A/B/\epsilon$. $A \to aAb/ab$ $B \to aBbb/abb$

→ Now the we got an unambiguous Grammar for the language.

The Grammar is ambiguous, but we can make it unambiguous.

∴ The given language is unambiguous.

NOTE:

Beyond DCFL's also there are languages which are unambiguous.

Membership Algorithms:

Membership problem is **decidable** (→ Means algo exist.) in Reg, DCFL, CFL, CSL & REC.

The algo will convert:

Reg to FA
DCFL to DPDA
CFL to NPDA
CSL to LBA
REC to HTM

→ All these machines are halting machines. Means decidable.

RE to TM → Turing Machine may or may not halt. Which means, it's **undecidable.**

For membership we have 3 algos:

i) Brute Force.

ii) CYK.

iii) LL(K) & LR(K).

i) Brute Force:

→ Exponential time.

→ It belongs to NP Class.

→ Not practical.

→ Very slow.

→ In any membership algo. first remove ϵ-productions, and unit productions, then remove useless productions if available.

Important point for Brute Force:

Time Complexity of Brute Force → $O(p^n)$ → $O(2n)$

↓

It's exponential.

ii) CYK:

→ It's polynomial

→ Time Complexity = $O(n^3)$

→ Tractable.

→ We need Chomsky Normal Form for CYK.

Why we need LL(K) & LR(K)?

⇒ If we have CFLs, inside it we have DCFLs. CYK is able to parse all the CFLs. And time complexity of it, is $O(n^3)$.

But most of the programming languages features fall under DCFL.

CYK's time complexity is high. Now if we only want to find membership of DCFL, we'll need LR(K) & LL(K).

Time complexity of $\boxed{LR(K) \text{ \& } LL(K) = O(n)}$.

When simplifying a Grammar, follow the below rules in the same order:

1) Delete epsilon first.
2) Delete unit productions.
3) Delete useless productions.

33:59

iii) DLL(K):

A grammar 'G' is LL(K), when 'K' symbols of an input is presented to the compiler at a time. The production that is used in every step of the derivation is uniquely determined.

Example:

aaabb → If we use LL(2) here, then first the compiler will take aa then ab then b. It'll know what to choose next.

Time complexity: $O(n)$ for LL(K).

Theory NOTES:

In every step of derivation we can use one and only one production, unlike Brute force which uses |P| productions at a time.

So, LL(K) is O(n) time algo.

2) LR(K):

A grammar "G" is LR(K), when "K" symbols of an i/p input are presented to compiler at a time, the production that is used in every step of derivation is uniquely determined.

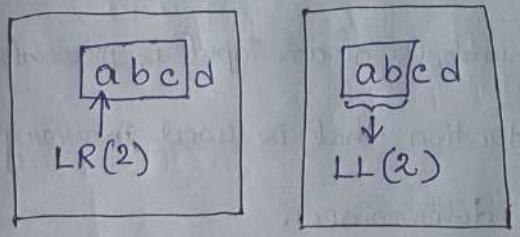
Difference b/w LL(K) & LR(K):

In LR(K), the K means lookaheads. Which means →

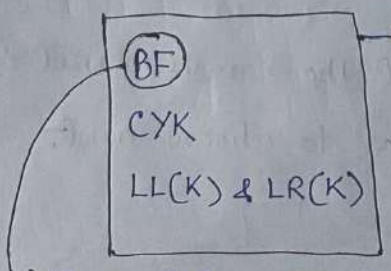
1 current symbol + "K- look aheads".

For the string "abcd":

If we use LR(2) here then it'll take a, with that bc will be also taken. So, the first string we get here → abc.



11:33



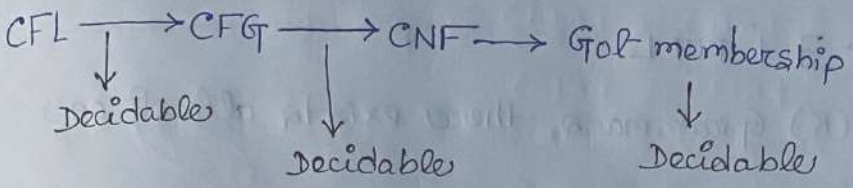
→ For all these, first eliminate epsilon, then unit productions and then remove useless productions if exist.

→ For brute force, O(2n) steps needed to generate a string.

NOTE:

CNF has same power as CFG.

For every CFL there's a CFG. And for every CFG, there's a CNF.



Therefore, CFL membership is decidable.

Properties of LL(K) and LR(K):

1) LL(K) and LR(K) are unambiguous. ∴ Suitable for Compiler Design.

2) Some unambiguous grammars are neither LL(K) nor LR(K).

3) If a Grammar is ambiguous, then surely it is not LL(K) or LR(K).

*4) LL(K) or LR(K) can parse in linear time (O(n)).

*5) ∀ DCFL ⟶ ∃ 1 LR(K)

[For every DCFL, there exists atleast one LR(K).

Most of the Programming Language's features are in DCFL. ∴ They have LR(K) grammars and so, parsing can be done in O(n) time.

NOTE:

LR(K) Parser → This is the Algorithm.

LR(K) Grammar → This is the Grammar.

6) $\forall$ LR(K) → $\exists$ 1 DCFL.

For every LR(K) Grammars, there exists atleast one DCFL.

7) $\forall$ LL(K) → $\exists$ 1 DCFL.

For every LL(K) Grammars, there exists atleast one DCFL.

But for every DCFL there need not be any LL(K).

In simple words, we can say →

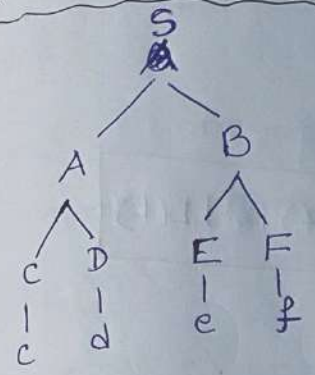
For all LL(K) grammars, there exists a DCFL.

But there are some DCFL's which are not LL(K).

LL(K)
- L → Left Most Derivation (LMD)
- K → Number of look aheads.
- Left to right scanning of input

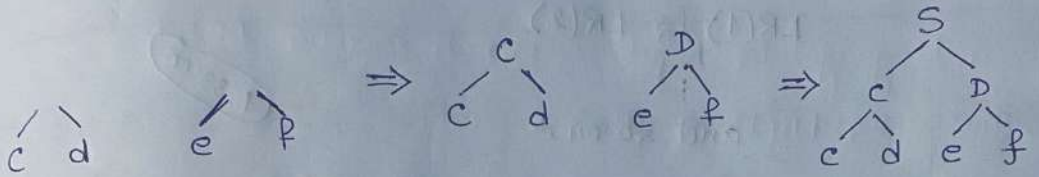
LR(K)
- R → Right Most Derivation (RMD)
- K → Number of look aheads.
- Left to right scanning of input

LMD: Top down parsing>



LMD uses Top down parsing.

RMD: Bottomup Parsing>



RMD uses bottomup parsing.

Theory Points:

→ In LL(K) and LR(K) every string will have exactly one derivation tree. Because at every step only one production is used and there is no choice.

LL(K) → O(n)
LR(K) → O(n)

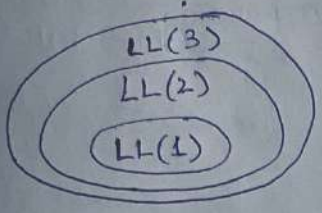
→ Practically every feature of any programming language is in DCFL, except few. So, we do not use CYK. But we use other parsers like, CLR, SLR which are faster than CYK.

→ Every DCFL has a corresponding LR(K).

→ Every LL(K) grammar is DCFL.

→ Every DCFL is not LL(K).

→ If a grammar is LL(K), we can surely call it as LL(K+1).



LL(1) is LL(2).

But we can't say → LL(2) is LL(1)

→ LR(K) = LR(K') / K' ≥ K.

So we can say → LR(0) is LR(1)

LR(1) is LR(2)

⋮

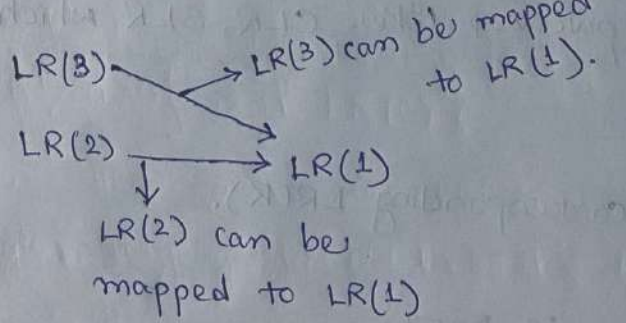
and so on.

1:33:00

→ There exists an algo to check if a given Grammar is LL(K) for a given value of K. Which means this is decidable.

→ There exists an algo to check if a given Grammar is LR(K) for a given value of K. Which means this is decidable.

→ If Grammar is LR(K)/K ≥ 2, then there exists G_1 which is LR(1) for the same language.



Which means we only need LR(0) and LR(1). Using these two, we can easily know what the higher parsers are doing.

Modified and easier explanation of the previous point:

→ If Grammar (G) is LR(K) / K ≥ 2, then there exists some grammar G_1, which is LR(1), such that $L(G_1) = L(G)$

∴ LR(0) and LR(1) are sufficient for all grammars. Therefore, for every DCFL there exists LR(0) or LR(1).

→ For all DCFL's with prefix property, we can generate LR(0).

Prefix Property and Proper Prefix Property:

DCFL has prefix property if $\forall w \in L$, there is no proper prefix $w \in L$ exists.

Example and explanation:

$L = \{011, 10\}$

Proper prefix of $011 \rightarrow \epsilon, 0, 01$ [NOTE: 011 is not a proper prefix of 011, so don't write it]

Now check whether $\epsilon, 0, 01$ is present in $\{011, 10\}$

→ $\epsilon, 0, 01$ don't exist in $\{011, 10\}$

Now check proper prefix of $10 \rightarrow \epsilon, 1$

Now check whether $\epsilon, 1$ is present in $\{011, 10\}$

→ $\epsilon, 1$ don't exist in $\{011, 10\}$

Therefore, $L = \{011, 10\}$ does not has proper prefix. So, it has prefix property. ∴ LR(0) Grammar is possible for this L.

Q: $L=\{011, 10, 1\}$

⇒ $011 \rightarrow 0, 01, \epsilon$

$10 \rightarrow \boxed{1} \rightarrow 1$, is already there in the language, which means the languages has proper prefix property.

∴ no prefix property.

Which means LR(0) is not possible.

The given language is Regular, which means DCFL also. Therefore, LR(1) is there.

The given language is LR(1), but not LR(0).

