Finite State Machine (Prerequisites)

Symbol - a, b, c, 0, 1, 2, 3, ...

Alphabet - Σ - collection of symbols - Eg. {a, b}, {d, e, f, g}
{0, 1, 2} ....

String - sequence of symbols. Eg.
a, b, 0, 1, aa, bb, ab, 01, ..

Language - set of strings

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

L_1 = set of all strings of length 2

= {00, 01, 10, 11}

Σ^3 = set of all strings of length 3 : $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Σ^n = set of all strings of length n

Cardinality - number of elements in a set

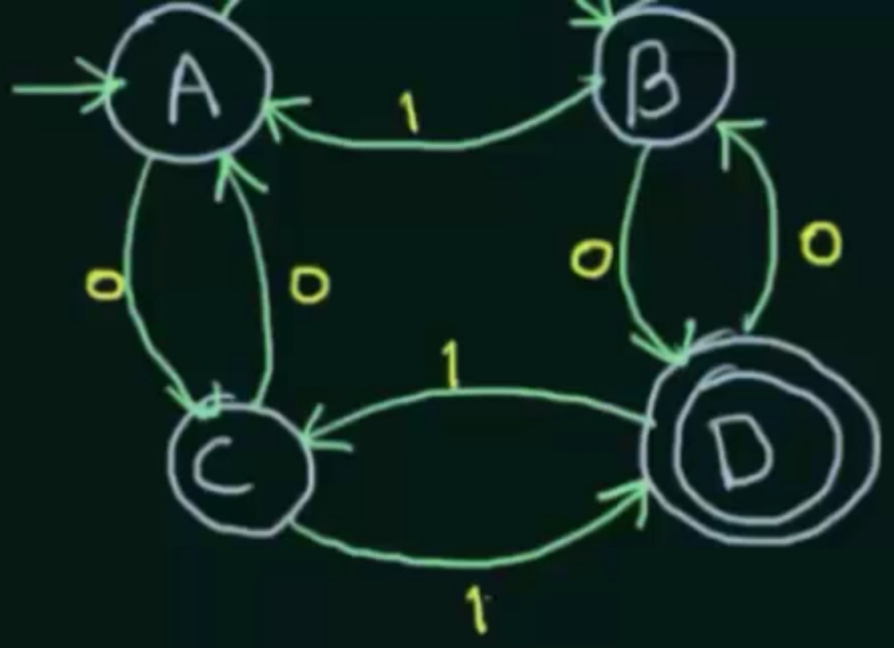
↳ $\Sigma^n = 2^n$

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

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

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

↳ infinite.



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

Q = set of all states

Σ = inputs

q_0 = start state / initial state

F = set of final states.

δ = transition function from $Q \times \Sigma \to Q$

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

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

$q_0 = A$

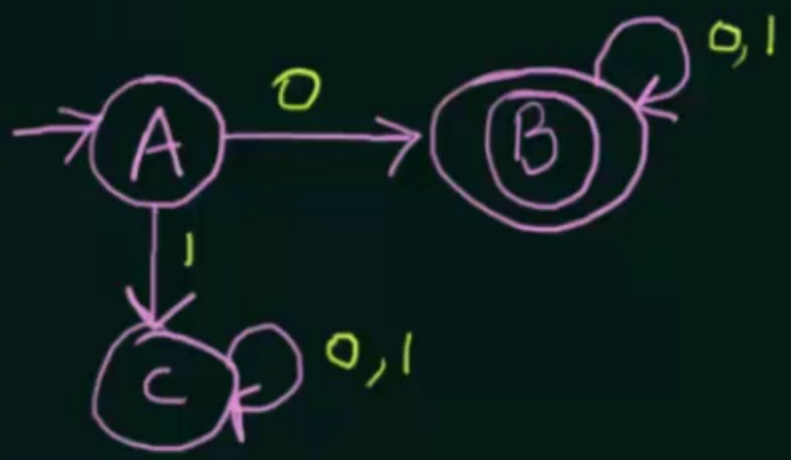
$F = \{D\}$

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

Deterministic Finite Automata (Example-1)

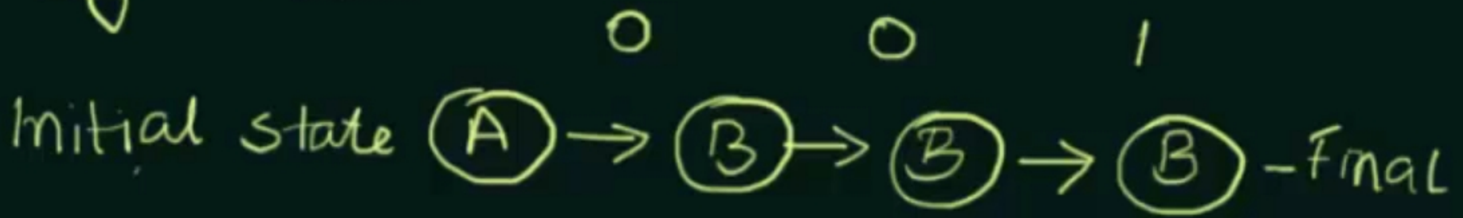
L1 = Set of all strings that start with '0'

$= \{0, 00, 01, 000, 010, 011, 0000, \ldots\}$

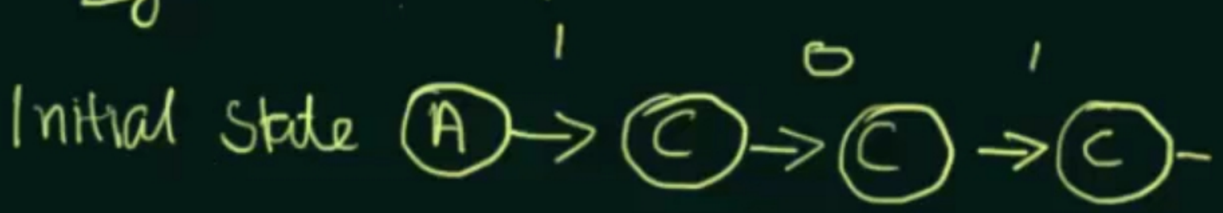


Dead State
or
trap State

Eg. 001 ✓



Eg. 101 ✗



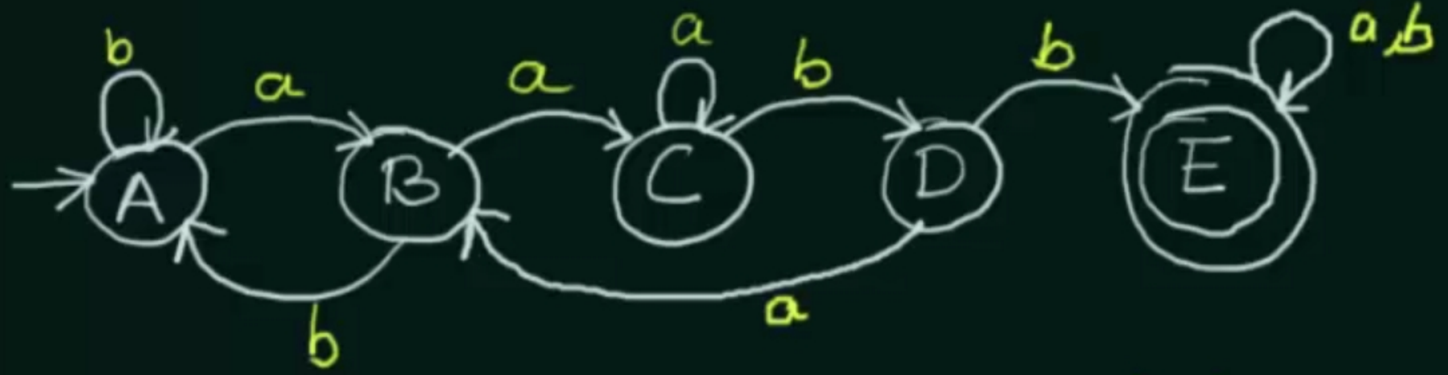
Deterministic Finite Automata (Example-3)

Construct a DFA that accepts any strings over {a,b} that does not contain the string aabb in it.

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

Try to design a simpler problem

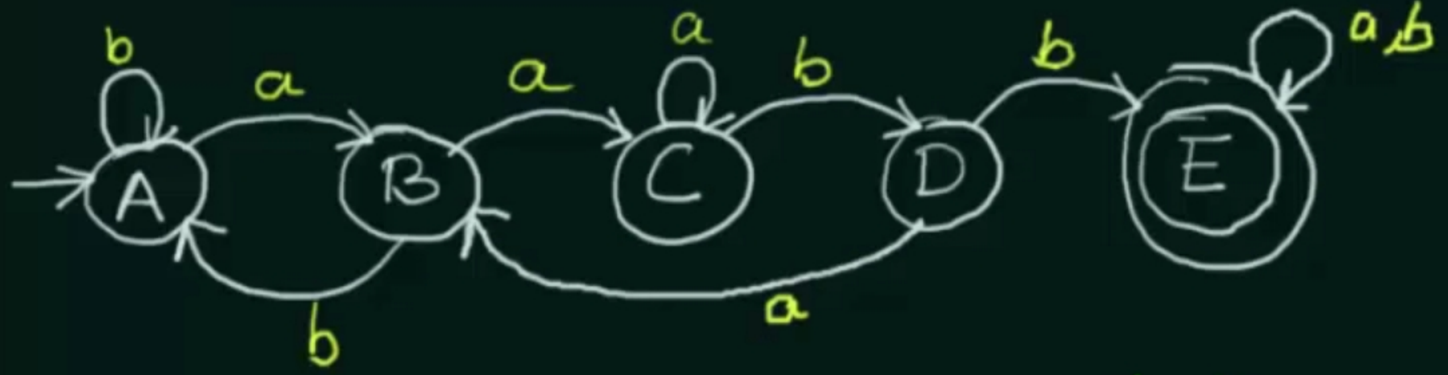
Let us construct a DFA that accepts all strings over {a,b} that contains the string aabb in it



aaabb

aaba

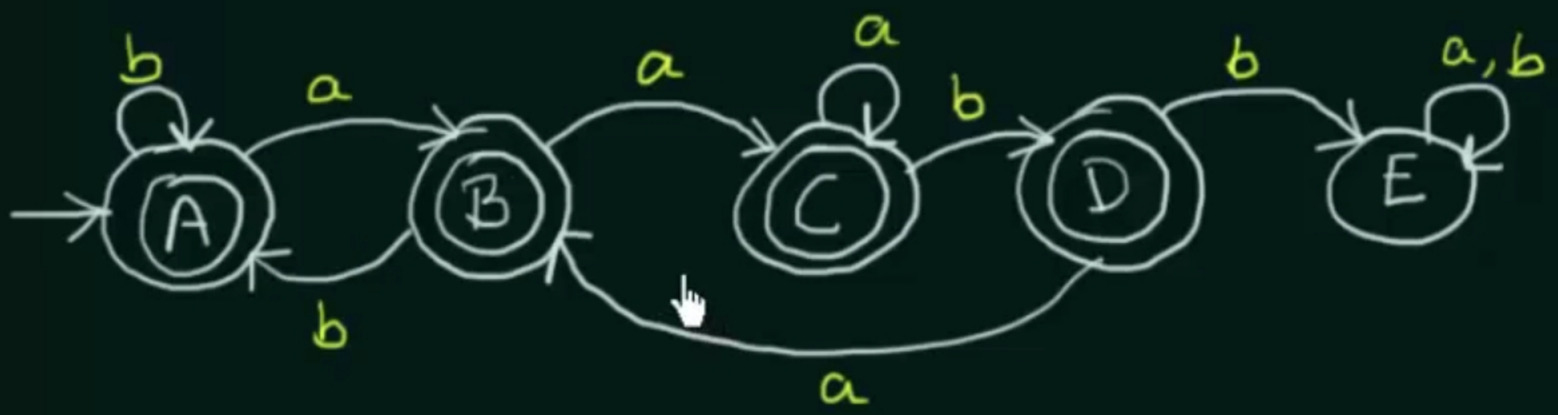
aabb in it

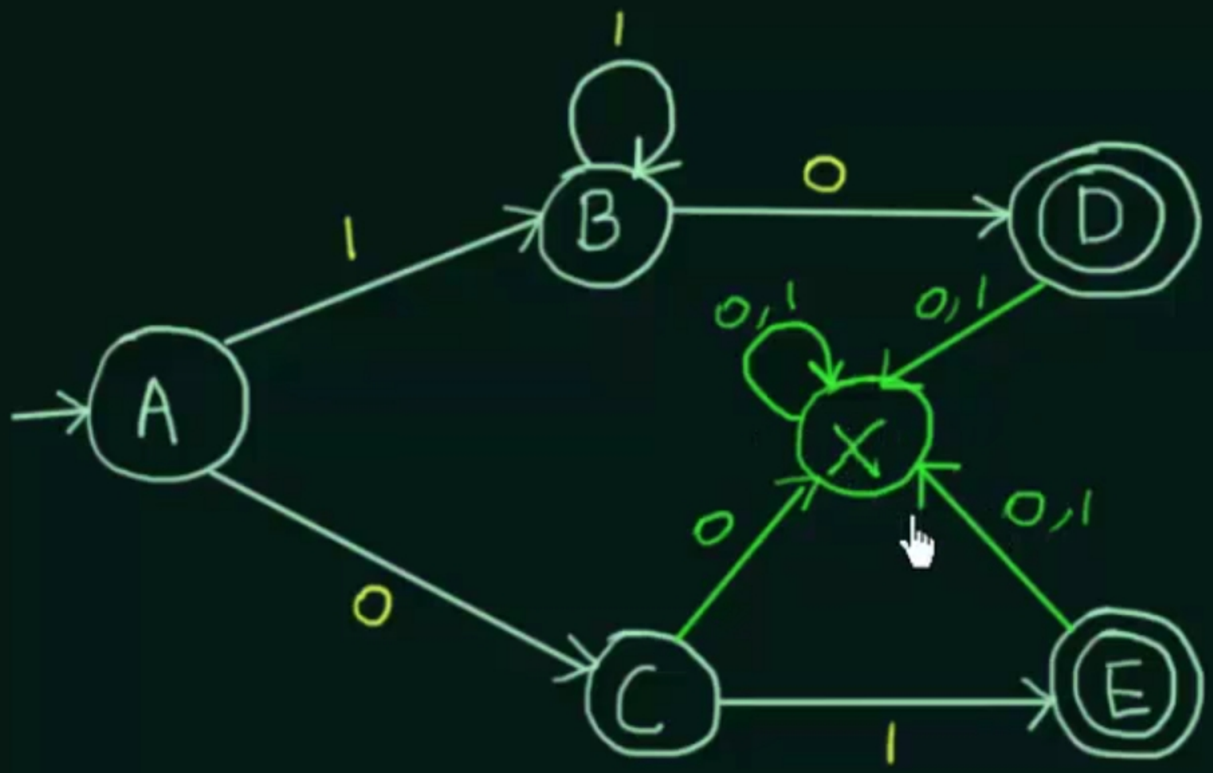


aaabb

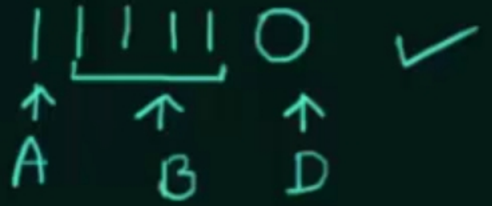
aaba

- Flip the States
- Make the Final State into non final state
and
- Make the non final states into final states





10 ✓



✓

01 ✓

one binary digit '1'

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

Eg. 001 , 010 , 011, 1101, 1100

X- Dead state

Regular Languages

- A language is said to be a REGULAR LANGUAGE if and only if some Finite State Machine recognizes it

So what languages are NOT REGULAR ?

The languages
>> Which are not recognized by any FSM
>> Which require memory

- Memory of FSM is very limited
-It cannot store or count strings

Eg. ababb abab b .

Eg. $a^N b^N$.

aaa bbb

aaaa bbbb

Operations on Regular Languages

UNION – $A \cup B = \{ x \mid x \in A \text{ or } x \in B \}$

CONCATENATION – $A \circ B = \{ xy \mid x \in A \text{ and } y \in B \}$

STAR – $A^* = \{ x_1 x_2 x_3 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A \}$

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

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

$A \circ B = \{ pqt, pquv, rt, ruv \}$

$A^* = \{ \epsilon, pq, r, pqr, rpq, pqpq, rr, pqpqpq, rrr, \ldots \}$

>> In DFA, given the current state
we know what the next state will be

>> It has only one unique next state

>> It has no choices or randomness

>> It is simple and easy to design

Non-deterministic Finite Automata

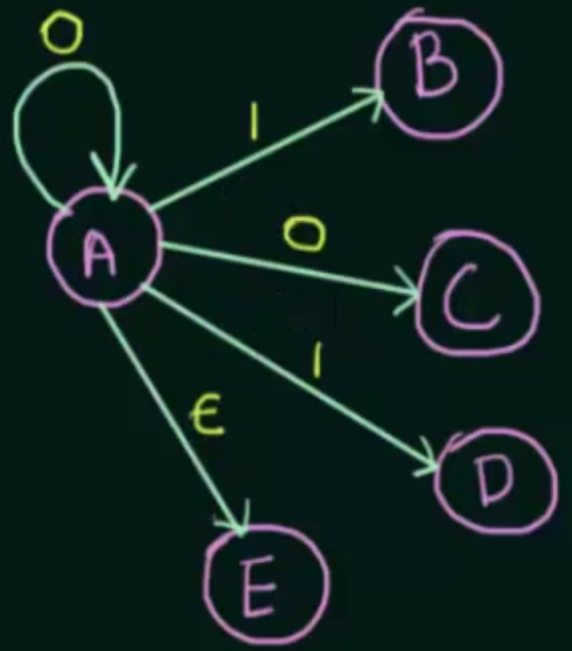
↓

NON-DETERMINISM

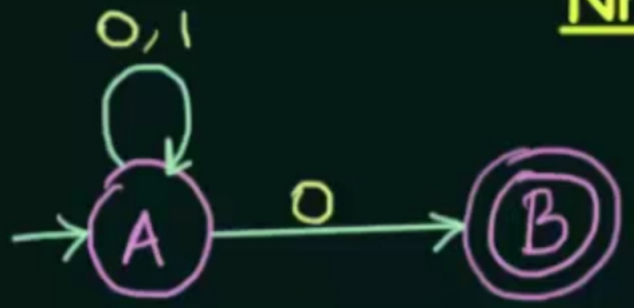
>> In NFA, given the current state
there could be multiple next states

>> The next state may be chosen at random

>> All the next states may be chosen in parallel



NFA - Formal Definition



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

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

Q = Set of all states - {A, B}

Σ = inputs - {0, 1}

q_0 = Start State / initial State - A

F = Set of final States - B

$\delta = Q \times \Sigma \rightarrow$ ___ - ?

A × 0 → A

A × 0 → B

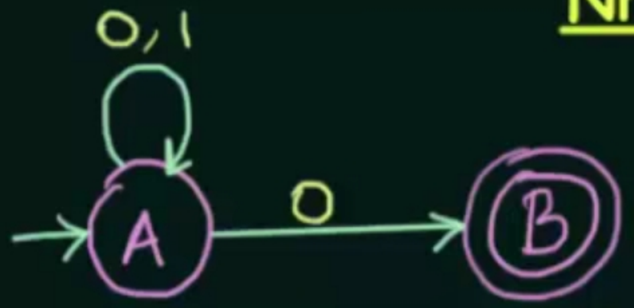
A × 1 → A

B × 0 → ϕ

B × 1 → ϕ

$A \xrightarrow{1} A, B, AB, \phi$

NFA - Formal Definition



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

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

Q = Set of all states - {A, B}

Σ = inputs - {0, 1}

q_0 = Start State / initial State - A

F = Set of final States - B

$\delta = Q \times \Sigma \rightarrow 2^Q$ - ?

A × 0 → A, B

A × 0 → B

A × 1 → A

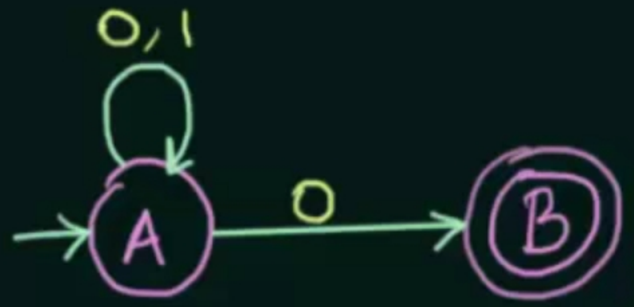
B × 0 → ϕ

B × 1 → ϕ

$A \xrightarrow{1} A, B, AB, \phi$ - $2^2 - 4$

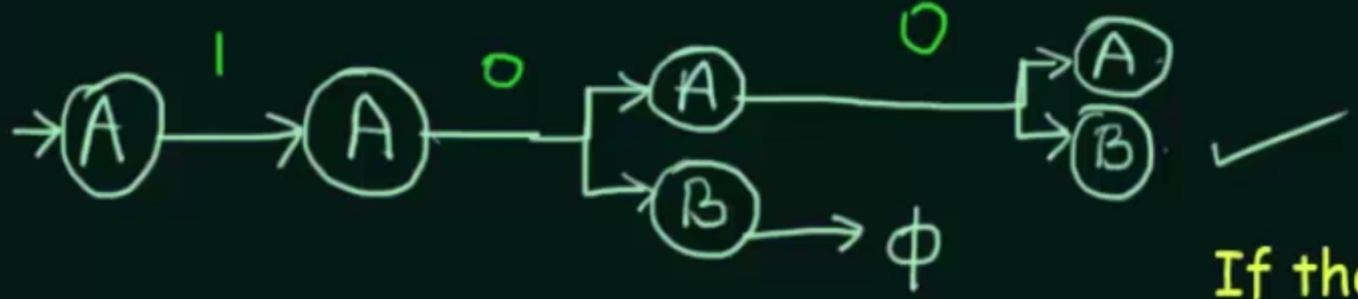
3 States - A, B, C

NFA - Example-1

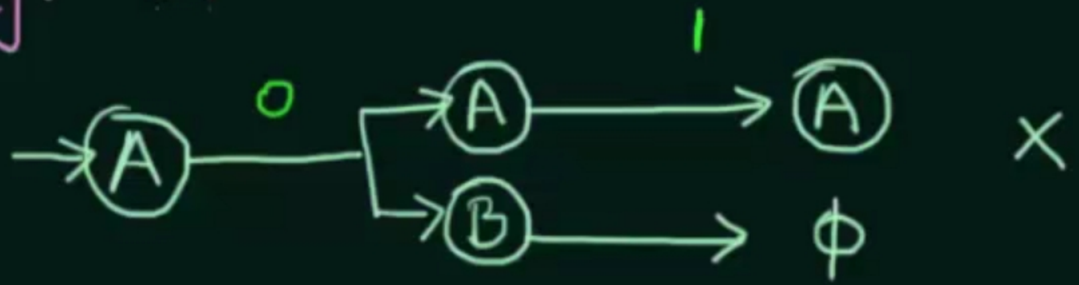


L = { Set of all strings that end with 0 }

Eg. 100

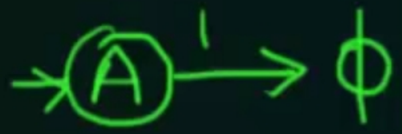


Eg. 01



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

Eg. 101 X

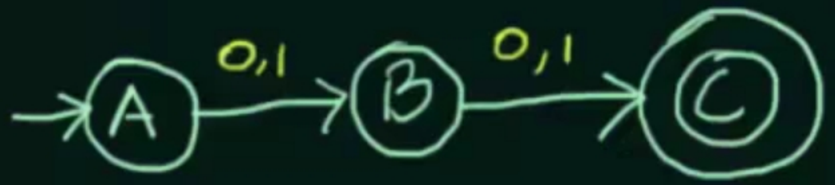


Dead configuration

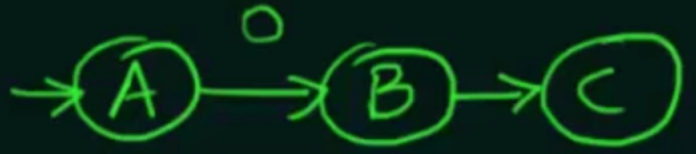
>> Construct a NFA that accepts sets of all strings over {0,1} of length 2

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

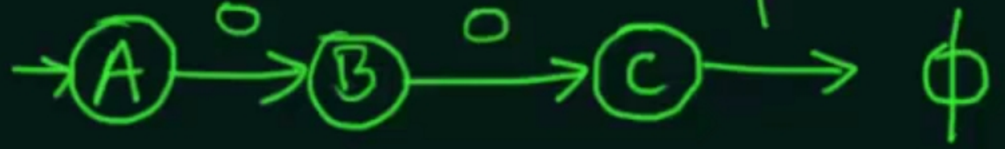
$L = \{00, 01, 10, 11\}$



Eg. 00 ✓

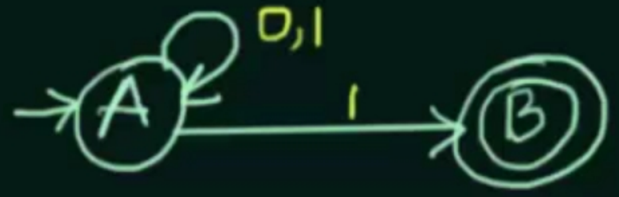


Eg. 001 X



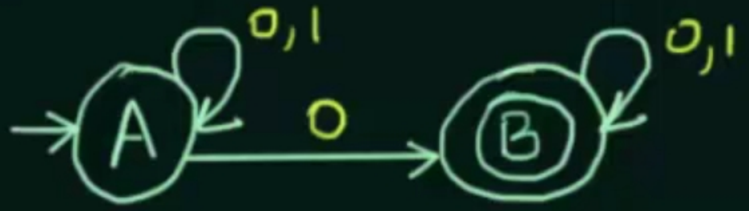
NFA - Example-3

Ex 1) L1 = { Set of all strings that ends with '1' }

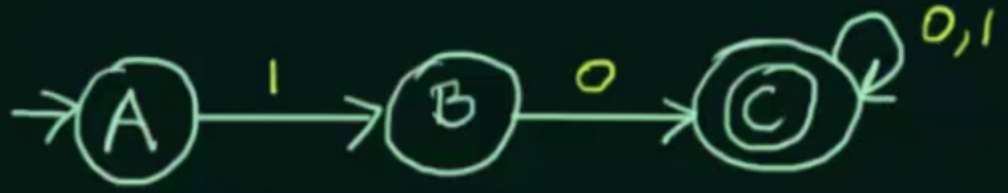


01, 001, 0001, 0*1, 1,
101, 1101,

Ex 2) L2 = { Set of all strings that contain '0' }

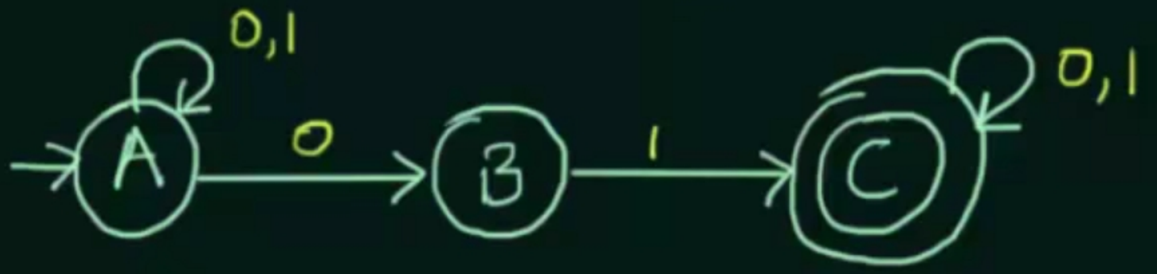


Ex 3) L3 = { Set of all strings that starts with '10' }

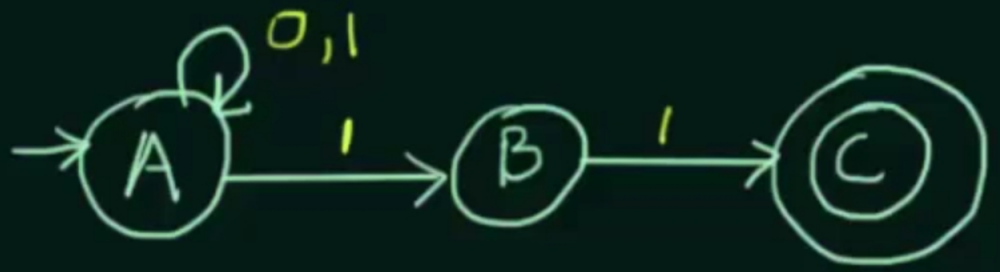


Ex 4) L4 = { Set of all strings that contain '01' }

Ex 4) L4 = { Set of all strings that contain '01' }



Ex 5) L5 = { Set of all strings that ends with '11' }



Assignment: If you were to construct the equivalent DFAs for the above NFAs, then tell me how many minimum number of states would you use for the construction of each of the DFAs

Conversion of NFA to DFA

Every DFA is an NFA, but not vice versa

But there is an equivalent DFA for every NFA

DFA

$\delta = Q \times \Sigma \rightarrow Q$

NFA

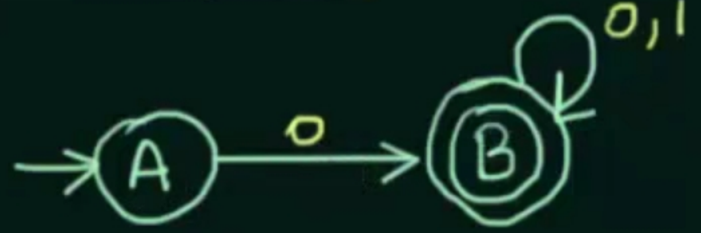
$\delta = Q \times \Sigma \rightarrow 2^Q$

$NFA \cong DFA$

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

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

NFA

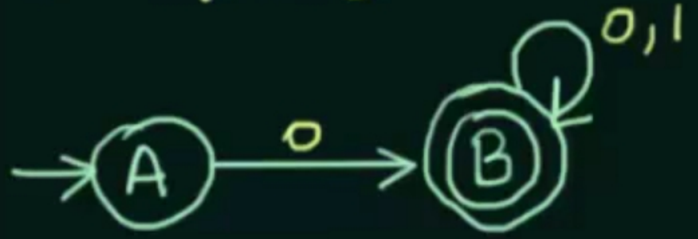


NFA ≅ DFA

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

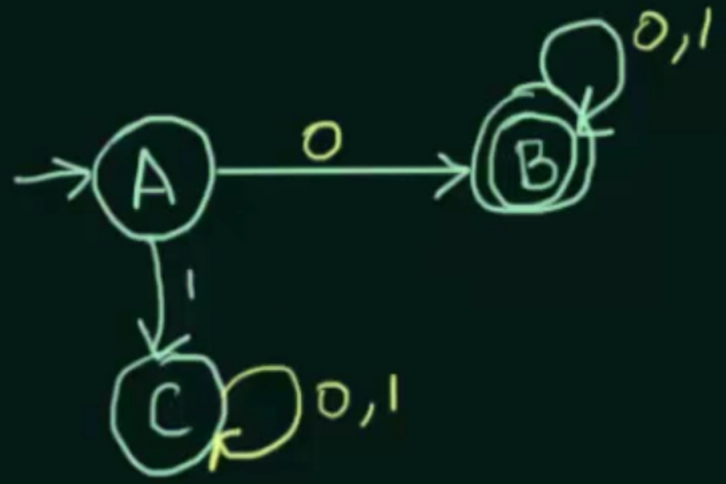
$\Sigma = \{0, 1\}$

NFA



	0	1
A	B	ϕ
B	B	B

DFA



	0	1
A	B	C
B	B	B
C	C	C

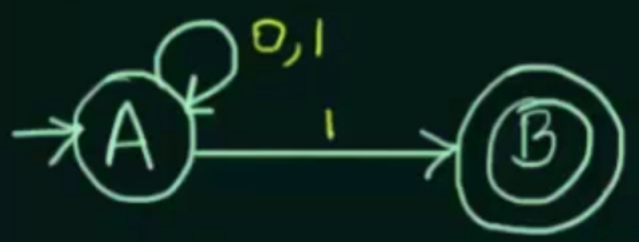
C - Dead State / Trap State

Conversion of NFA to DFA - Examples (Part 1)

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

$\Sigma = 0, 1$

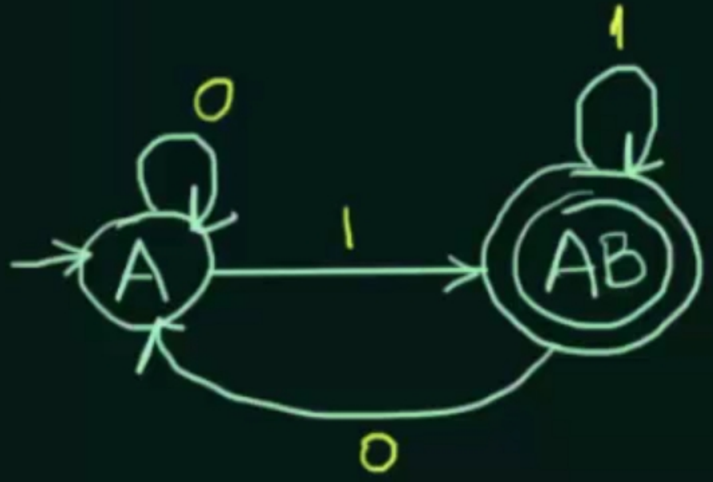
NFA



	0	1
A	{A}	{A, B}
B	ϕ	ϕ

Subset construction method

DFA

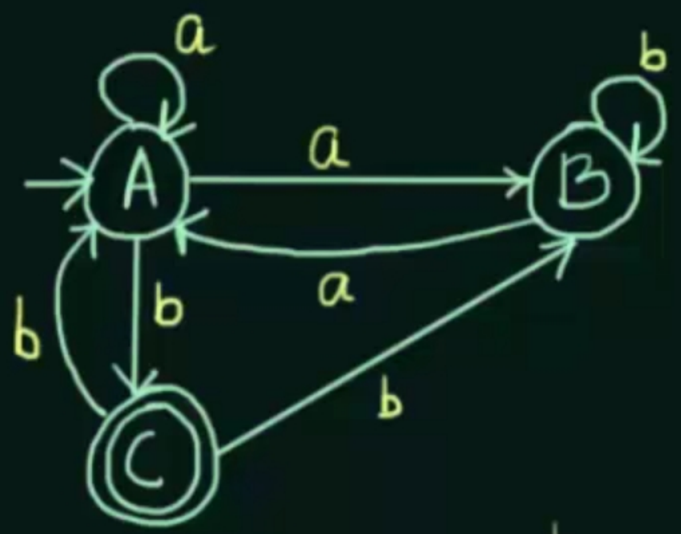


	0	1
A	{A}	{AB}
AB	{A}	{AB}

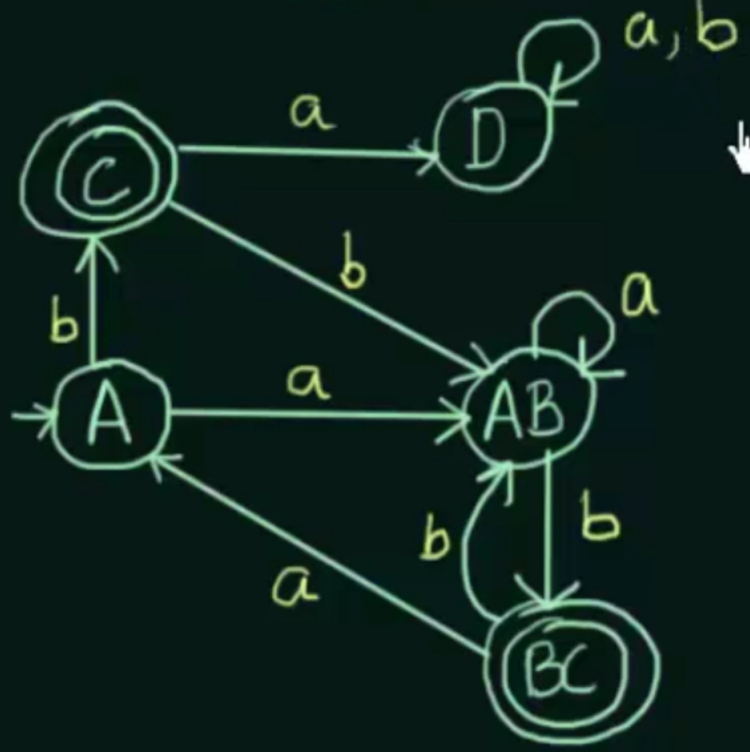
AB - Single State

Find the equivalent DFA for the NFA given by M= [{A,B,C}, {a,b}, δ, A, {C}] where δ is given by:

	a	b
→A	A,B	C
B	A	B
(C)	–	A,B



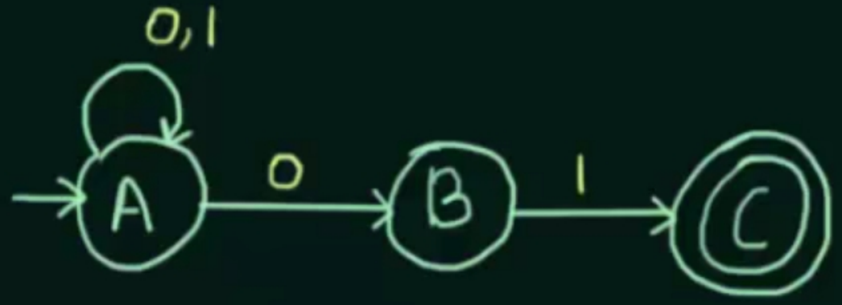
	a	b
→A	AB	C
AB	AB	BC
(BC)	A	AB
(C)	D	AB
D	D	D



Given below is the NFA for a language

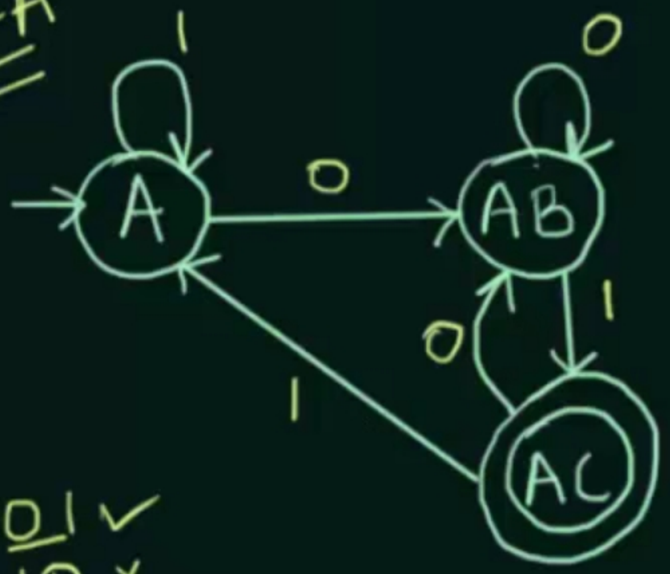
L = { Set of all strings over (0,1) that ends with '01' }. Construct its equivalent DFA

NFA



	0	1
→A	A, B	A
B	ϕ	C
(C)	ϕ	ϕ

DFA



	0	1
→A	AB	A
AB	AB	AC
(AC)	AB	A

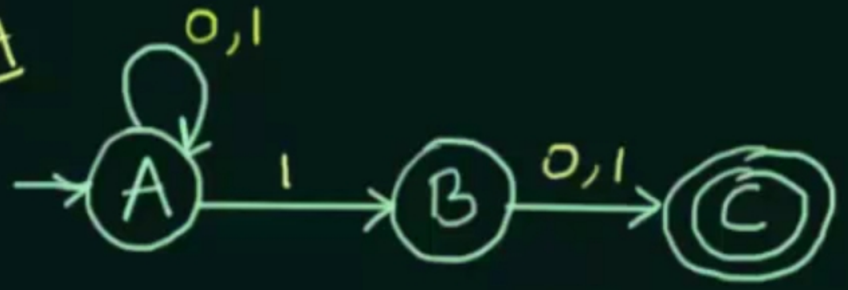
1101 ✓

1110 ✗

second last symbol is always '1'. Then convert it to its equivalent DFA.

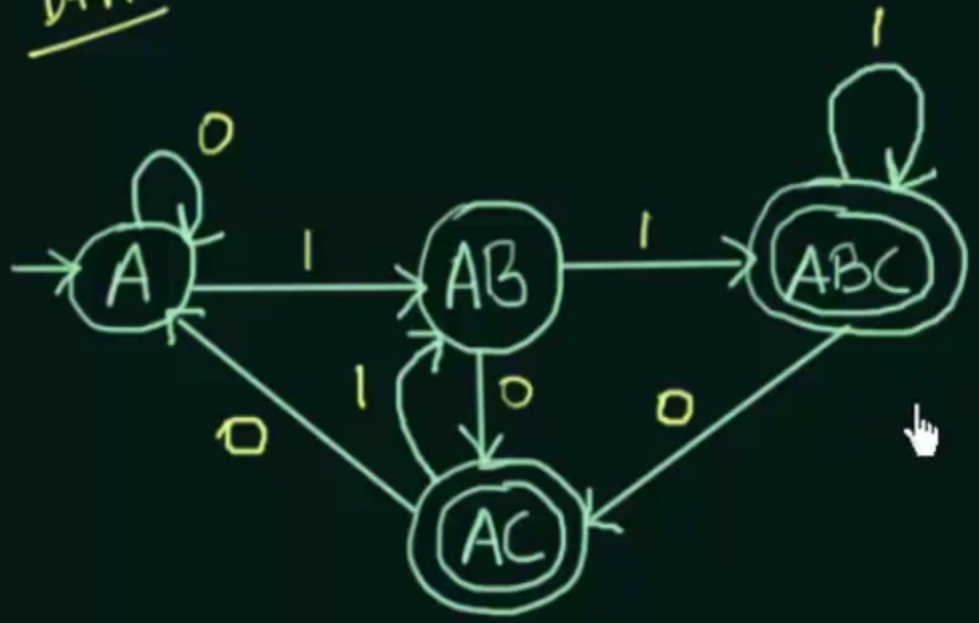
Eg. 1010 ✓
110 ✓
1101010 ✓

NFA



	0	1
→A	A	A,B
B	C	C
(C)	ϕ	ϕ

DFA



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

Minimization of DFA

Minimization of DFA is required to obtain the minimal version of any DFA which consists of the minimum number of states possible

DFA 5 states 4 States

Equivalent

Two states 'A' and 'B' are said to be equivalent if

$\delta(A, X) \rightarrow F$ and $\delta(B, X) \rightarrow F$

OR

$\delta(A, X) \not\rightarrow F$ and $\delta(B, X) \not\rightarrow F$

where 'X' is any input String

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

0 Equivalence : {A, B, C, D} {E}

1 Equivalence : {A, B, C} {D} {E}

2 Equivalence : {A, C} {B} {D} {E}

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

A, B ✓

A, C ✓

C, D ✗

0 Equivalence : {A, B, C, D} {E}

1 Equivalence : {A, B, C} {D} {E}

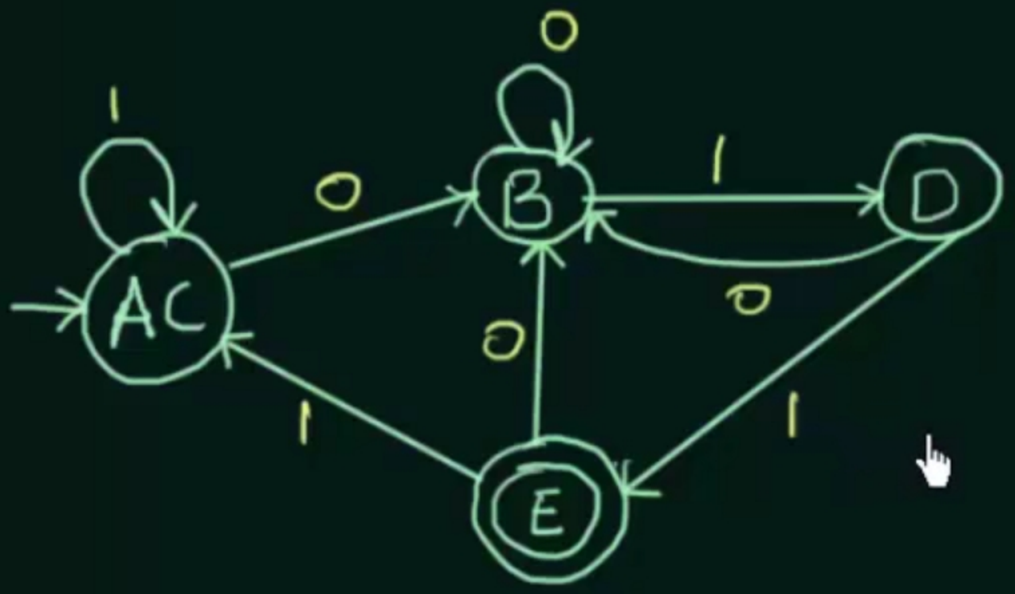
2 Equivalence : {A, C} {B} {D} {E}

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

A, B ✓

A, C ✓

C, D ✗



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

Minimization of DFA - Examples (Part-2)

Construct a minimum DFA equivalent to the DFA described by

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

0 Equivalence

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

1- Equivalence

$\{q_0, q_4, q_6\}$

$\{q_1, q_7\}$

$\{q_3, q_5\}$ $\{q_2\}$

2- Equivalence

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

q_7	q_6	q_2

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

3. Equivalence

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

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

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

	0	1
→A	B	C
B	D	E
C	E	D
D	G	G
E	G	G
G	G	G

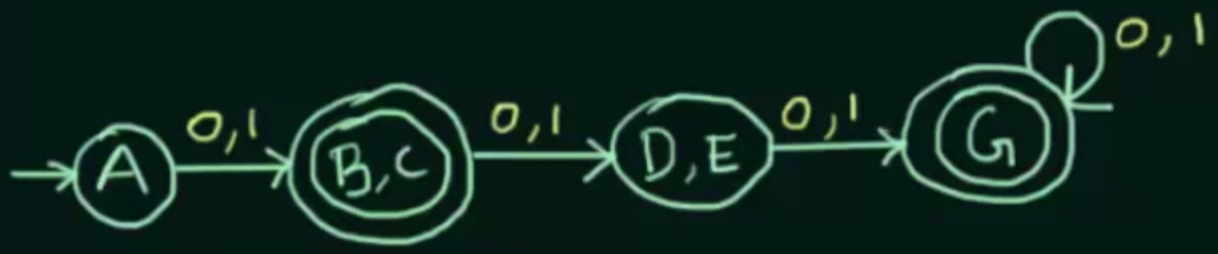
Initial State

0-Equivalence : {A, D, E} {B, C, G}

1-Equivalence : {A, D, E} {B, C} {G}

2-Equivalence : {A} {D, E} {B, C} {G}

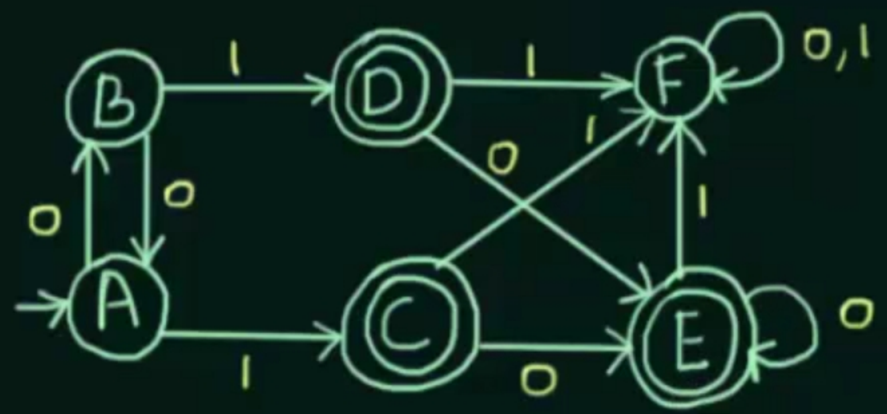
3-Equivalence : {A} {D, E} {B, C} {G}



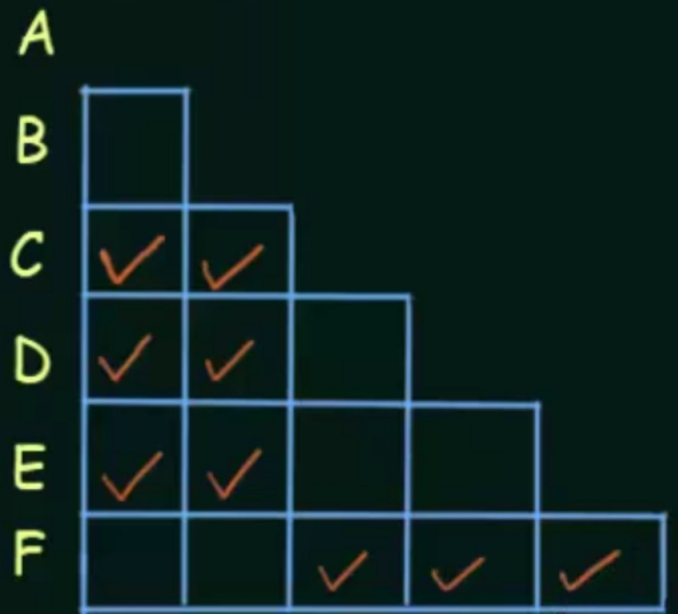
	0	1
→{A}	{B, C}	{B, C}
{D, E}	{G}	{G}
{B, C}	{D, E}	{D, E}
{G}	{G}	{G}

Minimization of DFA - Table Filling Method

(Myhill-Nerode Theorem)



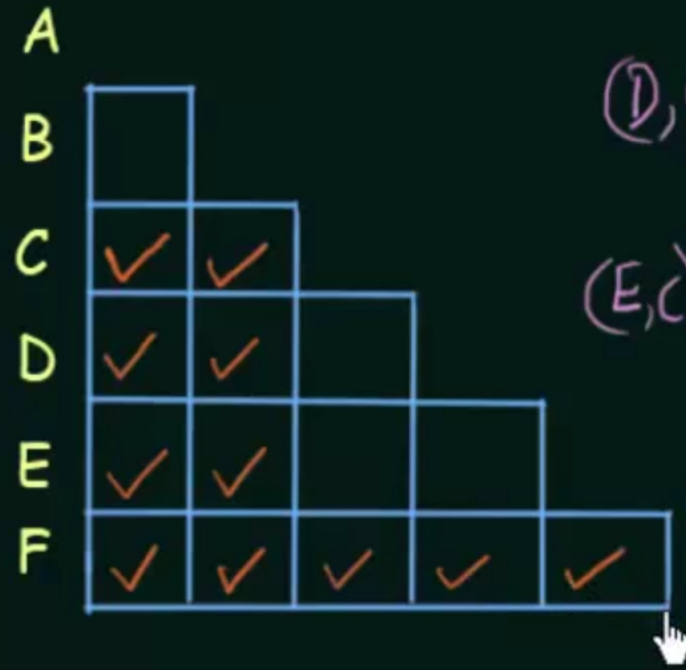
A B C D E F



Steps:

1) Draw a table for all pairs of states (P,Q)
2) Mark all pairs where $P \in F$ and $Q \notin F$
3) If there are any Unmarked pairs (P,Q) such that $[\delta(P,x), \delta(Q,x)]$ is marked, then mark [P,Q] where 'x' is an input symbol
   REPEAT THIS UNTIL NO MORE MARKINGS CAN BE MADE
4) Combine all the Unmarked Pairs and make them a single state in the minimized DFA

A B C D E F



1) Make pairs of all the states in the form (P,Q)
2) Mark all pairs where $P \in F$ and $Q \notin F$
3) If there are any Unmarked pairs (P,Q) such that $[\delta(P,x), \delta(Q,x)]$ is marked, then mark [P,Q] where 'x' is an input symbol
   REPEAT THIS UNTIL NO MORE MARKINGS CAN BE MADE
4) Combine all the Unmarked Pairs and make them a single state in the minimized DFA

(D,C) - $\delta(D,0) = E$, $\delta(D,1) = F$; $\delta(C,0) = E$, $\delta(C,1) = F$

(E,C) - $\delta(E,0) = E$, $\delta(E,1) = F$; $\delta(C,0) = E$, $\delta(C,1) = F$

(E,D) - $\delta(E,0) = E$, $\delta(E,1) = F$; $\delta(D,0) = E$, $\delta(D,1) = F$

(F,A) - $\delta(F,0) = F$, $\delta(F,1) = F$; $\delta(A,0) = B$, $\delta(A,1) = C$

(F,B) = $\delta(F,0) = F$; $\delta(B,0) = A$

(B,A) - $\delta(B,0) = A$, $\delta(B,1) = D$; $\delta(A,0) = B$, $\delta(A,1) = C$