Q: Is a^*b^* LR(0)?

⇒ $a^*b^* \rightarrow \{\epsilon, a, aa, b, bb....\}$

NOTE:

Whenever epsilon is there in the Language, the language will never have prefix property.

∴ The given language is not LR(0), but it is LR(1), because it's Regular, hence DCFL. Every DCFL has LR(1).

Q: $\{a^n b^n / n \geq 0\}$ is it LR(0)?

⇒ Epsilon is there, so it's not LR(0).

The given language is DCFL, so it's LR(1).

Q: $a^n b^n / n \geq 1$ is it LR(0)?

⇒ Assume a string → aaabbb

The language does not have proper prefix. So, we can say the language is LR(0).

aaabbb's proper prefix = {a, aa, aaa, aaab, aaabb}

These are not present in the language.

Q: $a^+ b^*$ → Is it LR(0)?

⇒ Assume a string → abb

$a^+ b^* \Rightarrow$ {a, aa, aaa, ab, abb, ...}

Proper prefix of abb → a, ab

'a' is present in the language $a^+ b^*$. Which means the language has proper prefix property.

∴ The language is not LR(0) but it's LR(1).

NOTE:

If a language has **proper prefix property** then it'll never be LR(0).

Only if the language has **prefix property** then only it'll be LR(0).

LECTURE-13:

Algorithms in CFG>

1) a) Removal of ϵ-production.
   b) Removal of unit-production.
   c) Removal of useless production.
   d) Removal of Left recursion.
   e) Removal of Left factoring.

Conversions>

CFG → CNF

CFG → GNF

Theory Points:

→ ϵ and unit productions have to be removed before a grammar can be converted into a CNF or GNF.

→ Removing ϵ-production is not always possible. When the language contains 'ϵ', the ϵ-production can't be eliminated.

Example: $L = \{\epsilon, a\}$;

$S \rightarrow \epsilon / a$

→ For all CFG's which are ϵ-free grammars, there exists a CNF.

→ For all CFG's which are ϵ-free grammars, there exists a GNF.

→ Presence of ϵ-production in the language does not mean 'ϵ' is in the language.

Example:

$S \rightarrow aA$
$A \rightarrow \epsilon$

→ There may be ϵ in the Grammar. But language is simply. $L = \{a\}$.

→ If ϵ is present in the language, then there will be atleast one ϵ-production in the Grammar.

How to check if 'ϵ' is there or not?

⇒ There are two methods →

1) Check using nullable variables (Explanation in LECTURE4 notes).

2) Check the chain of every production if something is leading to 'ϵ'.

Example:

$S \rightarrow aAB/aB/B$
$B \rightarrow ab/Ab/bA$
$A \rightarrow \epsilon$

* finding if ϵ is present in the language or not using nullable variables:

In the above Grammar only A is generating 'ϵ'. So, 'A' is the only nullable variable. 'B' & 'S' are not nullable.

Because 'S' is not nullable, we can say 'ϵ' is not present in the language.

Finding if ϵ is present or not in the language using the second method:

$S \longrightarrow aAB/aB/B$

$B \longrightarrow ab/Ab/bA$

$A \longrightarrow \epsilon$ → Because of 'a' it's not nullable.

$\Rightarrow S \longrightarrow (aAB)/(aB)/(B)$ → This may be nullable. To check, we have to see productions of 'B'.

Here because of 'a' it's not nullable.

$B \longrightarrow (ab/Ab/bA)$ → B is not nullable, because it does not has epsilon.

$A \longrightarrow [\epsilon]$ → Because of epsilon 'A' is nullable, but it's not in 'S'.

Therefore, ϵ is not present in the language.

Q: $S \longrightarrow aAb/aA/B$
$B \longrightarrow ab/Ab/A$
$A \longrightarrow \epsilon$

ϵ is there in the language or not?

$\Rightarrow$ 'A' is nullable. In 'B', 'A' is present individually, so now 'B' has become nullable.

'B' is present individually in 'S'. So, now 'S' has become nullable. Hence, ϵ is present in the language.

Theory Point:

→ A variable 'V' is useless if it does not appear in any sentential form.

Useful symbols (u) = $\{V / S \overset{*}{\Rightarrow} \boxed{xVy} \overset{*}{\Rightarrow} w, w \in T^*\}$

'V' is useful only if it's generating a string with x and y.

Which means 'V' should be reachable and 'V' should derive something.

→ For every grammar 'G' there is an equivalent grammar $\bar{G}$, without Left Recursion.

→ For every grammar 'G' there is an equivalent grammar $\bar{G}$, without Left Factoring.

→ For every grammar 'G' there is an equivalent grammar $\bar{G}$ without LR & LF.

Removal of Left Recursion:

$A \rightarrow \underbrace{A\alpha_1 / A\alpha_2 / A\alpha_3 \ldots / A\alpha_n}_{\text{This whole part is in Left Recursion, which we want to remove.}} / \underbrace{\beta_1 / \beta_2 / \beta_3 \ldots \beta_n}_{\text{This part is stopper}}$

1st Step: We'll create one A', where all the 'α^*' will be generated.

2nd Step: We'll add the 'β' infront of the respective 'α's.

Because the above mentioned language is giving us = $\boxed{\beta_n \alpha_n^*}$

So, after doing the previous two steps, we'll get:

⇒

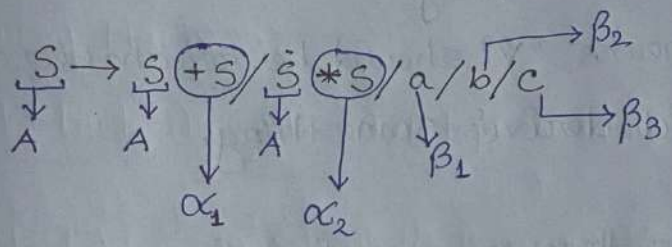
$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' \dots / \beta_n A'$ → Step 2

$A' \to \alpha_1 A' / \alpha_2 A' / \dots \alpha_n A' / \epsilon$

Step 1

Q: $S \to S+S / S*S / a / b / c$

⇒



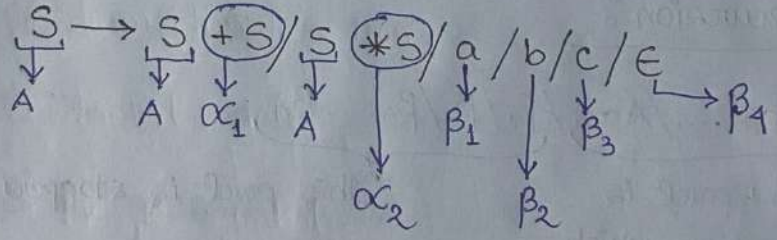
Final Result after removing LR:

$A \to aA' / bA' / cA'$

$A' \to +SA' / *SA' / \epsilon$

Q: $S \to S+S / S*S / a / b / c / \epsilon$

⇒



Final Result after removing LR:

$A \to aA' / bA' / cA' / A'$ → $\epsilon \times A' = A'$

$A' \to +SA' / *SA' / \epsilon$

Removal of Left Factoring:

If we have>

$A \rightarrow \alpha x / \alpha y / \alpha z$

then we can pull out the 'α' as common.

So, we'll get →

$$A \rightarrow \alpha A'$$
$$A' \rightarrow x / y / z$$

Q: $S \rightarrow aA / aAb / aAB / B$

⇒ $S \rightarrow aA / aAb / aAB / B$

From this part we can take 'aA' as common.

$$S \rightarrow aAA' / B$$ → Final Result
$$A' \rightarrow \epsilon / b / B$$

Q: $S \rightarrow aA / aAB / aACD / a$

⇒ Here we can take 'a' as common or 'aA' as common.

Taking 'a' as common:

$S \rightarrow aA'$

$A' \rightarrow A / AB / ACD / \epsilon$

Taking 'aA' as common:

$S \rightarrow aAA' / a$ → From here now we can take 'a' as common

$A' \rightarrow \epsilon / B / CD$

Final Result:

$S \rightarrow aAA''$ ~~$A' \rightarrow \epsilon / B / CD$~~ ~~$A'' \rightarrow A' / \epsilon$~~

$S \rightarrow aA''$

$A'' \rightarrow AA' / \epsilon$

$A' \rightarrow \epsilon / B / CD$

THEORY POINT:

→ Left Factoring and Left Recursion can be removed easily. But it does not remove ambiguity. Infact removing ambiguity is an undecidable problem.

Turing Machine

Turing Machine:

TM = Finite Automata + I/P tape with Read/Write head + Left-Right Movement.

TM = FA + 2 Stack
PDA = FA + 1 Stack

→ Both two are also right.

TM = M (Machine) $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$Q \rightarrow$ Finite States

$\Sigma \rightarrow$ Input alphabet

$\Gamma \rightarrow$ Tape alphabets

$\delta \rightarrow$ Transition function

$q_0 \rightarrow$ Initial State

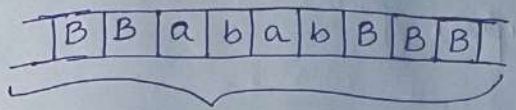
$B \rightarrow$ Blank symbol

$F \rightarrow$ Set of final states

In Finite Automata Read/Write head goes only in one direction (right direction), but in Turing Machine it moves in both left & right directions.

What is the value of 'Γ' in TM?

⇒



This is the Tape

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

B = Blank symbols of the tape.

My intention might be to replace the 'a's with '0' & 'b's with 1.

Therefore $\Gamma = \{a, b, B, 0, 1\}$ → These are the Tape Alphabets.

∴ Value of Γ:

$\Sigma \subseteq \Gamma - B$

Σ → {a,b}

Γ − B → Input symbols need not contain 'B', so we are subtracting it.

A subset of Γ is our Σ. Which is {a, b}.

NOTE:

We are talking about Deterministic Turing Machine.

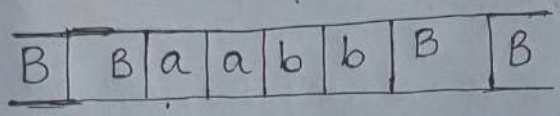
'δ' in DTM>

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times (R, L)$

You are in some state 'q' and you'll be seeing something on the tape (Γ) then you'll go to some state '$\tilde{q}$' and overwrite the tape ($\tilde{\Gamma}$) with some symbol from the tape alphabets and then you'll go either to the Right or Left.

NOTE:

Any Turing Machine will start from the very first non-blank symbol of the tape.



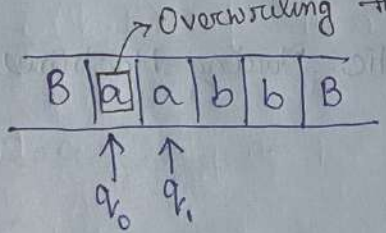
$\delta(q_0, a) = (q_1, a, R)$ ——— (i)

$\delta(q_1, a) = (q_1, a, R)$ ——— (ii)

$\delta(q_1, b) = (q_1, b, L)$ ——— (iii)

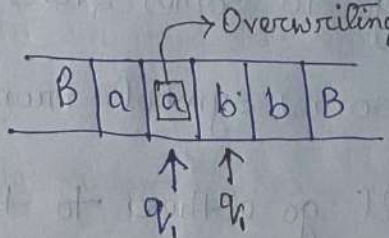
Explanation of (i):

q_0 on 'a' is going to q_1 and overwriting 'a' with 'a' & moving right afterwards.



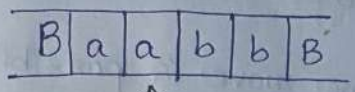
Explanation of (ii):

q_1 on 'a' is going to q_1 and overwriting 'a' with 'a' & moving right afterwards.



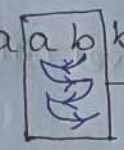
Explanation of (iii):

q_1 on 'b' it's leaving 'b' as it is. And coming back to left.



q_1 → q_1 is back to 'a' again.

After (iii) q_1 will be stuck b/w a & b, which means the Turing Machine is hanging.



→ This thing will keep on happening.

In the above example, the turing machine hangs.

∴ A string is rejected either by dead configuration or by hanging.

It's the difference between Regular, CFL and TM.

~~Note~~ Nondeterministic Turing Machine:

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{R, L\}}$$

Power-wise Relations:

DFA ≅ NFA
DPDA < NPDA
DTM ≅ NTM

Theory Point:

→ The set of languages accepted by NTM is same as the set of languages accepted by DTM.

→ Turing machine always starts with first non-blank symbol.
The tape is unbounded i.e. you can make any number of left or right moves.

→ In deterministic TM almost one move is possible for each configuration.

Explanation:

$\delta(q_0, b)$
this is the configuration.
for q_0, b there should be one move only.

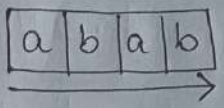
$\delta(q_0, b) = (q_1, b, R)$ OR (q_2, b, R) } this is wrong.

$\delta(q_0, b) = (q_1, b, R)$ ⟶ this is right.

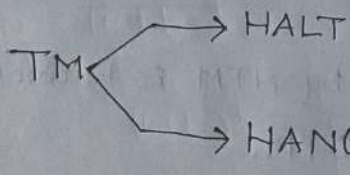
If $\delta(q_0, b) = \phi$, then it's Dead configuration.

→ TM can only do two things, either hang or halt.

→ In FA & PDA they only move to the right and when they meet the end, they will halt.

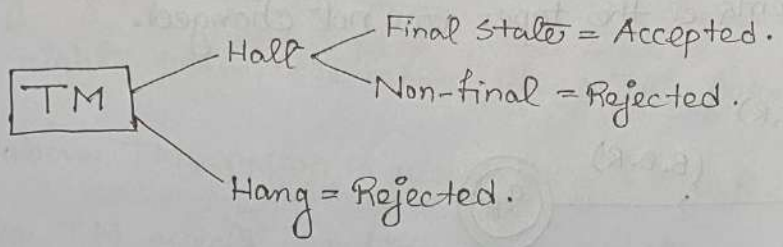


But in Turing Machine, we do not have end for the tape.



→ If TM is in hang, then we can say the string is not accepted.

But if it halts, then string is accepted only if it halts in final state, otherwise it's rejected.

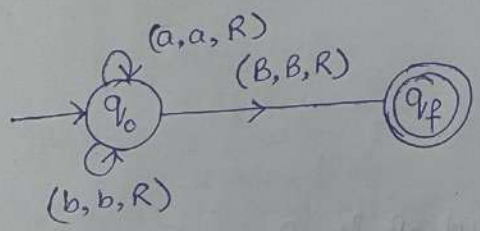


→ So, if you want to accept a string in TM, then send it to final state and kill it there by dead configuration

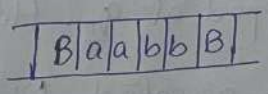
If you want to reject a string in TM, send it to non-final state and kill it there by dead configuration.

Example and Explanation:

i)



→ Assume a string aabb.



On seeing an 'a' leave it as 'a' and move right.
Again seeing an 'a' leave it as 'a' and move right.
Seeing a 'b' leave it as 'b' and move right.
Seeing a 'b' leave it as 'b' and move right.
Now seeing a Blank symbol you are going qf and halting is done. Which means halting is done at final state.
So, string is accepted.

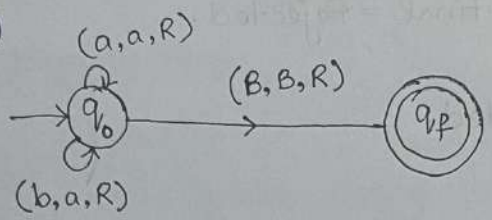
∴ The language = $(a+b)^*$

The above machine halts on : $(a+b)^*$

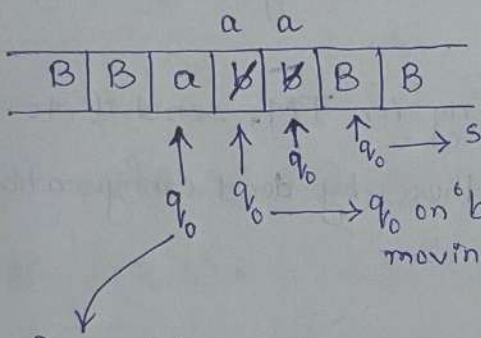
The machine hangs on : ϕ → which means it never hangs.

Here contents of the tape are not changed.

ii)



Assume a string → abb



it's getting into q_f.

q_0 → seeing one 'B' it's moving to right

q_0 → q_0 on 'b' is changing 'b' to 'a' & then moving right. State is same.

q_0 on 'a' is not changing 'a' and moving to the right.

So, we stopped at q_f. Halt happened at final states. So, abb is accepted. Whatever string we'll give, are getting into q_f, which means all strings are accepted.

∴ Language = $(a+b)^*$

The 'L' halts on : $(a+b)^*$

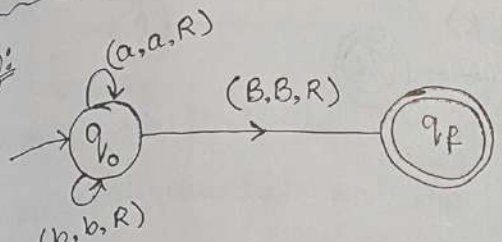
The 'L' does not hang.

The TM is converting all i/p into all 'a's.

This TM is acting like an acceptor and also as a transducer.

LECTURE : 1

Q:



Here only right moves are there. So, in TM whenever there are only right moves, it'll behave like a Finite Automata only.

In the above TM, epsilon is accepted using → B, B, R.

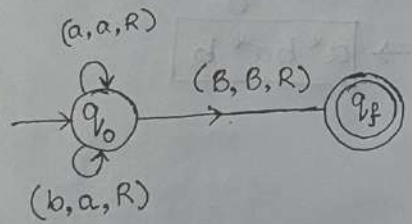
The above TM accepts the language → (a+b)*

BaB → 'a' is accepted.

BaaaaB → Everything is accepted.

BbbB → Everything is accepted.

Q:



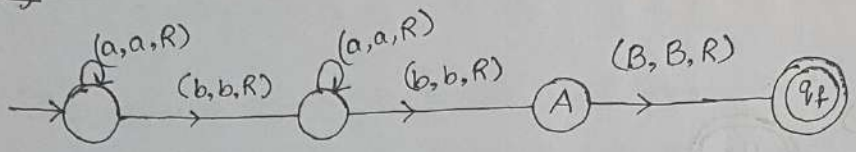
⇒ The above Turing Machine on seeing an 'a' it's leaving the 'a' as it is, & moving to right on seeing a 'b' it's changing it to 'a' and moving right.

Which means the TM is accepting the language → (a+b)*.

The TM is changing all inputs into "all a's".

So, it's acting as an acceptor & also as a transducer.

Q:



⇒ (a,a,R) → Acting as "a*".

(b,b,R) → Acting as "b".

(B,B,R) → If valid string is then this will help accepting the string.

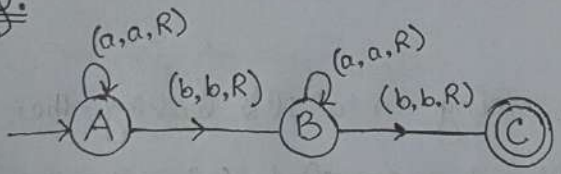
The above TM accepts strings which are in the form of →

$\boxed{a^*ba^*b}$

Assume a string → ababa

Here when we see the last 'a' of the string, that time we'll be in 'A' state of the TM. Which is a dead configuration & also a non-final state. So, the string will be rejected because it's not following → $\boxed{a^*ba^*b}$

Q:



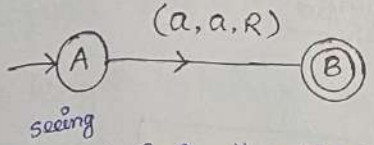
⇒ Assume a string 'abab' → This will be accepted.

Assume a string 'ababa' → This will also be accepted.

The above TM accepts the language → $a^*ba^*b(a+b)^*$

So, any string which follows $a^*ba^*b(a+b)^*$ will be accepted.

Q:

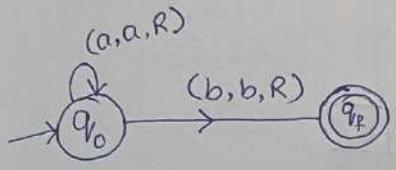


seeing

⇒ After one 'a' the TM will accept any string, because there's no configuration on the final state for any string.

∴ The laguage which is accepted by the TM = $\boxed{a(a+b)^*}$

Q: Draw the turing machine for $a^*b(a+b)^*$>



This TM is accepting language = $\boxed{a^*b(a+b)^*}$

How we'll find the language of a Turing Machine?

→ If a TM is only moving to the right, then it's working like a Finite Automata. In this case, it'll be easy to analyze.

But if the TM is moving both Left and Right, then in this case, it'll be difficult to analyze it.

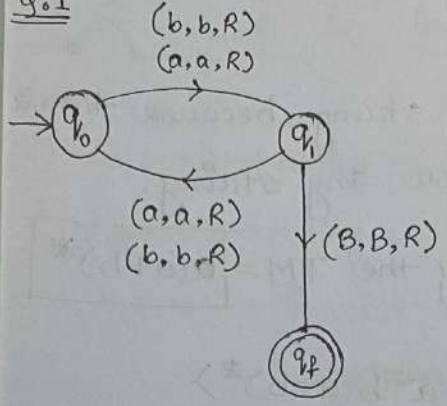
Therefore, we'll follow Numerical Method for this.