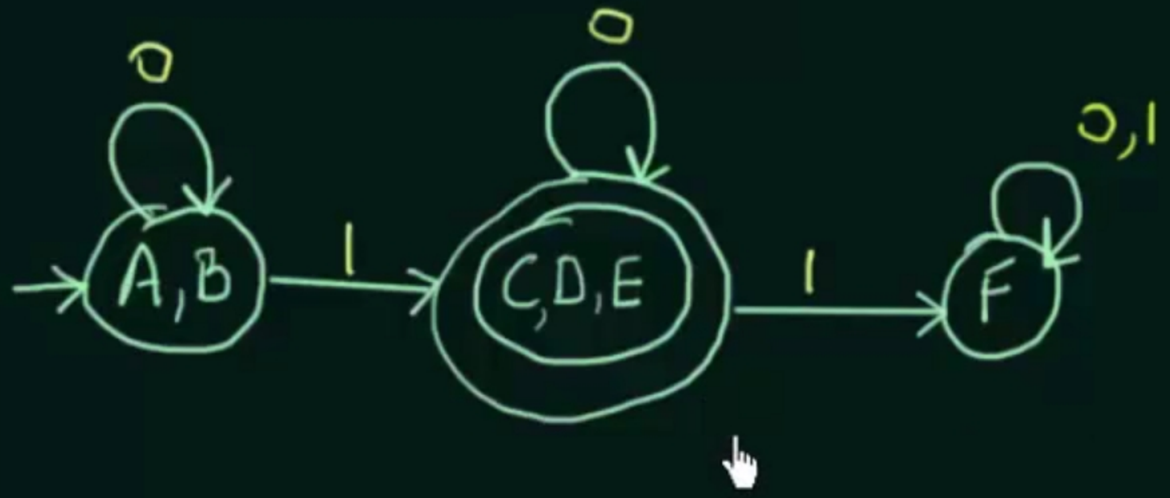
$\delta(B,0) = E$ $\delta(D,1) = F$

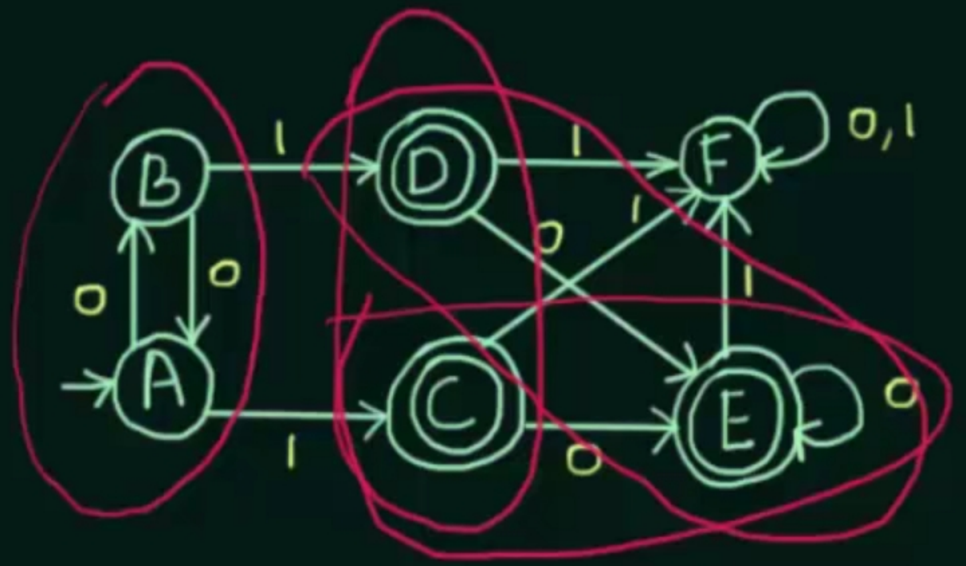
$\delta(B,0) = A$

(B,A) - $\delta(B,0) = A$ $\delta(B,1) = D$

$\delta(A,0) = B$ $\delta(A,1) = C$

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

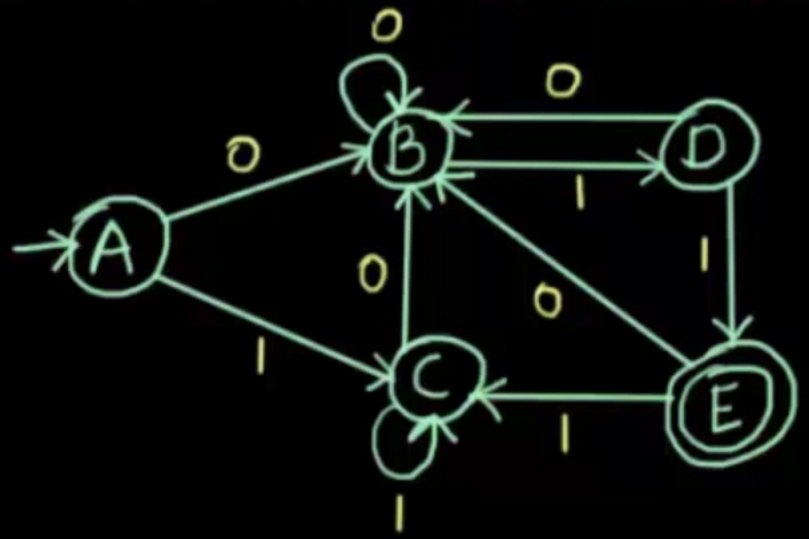




Minimization of DFA - Table Filling Method (Myhill Nerode Theorem)

(Example)

Minimize the following DFA using Table Filling Method



	A	B	C	D	E
A					
B					
C					
D	✓	✓			
E	✓	✓	✓	✓	

(B,A) - $\delta(B,1) = D$, $\delta(A,1) = C$ } $\delta(B,0) = B$, $\delta(A,0) = B$ }

(C,B) - $\delta(C,0) = B$, $\delta(B,0) = B$ } $\delta(C,1) = C$, $\delta(B,1) = D$ }

(D,B) - $\delta(D,0) = B$, $\delta(B,0) = B$ }

(C,A) - $\delta(C,0) = B$, $\delta(A,0) = B$ } $\delta(C,1) = C$, $\delta(A,1) = C$ }

(D,A) - $\delta(D,0) = B$, $\delta(A,0) = B$ } $\delta(D,1) = E$, $\delta(A,1) = C$ } $\delta(D,1) = E$, $\delta(B,1) = D$ }

(C,A) - $\delta(C,0)=B$, $\delta(C,1)=C$; $\delta(A,0)=B$, $\delta(A,1)=C$

(D,A) - $\delta(D,0)=B$, $\delta(D,1)=E$; $\delta(A,0)=B$, $\delta(A,1)=C$

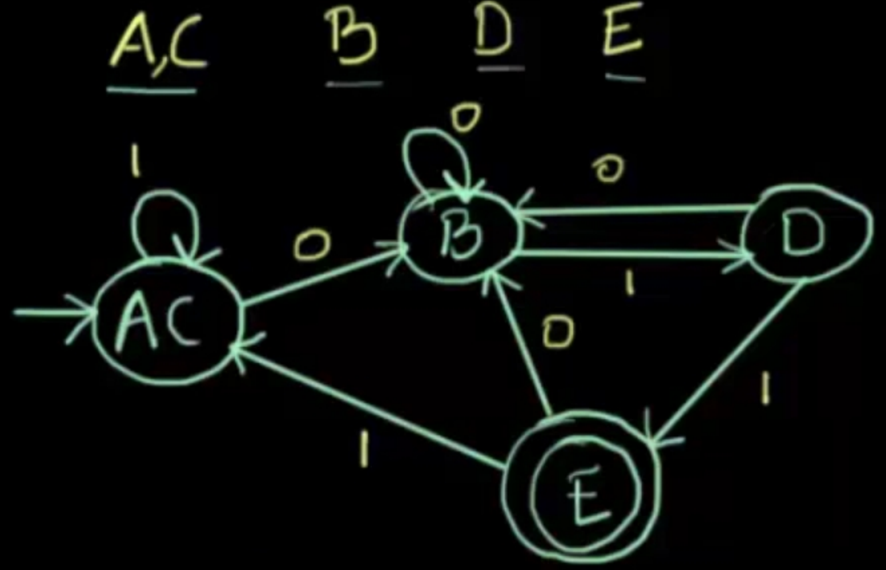
(D,C) - $\delta(D,0)=B$, $\delta(D,1)=E$; $\delta(C,0)=B$, $\delta(C,1)=C$

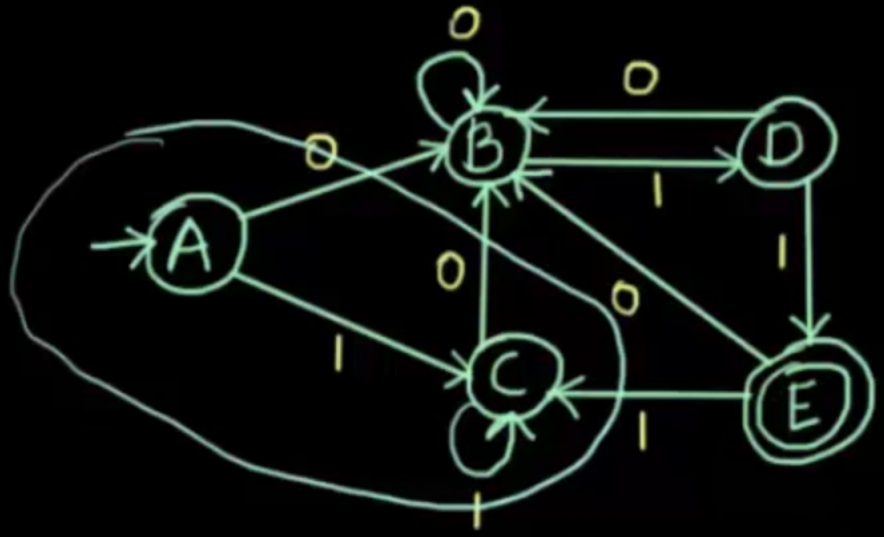
(B,A) - $\delta(B,0)=B$, $\delta(B,1)=D$; $\delta(A,0)=B$, $\delta(A,1)=C$

(C,A) - $\delta(C,0)=B$, $\delta(C,1)=C$; $\delta(A,0)=B$, $\delta(A,1)=C$

(C,B) - $\delta(C,0)=B$, $\delta(C,1)=C$; $\delta(B,0)=B$, $\delta(B,1)=D$

(C,A) - $\delta(C,0)=B$, $\delta(C,1)=C$; $\delta(A,0)=B$, $\delta(A,1)=C$





Finite Automata With Outputs

MEALY MACHINE

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

where

Q = Finite Set of States

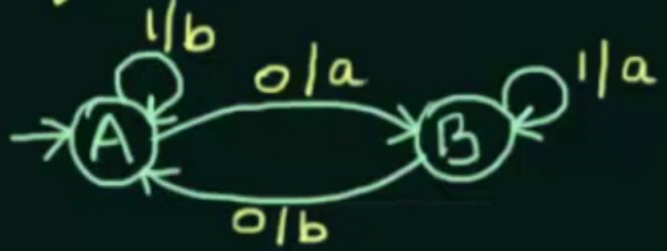
Σ = Finite non-empty set of Input Alphabets

Δ = The set of Output Alphabets

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

λ = Output function: $\Sigma \times Q \rightarrow \Delta$

q_0 = Initial State / Start State



MOORE MACHINE

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

where

Q = Finite Set of States

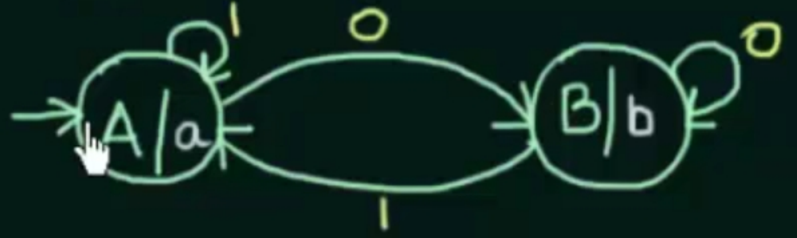
Σ = Finite non-empty set of Input Alphabets

Δ = The set of Output Alphabets

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

λ = Output function: $Q \rightarrow \Delta$

q_0 = Initial State / Start State



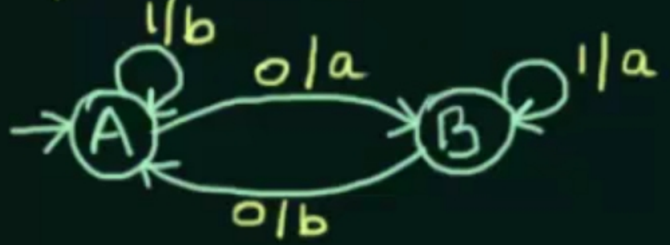
Alphabets

Δ = The set of Output Alphabets

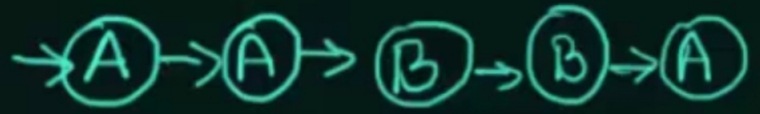
δ = Transition function: $Q \times \Sigma \rightarrow Q$

λ = Output function: $\Sigma \times Q \rightarrow \Delta$

q_0 = Initial State / Start State



Eg. 1 0 1 0



b a a b

n – n

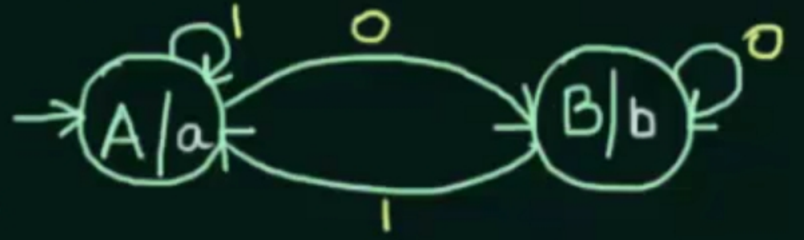
Alphabets

Δ = The set of Output Alphabets

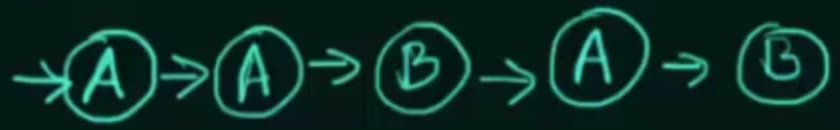
δ = Transition function: $Q \times \Sigma \rightarrow Q$

λ = Output function: $Q \rightarrow \Delta$

q_0 = Initial State / Start State



Eg. 1 0 1 0

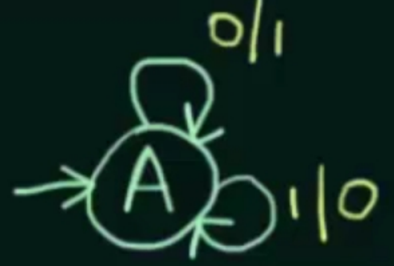


a a b a b

n → n+1

Construction of Mealy Machine

Ex-1) Construct a Mealy Machine that produces the 1's Complement of any binary input string.

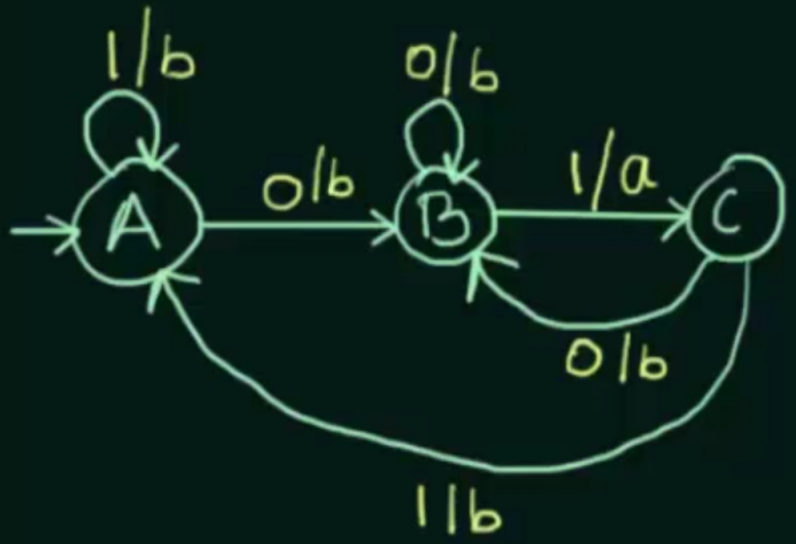


10100

01011

Ex-2) Construct a Mealy Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string.

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



0110

babb

1000

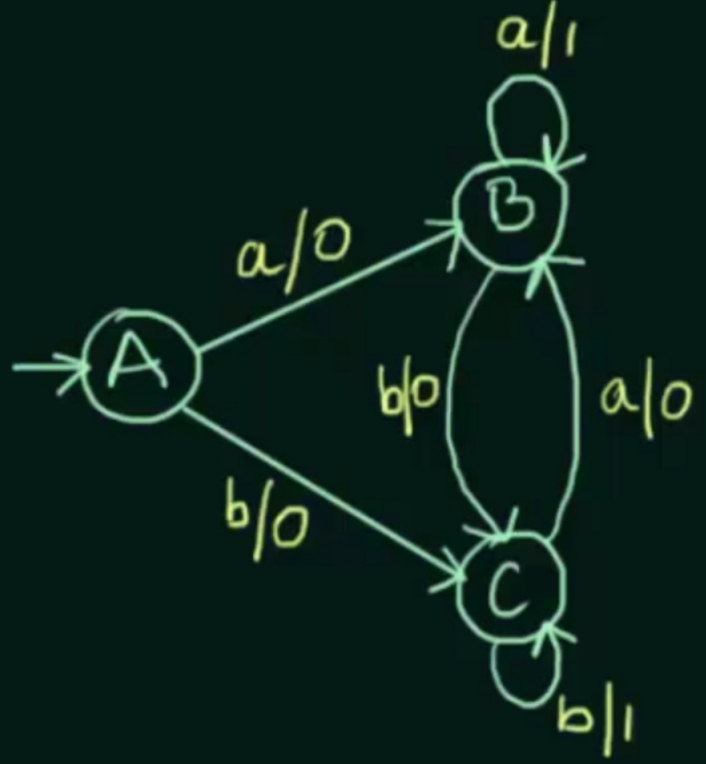
bbbb

Construction of Mealy Machine - Examples (Part-1)

Design a Mealy Machine accepting the language consisting of strings from Σ^*, where Σ = {a,b} and the strings should end with either aa or bb

aa - 1

bb - 1



abb

baa

ba

00

aa

0 1

Construction of Mealy Machine -Examples (Part-2)

Construct a Mealy Machine that gives 2's Complement of any binary input. (Assume that the last carry bit is neglected)

2's complement = 1's complement + 1

MSB ← LSB

Eg. 10100

1's. 01011

+1

2's = 01100

Eg. 11100

1's 00011

+ 1

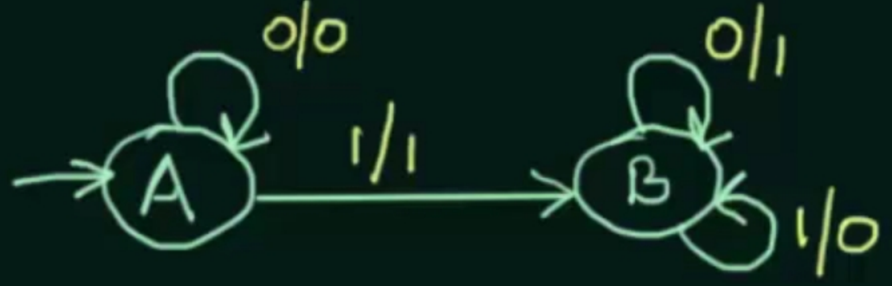
2's: 00100

Eg. 1111(1

1's 0000

2's +1

0001

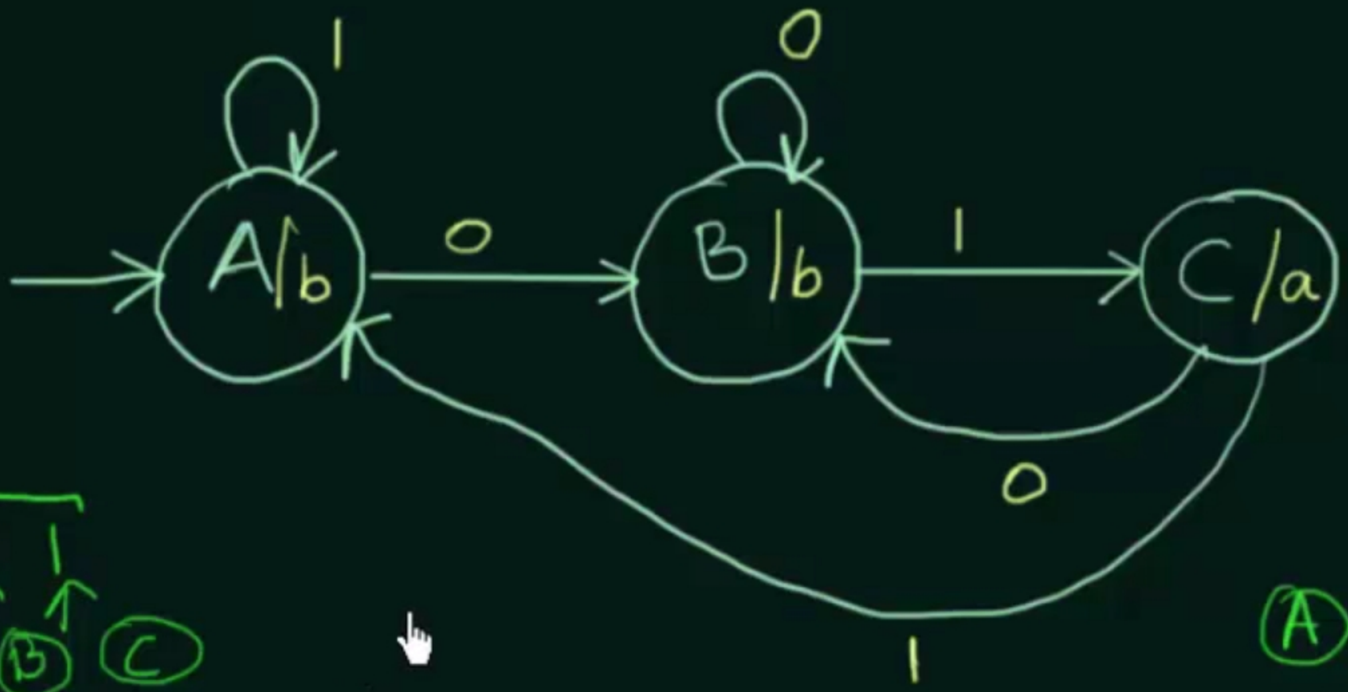


Construction of Moore Machine

Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string

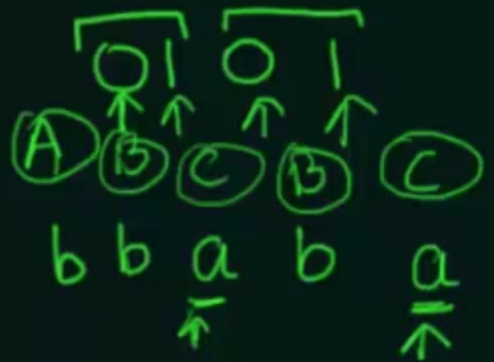
$\Sigma = \{0, 1\}$

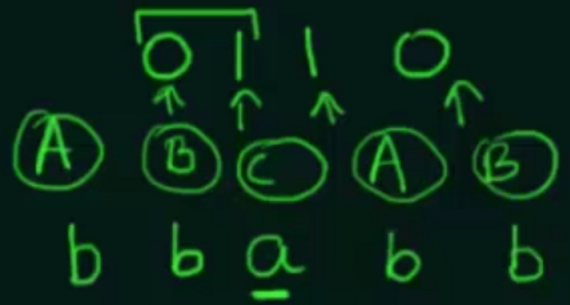
$\Delta = \{a, b\}$



0 1 0 1

0 1 1 0 1





Construction of Moore Machine - Example (Part-2)

For the following Moore Machine the input alphabet is $\Sigma = \{a,b\}$ and the output alphabet is $\Delta = \{0,1\}$. Run the following input sequences and find the repective outputs:

(i) aabab (ii) abbb (iii) ababb

States	a	b	Outputs
$\rightarrow q_0$	q_1	q_2	0
q_1	q_2	q_3	0
q_2	q_3	q_4	1
q_3	q_4	q_4	0
q_4	q_0	q_0	0

i) a a b a b

q_0 q_1 q_2 q_4 q_0 q_2

0 0 1 0 0 1

ii) a b b b

q_0 q_1 q_3 q_4 q_0

0 0 0 0 0

iii) a b a b b

q_0 q_1 q_3 q_4 q_0 q_2

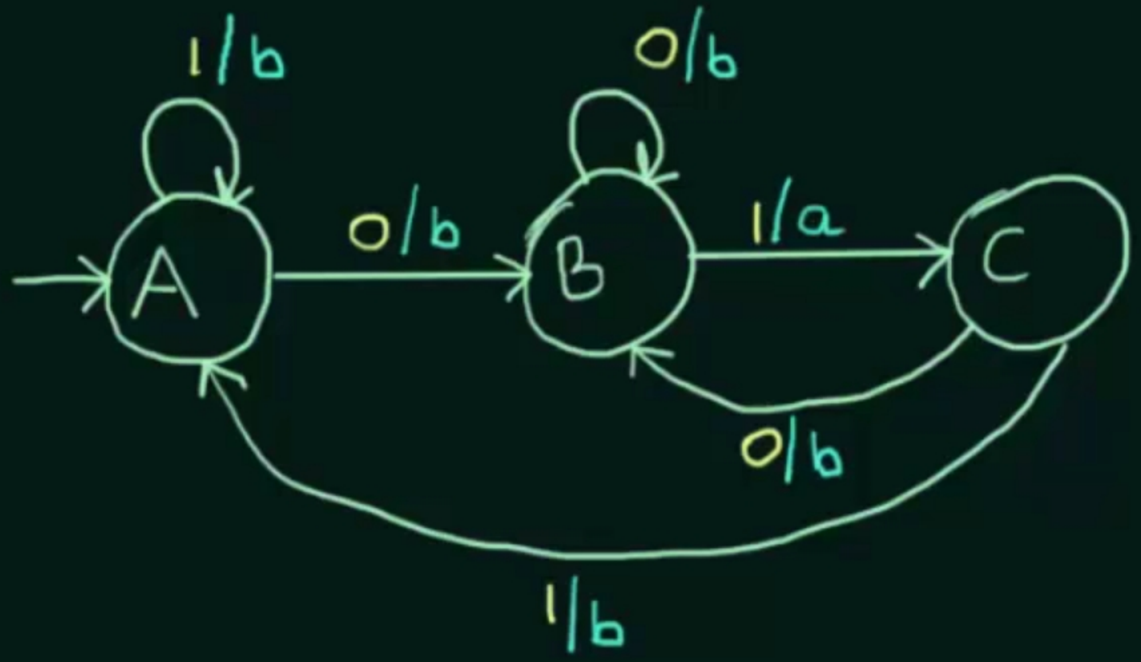
0 0 0 0 0 1

Conversion of Moore Machine to Mealy Machine

Construct a Moore Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string and then CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

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

Moore Machine $\Longleftrightarrow$ Mealy Machine

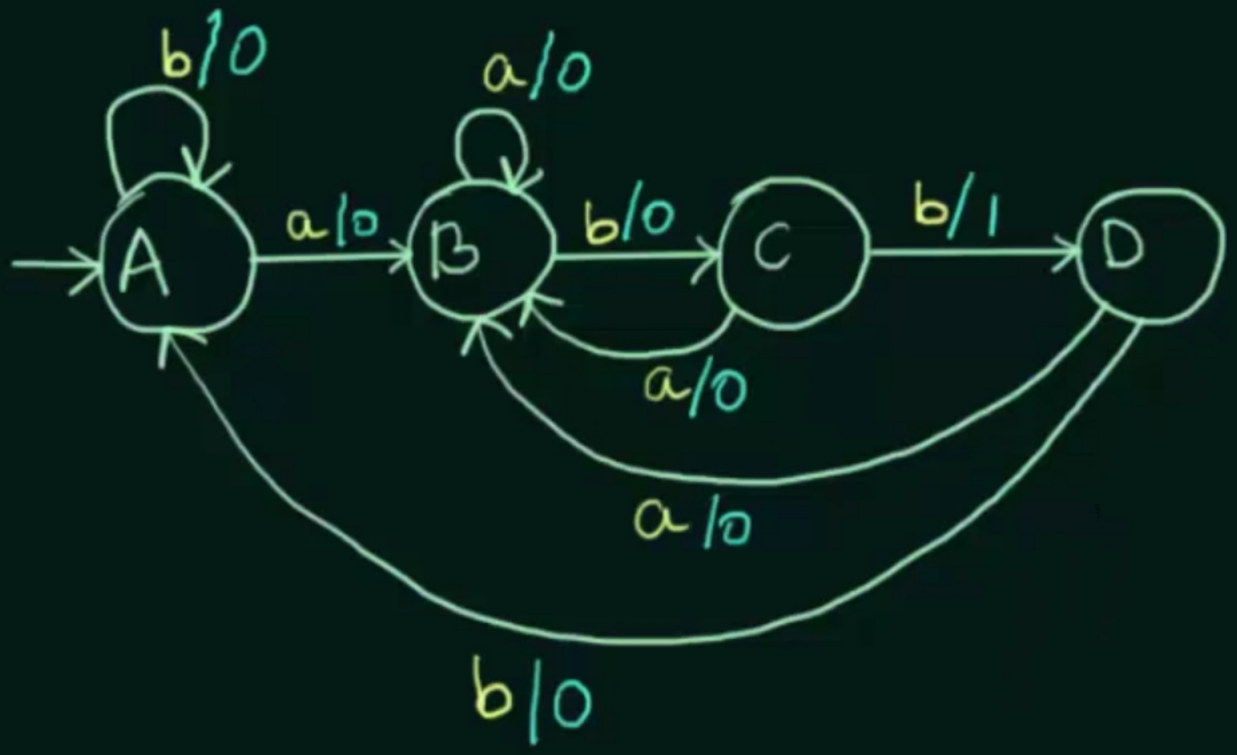


State	0	1
→ A	B, b	A, b
B	B, b	C, a
C	B, b	A, b

Conversion of Moore Machine to Mealy Machine - Examples (Part-1)

The given Moore Machine counts the occurences of the sequence 'abb' in any input binary strings over {a,b}. CONVERT IT TO ITS EQUIVALENT MEALY MACHINE

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



State	a	b
→ A	B, 0	A, 0
B	B, 0	C, 0
C	B, 0	D, 1
D	B, 0	A, 0

Conversion of Moore Machine to Mealy Machine- Examples (Part-2)

Convert the given Moore Machine to its equivalent Mealy Machine.

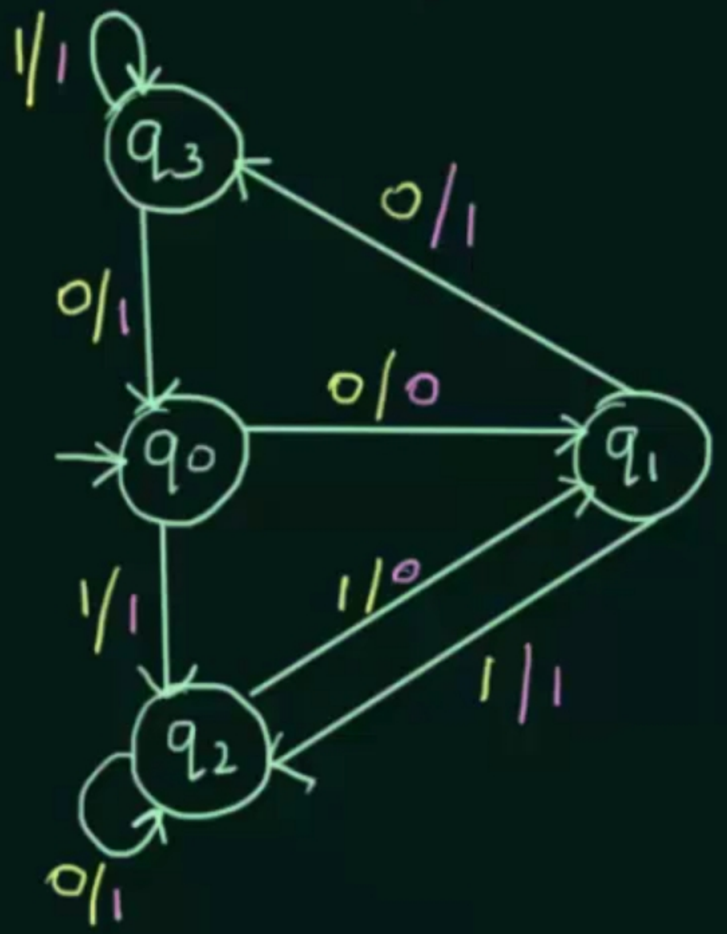
Moore

State	0	1	Output
→ q_0	q_1	q_2	1
q_1	q_3	q_2	0
q_2	q_2	q_1	1
q_3	q_0	q_3	1

Mealy

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

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

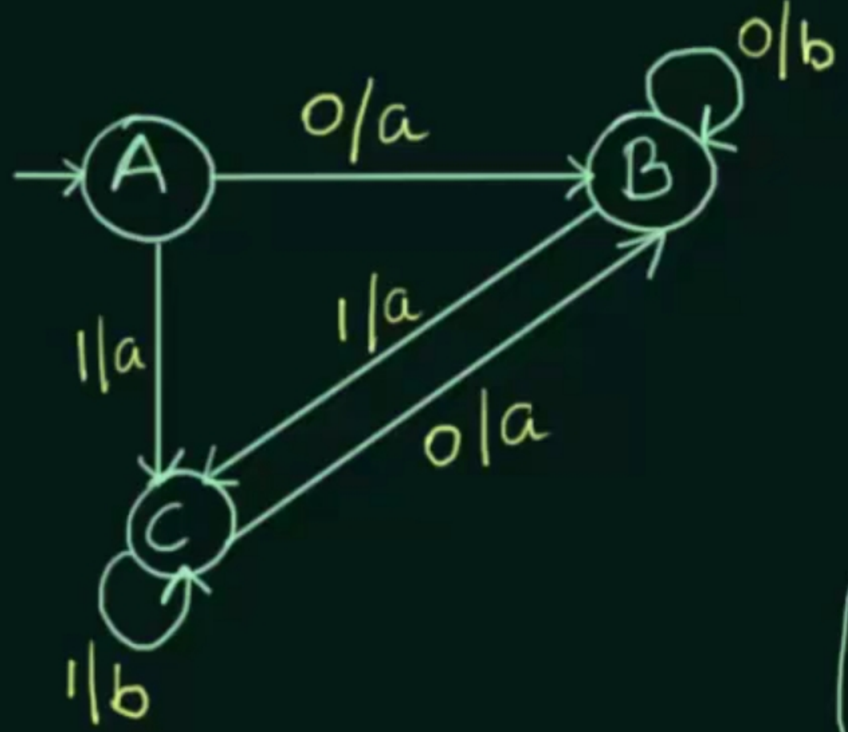


Conversion of Mealy Machine to Moore Machine

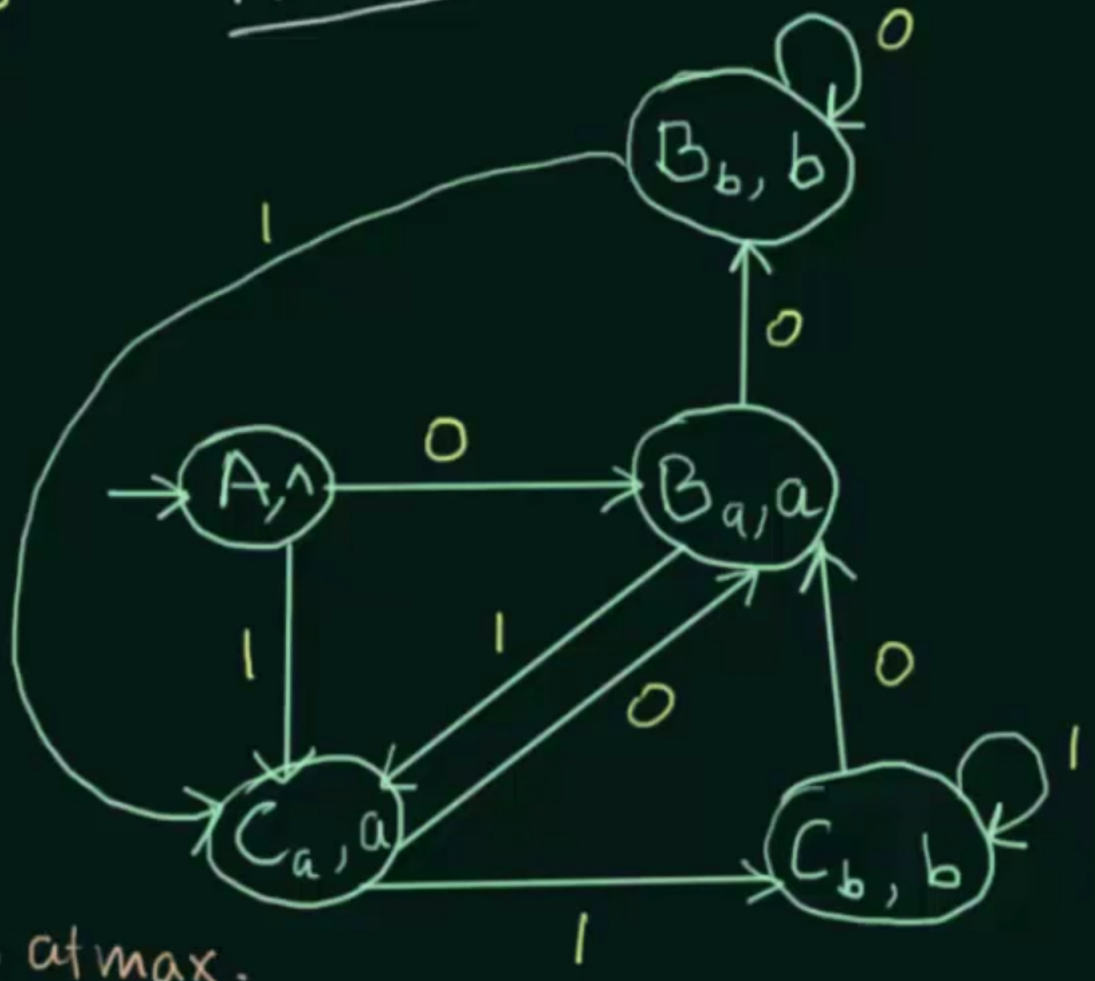
Convert the following Mealy Machine to its equivalent Moore Machine

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

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



Moore machine



Moore -> Mealy => No. of states were same

Mealy -> Moore => No. of states increased

↓

x and y ⇒ (x x y) no. of states at max.

x → No. of states

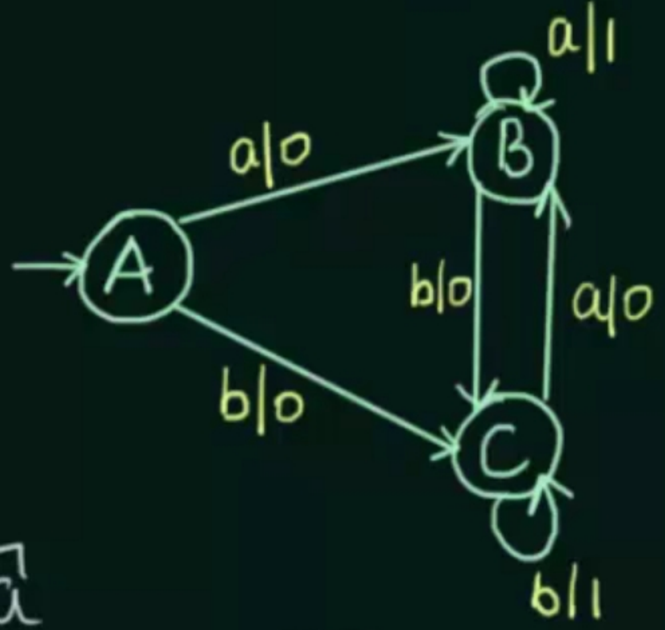
y → No. of outputs

Conversion of Mealy Machine to Moore Machine - Examples (Part-1)

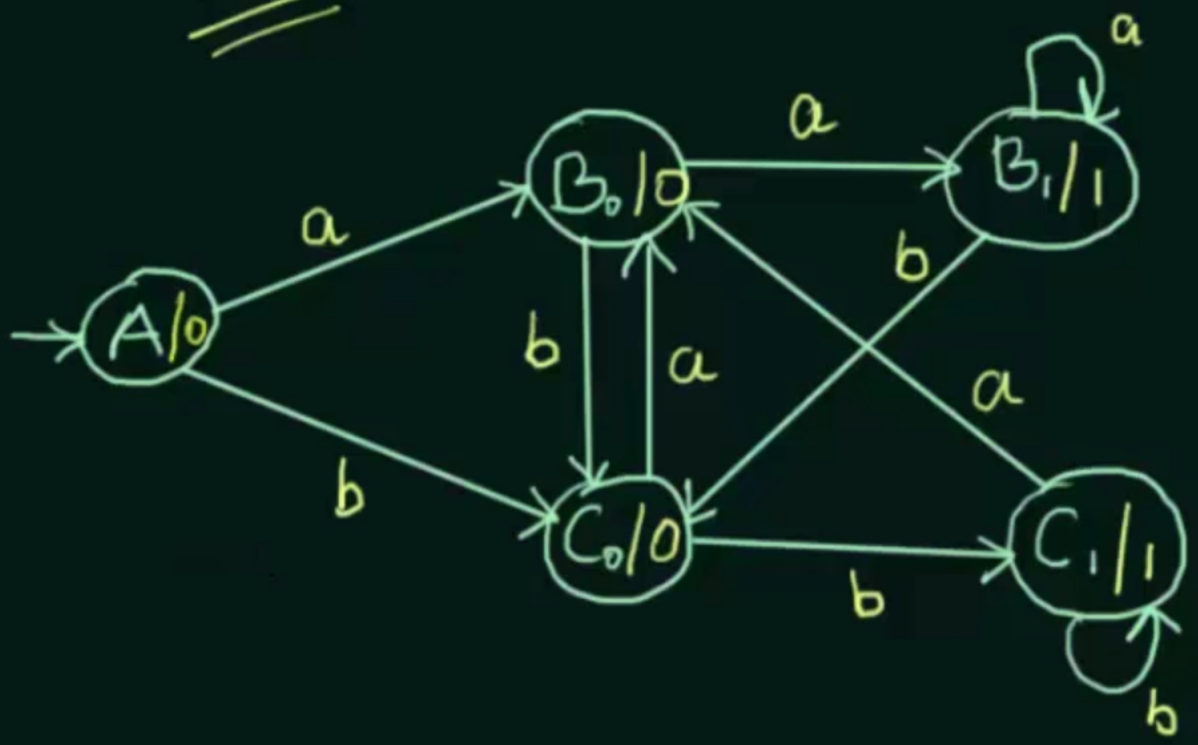
Given below is a Mealy Machine that prints '1' whenever the sequence 'aa' or 'bb' is encountered in any input binary string from Σ^* where $\Sigma = \{a,b\}$.
DESIGN THE EQUIVALENT MOORE MACHINE FOR IT.

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

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



Moore



a a
0 0 1 ✓

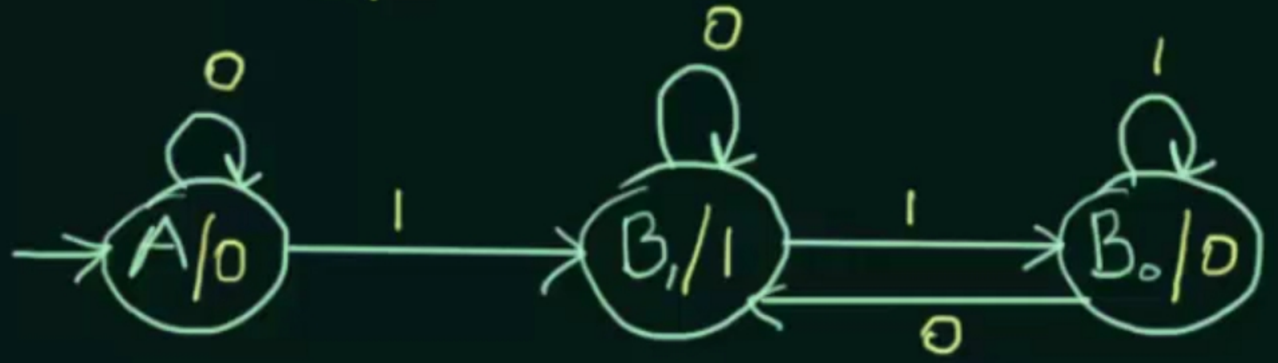
b b
0 0 1 ✓

a b
0 0 0

Conversion of Mealy Machine to Moore Machine- Examples (Part-2)

Convert the given Mealy Machine that give the 2's complement of any binary input to its equivalent MOORE MACHINE.

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



Moore

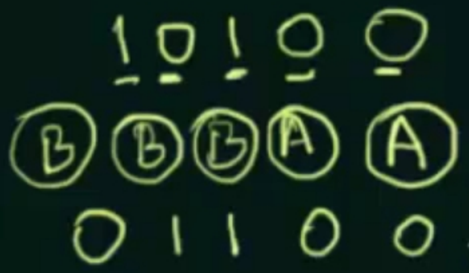
Eg. 10100

1's. 01011

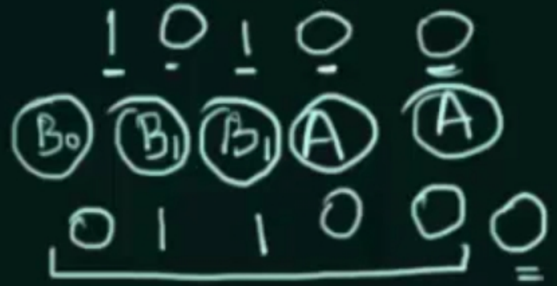
+ 1

01100

Mealy:



Moore:



Conversion of Mealy Machine to Moore Machine -Examples (Part-3)

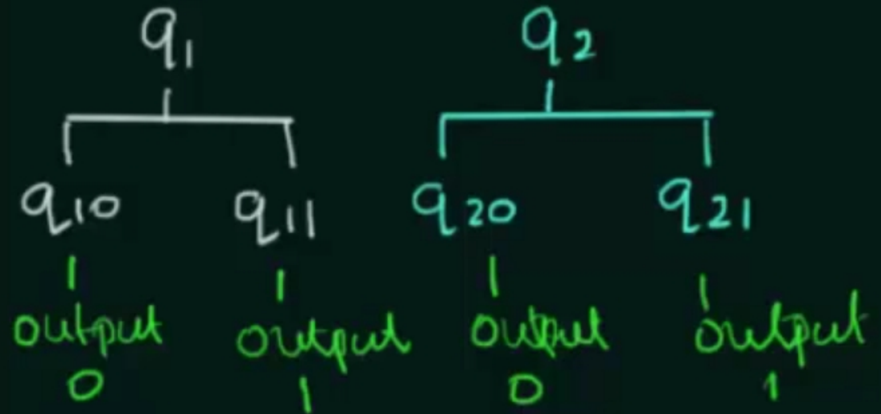
Using Transition Table

Convert the given Mealy Machine to its equivalent Moore Machine

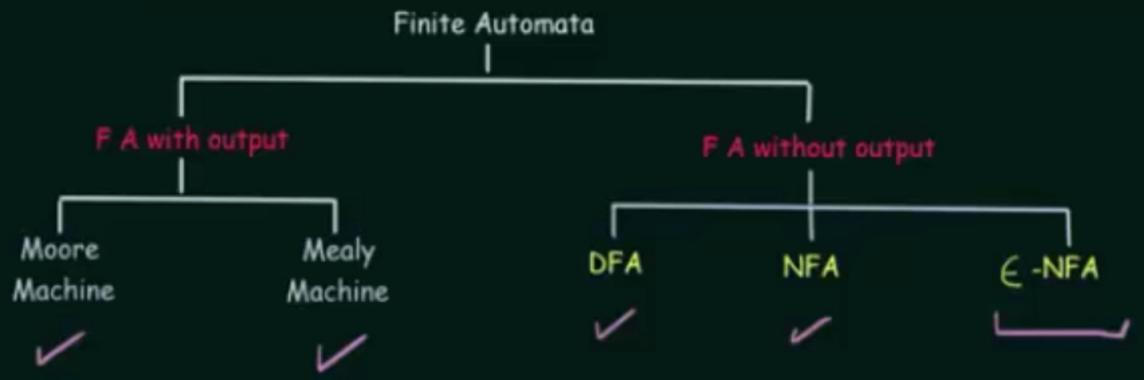
State	a	b
→ q_0	$q_3, 0$	$q_1, 1$
q_1	$q_0, 1$	$q_3, 0$
q_2	$q_2, 1$	$q_2, 0$
q_3	$q_1, 0$	$q_0, 1$

Moore

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



Epsilon (ϵ) - NFA

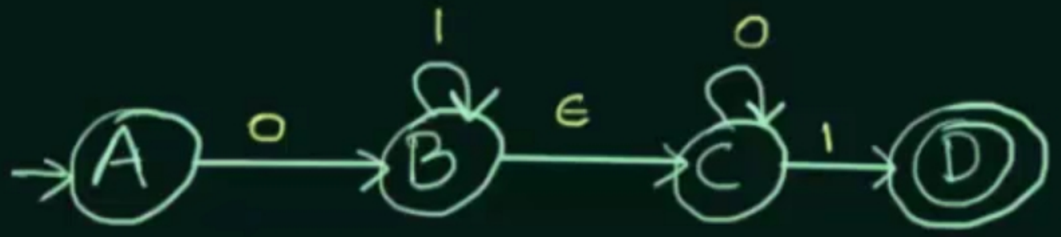


ϵ- NFA

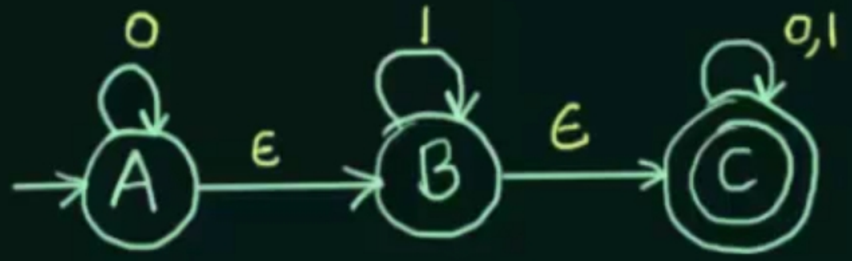
↳ empty symbols

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

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

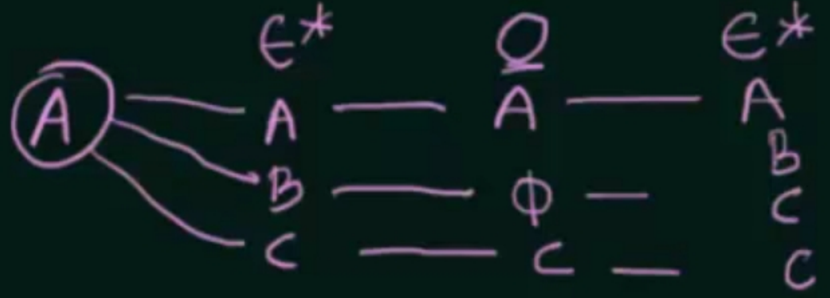


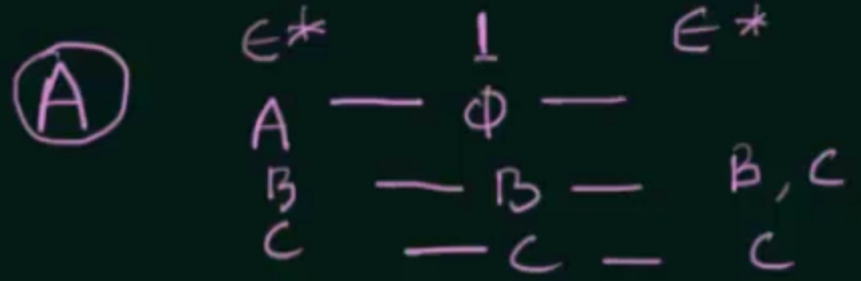
Every state on ϵ goes to itself.