Numerical Method: We have to check, if below strings are getting accepted or not>

1) ∈
2) a
3) b
4) aa
5) ab
6) ba
7) bb

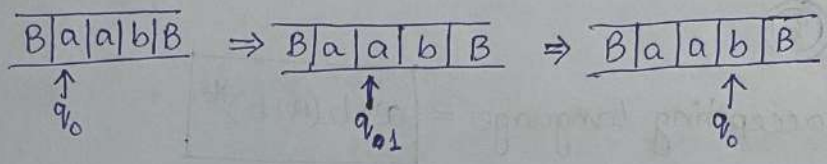
Example & Explanation of the previous point:

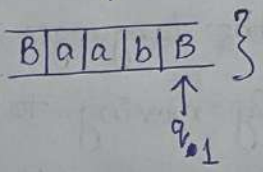
Q:1



Only Right moves, so acting as an FA

⇒ Assume a string → aab

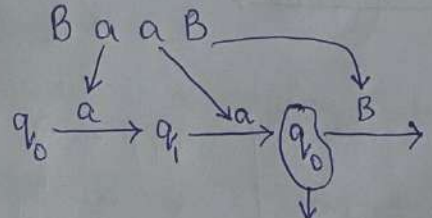




Here q_1 is seeing 'B' so without changing 'B' it's going to final state and TM halts here.

So, TM has accepted 'aab'.

Assume a string 'aa':



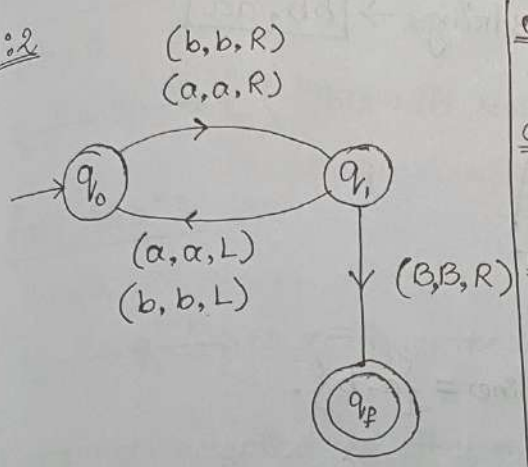
Now, when q_0 is seeing a blank, nothing is defined for q_0, which means it's a dead configuration, so TM halts at a non-final state. Hence the string is rejected.

Therefore, the TM accepts → Odd length strings.

Even length strings are rejected, not even epsilon is accepted.

This TM never hangs. ∴ Hang = ϕ.

Q:2



Q: What language is accepted by the machine?

Q: What are the strings where we have halted?

Q: What are the strings if was hanged?

⇒ Check for 'ϵ':

ϵ will look like → BBB

→ On q_0, 'B' move is not there.

Halt/Reject.

∴ this is dead configuration. So, ϵ it'll halt on non-final state & ϵ'll be rejected.

Check for 'a':

'a' will look like → BaBB

Halt/Accept.

$q_0 \xrightarrow{a} q_1 \xrightarrow{B} q_f$ ⇒ For 'a' we are halting at final state, so 'a' is accepted.

Check for 'b':

'b' will look like → BbBB

Halt/Accept.

$q_0 \xrightarrow{b} q_1 \xrightarrow{B} q_f \xrightarrow{B}$ } ⇒ Same like 'a' as above written. So, 'b' is accepted.

Check for 'ab':

'ab' will look like → BabB.

Hang / Reject

$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0 \xrightarrow{a} q_1$ } This will happen forever. ∴ This is a hang & it's rejected.

Just like 'ab', the TM will hang for 'ba' string also, & it'll we rejected.

∴ For string 'ba' → Hang / Rejected

Likewise the TM will hang for strings → bb, aa

For string 'bb' → Hang / Reject

For string 'aa' → Hang / Reject

Answers:

(i) The language accepted by the machine = {a, b}.

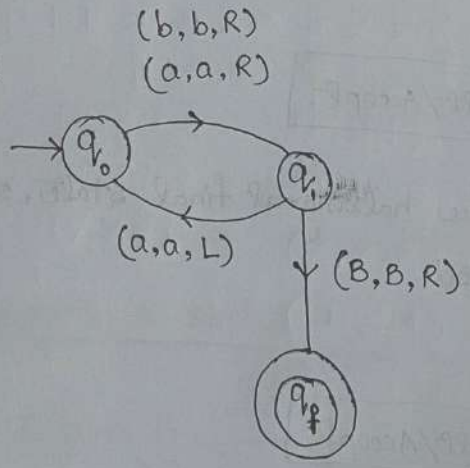
(ii) The strings where the machine halted = {ϵ, a, b}.

(iii) The strings where the machine hanged = {(a+b)(a+b)(a+b)*}

↓

It means, whenever the string length ≥ 2, the machine hanged.

Q:



Q: L(M) = ?

Q: Hang = which languages?

Q: Halt = ,, ,,

⇒ ϵ → BBB

↓

Not accepted

for 'a':

BaB → Accepted

for 'b':

BbB → Accepted

for 'ab':

BabB → No configuration for 'b' on q_1, so halt and reject

for 'aa':

BaaB

$q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_0$ } This will go on, so hang & reject.

for 'ba':

BbaB

$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_0$ } This will hang & reject.

for 'bb':

BbbB

$q_0 \xrightarrow{b} q_1 \xrightarrow{b}$ No configuration, which means halt & reject.

Ans:

(i) Language accepted by the machine = {a, b}

(ii) Language where the machine hangs:

Whenever the second symbol is 'a', it's hanging.

∴ Language = $\{(a+b)\,a\,(a+b)^*\}$

(iii) Language where the machine is halting:

Whenever the second symbol is 'b', it's halting, with that it's halting for ϵ, a, b, [illegible].

∴ Language = $\{(a+b)\,b\,(a+b)^*, \epsilon, a, b, [illegible]\}$

Theory Point:

$L(M) \subseteq halt(M)$

↓ ↓ → Machine

$Halt(M) + Hang(M) = \Sigma^*$

Q: Draw Turing Machine for $a^n b^n \geq 1$.

⇒ Strings can be → ab, aabb, aaabbb, ...

Approach & technique to solve questions like these:

BBB aa bb BBB → Start marking the 1st 'a' as X.

BBBXa bb

→ Then move right until you find a 'b'. After getting the 'b' mark it Y, & move left while ignoring any no. of y's & a's till you find the 1st 'X'.

While moving right ignore all 'a's & 'y's till you find a 'b'.

y y
BBXXa YYb BBB
↓
(a, Y, R)

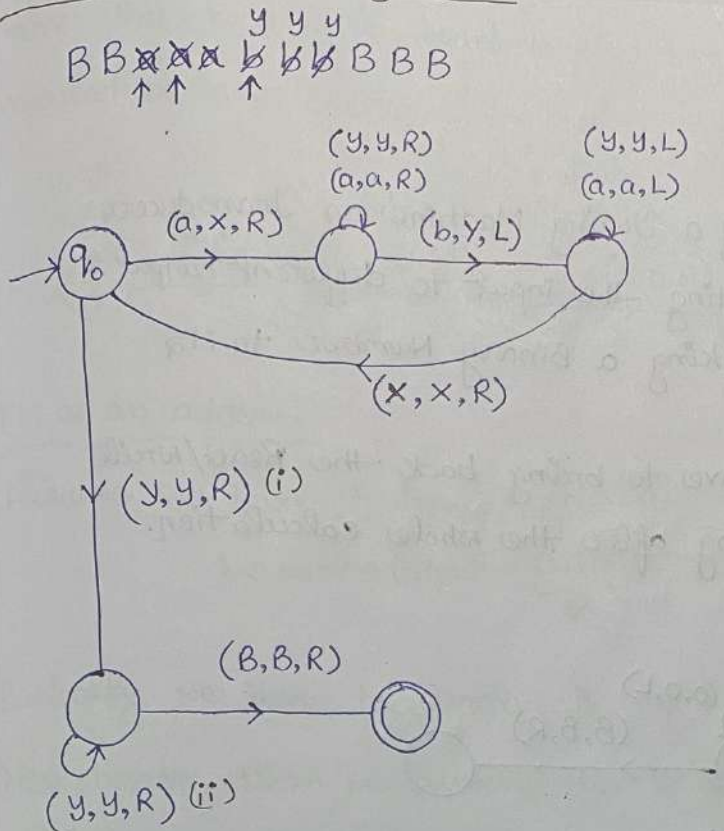
→ Now while coming back from 'b' to 'a', when you see the first 'X', use that to mark the 'a' on the right side of the 'X'.

Proper work flow:

y y y y
B BB X X X X Y Y Y Y B B B B
↑ ↑ ↑ → When you are in the last 'b', you keep moving left, ignore all y's & when you see the first 'X', then you stop there. Here you check that after an 'X', there's a 'y'. Which means all the a's are over. Now go to the right & check if all the b's are over.
If all a's & b's are over, you can accept the string.

(a, X, R)	(b, Y, L)
(a, a, R)	(a, a, L)
(Y, Y, R)	(X, X, R)
	(Y, Y, L)

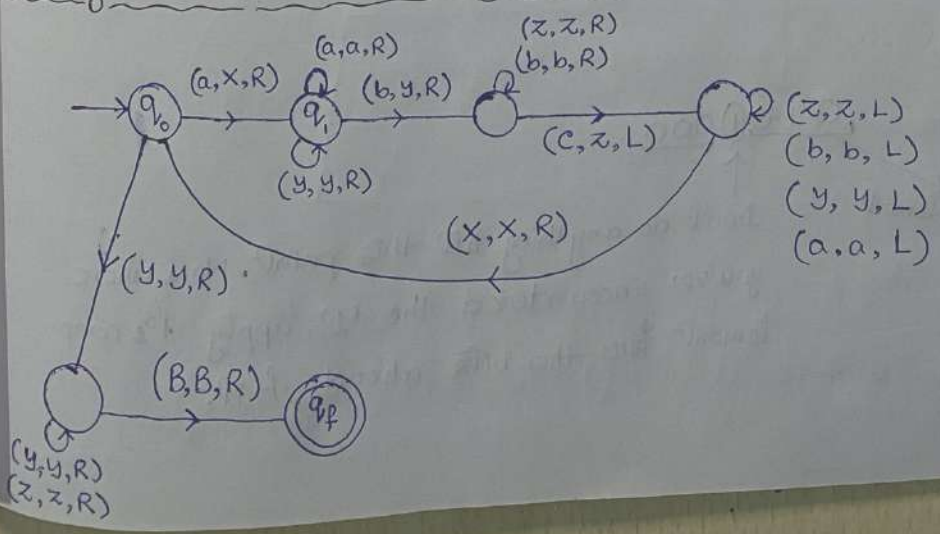
Explanation with diagram:



(i) After seeing the first 'X' using (X, X, R) you've encountered a 'y'. You keep moving to right.

(ii) Using the second (Y, Y, R) we are checking if all the 'y's are over or not.
If 'y's are over then we'll enter the final state.
We'll see a blank after 'y's are over.

Turing Machine for $a^n b^n c^n / n \geq 1$:

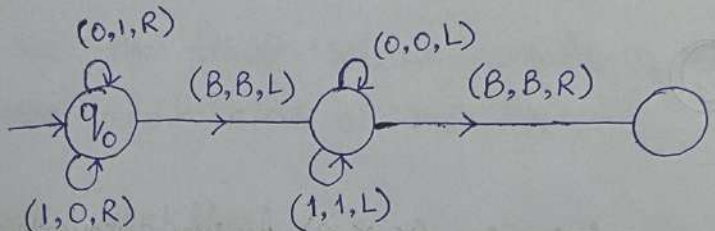


Turing Machine to find 1's complement of a Binary Number:

NOTE:

Whenever you are using a Turing Machine as Transducer, which means converting the input to different output. Like here we are making a Binary Number to it's 1's complement.

In such cases, we have to bring back the Read/Write head to the beginning after the whole calculation.



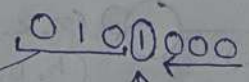
LECTURE: 2

Turing Machine to find 2's complement>

NOTE:

We'll ignore the carry-bit while doing it. Which means we will use the given bits only. No extra bit we'll take.

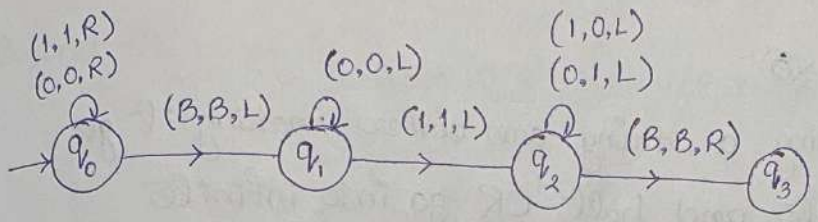
Example:



Do 1's complement here.

Don't do anything till this point. Now, when you've encountered the '1', apply 1's complement for the bits ahead of it.

This TM will act like a transducer. So, we have to move the Read/Write head to the beginning after all the calculations are over.



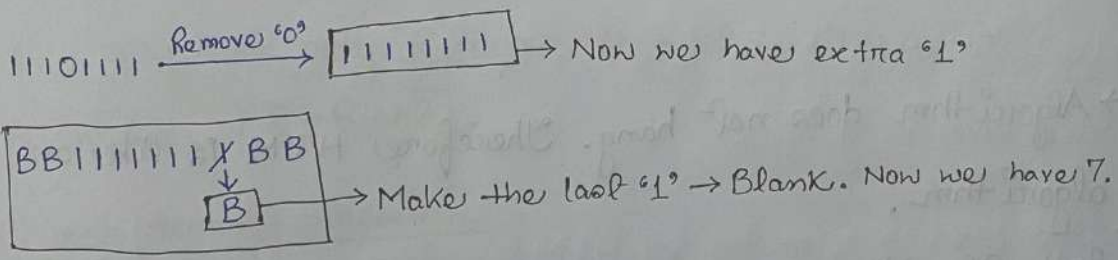
TM as an adder:

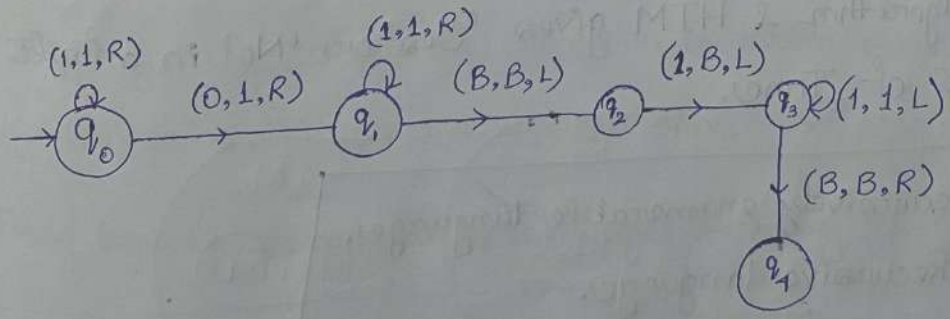
Assume we have → Input = 111011 (3, 1)

We want → Output = 1111111 (7)

Basically we have to remove the '0' from the input.

Now observe, when we remove the '0' by making it '1', we'll have one extra '1'. So, we have to make the last 1 as 'B' to resolve it.





THEORY PART:

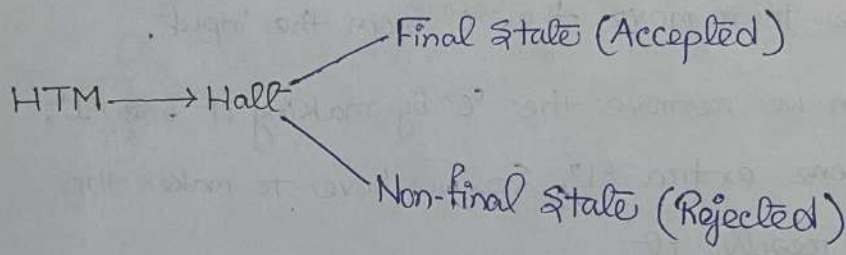
In TM, final states are always dead configurations.
No configuration is given to final state in general.

→ ◎

Two ways of rejecting a string are either making it go into non-final state and halt OR go into infinite loop.

→ Halting Turing Machine: (HTM)

HTM never goes into hang.



→ Algorithm does not hang. Therefore HTM is like an algorithm.
Both Algorithm & HTM gives 'Yes' or 'No' in a finite amount of Time.

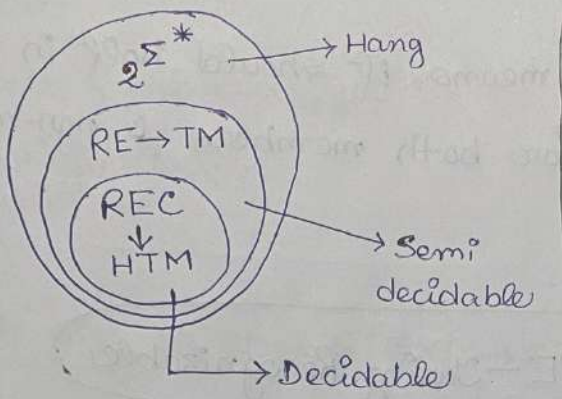
RE = Recursively enumerable language.
REC = Recursive language.

→ RE is the language accepted by Turing Machine.
∴ RE is semi-decidable (Because TM may hang)

REC is the language accepted by HTM.

∴ REC is Decidable. → Because HTM always halts & say → Yes or No.

→ Beyond RE even for yes, machine will hang.



Σ = alphabet.
Σ^* = all strings.
2^{Σ^*} = all languages possible.

→ Given a word 'w' to TM, can we guess whether it will halt or hang?

⇒ Ans: No, this problem is undecidable, which means there is no algorithm.

→ If a TM has only 'R' moves, then it is like a Finite Automata. [R = Right].

Church turing thesis:

→ Every logically computable function is turing computable. Which means, TM can compute $\sin x, \cos x, e^x, x^2, x^p, \ldots$

→ Every logically recognisable language is turing recongnisable.

Logically recognisable language = RE Language.
Turing recognisable language = RE Language.

↓
Alternate ways of saying RE Language.

→ REC is called turing decidable.

→ Recognising a language means recognising the strings in the language, i.e. halting on strings in the language. We don't care about non-member.

→ Turing decidable means, it should halt in finite amount of time for both members & non-members of the language.

Turing Machine – RE – Turing Recognizable

↓

All are same

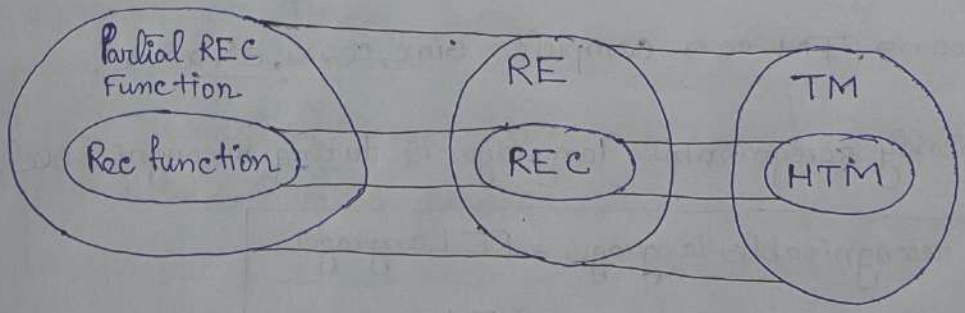
Halting Turing Machine – REC – Turing Decidable

↓

All are same

→ Logically recognisable function is also called partially recursive language.

→ The function computed by HTM is called recursive function.



→ Whatever turing machine can do, computer can ~~do~~ do only that. Whatever TM cannot do now, computer cannot do it even after 1000 years, as both TM & Computer are based on the same logic.

Variations of Turing Machines:

TM is → FA + R/W Head + Right ↔ Left movement.

Standard TM: ≅ DTM

1) It is deterministic TM.
2) R/W Head
3) Can move Right ↔ Left.
4) It has single tape.

NOTE:
DTM ≅ NTM

Variations >

1) TM with stay option → $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$

It can move left, it can move right or it can stay.

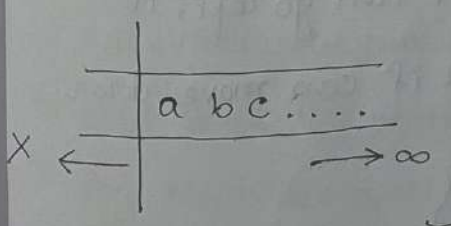
'Stay' means:

Sometimes when you are looking at an input, it'll stay there only. It won't move left or right.

This TM with stay option is equivalent in power when compared with Standard Turing Machine. It's not going to add any power.