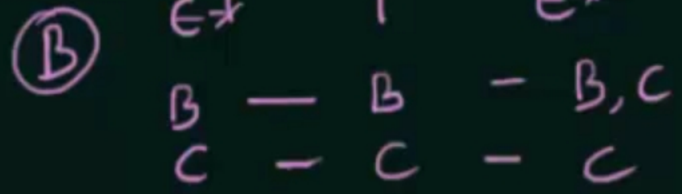


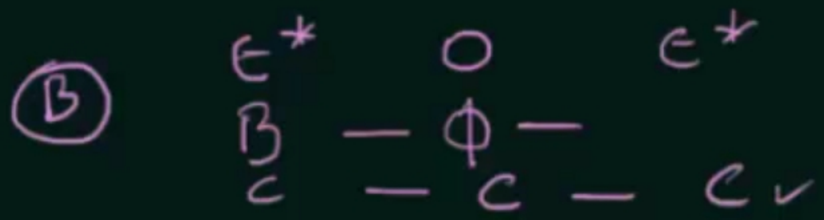
ε-Closure (ϵ^*) - All the states that can be reached from a particular state only by seeing the ε symbol

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









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

(B)

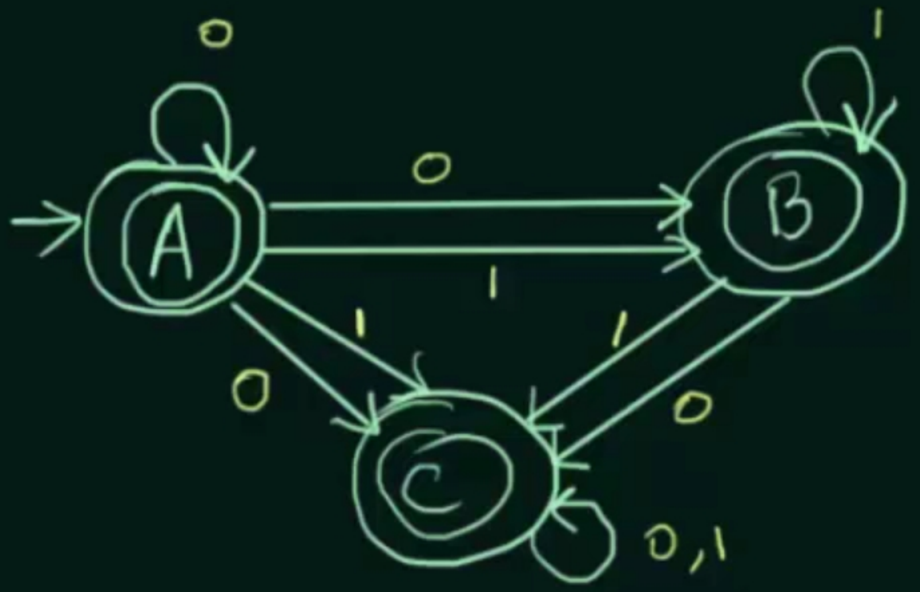
∈*	1	∈*
B	B	B, C
C	C	C

(A)

∈*	1	∈*
A	Φ	
B	B	B, C
C	C	C

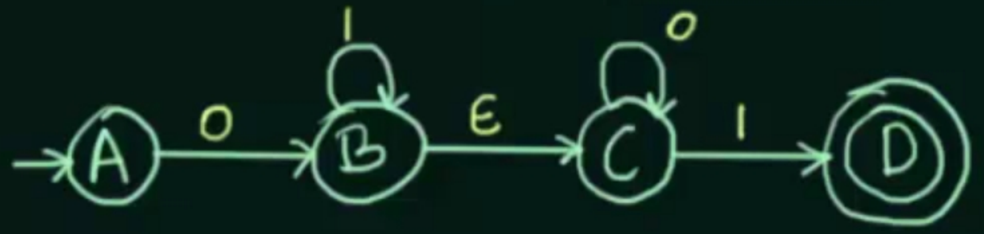
(B)

∈*	0	∈*
B	Φ	
C	C	C ✓



Conversion of ϵ -NFA to NFA -Examples (Part-1)

Convert the following ϵ-NFA to its equivalent NFA



NFA

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

	ε*	0	ε*	
A	A	B	B C	
B	B	Φ	Φ	
	C	C	C	
C	C	C	C	
D	D	Φ	Φ	

	ε*	1	ε*	
A	A	Φ	Φ	
B	B	B	B C	
	C	D	D	
C	C	D	D	
D	D	Φ	Φ	

NFA

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

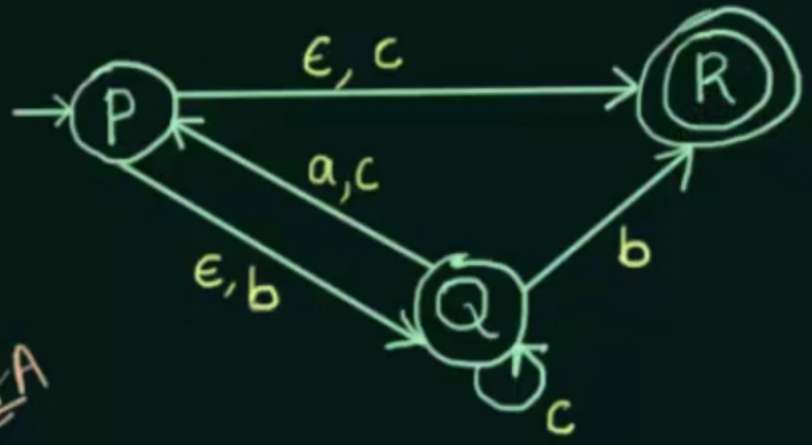
NFA

B	B	Φ	Φ
	C	C	C
C	C	C	C
D	D	Φ	Φ

B	B	B	B C
	C	D	D
C	C	D	D
D	D	Φ	Φ

Conversion of ϵ -NFA to NFA -Examples (Part-2)

Convert the following ϵ-NFA to its equivalent NFA



NFA

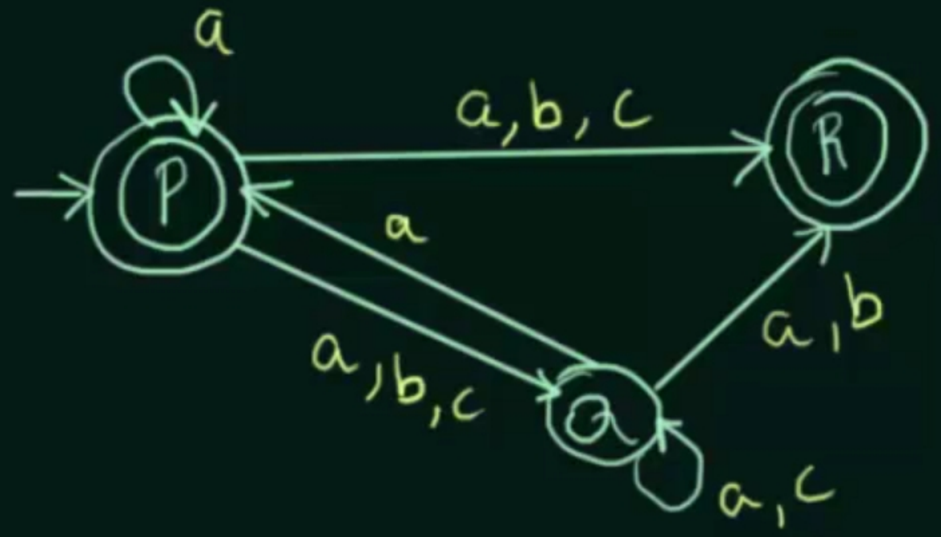
	a	b	c
→ (P)	{P, Q, R}	{Q, R}	{Q, R}
Q	{P, Q, R}	{R}	{Q}
(R)	ϕ	ϕ	ϕ

	ϵ^*	a	ϵ^*
P	P	ϕ	ϕ
	Q	P	P Q R
	R	ϕ	-
Q	Q	P	P Q R
R	R	ϕ	-

	ϵ^*	b	ϵ^*
P	P	Q	Q
	Q	R	R
	R	ϕ	-
Q	Q	R	R
R	R	ϕ	-

	ϵ^*	c	ϵ^*
P	P	R	R
	Q	Q	Q
	R	ϕ	-
Q	Q	Q	Q
R	R	ϕ	-

	a	b	c
→ (P)	{P, Q, R}	{Q, R}	{Q, R}
Q	{P, Q, R}	{R}	{Q}
(R)	ϕ	ϕ	ϕ



NFA

Regular Expression

Regular Expressions are used for representing certain sets of strings in an algebraic fashion.

1) Any terminal symbol i.e. symbols $\in \Sigma$ including $\wedge$ and Φ are regular expressions. $a, b, c, \ldots \wedge, \Phi$

2) The Union of two regular expressions is also a regular expression. $R_1, R_2 \quad (R_1 + R_2)$

3) The Concatenation of two regular expressions is also a regular expression. $R_1, R_2 \rightarrow (R_1 . R_2)$

4) The iteration (or Closure) of a regular expression is also a regular expression. $R \rightarrow R^* \quad a^* = \wedge, a, aa, aaa, \ldots$

5) The regular expression over Σ are precisely those obtained recursively by the application of the above rules once or several times.

Regular Expression - Examples

Describe the following sets as Regular Expressions

1) {0,1,2} 0 or 1 or 2

$R = 0 + 1 + 2$

2) {∧, ab}

$R = \wedge + ab$

3) {abb, a, b, bba} abb or a or b or bba

$R = abb + a + b + bba$

4) {∧, 0, 00, 000, ...} closure of 0

$R = 0^*$

5) {1, 11, 111, 1111, ...}

$R = 1^+$

Identities of Regular Expression

1) $\emptyset + R = R$

2) $\emptyset R + R\emptyset = \emptyset$

3) $\epsilon R = R\epsilon = R$

4) $\epsilon^* = \epsilon$ and $\emptyset^* = \epsilon$

5) $R + R = R$

6) $R^*R^* = R^*$

7) $RR^* = R^*R$

R^+

8) $(R^*)^* = R^*$

9) $\epsilon + RR^* = \epsilon + R^*R = R^*$

10) $(PQ)^*P = P(QP)^*$

11) $(P + Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$

12) $(P + Q)R = PR + QR$ and

$R(P + Q) = RP + RQ$

Identities of Regular Expression

1) $\emptyset + R = R$

2) $\emptyset R + R\emptyset = \emptyset$

3) $\epsilon R = R\epsilon = R$

4) $\epsilon^* = \epsilon$ and $\emptyset^* = \epsilon$

5) $R + R = R$

6) $R^*R^* = R^*$

7) $RR^* = R^*R$

R^+

8) $(R^*)^* = R^*$

9) $\epsilon + RR^* = \epsilon + R^*R = R^*$

10) $(PQ)^*P = P(QP)^*$

11) $(P + Q)^* = (P^* Q^*)^* = (P^* + Q^*)^*$

12) $(P + Q)R = PR + QR$ and

$R(P + Q) = RP + RQ$

ARDEN'S THEOREM

If P and Q are two Regular Expressions over Σ , and if P does not contain ϵ, then the following equation in R given by $R = Q + RP$ has a unique solution i.e. $R = QP^*$

$$R = Q + RP \longrightarrow (1)$$

$$= Q + QP^*P$$

$$= Q(\epsilon + P^*P)$$

$$= QP^*$$

Proved

$$R = QP^*$$

$$[\epsilon + R^*R = R^*]$$

$$R = Q + RP$$

$$= Q + [Q + RP]P$$

Padded

$$R = Q + RP$$

$$= Q + [Q + RP]P$$

$$= Q + QP + RP^2$$

$$= Q + QP + [Q + RP]P^2$$

$$= Q + QP + QP^2 + RP^3$$

$$\vdots$$

$$= Q + QP + QP^2 + \dots QP^n + RP^{n+1}$$

$$[R = QP^*]$$

$$= Q + QP + QP^2 + \dots QP^n + QP^* P^{n+1}$$

$$= Q[\epsilon + P + P^2 + \dots P^n + P^* P^{n+1}]$$

$$R = Q + RP$$

$$= Q + [Q + RP]P$$

$$= Q + QP + RP^2$$

$$= Q + QP + [Q + RP]P^2$$

$$= Q + QP + QP^2 + RP^3$$

$$\vdots$$

$$= Q + QP + QP^2 + \cdots QP^n + RP^{n+1}$$

$$[R = QP^*]$$

$$= Q + QP + QP^2 + \cdots QP^n + QP^* P^{n+1}$$

$$= Q[\epsilon + P + P^2 + \cdots P^n + P^* P^{n+1}]$$

$$R = QP^*$$

An Example Proof using Identities of Regular Expressions

Prove that (1+00*1) + (1+00*1) (0+10*1)* (0+10*1) is equal to 0*1 (0+10*1)*

$$LHS = (1+00^*1) + (1+00^*1)(0+10^*1)^*(0+10^*1)$$

$$= (1+00^*1)\left[\epsilon + (0+10^*1)^*(0+10^*1)\right]$$

$$= (1+00^*1)(0+10^*1)^*$$

$$= (\epsilon . 1+00^*1)(0+10^*1)^*$$

$$= (\epsilon+00^*)1(0+10^*1)^*$$

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

$\epsilon + R^*R = R^*$

$\epsilon . R = R$

Designing Regular Expressions - Examples (Part-1)

Design Regular Expression for the following languages over {a,b}

1) Language accepting strings of length exactly 2
2) Language accepting strings of length atleast 2
3) Language accepting strings of length atmost 2

Soln

1) $L_1 = \{aa, ab, ba, bb\}$

$R = aa + ab + ba + bb$

$= a(a+b) + b(a+b)$

$= (a+b)(a+b)$

2) $L_1 = \{aa, ab, ba, bb, aaa, \ldots\}$

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

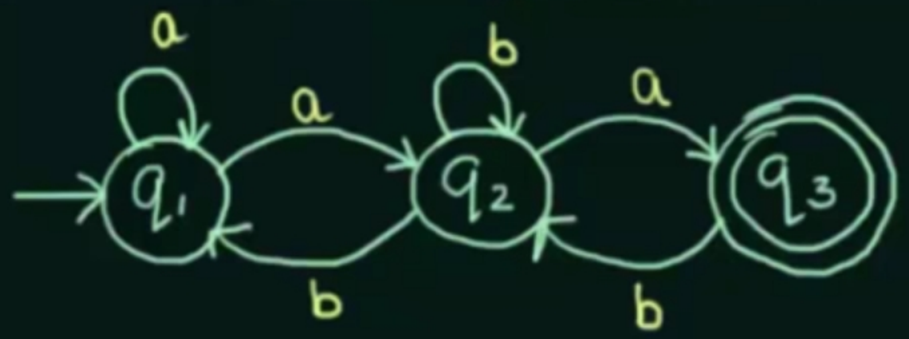
3) $L_1 = \{\epsilon, a, b, aa, ab, ba, bb\}$

$R = \epsilon + a + b + aa + ab + ba + bb$

$= (\epsilon + a + b)(\epsilon + a + b)$

Designing Regular Expression - Examples (Part-2)

Find the Regular Expression for the following NFA



$$q_3 = q_2 a \longrightarrow (1)$$

$$q_2 = q_1 a + q_2 b + q_3 b \longrightarrow (2)$$

$$q_1 = \epsilon + q_1 a + q_2 b \longrightarrow (3)$$

(1) $\rightarrow$ $q_3 = q_2 a$

$= (q_1 a + q_2 b + q_3 b) a$

$= q_1 aa + q_2 ba + q_3 ba \longrightarrow (4)$

(2) $\rightarrow$ $q_2 = q_1 a + q_2 b + q_3 b$ Putting value of q_3 from (1)

$= q_1 a + q_2 b + (q_2 a) b$

$= (q_1a + q_2b + q_3b)a$

$= q_1aa + q_2ba + q_3ba \longrightarrow (4)$

(2) → $q_2 = q_1a + q_2b + q_3b$ Putting value of q_3 from (1)

$= q_1a + q_2b + (q_2a)b$

$= q_1a + q_2b + q_2ab$

$\underbrace{q_2}_{R} = \underbrace{q_1a}_{Q} + \underbrace{q_2}_{R}\underbrace{(b + ab)}_{P}$

$R = Q + RP$ Arden's Theorem

$R = QP^*$

$q_2 = (q_1a)(b + ab)^* \longrightarrow (5)$

(3) → $q_1 =$

$$q_2 = (q_1 a)(b + ab)^* \rightarrow (5)$$

(3) $\Rightarrow$ $q_1 = \epsilon + q_1 a + q_2 b$

Putting value of q_2 from (5)

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

$R = Q + RP$

$R = QP^*$

$$\underbrace{q_1}_{R} = \underbrace{\epsilon}_{Q} + \underbrace{q_1}_{R} \underbrace{(a + a(b+ab)^*)b}_{P}$$

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

$\epsilon \cdot R = R$

$$q_1 = (a + a(b+ab)^* b)^* \rightarrow (6)$$

$\epsilon \cdot R = R$

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

$$q_1 = (a + a(b+ab)^* b)^* \rightarrow (6)$$

Final state (q_3)

$$q_3 = q_2 a$$

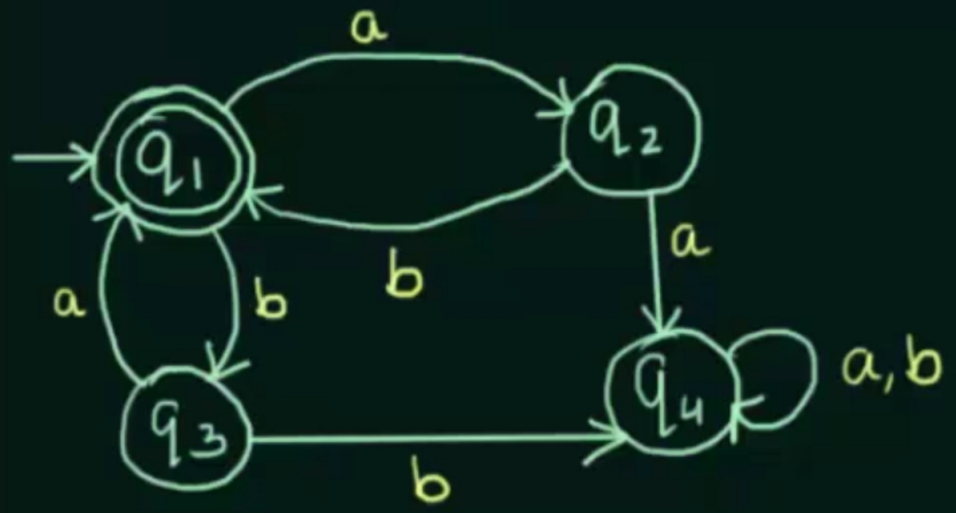
$$= q_1 a(b+ab)^* a \quad \text{Putting value of } q_2 \text{ from } (5)$$

$$q_3 = (a + a(b+ab)^* b)^* \; a(b+ab)^* a \quad \text{Putting value of } q_1 \text{ from } (6)$$

= Required Regular Expression for the given NFA

Designing Regular Expression - Examples (Part-3)

Find the Regular Expression for the following DFA



$q_1 = \epsilon + q_2 b + q_3 a \rightarrow (I)$

$q_2 = q_1 a \rightarrow (II)$

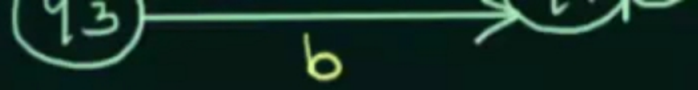
$q_3 = q_1 b \rightarrow (III)$

$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \rightarrow (IV)$

(I) $\rightarrow$ $q_1 = \epsilon + q_2 b + q_3 a$

Putting values of q_2 and q_3 from (II) and (III)

$q_1 = \epsilon + q_1 ab + q_1 ba$



$q_4 = q_2a + q_3b + q_4a + q_4b \rightarrow (IV)$

(I) $\rightarrow$ $q_1 = \epsilon + q_2b + q_3a$

Putting values of q_2 and q_3 from (II) and (III)

$q_1 = \epsilon + q_1ab + q_1ba$

$q_1 = \epsilon + q_1(ab + ba)$

R Q R P

$R = Q + RP$

$R = QP^*$ Arden's theorem

$q_1 = \epsilon\,(ab + ba)^*$

$\epsilon R = R$

$q_1 = (ab + ba)^*$

Pumping Lemma (For Regular Languages)

>> Pumping Lemma is used to prove that a Language is NOT REGULAR

>> It cannot be used to prove that a Language is Regular

If A is a Regular Language, then A has a Pumping Length 'P' such that any string 'S' where $|S| \geq P$ may be divided into 3 parts $S = x\ y\ z$ such that the following conditions must be true:

(1) $x\ y^i z \in A$ for every $i \geq 0$

(2) $|y| > 0$

(3) $|xy| \leq P$

To prove that a language is not Regular using PUMPING LEMMA, follow the below steps:

(We prove using Contradiction)

-> Assume that A is Regular

-> It has to have a Pumping Length (say P)

(2) $|y| > 0$

(3) $|xy| \leq P$

To prove that a language is not Regular using PUMPING LEMMA, follow the below steps:

(We prove using Contradiction)

-> Assume that A is Regular

-> It has to have a Pumping Length (say P)

-> All strings longer than P can be pumped $|S| \geq P$

-> Now find a string 'S' in A such that $|S| \geq P$

-> Divide S into x y z

-> Show that $x\, y^i z \notin A$ for some i

-> Then consider all ways that S can be divided into x y z

-> Show that none of these can satisfy all the 3 pumping conditions at the same time

-> S cannot be Pumped == CONTRADICTION

Regular Grammar

Noam Chomsky gave a Mathematical model of Grammar which is effective for writing computer languages

The four types of Grammar according to Noam Chomsky are:

Grammar Type	Grammar Accepted	Language Accepted	Automaton
TYPE-0	Unrestricted Grammar	Recursively Enumerable Language	Turing Machine
TYPE-1	Context Sensitive Grammar	Context Sensitive Language	Linear Bounded Automaton
TYPE-2	Context Free Grammar	Context Free Language	Pushdown Automata
TYPE-3	Regular Grammar	Regular Language	Finite State Automaton

Grammar:

A Grammar 'G' can be formally described using 4 tuples as $G = (V,T,S,P)$ where,

V = Set of Variables or Non-Terminal Symbols

T = Set of Terminal Symbols

S = Start Symbol

P = Production rules for Terminals and Non-Terminals

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

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

$V = \{S, A, B\}$

$T = \{a, b\}$

$S = S$

$P = S \rightarrow AB,\ A \rightarrow a,\ B \rightarrow b$

Eg.

$S \rightarrow AB$

$\rightarrow aB$

$\rightarrow ab$

Regular Grammar:

Regular Grammar can be divided into two types:

Right Linear Grammar

A grammar is said to be Right Linear if all productions are of the form

$A \rightarrow xB$

$A \rightarrow x$

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

Left Linear Grammar

A grammar is said to be Left Linear if all productions are of the form

$A \rightarrow Bx$

$A \rightarrow x$

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

Eg: $S \rightarrow abS \mid b$ - Right linear

$S \rightarrow Sbb \mid b$ - Left linear

Derivations from a Grammar

The set of all strings that can be derived from a Grammar is said to be the LANGUAGE generated from that Grammar

Example 1: Consider the Grammar $G1 = (\{S,A\}, \{a,b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \epsilon\})$

$S \rightarrow \underline{aA}b$ [by $S \rightarrow aAb$]

$\rightarrow aa\underline{A}bb$ [by $aA \rightarrow aaAb$]

$\rightarrow aaa\underline{A}bbb$ [by $aA \rightarrow aaAb$]

$\rightarrow aaabbb$ [by $A \rightarrow \epsilon$]

Example 2: $G2 = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\})$

$S \rightarrow AB$

$\rightarrow ab$

$L(G2) = \{ab\}$

$\rightarrow aaAbbb$ [by $aA \rightarrow aaAb$]

$\rightarrow aaabbb$ [by $A \rightarrow \epsilon$]

Example 2: $G2 = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\})$

$S \rightarrow AB$

$\rightarrow ab$

$L(G2) = \{ab\}$

Example 3: $G3 = (\{S,A,B\}, \{a,b\}, S, \{S \rightarrow AB, A \rightarrow aA|a, B \rightarrow bB|b\})$

$S \rightarrow AB$

$\rightarrow ab$

$S \rightarrow AB$

$\rightarrow aAbB$

$\rightarrow aabb$

$S \rightarrow AB$

$\rightarrow aAb$

$\rightarrow aab$

$S \rightarrow AB$

$\rightarrow abB$

$\rightarrow abb$

$L(G3) = \{ab, a^2b^2, a^2b, ab^2, \ldots\}$

$= \{a^m b^n \mid m \geq 0 \text{ and } n \geq 0\}$

Context Free Language

In formal language theory, a Context Free Language is a language generated by some Context Free Grammar.

The set of all CFL is identical to the set of languages accepted by Pushdown Automata.

Context Free Grammar is defined by 4 tuples as $G = \{ V, \Sigma, S, P \}$ where

V = Set of Variables or Non-Terminal Symbols

Σ = Set of Terminal Symbols

S = Start Symbol

P = Production Rule

Context Free Grammar has Production Rule of the form

$$A \rightarrow \alpha$$

where, $\alpha = \{V \cup \Sigma\}^*$ and $A \in V$

Example: For generating a language that generates equal number of a's and b's in the form $a^n b^n$, the Context Free Grammar wil be defined as

$G = \{ (S,A), (a,b), (S \rightarrow aAb, A \rightarrow aAb | \in) \}$

Context Free Grammar is defined by 4 tuples as $G = \{V, \Sigma, S, P\}$ where

V = Set of Variables or Non-Terminal Symbols

Σ = Set of Terminal Symbols

S = Start Symbol

P = Production Rule

Context Free Grammar has Production Rule of the form

$$A \rightarrow \alpha$$

where, $\alpha = \{V \cup \Sigma\}^*$ and $A \in V$

Example: For generating a language that generates equal number of a's and b's in the form $a^n b^n$, the Context Free Grammar wil be defined as

$G = \{ (S,A), (a,b), (S \rightarrow aAb, A \rightarrow aAb | \in) \}$

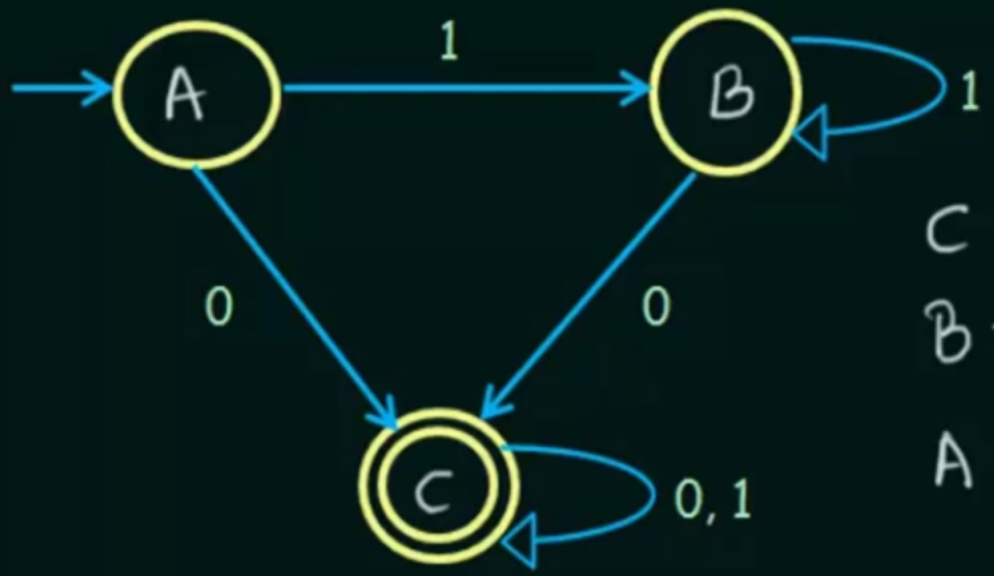
$S \rightarrow a\underline{A}b$

$\rightarrow aa\underline{A}bb$ (by $A \rightarrow aAb$)

$\rightarrow aaa\underline{A}bbb$ (")

$\Rightarrow aaabbb$ (by $A \rightarrow \in$)

$\rightarrow \underline{a^3}\underline{b^3} \Rightarrow a^n b^n$



$L(A) = L((11^*0 + 0)(0+1)^*0^*1^*)$

$C = A0 + B0 + C0 + C1 \longrightarrow (I)$

$B = A1 + B1 \longrightarrow (II)$

$A = \epsilon \longrightarrow (III)$

(II) $B = A1 + B1$

$= \epsilon 1 + B1$

$B = 1 + B1$

$R = Q + RP \Rightarrow R = QP^*$

$B = 11^*$

(I) $C = A0 + B0 + C0 + C1$

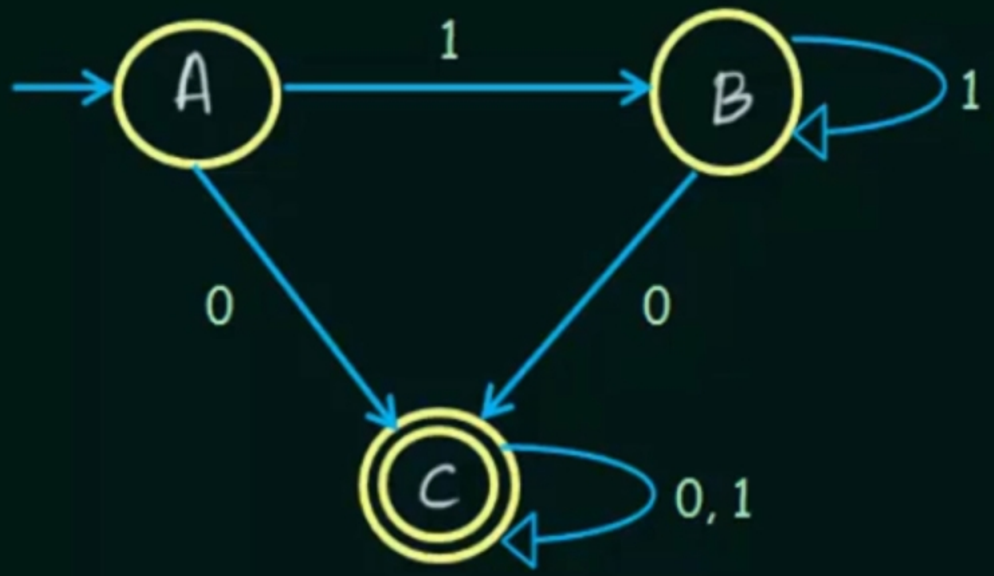
$C = \epsilon 0 + 11^*0 + C(0+1)$

$C = 0 + 11^*0 + C(0+1)$

$R = Q + RP$

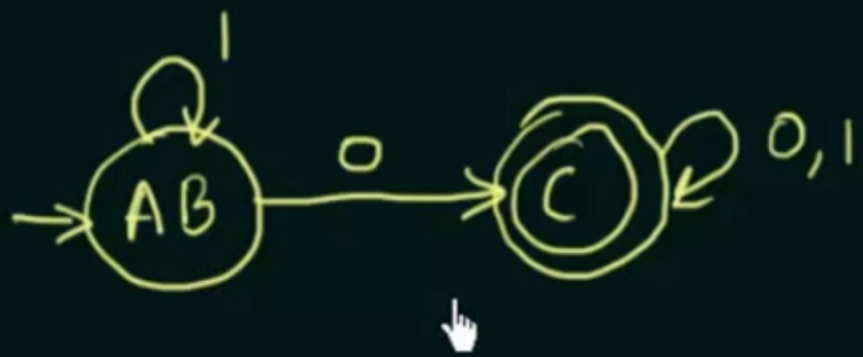
$C = 0 + 11^*0\,(0+1)^*$

$C = 11^*0 + 0\,(0+1)^*$ ✓



0 Equ : {A, B} {C}

1 Equ : {A, B} {C}

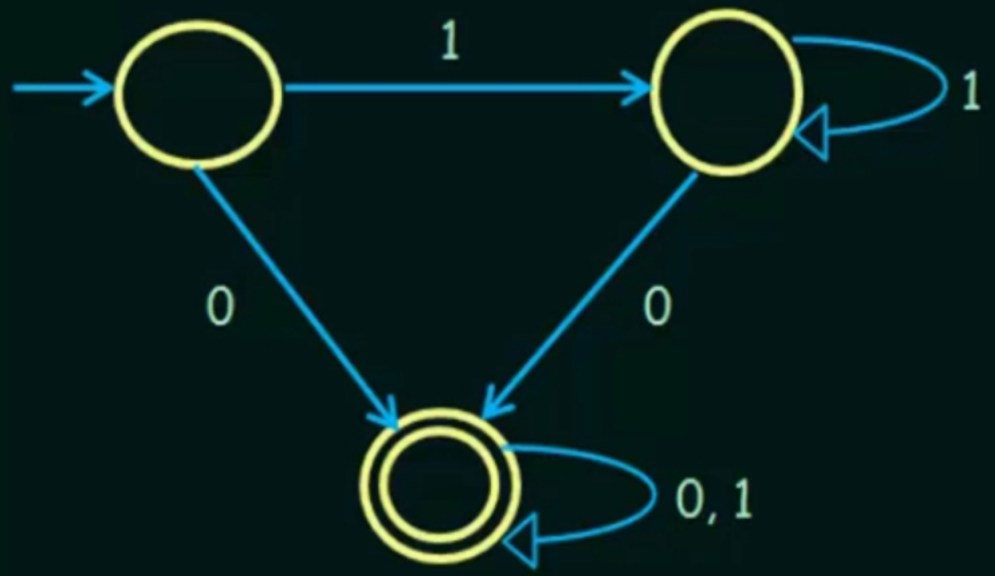


Regular Languages and Finite Automata

(Solved Problem - 1)

GATE 2013

Consider the DFA A given below:



Which of the following are FALSE?