> TM with stay option ≅ Standard Turing Machine

2) TM with semi infinite tape:

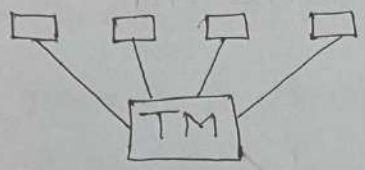


On the right side it is infinite, but on the left side, it's not infinite. This is semi infinite tape.

> TM with semi infinite tape ≅ Standard Turing Machine

3) Mulli Tape TM:

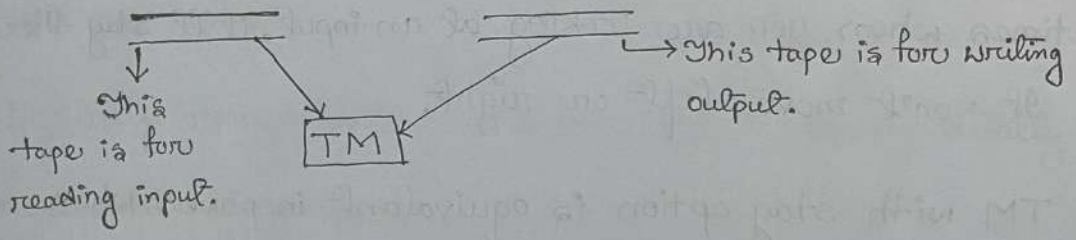
It means, you have a Turing Machine and there are mulliple tapes attached to it & it'll be reading & writing on mulliple tapes.



Mulli Tape TM ≅ Standard Turing Machine

4) Offline Turing Machine:

It means there will be Turing Machine and there'll be two tapes. One tape is for reading & the other one is for writing.

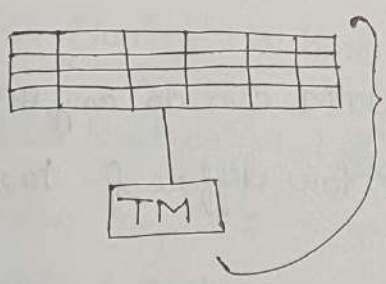


Offline Turing Machine ≅ Standard Turing Machine

5) Mullidimensional Turing Machine:

It means a turing machine will have a tape which has many dimensions. Therefore, the TM can go left, it can move right, it can move up & it can move down.

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$



Mullidimensional TM ≅ Standard Turing Machine

6) Non-deterministic TM:

This is same as Deterministic Turing Machine.

NTM ≅ DTM

7) Universal Turing Machine:

Here Turing Machine will take another Turing Machine as input & then it'll take what is the input that you have to trace on this Turing Machine.

Indetails Explanation>

Assume there are three TM's, one is adder, next one is doing subtractor & last one is multiplier. The adder can only add. And same goes for the other two. Now to do multiple things they introduced a machine called Universal Turing Machine.

Using UTM for the adder TM: (Same will happen with the other TMs)

UTM has total 3 tapes. In the first tape it will take the adder TM as binary inputs. In the second tape it will take actual inputs like a, b and it'll try to add 'a' & 'b' using the logic present in the adder Turing Machine and in the last tape it'll try to maintain the current state of the adder TM.

NOTE:

Just like the adder TM, the UTM can do anything. We just have to reprogram it for different tasks.

UTM is reprogrammable.

Universal Turing Machine ≅ Standard Turing Machine

4 more variations which are less powerful than STM:

1) HTM < TM

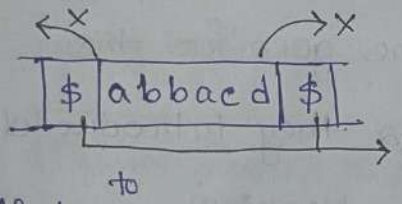
HTM = REC
TM = RE

↓
Always halts

2) LBA < HTM < TM

LBA = CSL
HTM = REC
TM = RE

In LBA, tape is bounded.



It has to use the space available b/w the two '$'s.

NOTE:

Tape size in LBA is not fixed. It depends on the input size.

The working space of LBA is limited or dependant to the size of the input.

3) Read only Turing Machine:

FA + Read head only + Left ↔ Right movement.

So, we can say, it's simply a Finite Automata which can move left & right.

This is also called 2 way Finite Automata.

> Read only TM or 2 way FA < Standard Turing Machine

4) One way Turing Machine:

FA + Read/Write head + Right movement only

This is equivalant to Finite Automata, hence less in power than Standard Turing Machines.

REC and RE

Definition: A language 'L' is RE if ∃ TM which accepts 'L'.

RE = Turing Recognizable Language

↓

It means the TM is accepting the members. We don't care about non-members. For non-members it may halt or hang.

Definition of REC:

'L' is REC if ∃ TM that ~~accp~~ accepts 'L' and which halts for all $\forall w \in \Sigma^*$.

Which means, whatever input you give, it has to halt.

> Turing Recognisable = RE = Formal Language = Type '0' language.
>
> Decidable = Turing decidable = REC

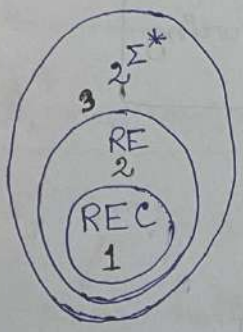
LECTURE: 3

GATE Q:

For a language there is no HTM but has TM, what is the language?

⇒ RE but not REC.

Properties of RE & REC:



1 → REC
1+2 → RE
3 → Not RE
2+3 → Not REC, so not decidable
2 → RE but not REC

Enumuration Procedure:

L is RE, if 'L' has turing enumuration procedure.

So, RE are **turing enumurable.**

What is enumuration:

To list a language all you need to do is, list the members of the language. We need not list all the non-members.

"L" is countable: if 'L' has enumuration procedure.

Set of Real Numbers are → Uncountably infinite.
Set of Natural Numbers are → Countably infinite.

Real numbers are not countably infinite, because we can't list all Real Numbers.

Example: 1 to 2 there are a lot of numbers possible.

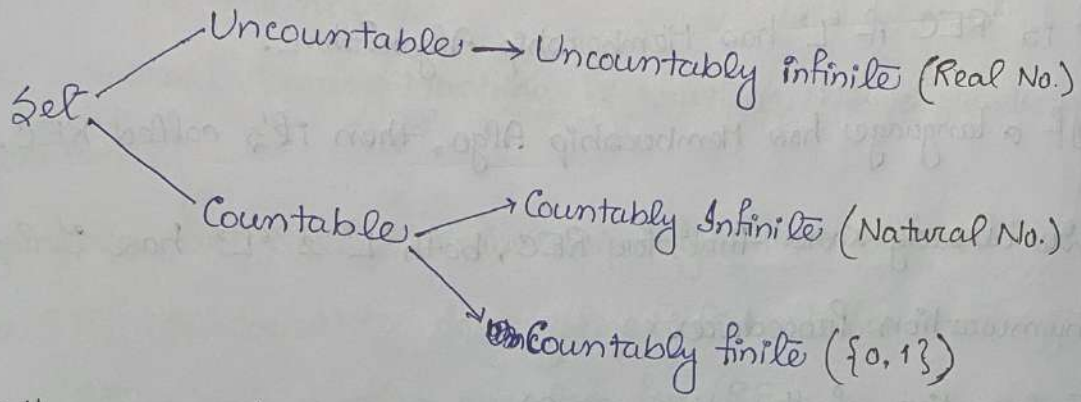
1, 1.00001, 1.00010, 1.10000 ..... 2 → We can't list them.

Natural Numbers are countable because we can list them.

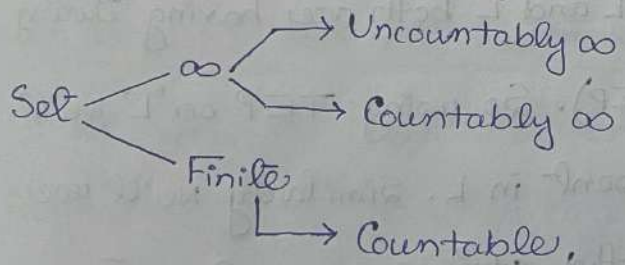
Example: 1 1+1 1+1+1 1+1+1+1 .....

Therefore, Natural Numbers are countably infinite.

We can classify any "set" in 3 types:



Another way of classifying "set":



Theory Points:

→ 'L' is recursive if L and $\bar{L}$ both has turing enumuration procedure.

→ Some language is turing enumurable but it's complement is not.

So, we can say → Language is RE but not REC.

→ If a language is turing enumurable, then it is RE.

→ If a language is not turing enumurable, then it's not RE.

Membership Algo:

→ Enumuration Procedure has to list all the members, but Membership Algo has to list all the members, as well as non members.

→ 'L' is REC if 'L' has Membership Algorithm.

→ If a language has Membership Algo, then it's called REC.

→ We already know that, for REC, both 'L' & '$\bar{L}$' has Turing Enumuration Procedure.

Explanation of the above point>

We know that for REC, L and $\bar{L}$ both are having Turing Enumuration Procedure (TEP). So, using TEP on 'L' we'll get all the strings present in L. Similarly we'll use TEP on '$\bar{L}$' to get all the strings present in $\bar{L}$. For Membership algo we need both members and non-members. Using TEP we are able to get that with REC. So, now TEP has created a Membership Algo for REC.

Hence we can say REC languages are having Membership Algorithm. So, they are decidable.

→ L is REC if L has Membership Algorithm.

Lexicographic Order:

'L' is REC if L can be enumurated in Lexicograpic Order.

Example:

$\{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$ → lexicographic order.

Dictonary follows this.

If we have a universal turing machine, we can give the 'Machine' to the UTM along with $\epsilon, 0, 1, 00, \ldots$ one by one and find the members.

Here Universal Turing Machine is working like a Membership Algorithm.

Therefore, for REC, Membership Algorithm exists. But for RE, Membership does not exist because there will be halting or hanging problem for RE.

Reason why RE does not has Membership Algo:

So, if RE language has to be enumurated, but not Lexicographically. Then we can start on all string in parallal by using the Turing Machine. But we'll not get the lexicographic order, but we will get some order.

We won't be getting result for all the strings. Hence we can say RE does not has Membership Algorithm.

Closure Property:

If 'L' is REC it implies $\bar{L}$ is also REC. Because we already know that REC is closed under complement.

We can also say → If L is decidable then $\bar{L}$ is also decidable in case of REC.

→ If 'L' is RE then $\bar{L}$ may or may not be RE.

Because RE is not closed under complement.

GATE Q:

If there are two complementary languages L & $\bar{L}$, which of the following is not possible?

a) Both are decidable.

b) Both are undecidable.

c) One is decidable and other one is undecidable. ✓

L and $\bar{L}$ Theorem:

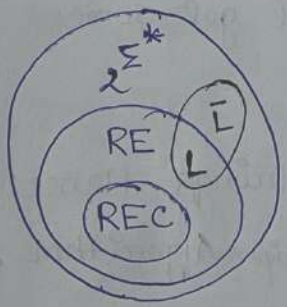
If you look at closure properly, then →

'L' is RE ⇒ $\bar{L}$ may or may not be RE.

'L' is REC ⇔ $\bar{L}$ is REC.

If 'L' & '$\bar{L}$' both in RE, then only possibility is, that L and $\bar{L}$ are REC.

Other meaning of the above theorem:



In this case if L & $\bar{L}$ are in RE, then it'll make RE closed under complementation. Which is not possible in case of RE.

→ If L is RE but not REC then $\bar{L}$ is not RE.

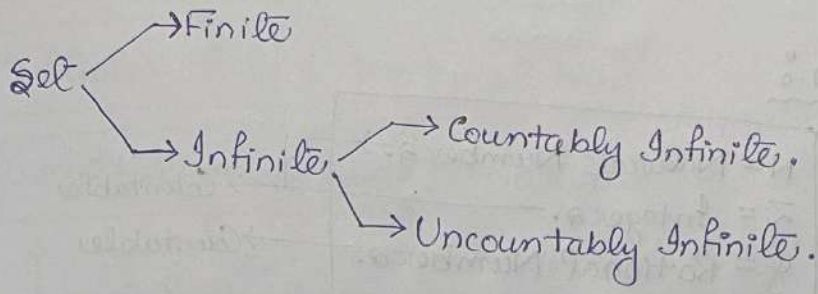
⇒ $\bar{L} \neq$ RE

→ L is not RE ⇒ $\bar{L}$ is not RE.

→ L is RE then $\bar{L}$ is RE or not RE.

→ If L is not REC, then $\bar{L}$ is not REC.

Countability :



→ A Set 'S' is countable if 'S' has enumuration procedure.

Example :

Natural No. = 1, 2, 3, 4, 5, ...

1, 1+1, 1+1+1, 1+1+1+1, ...

∴ Countably Infinite.

Real No. = 1, ..., 2

This part is infinite. ∴ Uncountably ∞.

Enumuration Procedure :

EP is a procedure which lists a set with a guarantee that any particular element should be listed in a finite amount of time.

Example : In case of Natural Numbers if we want to find till 100, then we can do it in finite amount of time. Which means, enumuration procedure is there in Natural Numbers.

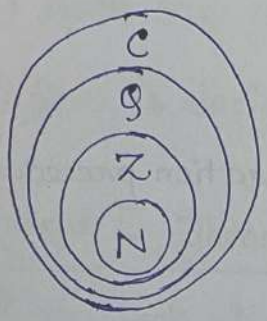
But in case of Real Numbers, enumuration procedure is not there.

Natural No.
$N = \{1, 2, 3, 4, \ldots\}$
Discrete sets

Real No.
$R = \{1, 1.0001, 1.0002, \ldots\}$
Continuous sets

∴ Countably infinite sets are always discrete.
Uncountably " " " " continuous.

Number System:



N = Natural Numbers. → Countable
Z = Integers. → Countable
Q = Rational Numbers.
R = Real Numbers.
C = Complex Numbers.

Rational Numbers:

$\frac{1}{1}\ \frac{1}{2}\ \frac{1}{3}\ \frac{1}{4}\ \frac{1}{5}\ \ldots\ldots$
$\frac{2}{1}\ \frac{2}{2}\ \frac{2}{3}\ \frac{2}{4}\ \frac{2}{5}\ \ldots\ldots$
$\frac{3}{1}\ \frac{3}{2}\ \frac{3}{3}\ \frac{3}{4}\ \frac{3}{5}\ \ldots\ldots$

If we go in diagonal we'll cover all the rows and every number will occur after finite no. of steps.

∴ We are able to give an enumuration procedure.
So, we can say that, it's **countably ∞**.

→ If $A \subseteq B$ and B is countably ∞ or finite, then A is also countably ∞ or finite.

$Z \subseteq Q$ → We saw that 'Q' is countably ∞. So, we can say 'Z' is also countably ∞.

...

→ We know that Prime No. are subset of Natural Numbers & Natural No. are countably infinite. Hence, we can say **Prime No. are also countably infinite.**

Same applies for the below:

i) **Multiple of 3** ⊆ Natural No.

ii) **Composite numbers** ⊆ Natural No.

→ Both these two are countably infinite.

Cantor's Diagonal Argument:

It is used to prove Real Numbers are countably infinite.

The Argument>

Write down all the Real Numbers and say that these are the only Real Number possible.

Now, we'll take a diagonal & now we'll chang each number to 9's complement. In this way we'll get a set of new numbers which are not in there.

Example:

0.1 2 3 4 0 0 0
0.1 2 3 4 0 0 1
0.1 2 3 4 0 1 0
0.1 3 4 5 0 0 0

⇒ 1 2 3 5
↓ ↓ ↓ ↓
8 7 6 4 → 9's complement, which is a set of new no.

Position wise values are changed.

9's complement = 9 − given number

∴ Real no. are uncountably infinite.

Complex Numbers:

These are combination of two Real Numbers.

Example:

$a+ib$ → We can write it as (a, b)

a → Real Number, b → Real Number

Real Numbers are uncountably infinite. So, multiplication of two Real Numbers will give us uncountably infinite only.

UC → Uncountable. CI = Countably Infinite.
C → Countable.

$C \times C = C$	$2^{CI} = UCI$
$UC \times UC = UC$	
$UC \times C = UC$	

Class → $CI = N, Z, Q, Z \times Z$

Class → $UCI = R, C, R \times R, 2^N, 2^Q, 2^Z$, Irrational Numbers.

C = Countable

$C \cup C \cup C \cup C \ldots = $ Countable.

Cantor's Theorem:

'S' is Countably infinite implies 2^S is uncountably infinite.

'S' is countably finite $\Rightarrow 2^S$ is countably finite.

→ This part is Irrational Numbers. (IR)

$Q \cup IR = R$

$\downarrow \quad \downarrow$

$CI \cup \{\} = UCI$

We already know, that $CI \cup UCI = UCI$.

∴ IRs are Uncountably Infinite.

LECTURE: 4

Now we want to see which languages are Countably infinite, Countably finite and Uncountably infinite.

1) Σ^* is always Countably Infinite. Σ^* = Set of all strings possible over 'Σ'.

How to prove it?

→ We have to give Enumuration procedure to prove that something is Countably Infinite.

Proof:

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

↳ Enumuration Procedure → Generate all the strings in increasing order of length. This is called Proper Ordering

$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$

↓

We got one length, two length strings and other will be there also. So, Σ^* has enumuration procedure.

$\therefore \Sigma^*$ is Countably Infinite.

NOTE:

Every language is subset of Σ^*.

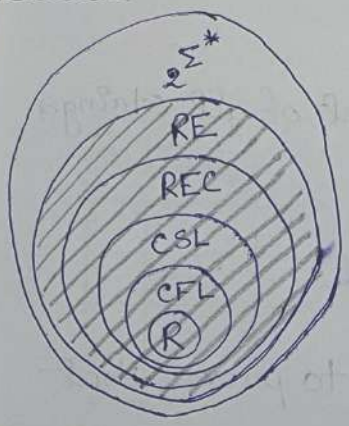
∴ Every language is countably infinite or countably finite.

Example: $L_1 = \{a, b\}$ → Countably Finite

$L_2 = a^*b^*$ → Countably Infinite

Set of all Regular languages, set of all CFL, Set of all CSL, Set of all REC, Set of all RE are all countably infinite.

Proof:



→ If we prove this whole part is countably infinite then the other languages inside it, will also be countably infinite.

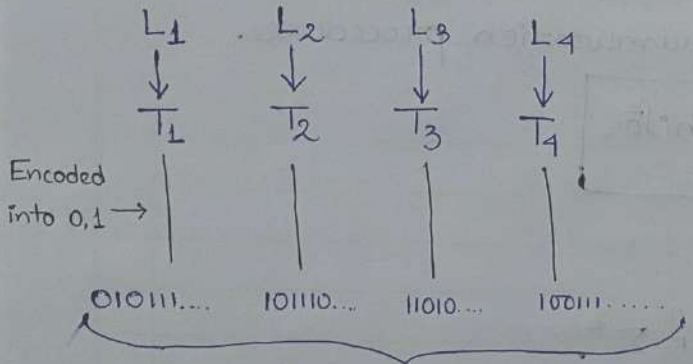
First we need to show that set of all RE languages is Countably Infinite.

Every RE language has a Turing Machine.

* Set of all RE $(L_{RE}) = \{L_1, L_2, L_3, L_4, \ldots\}$

Now every language will have a turing machine. Now every Turing Machine can be encoded into 0's & 1's.

Diagram>



Now if we put together all these Turing Machines, it'll become a language.

$\therefore L_{TM} = \{010111\ldots, 101110\ldots, 11010\ldots, 100111\ldots\}$

Now, this $L_{TM} \subseteq \Sigma^*$

If a language is a subset of Σ^*, then we can say the language is countably infinite. Because Σ^* is countably infinite, which we already know.

$\therefore L_{TM}$ is Countably Infinite. Therefore we can say →

Set of all RE languages is Countably Infinite.

Hence proved →

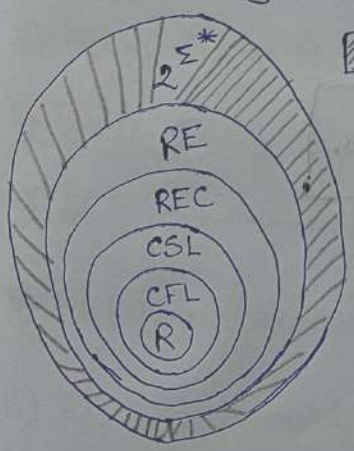
L_{RE}, L_{CFL}, L_{CSL}, $L_{Regular}$, L_{DCFL} ⇒ All are Countably Infinite.

Now we want to prove that, Set of all languages possible 2^{Σ^*} is Uncountably Infinite :

Cantor's Theorem :

It says that → If 'A' is Countably Infinite, then '2^A' is Uncountably Infinite.

$\therefore \Sigma^*$ is Countably Infinite. Therefore, 2^{Σ^*} is Uncountably Infinite.



'L' of → This part is not RE

2^{Σ^*} → UCI

L_{RE} → CI

$L_{RE} \cup L_{not\ RE} = 2^{\Sigma^*}$

↓ CI ↓ UCI

$\therefore$ CI ∪ UCI = UCI

$\therefore L_{not\ RE}$ = UCI

Now, we can say →

$L_{not\ RE}$, $L_{not\ REC}$, $L_{not\ CSL}$, $L_{not\ CFL}$, $L_{not\ Reg}$ = UCI.