1. Complement of L (A) is context-free

2. L (A) = L ((11* 0 + 0) (0 +1)* 0 *1*)

3. For the language accepted by A, A is the minimal DFA

4. A accepts all strings over {0, 1} of length at least 2

(A) 1 and 3 only
(B) 2 and 4 only
(C) 2 and 3 only
(D) 3 and 4 only

Regular Languages and Finite Automata

(Solved Problem - 2)

GATE 2013

Consider the languages $L_1 = \Phi$ and $L_2 = \{a\}$. Which one of the following represents $L_1L_2^* \cup L_1^*$?

(A) $\{\epsilon\}$ ✓
(B) Φ
(C) a^*
(D) $\{\epsilon, a\}$

$L_1 L_2^* \cup L_1^*$

$\Phi . a^* \cup L_1^*$

$\Phi \cup \Phi^*$

$\Phi \cup \epsilon$

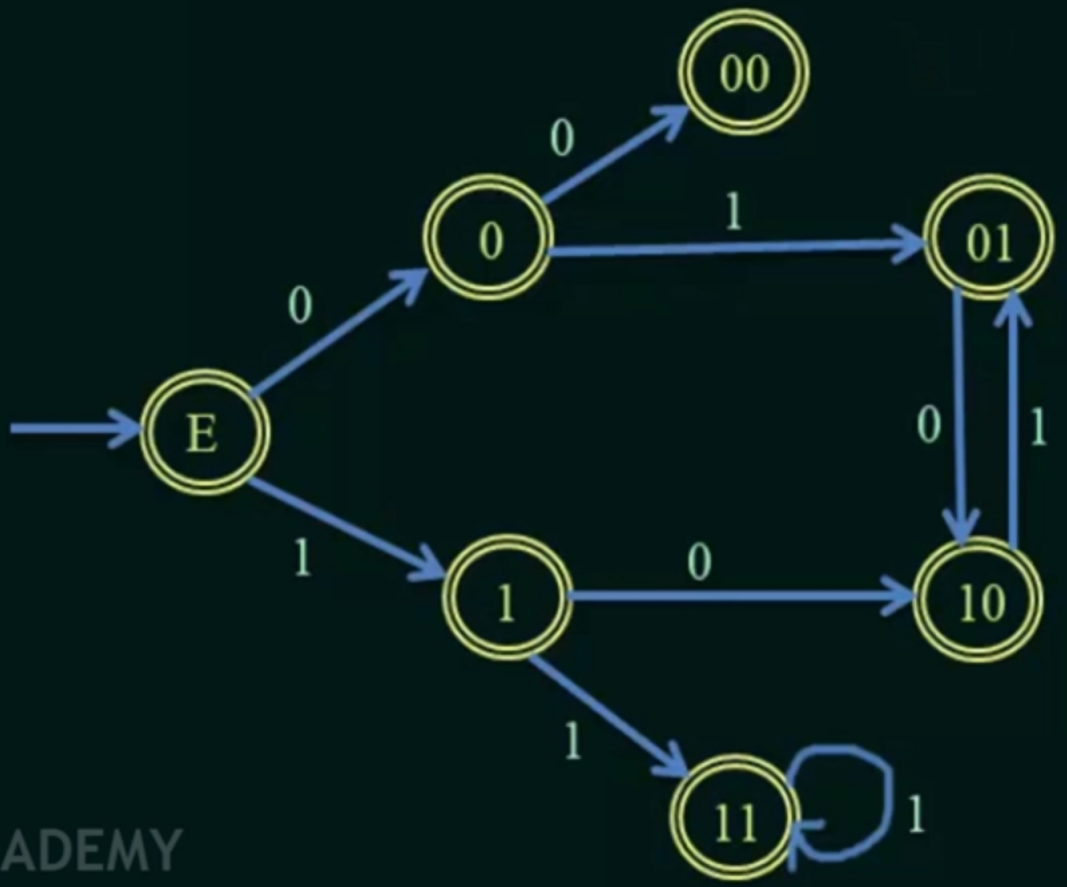
ϵ

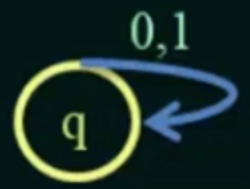
Regular Languages and Finite Automata

(Solved Problem - 3)

GATE 2012

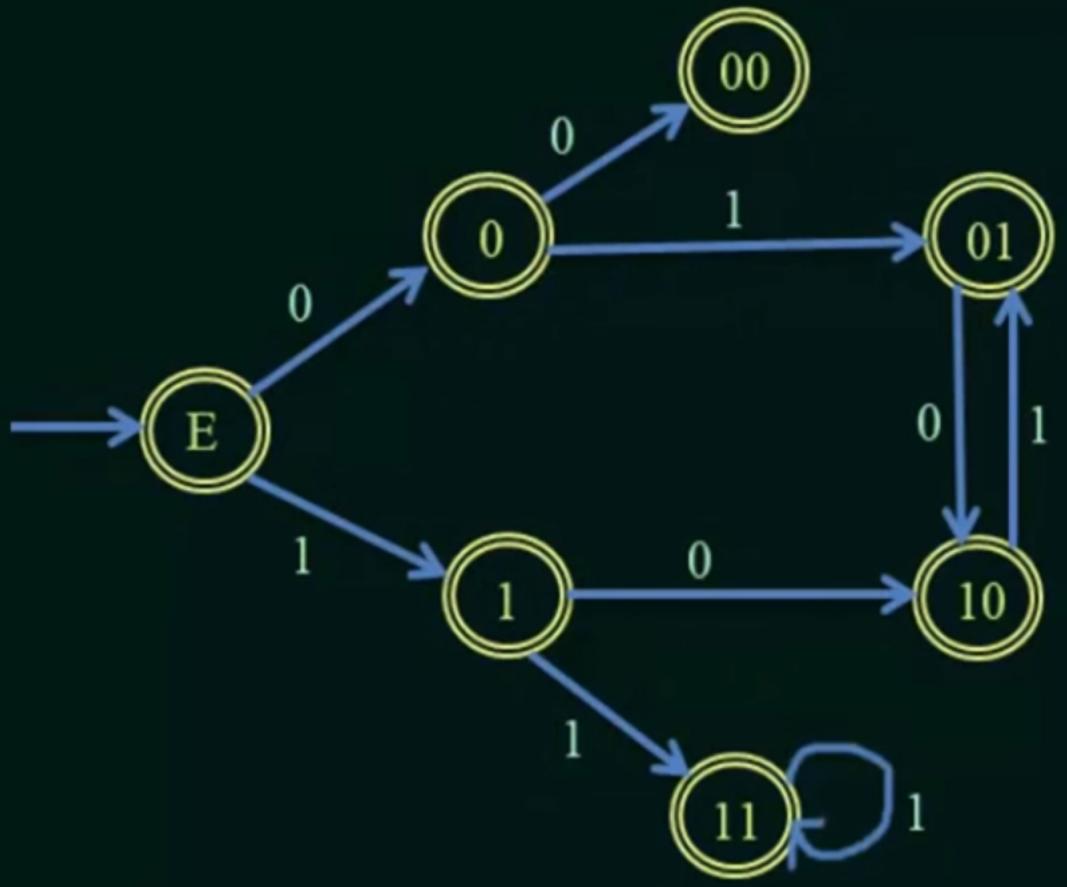
Consider the set of strings on {0, 1} in which, every substring of 3 symbols has at most two zeros. For example, 001110 and 011001 are in the language, but 100010 is not. All strings of length less than 3 are also in the language. A partially completed DFA that accepts this language is shown below.





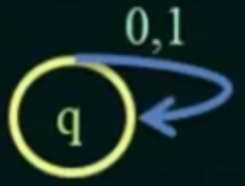
The missing arcs in the DFA are:

(A) A
(B) B
(C) C
(D) D



The missing arcs in the DFA are:

(A) A
(B) B
(C) C
(D) D



(C)

	00	01	10	11	q
00		1			0
01		1			
10			0		
11		0			

(A)

	00	01	10	11	q
00	1	0			
01				1	
10	0				
11			0		

(B)

	00	01	10	11	q
00		0			1
01		1			
10				0	
11		0			

(D)

	00	01	10	11	q
00		1			0
01				1	
10	0				
11			0		

1) Start with the Start Symbol and choose the closest production that matches to the given string.
2) Replace the Variables with its most appropriate production. Repeat the process until the string is generated or until no other productions are left.

Example: Verify whether the Grammar $S \rightarrow 0B|1A,\ A \rightarrow 0|0S|1AA|\hat{},\ B \rightarrow 1|1S|0BB$ generates the string 00110101

$S \rightarrow 0B$ $(S \rightarrow 0B)$
$\rightarrow 00BB$ $(B \rightarrow 0BB)$
$\rightarrow 001B$ $(B \rightarrow 1)$
$\rightarrow 0011S$ $(B \rightarrow 1S)$
$\rightarrow 00110B$ $(S \rightarrow 0B)$
$\rightarrow 001101S$ $(B \rightarrow 1S)$
$\rightarrow 0011010B$ $(S \rightarrow 0B)$
$\rightarrow 00110101$ $(B \rightarrow 1)$

Example: Verify whether the Grammar $S \rightarrow aAb$, $A \rightarrow aAb \mid \wedge$ generates the string aabbb

$S \rightarrow aAb$

$\rightarrow aaAbb \quad (A \rightarrow aAb)$

$\rightarrow aabb \quad (A \rightarrow \wedge)$

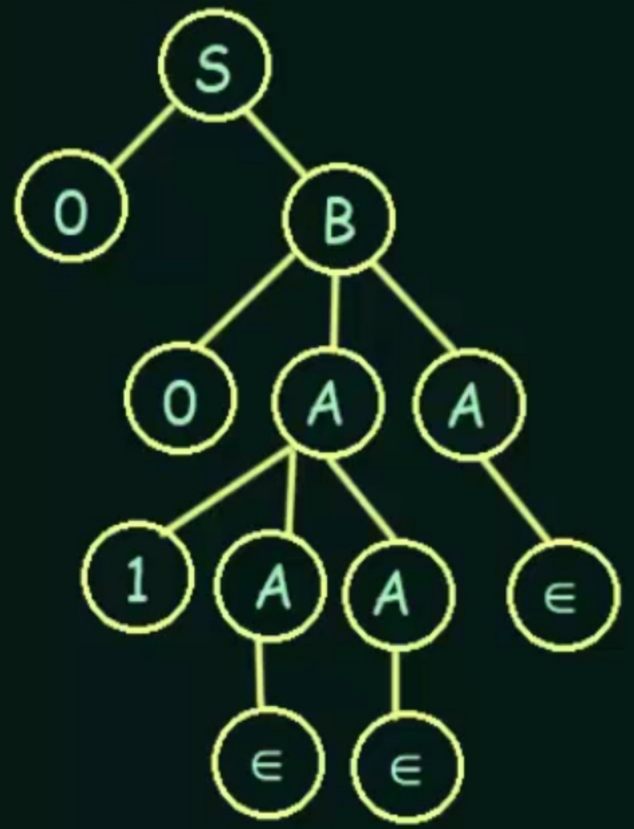
$aaaAbbb \quad (A \rightarrow aAb)$

$aaabbb \quad (A \rightarrow \wedge)$ X

Derivation Tree

A Derivation Tree or Parse Tree is an ordered rooted tree that graphically represents the semantic information of strings derived from a Context Free Grammar

Example: For the Grammar $G = \{V, T, P, S\}$ where $S \rightarrow 0B$, $A \rightarrow 1AA \mid \in$, $B \rightarrow 0AA$



Root Vertex: Must be labelled by the Start Symbol

Vertex: Labelled by Non-Terminal Symbols

Leaves: Labelled by Terminal Symbols or $\in$

Left Derivation Tree

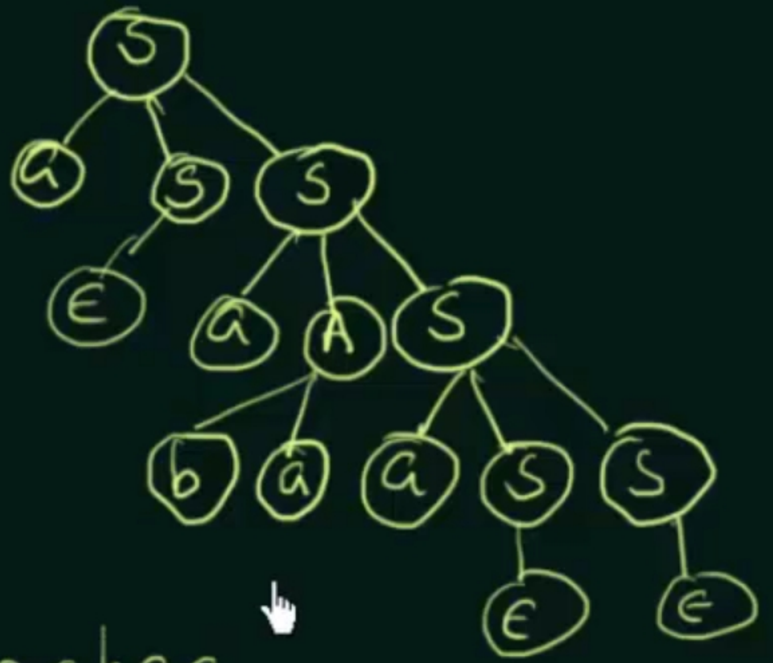
A Left Derivation Tree is obtained by applying production to the leftmost variable in each step.

Right Derivation Tree

A Right Derivation Tree is obtained by applying production to the rightmost variable in each step.

Eg. For generating the string aabaa from the Grammar $S \rightarrow aAS|aSS|\in$, $A \rightarrow SbA|ba$

aabaa



aabaa

Ambiguous Grammar

A Grammar is said to be Ambiguous if there exists two or more derivation tree for a string w (that means two or more left derivation trees)

Example: $G = (\{S\}, \{a+b, +, *\}, P, S)$ where P consists of $S \rightarrow S+S \mid S*S \mid a \mid b$

The String $a + a * b$ can be generated as:

$S \rightarrow S+S$
$\rightarrow a+S$
$\rightarrow a+S*S$
$\rightarrow a+a*S$
$\rightarrow a+a*b$

$S \rightarrow S*S$
$\rightarrow S+S*S$
$\rightarrow a+S*S$
$\rightarrow a+a*S$
$\rightarrow a+a*b$

Thus, this Grammar is Ambiguous

Simplification of Context Free Grammar

Reduction of CFG

In CFG, sometimes all the production rules and symbols are not needed for the derivation of strings. Besides this, there may also be some NULL Productions and UNIT Productions. Elimination of these productions and symbols is called Simplification of CFG.

Simplification consists of the following steps:

1) Reduction of CFG 2) Removal of Unit Productions 3) Removal of Null Productions

REDUCTION OF CFG

CFG are reduced in two phases

Phase 1: Derivation of an equivalent grammar G' , from the CFG, G, such that each variable derives some terminal string

Derivation Procedure:

Step 1: Include all Symbols W_1 ,that derives some terminal and initialize i = 1

Step 2: Include symbols W_{i+1} , that derives W_i

Step 3: Increment i and repeat Step 2, until $W_{i+1} = W_i$

Step 4: Include all production rules that have W_i in it

REDUCTION OF CFG

CFG are reduced in two phases

Phase 1: Derivation of an equivalent grammar G' , from the CFG, G, such that each variable derives some terminal string

Derivation Procedure:

Step 1: Include all Symbols W_1 ,that derives some terminal and initialize i = 1

Step 2: Include symbols W_{i+1} , that derives W_i

Step 3: Increment i and repeat Step 2, until $W_{i+1} = W_i$

Step 4: Include all production rules that have W_i in it

Phase 2: Derivation of an equivalent grammar G'' , from the CFG, G', such that each symbol appears in a sentential form

Derivation Procudure:

Step 1: Include the Start Symbol in Y_1 and initialize i =1

Step 2: Include all symbols Y_{i+1} , that can be derived from Y_i and include all production rules that have been applied

Step 3: Increment i and repeat Step 2, until $Y_{i+1} = Y_i$

Example: Find a reduced grammar equivalent to the grammar G, having production rules

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

Phase 1

$T = \{a, c, e\}$

$W_1 = \{A, C, E\}$

$W_2 = \{A, C, E, S\}$

$W_3 = \{A, C, E, S\}$

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

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

$W_2 = \{A, C, E, S\}$

$W_3 = \{A, C, E, S\}$

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

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

Phase 2 : $Y_1 = \{S\}$

$Y_2 = \{S, A, C\}$

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

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

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

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

Simplification of Context Free Grammar

Removal of Unit Productions

Any Production Rule of the form $A \rightarrow B$ where $A, B \in$ Non Terminals is called Unit Production

Procedure for Removal

Step 1: To remove $A \rightarrow B$, add production $A \rightarrow x$ to the grammar rule whenever $B \rightarrow x$ occurs in the grammar. [$x \in$ Terminal, x can be Null]

Step 2: Delete $A \rightarrow B$ from the grammar.

Step 3: Repeat from Step 1 until all Unit Productions are removed.

Example: Remove Unit Productions from the Grammar whose production rule is given by

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

$Y \rightarrow Z$, $Z \rightarrow M$, $M \rightarrow N$

1) Since $N \rightarrow a$, we add $M \rightarrow a$

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

Example: Remove Unit Productions from the Grammar whose production rule is given by

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

$Y \to Z,\quad Z \to M,\quad M \to N$

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

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

2) Since $M \to a$, we add $Z \to a$

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

3) Since $Z \to a$, we add $Y \to a$

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

Remove the Unreachable symbols

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

Simplification of Context Free Grammar

Removal of Null Productions

In a CFG, a Non-Terminal Symbol 'A' is a nullable variable if there is a production $A \rightarrow \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ. (Like $A \rightarrow \ldots\ldots\ldots\ldots \rightarrow \epsilon$)

Procedure for Removal:

Step 1: To remove $A \rightarrow \epsilon$, look for all productions whose right side contains A

Step2: Replace each occurences of 'A' in each of these productions with ϵ

Step 3: Add the resultant productions to the Grammar

Example: Remove Null Productions from the following Grammar

$S \rightarrow ABAC,\quad A \rightarrow aA \mid \epsilon,\quad B \rightarrow bB \mid \epsilon,\quad C \rightarrow c$

$A \rightarrow \epsilon$, $B \rightarrow \epsilon$

1) To eliminate $A \rightarrow \epsilon$

$S \rightarrow ABAC$

$S \rightarrow ABC \mid BAC \mid BC$

$A \rightarrow aA$

$A \rightarrow a$

New production: $S \rightarrow ABAC \mid ABC \mid BAC \mid BC$

$A \rightarrow aA \mid a$, $B \rightarrow bB \mid \epsilon$, $C \rightarrow c$

2) To eliminate $B \rightarrow \epsilon$

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

New production: $S \rightarrow ABAC \mid ABC \mid BAC \mid BC \mid AAC \mid AC \mid c$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$C \rightarrow c$

Steps to convert a given CFG to Chomsky Normal Form:

Step 1: If the Start Symbol S occurs on some right side, create a new Start Symbol S' and a new Production $S' \rightarrow S$.

Step 2: Remove Null Productions. (Using the Null Production Removal discussed in previous Lecture)

Step 3: Remove Unit Productions. (Using the Unit Production Removal discussed in previous Lecture)

Step 4: Replace each Production $A \rightarrow B_1 .....B_n$ where $n > 2$, with $A \rightarrow B_1 C$ where $C \rightarrow B_2.....B_n$ Repeat this step for all Productions having two or more Symbols on the right side.

Step 5: If the right side of any Production is in the form $A \rightarrow aB$ where 'a' is a terminal and A and B are non-terminals, then the Production is replaced by $A \rightarrow XB$ and $X \rightarrow a$. Repeat this step for every Production which is of the form $A \rightarrow aB$

2) Remove the Null Productions: $B \to \in$ and $A \to \in$:

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

After Removing $A \to \in$: P: $S' \to S,\ S \to ASA|aB|a|AS|SA|S,\ A \to B|S,\ B \to b$

3) Remove the Unit Productions: $S \to S$, $S' \to S$, $A \to B$ and $A \to S$:

After Removing $S \to S$: P: $S' \to S,\ S \to ASA|aB|a|AS|SA,\ A \to B|S,\ B \to b$

After Removing $S' \to S$: P: $S' \to ASA|aB|a|AS|SA,$
$S \to ASA|aB|a|AS|SA,$
$A \to B|S,\ B \to b$

After Removing $A \to B$: P: $S' \to ASA|aB|a|AS|SA,$
$S \to ASA|aB|a|AS|SA,$
$A \to b|S,\ B \to b$

After Removing $A \to S$: P: $S' \to ASA|aB|a|AS|SA,$
$S \to ASA|aB|a|AS|SA,$
$A \to b|ASA|aB|a|AS|SA,$
$B \to b$

4) Now find out the productions that has more than TWO variables in RHS

S'→ASA, S→ASA and A→ASA

After removing these, we get: P: S'→AX|aB|a|AS|SA,
S →AX|aB|a|AS|SA,
A →b|AX|aB|a|AS|SA,
B →b,
X → SA

5) Now change the productions S'→aB, S→aB and A→aB

Finally we get: P: S'→ AX|YB|a|AS|SA,
S → AX|YB|a|AS|SA,
A → b|AX|YB|a|AS|SA,
B → b,
X → SA,
Y → a

which is the required Chomsky Normal Form for the given CFG

Greibach Normal Form

A CFG is in Greibach Normal Form if the productions are in the following forms:

$$A \rightarrow b$$
$$A \rightarrow bC_1C_2 \ldots\ldots C_n$$

where $A, C_1, \ldots\ldots, C_n$ are Non-Terminals and b is a Terminal

Steps to convert a given CFG to GNF:

Step 1: Check if the given CFG has any Unit Productions or Null Productions and Remove if there are any (using the Unit & Null Productions removal techniques discussed in the previous lecture)

Step 2: Check whether the CFG is already in Chomsky Normal Form (CNF) and convert it to CNF if it is not. (using the CFG to CNF conversion technique discussed in the previous lecture)

Step 3: Change the names of the Non-Terminal Symbols into some A_i in ascending order of i

Example: $S \rightarrow CA \mid BB$

$B \rightarrow b \mid SB$

$C \rightarrow b$

$A \rightarrow a$

Replace: S with A_1

C with A_2

A with A_3

B with A_4

We get:

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Step 4: Alter the rules so that the Non -Terminals are in ascending order, such that,

If the Production is of the form $A_i \rightarrow A_j\, x$, then,

$i < j$ and should never be $i \geqslant j$

$A_4 \rightarrow b \mid \underline{A_1} A_4$

$A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Step 4: Alter the rules so that the Non -Terminals are in ascending order, such that,

If the Production is of the form $A_i \rightarrow A_j\ x$, then,

$i < j$ and should never be $i \geqslant j$

$A_4 \rightarrow b \mid A_1 A_4$

$A_4 \rightarrow b \mid A_2 A_3 A_4 \mid A_4 A_4 A_4$

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

↓

Left Recursion

Step 5: Remove Left Recursion

Greibach Normal Form

(Conversion of CFG to GNF - Removal of Left Recursion)

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

$A_4 \rightarrow b \mid A_1 A_4$ ⟶ $A_4 \rightarrow b \mid b A_3 A_4 \mid A_4 A_4 A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Left Recursion