L_u and L_d :

L_u = Universal language of Turing Machine.

L_d = Diagonalization of Turing Machine.

$$L_u = \{e(T)/e(T) \in L(T)\}$$

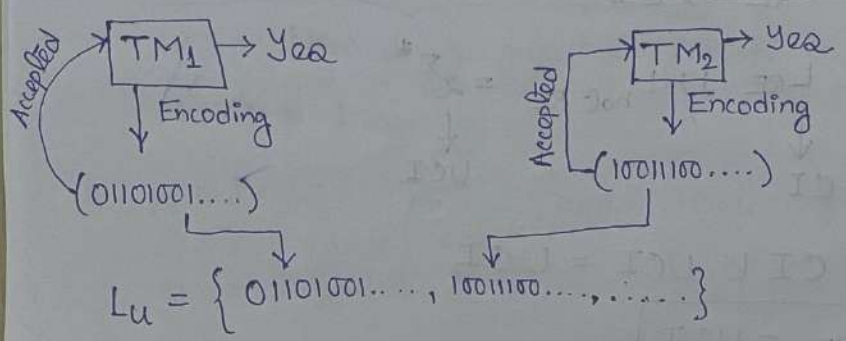
L_u is encoding of all the Turing Machines such that, the encoding of the Turing Machine belongs to the language accepted by the Turing Machine.

Example & Explanation :

If we take one or more TM, and start encoding them in 0's and 1's, we'll get strings which will be accepted by the same Turing Machines.

Now, with encoded strings we'll make a language which will be a subset of Σ^*.

∴ We can say L_u is Countably Infinite.



$L_u \subseteq \Sigma^*$. ∴ L_u is Countably Infinite

NOTE:

→ L_u is RE but not REC.

→ L_u is self accepting.

NOTE: If we take a single language from 2^{Σ^*}, then that language will be CI.

But if we take all languages of 2^{Σ^*}, then they'll be Uncountably Infinite.

L_d:

$L_d = \{e(T)/e(T) \notin L(T)\}$ → Non-self accepting.

encoding of

L_d is set of all Turing Machines, such that the encoding of Turing Machine does not belong to Language of the Turing Machine.

L_d is subset of Σ^* because L_d is combination of 0's & 1's.

∴ L_d is Countably Infinite.

Undecidable Problems in Turing Machine:

1) Halting Problem.
2) Blank Tape Halting Problem.
3) State Entry Problem.
4) Post Correspondence Problem.
5) Modified Correspondence Problem.
6) RE Membership.

All these problems are undecidable.

They are all RE but not REC.

RE, because we can make Turing Machine for them but not HTM.

1) Halting Problem:

Given a Turing Machine & a word (w), where $w \in \Sigma^*$. Does ~~$w \in L(TM)$~~ turing machine halts on "w"?

~~Language of Turing Machine~~

This is RE but not REC. Therefore, it's undecidable.

2) Blank Tape Halting Problem:

Which means, when blanks are given as the input to a Turing Machine, will the Turing Machine halts?

This problem is also undecidable.

3) State Entry Problem:

Given a Turing Machine "T", $w \in \Sigma^*$ and a state "q", will the turing machine enter q while processing "w"?

This problem is also undecidable.

4) Post Correspondence Problem:

Given two sets of substrings → (i) $(u_1, u_2, u_3, \ldots u_n)$

(ii) $(v_1, v_2, v_3, \ldots v_n)$, then is there any combination such that →

$$u_i u_j u_k = v_a v_b v_c$$

This problem is also undecidable.

5) Modified Post Correspondence Problem :

If you have two strings → (i) $(u_1 u_2 u_3 \ldots)$

(ii) $(v_1 v_2 v_3 \ldots)$, now can you find a sequence where →

$$u_1 u_i u_j \ldots = v_1 v_a v_b \ldots$$

This problem is also undecidable.

6) RE Membership :

$\overset{?}{w} \in L(T(M))$?

This is also undecidable.

Undecidable Tabe for RE : [The Big Table is in previous notes]

	RE
Membership	X
Emptiness	X
Finiteness	X
Equivalance	X
Regular or not	X
Ambiguous	X
Completeness	X
Disjoint	X

X → Undecidable

Rice's Theorem:

Every non-trivial question on RE are undecidable (or) only trivial questions on RE are Decidable.

Trivial question means → always yes or always no.

Non-trivial question means → Sometimes yes or sometimes no.

Reduction Theorem:

It means whenever a language L_1 is reducable to language L_2. If all strings of L_1 can be mapped to all strings of L_2, then we say that, L_1 is reducable to L_2.

If L_1 is Reducable to L_2>

In this case if →

L_1 is UD then L_2 is UD.

L_2 is decidable then L_1 is decidable.

L_2 is REC then L_1 is REC.

L_2 is Semi-decidable then L_1 is Semi-decidable.

L_2 is RE then L_1 is RE.

Tricks to remember the above Points>

$L_1 \leq L_2$ → Reducable

'⇒' forward direction

'⇐' backward direction

1) UD ⇒ UD

2) Decidable ⇐ Decidable

3) REC ⇐ REC

4) Semi-decidable ⇐ Semi-decidable

5) RE ⇐ RE

Whenever "p" implies "q" is true. $(P \rightarrow Q)$

Then $\boxed{\sim Q \rightarrow \sim P}$ is also true.

↓ This is called contrapositive.

$L_1 \leq L_2$	Negation Relation (Contrapositive)
UD ⇒ UD	DEC ⇐ DEC
DEC ⇐ DEC	UD ⇒ UD
REC ⇐ REC	Non REC ⇒ Non REC
SD ⇐ SD	Non SD ⇒ Non SD
RE ⇐ RE	Non RE ⇒ Non RE

1:39:00

Q: If $L_1 \leq L_2$ and if L_1 is decidable, what can we say about L_2 ?

→ We can't say anything about L_2, because 'decidable' is backward.

If L_2 is dec then we can say L_1 is decidable. Vice-versa is not possible.

Q: If $L_1 \leq L_2$ & $L_2 \leq L_3$, then $L_1 \leq L_3$. In the above case L_1 is undecidable. Then L_3 is, ?

→ Undecidable is forward direction. Therefore, L_3 is also undecidable.

Q: $L_1 \leq L_2$ and $L_2 \leq L_3$. L_2 is RE but not REC.

→ L_2 has two part → 1st one = RE, 2nd one = not REC which means undecidable.

∴ L_1 is RE and L_3 is undecidable.

Q: $L_1 \leq L_2$. If L_2 is REC then L_1 is RE. True / False.

→ It's true.

L_2 is REC & and REC is decidable. Decidable goes backwards, so, L_1 is REC. And we know that if a language is REC then we can say it's RE also.

Q: $L_1 \leq L_2$. If L_2 is RE then L_1 is REC. True or false.

→ It's false.

You can go up ↑ which means ⇒ REC ↑ = RE ✓

But you can't go down ⇒ RE ↓ ✗

Q: $L_1 \cup L_2 \leq L_3$. $L_3 = ?$ $L_1 = REC$, $L_2 = RE$.

→ REC ∪ RE

= REC ↑ ∪ RE

= RE ∪ RE

= RE

But RE works in backwards, so we can't say anything about L_3.

Q: $L_3 \leq L_1 \cup L_2$; $L_1 = REC$ & $L_2 = RE$. $L_3 = ?$

→ REC ∪ RE

= REC ↑ ∪ RE

= RE

RE works in backwards, so L_3 is RE.

LECTURE: 5

Unrestricted Grammars:

This grammar corresponds to Turing Machine. The language is called Recursively Enumurable language.

Definition >

A Grammar is called unrestricted if all the productions are of form "$u \rightarrow w$", where 'u' is $\underbrace{(V \cup T)^+}$, Left side should be atmost one symbol.

'w' is $(V \cup T)^*$.

- V → Variables
- T → Terminals

Left side should be atmost one symbol.

- Variable (or) Terminal

What does the following unrestricted grammar?

$S \rightarrow S_1 B$

$S_1 \rightarrow a S_1 B$

$bB \rightarrow bbbB$

$aS_1 b \rightarrow aa$

$B \rightarrow \epsilon$

$\Rightarrow S \rightarrow S_1 B$

$\Rightarrow a^n S_1 b^n B$

$\Rightarrow a^{n-1} a S_1 b b^{n-1} B$

$\Rightarrow a^{n-1} aa b^{n-1} B$

$\Rightarrow a^{n+1} b^{n-1} B$

$\Rightarrow a^{n+1} b^{n-2} bB$

$\Rightarrow a^{n+1} b^{n-2} bbbB$

Now we can write:

$S \rightarrow a^{n+1} b^{n-2+K} / K = 1, 3, 5, \ldots$

$n = 1, 2, 3, \ldots$

Context Sensitive Grammars:

A grammar is said to be context sensitive if all productions are of form $x \to y$, where $x, y \in (V \cup T)^+$ and $|x| \leq |y|$.

'ϵ' is not allowed.

CSL for $a^n b^n c^n / n \geq 1$:

$S \xrightarrow{1} abc \;/\; S \xrightarrow{2} aAbc$

$Ab \xrightarrow{3} bA$

$Ac \xrightarrow{4} Bbcc$

$bB \xrightarrow{5} Bb$

$aB \xrightarrow{6} aa \;/\; aB \xrightarrow{7} aaA$

$\Rightarrow$

$S \xRightarrow{2} aAbc$

$\xRightarrow{3} abAc$

$\xRightarrow{4} abBbcc$

$\xRightarrow{5} aBbbcc$

$\xRightarrow{7} aaAbbcc$

$\xRightarrow{3} aabAbcc$

$\xRightarrow{3} aabbAcc$

$\xRightarrow{4} aabbBbccc$

$\xRightarrow{5} aabBbbccc$

$\xRightarrow{5} aaBbbbccc$

$\xRightarrow{6} aaabbbccc$

Q: $L = \{a^n b^m c^n d^m / n, m \geq 1\} \longrightarrow CSL$.

⇒ Grammar:

$S \longrightarrow aAcD / aBcD$

$A \longrightarrow aAc / aBc$

Not Required

$Bc \longrightarrow cB$

$Bb \longrightarrow bB$

$BD \longrightarrow Ed$

$cE \longrightarrow Ec$

$bE \longrightarrow Eb$

$aE \longrightarrow ab$

$BD \longrightarrow FDd$

$cF \longrightarrow Fc$

$bF \longrightarrow Fb$

$aF \longrightarrow abB$

Chomsky hierarchy:

Type 0 → Unrestricted Grammars

Type 1 → Context Sensitive Grammars

Type 2 → Context Free Grammars

Type 3 → Regular Grammars