Step 5: Remove Left Recursion

Introduce a New Variable to remove the Left Recursion

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

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

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

$b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

Now the grammar is:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$

$Z \rightarrow bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4 \mid$

$bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z$

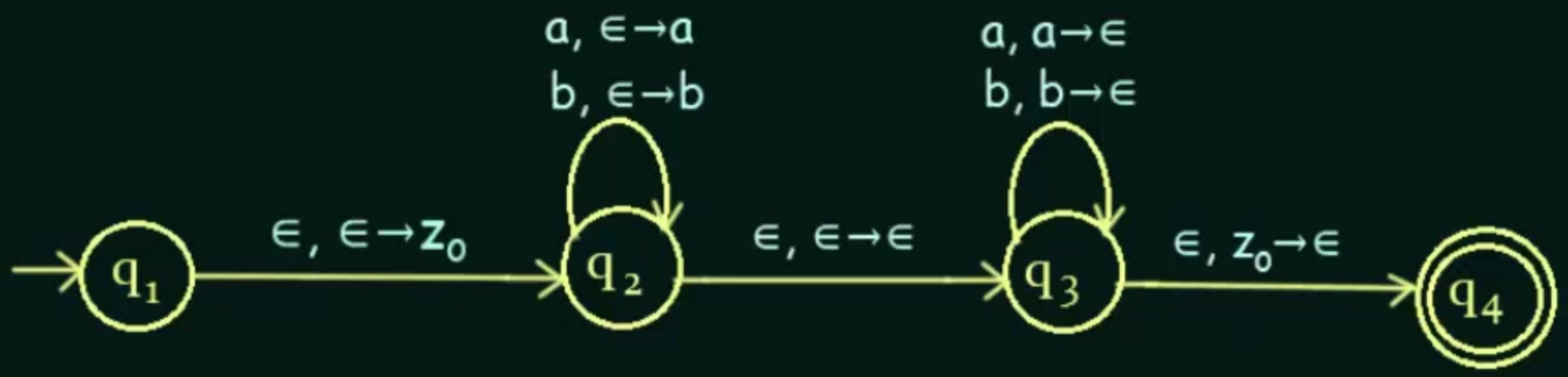
$A_2 \rightarrow b$

$A_3 \rightarrow a$

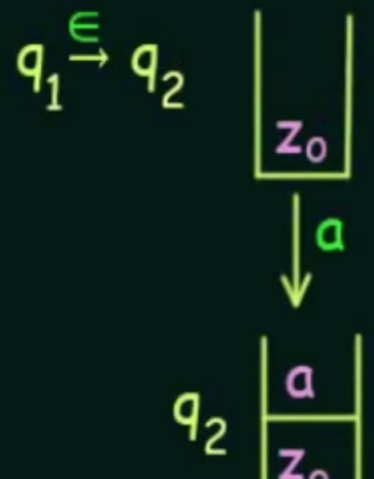
Pushdown Automata - Example (Even Palindrome) PART- 3

Construct a PDA that accepts Even Palindromes of the form

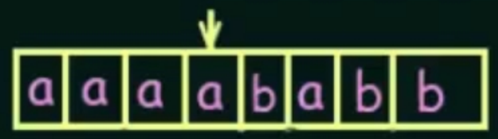
$$L = \{ ww^R \mid w = (a+b)^+ \}$$



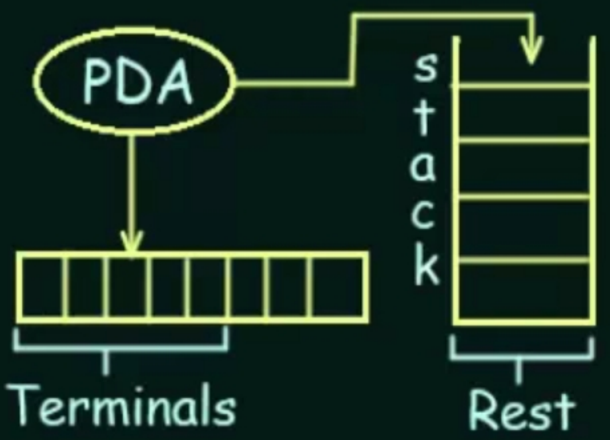
Example: ∈ a ∈ b ∈ a ∈ b ∈



FSM: The Input String

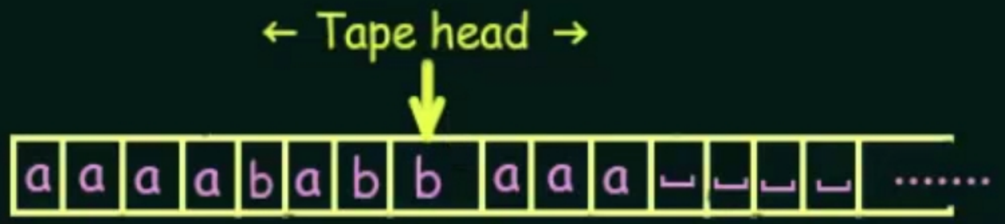


PDA: -> The Input String
-> A Stack



TURING MACHINE:

-> A Tape



Tape Alphabets: $\Sigma = \{ 0,1,a,b,x,Z_0 \}$

The Blank ␣ is a special symbol. ␣ $\notin \Sigma$

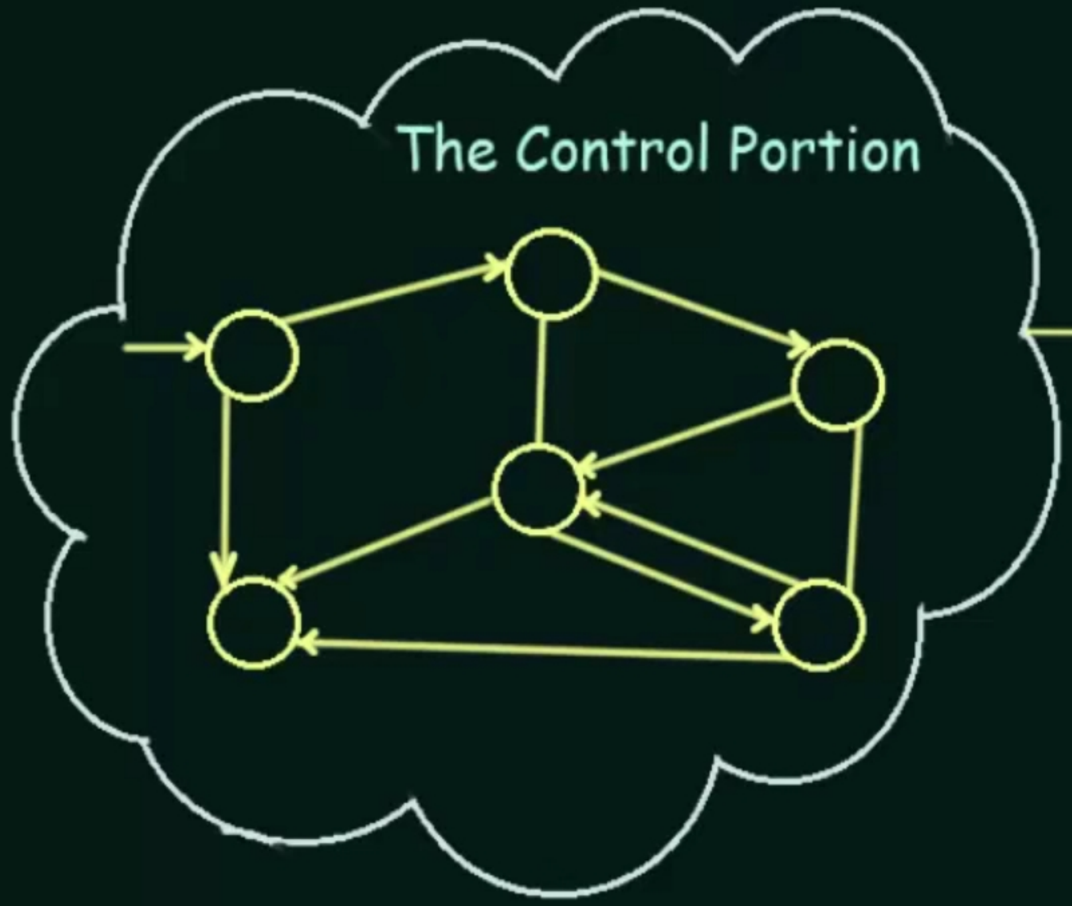
The blank is a special symbol used the fill the infinite tape

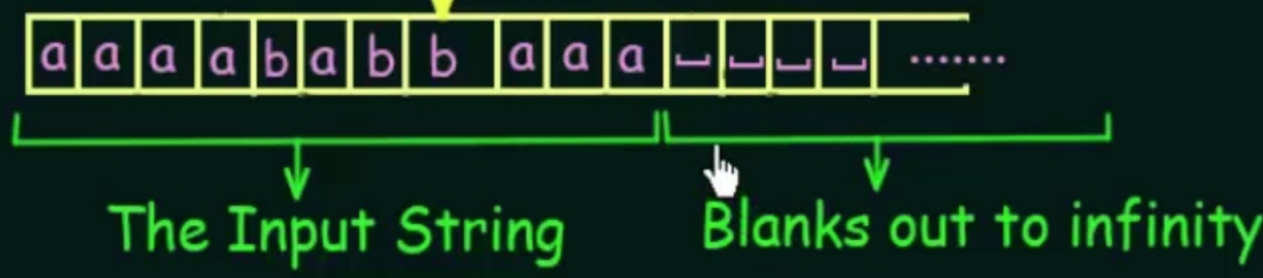
Initial Configuration:

Operations on the Tape:

-> Read /Scan symbol below the Tape Head
-> Update / Write a symbol below the Tape Head
-> Move the Tape Head one step LEFT
-> Move the Tape Head one step RIGHT

Turing Machine - Introduction (Part-2)





The Control Portion similar to FSM or PDA

The PROGRAM

It is deterministic

Rules of Operation - 1

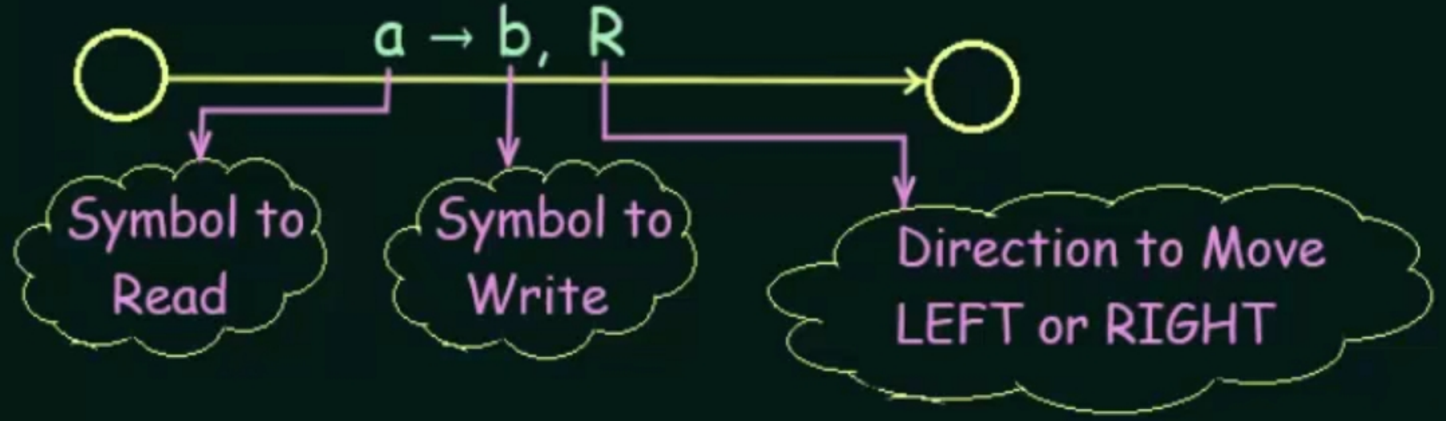
At each step of the computation:

-> Read the currect symbol

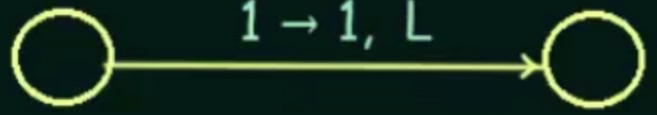
-> Update (i.e. write) the same cell

-> Move exactly one cell either LEFT or RIGHT

If we are at the left end of the tape, and trying to move LEFT, then do not move.
Stay at the left end



If you dont want to update the cell,
JUST WRITE THE SAME SYMBOL



Rules of Operation - 2

-> Control is with a sort of FSM

-> Initial State

-> Final States: (there are two final states)

1) The ACCEPT STATE
2) The REJECT STATE

-> Computation can either

1) HALT and ACCEPT
2) HALT and REJECT
3) LOOP (the machine fails to HALT)

Turing Machine (Formal Definition)

A Turing Machine can be defined as a set of 7 tuples

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

$Q \rightarrow$ Non empty set of States

$\Sigma \rightarrow$ Non empty set of Symbols

$\Gamma \rightarrow$ Non empty set of Tape Symbols

$\delta \rightarrow$ Transition function defined as

$$Q \times \Sigma \rightarrow \Gamma \times (R/L) \times Q$$

$q_0 \rightarrow$ Initial State

$b \rightarrow$ Blank Symbol

$F \rightarrow$ Set of Final states (Accept state & Reject State)

Thus, the Production rule of Turing Machine will be written as

$$\delta (q_0, a) \rightarrow (q_1, y, R)$$

Turing's Thesis:

Turing's Thesis states that any computation that can be carried out by mechanical means can be performed by some Turing Machine.

Few arguments for accepting this thesis are:

i. Anything that can be done on existing digital computer can also be done by Turing Machine.
ii. No one has yet been able to suggest a problem solvable by what we consider on algorithm, for which a Turing Machine Program cannot be written.

Recursively Enumerable Language:

A Language L and Σ is said to be Recursively Enumerable if there exists a Turing Machine that accepts it.

Turing Machine - Example (Part-1)

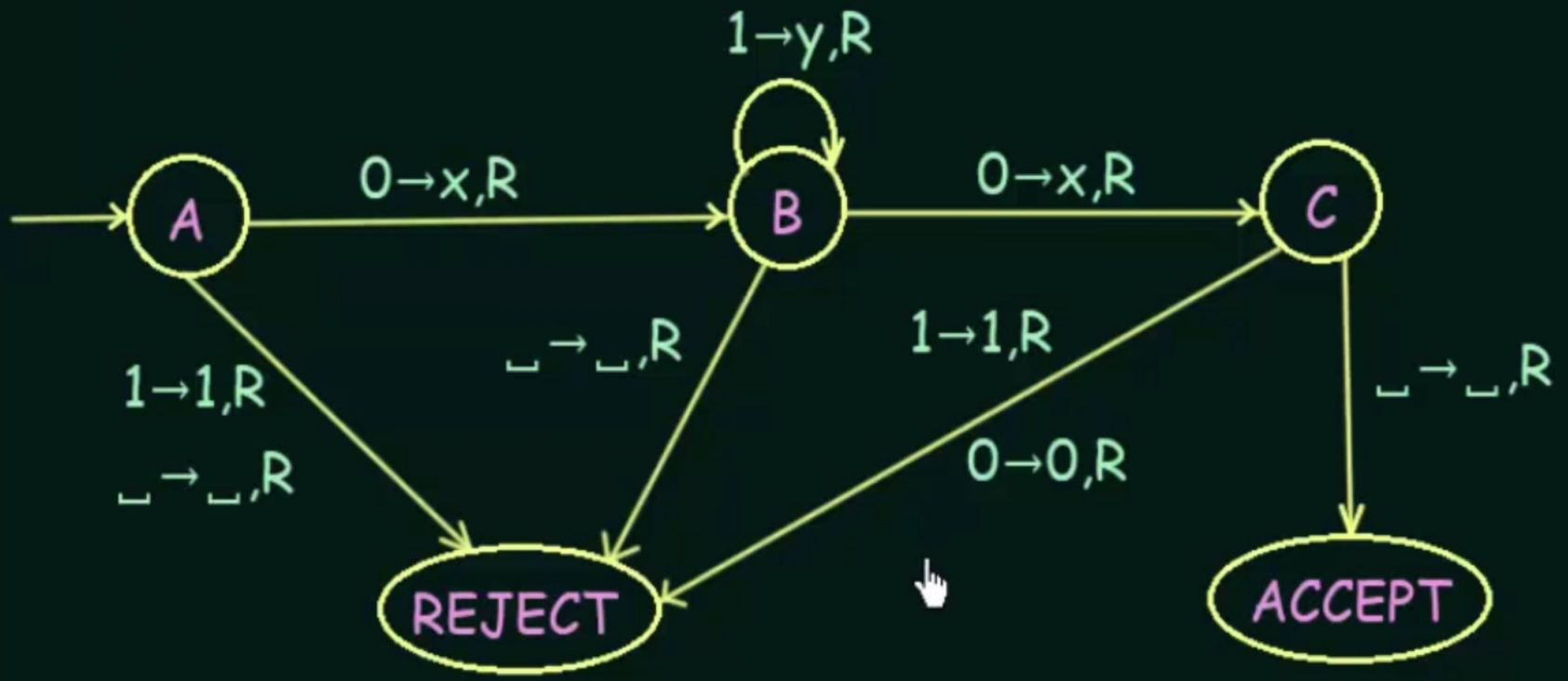
Design a Turing Machine which recognizes the language

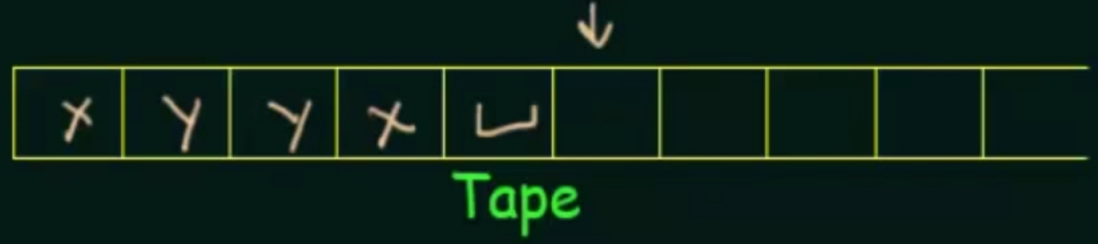
$L = 01^*0$

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

$b = \sqcup$

0110 ✓





Turing Machine - Example (Part-1)

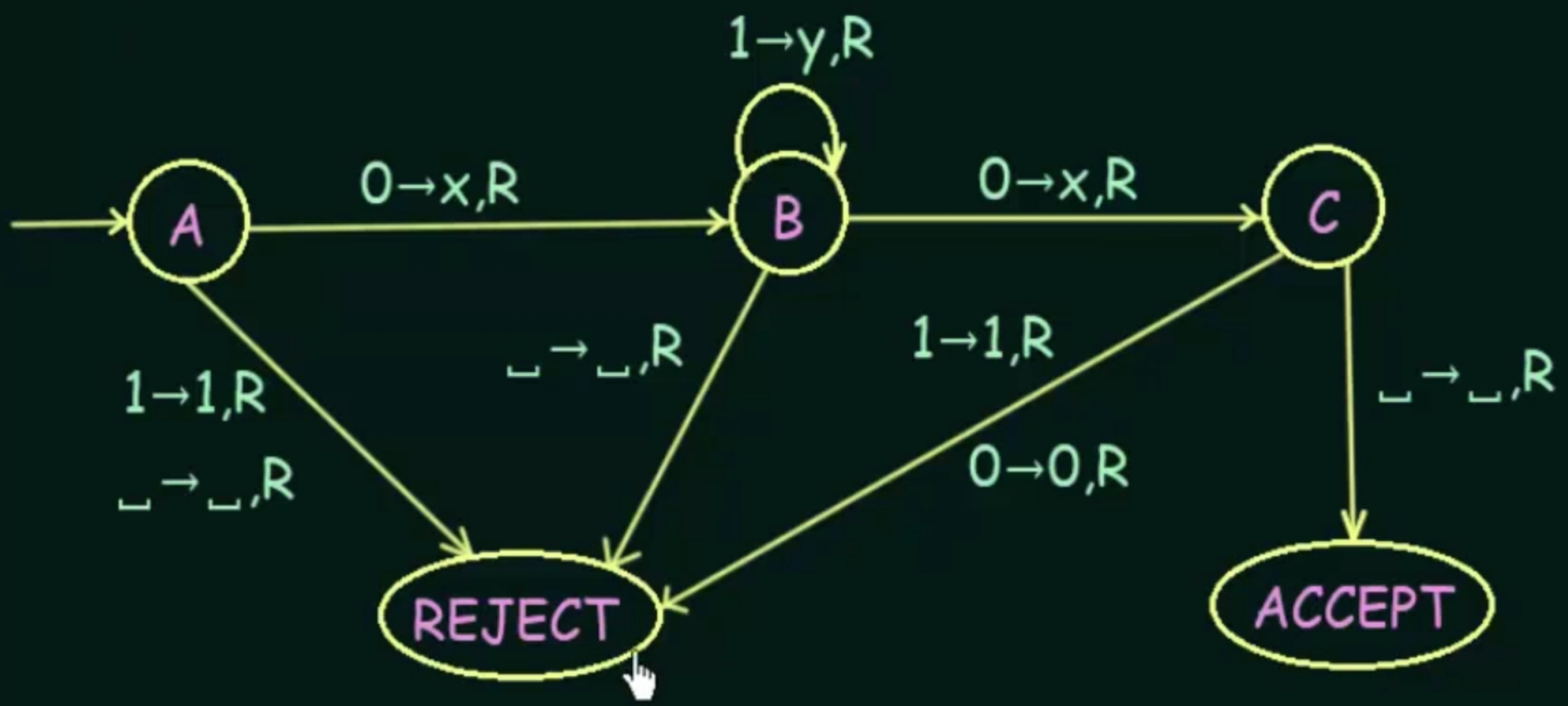
Design a Turing Machine which recognizes the language

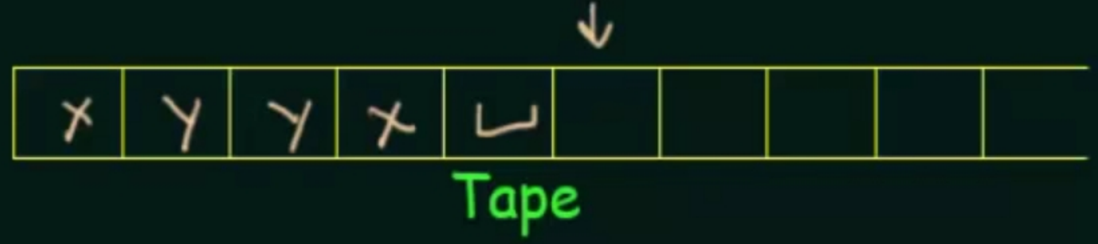
L = 01*0

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

b = ␣

0110 ✓



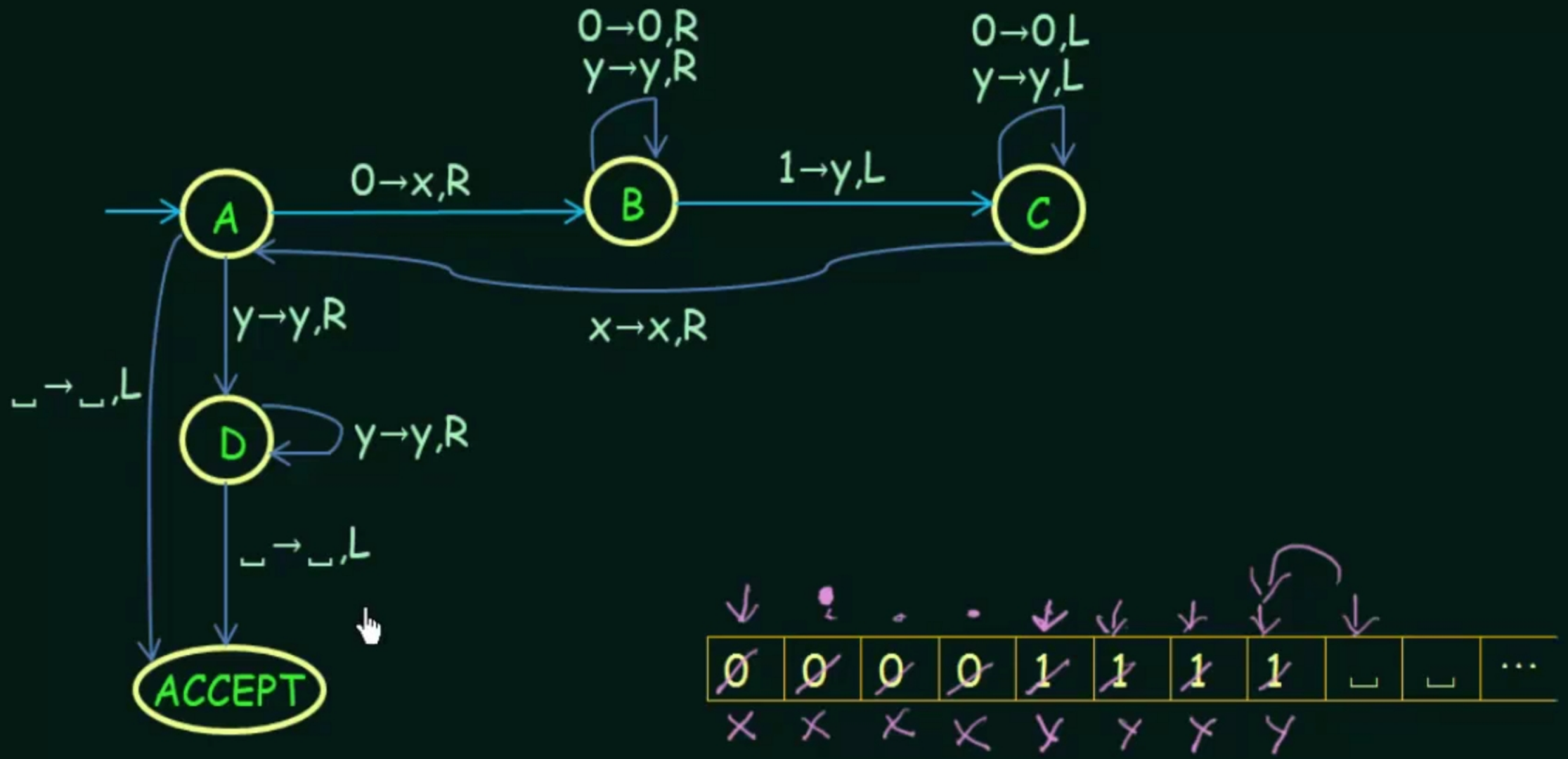


Turing Machine - Example (Part-2)

Design a Turing Machine which recognizes the language $L = 0^N 1^N$

Algorithm:

- Change "0" to "x"
- Move RIGHT to First "1"
  If None: REJECT
- Change "1" to "y"
- Move LEFT to Leftmost "0"
- Repeat the above steps until no more "0"s
- Make sure no more "1"s remain


0→0,R
y→y,R
0→0,L
y→y,L
0→x,R
1→y,L
A
B
C
y→y,R
x→x,R
␣→␣,L
D
y→y,R
␣→␣,L
ACCEPT
0 0 0 0 1 1 1 1 ␣ ␣ ...
x x x x y y x y

The CHURCH-TURING Thesis

What does COMPUTABLE mean?

Alonzo Church - LAMBDA CALCULUS

(June 14, 1903 - August 11, 1995)
American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science

Allen Turing - TURING MACHINE

(23 June 1912 - 7 June 1954)
English computer scientist, mathematician, logician, cryptanalyst, philosopher and theoretical biologist.

Several Variations of Turing Machine:

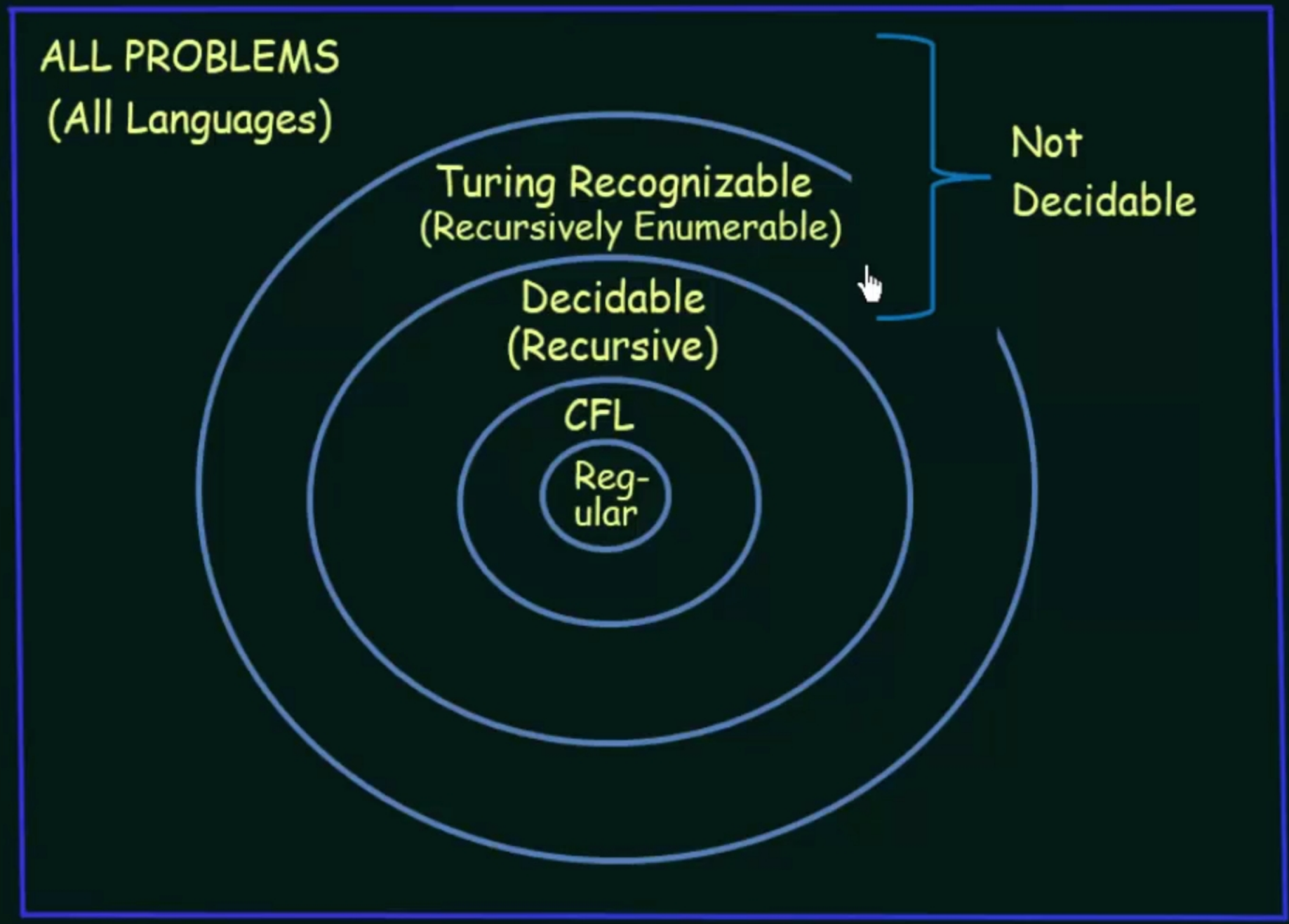
- One Tape or many
- Infinite on both ends
- Alphabets only {0, 1} or more?
- Can the Head also stay in the same place?
- Allow Non-Determinism

All variations are equivalent in computing capability

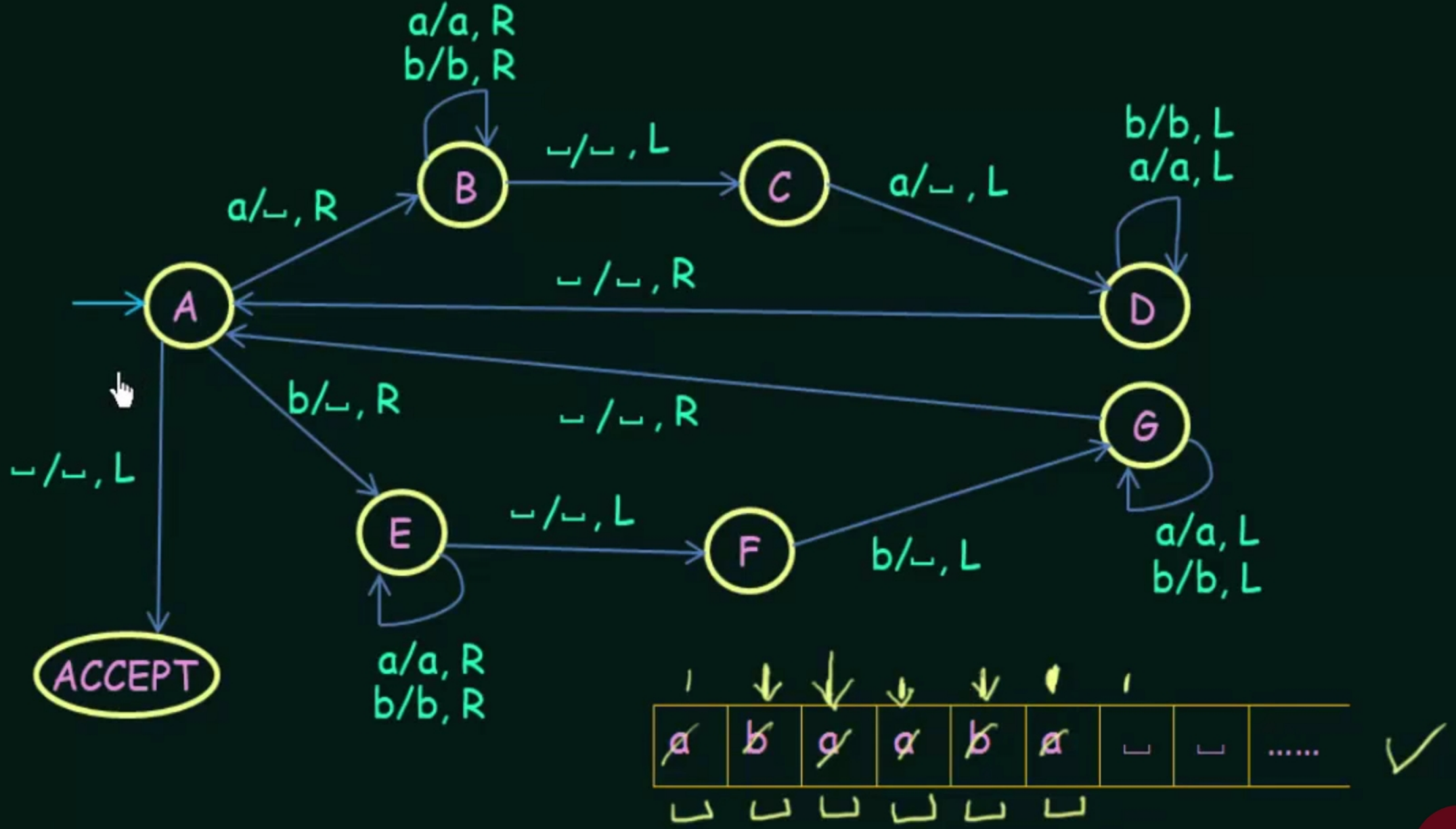
Turing Machine and Lambda Calculus are also equivalent in power

Algorithmically computable
Means
Computable by a TURING MACHINE

Turing Machine
≠
Turing Test



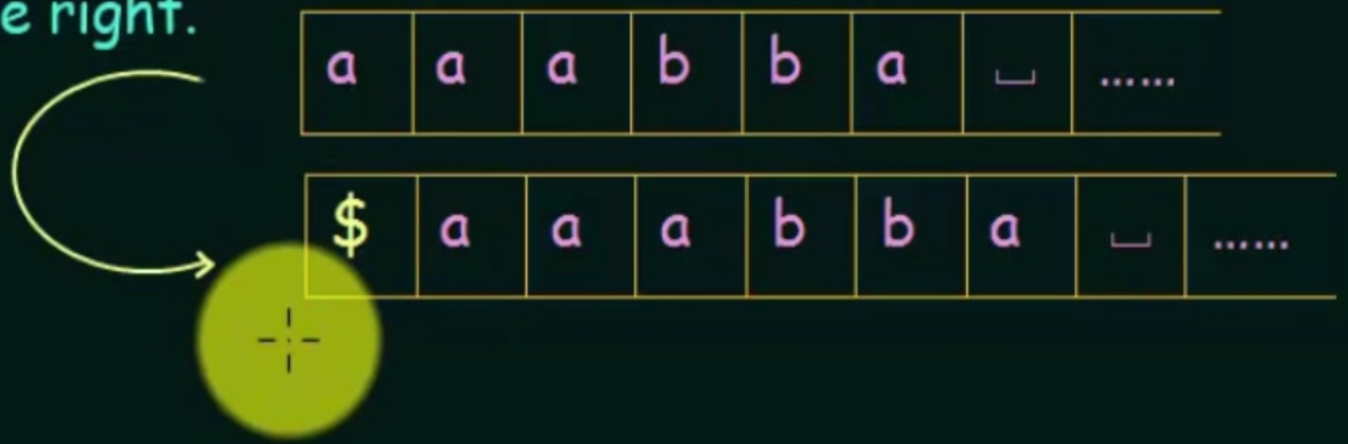
The different classes of Languages


a/a, R
b/b, R
␣/␣, L
b/b, L
a/a, L
a/␣, R
B
C
a/␣, L
␣/␣, R
A
D
b/␣, R
␣/␣, R
G
␣/␣, L
E
␣/␣, L
F
b/␣, L
a/a, L
b/b, L
ACCEPT
a/a, R
b/b, R
a
b
a
a
b
a
␣
␣
.....

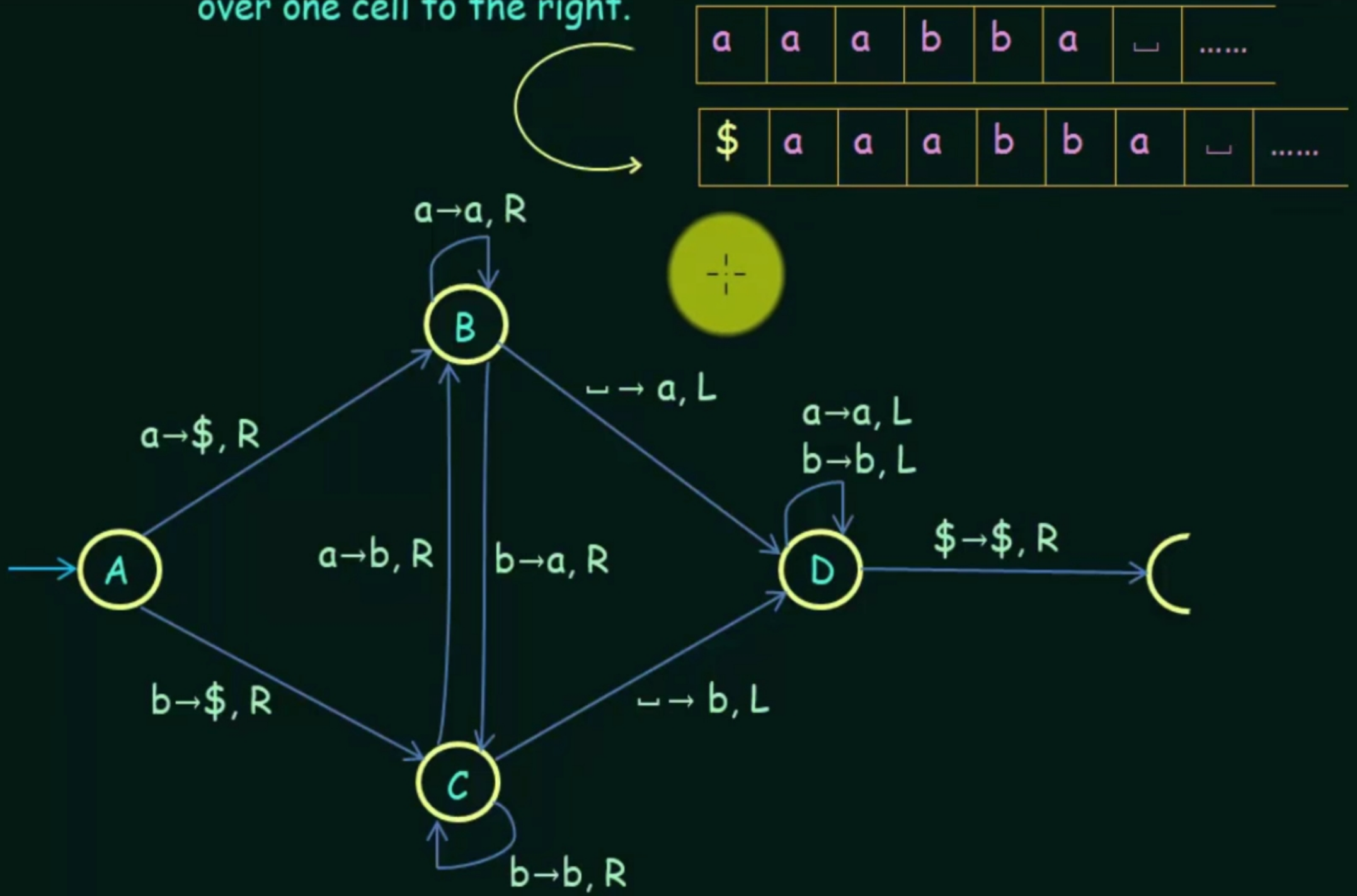
Turing Machine Programming Techniques (Part-1)

Problem: How can we recognize the left end of the Tape of a Turing Machine ?

Solution: Put a Special Symbol $ on the left end of the Tape and shift the input over one cell to the right.



over one cell to the right.



Turing Machine Programming Techniques (Part-2)

Example: Build a Turing Machine to recognize the language $0^N1^N0^N$

IDEA

We already have a Turing Machine to turn 0^N1^N to x^Ny^N
and to decide that language.

USE THIS TURING MACHINE AS A SUBROUTINE

Step 1: 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0

x x x x x y y y y y 0 0 0 0 0

Step 2: Build a similar Turing Machine to recognize y^N0^N

Step 3: Build the final Turing Machine by combining these two smaller Turing Machines together into one larger Turing Machine

Turing Machine Programming Techniques (Part-3)

COMPARING TWO STRINGS

A Turing Machine to decide $\{ w \# w \mid w \in \{a,b,c\}^* \}$

Solution:

- Use a new symbol such as 'x'
- Replace each symbol into an x after it has been examined

Solution:

- Use a new symbol such as 'x'
- Replace each symbol into an x after it has been examined

a b b a c # a b b a c

⇩

x b b a c # x b b a c

⇩

x x b a c # x x b a c

⇩

x x x a c # x x x a c

⇩

x x x x c # x x x x c

⇩

x x x x x # x x x x x

Problem:

Can we do it non-destructively? i.e. without loosing the original strings?

Solution:

Replace each unique symbol with another unique symbol instead of replacing all with the same symbol

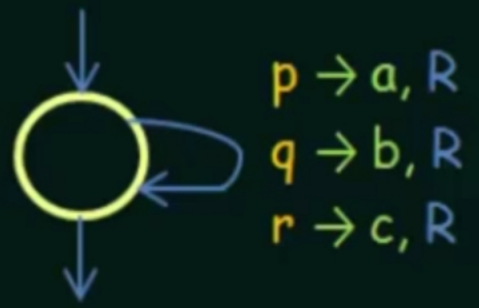
Eg. $a \rightarrow p$, $b \rightarrow q$, $c \rightarrow r$

a b b a c # a b b a c

⇩

p q q p r # p q q p r

Restore the original strings if required



Multitape Turing Machine

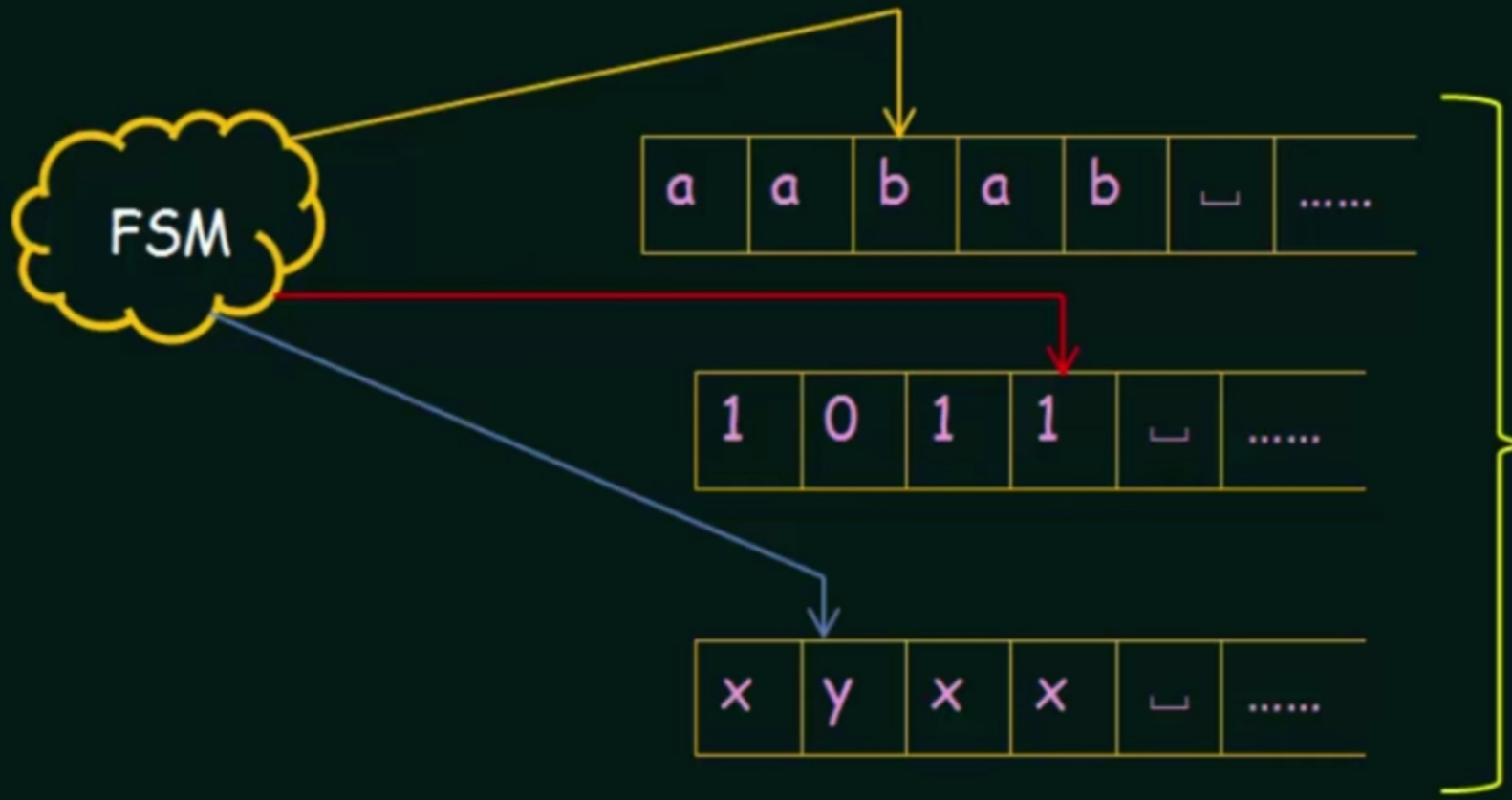
Theorem: Every Multitape Turing Machine has an equivalent Single Tape Turing Machine

Proof

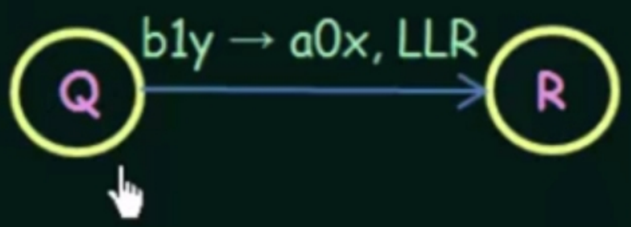
Given a Multitape Turing Machine show how to build a single tape Turing Machine

- Need to store all tapes on a single tape
  Show data representation
- Each tape has a tape head
  Show how to store that info
- Need to transform a move in the Multitape TM into one or moves in the Single Tape TM

Multitape Turing Machine



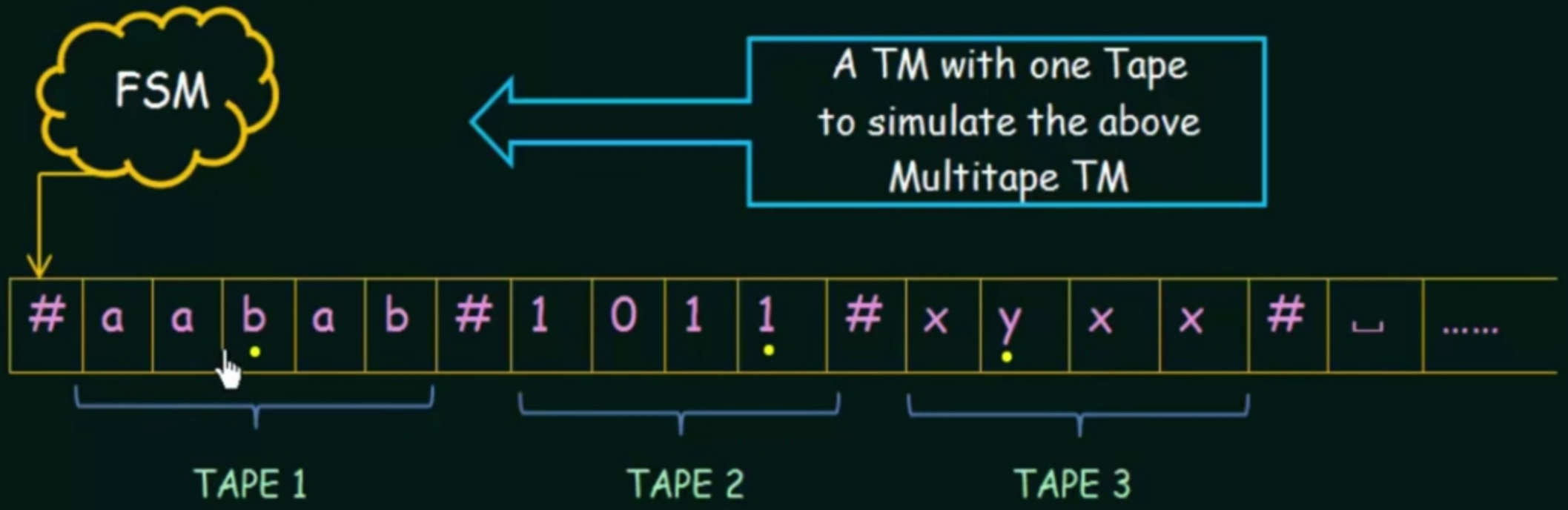
Several Tapes .
Each has its own
TAPE HEAD



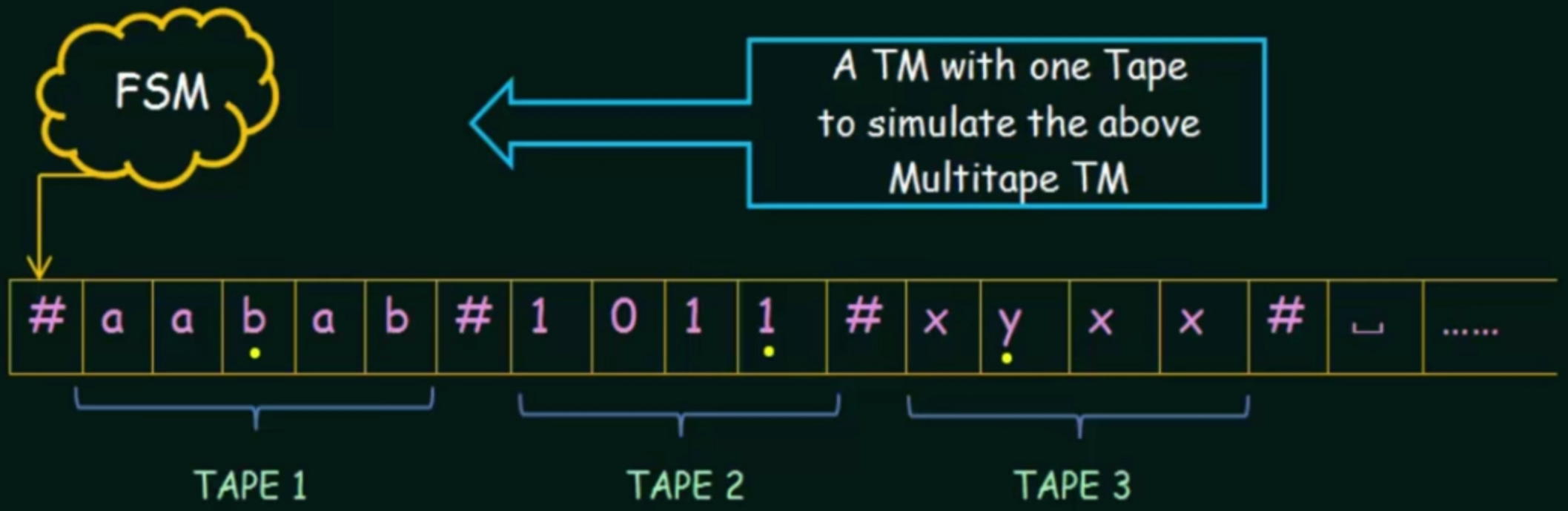
An Example Machine with
K=3 Tapes

An Example Machine with
K=3 Tapes

Single Tape Turing Machine



Single Tape Turing Machine



- Add "dots" to show where Head "K" is
- To simulate a transition from state Q, we must scan our Tape to see which symbols are under the K Tape Heads
- Once we determine this and are ready to MAKE the transition, we must scan across the tape again to update the cells and move the dots
- Whenever one head moves off the right end, we must shift our tape so we can insert a ␣

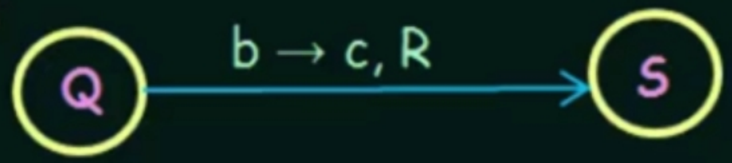
Nondeterminism in Turing Machine (Part-1)

Nondeterministic Turing Machines:

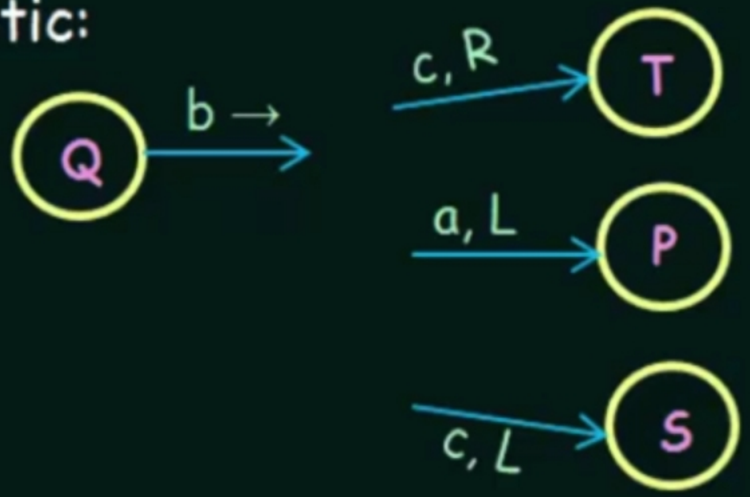
Transition Function:

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

Deterministic:

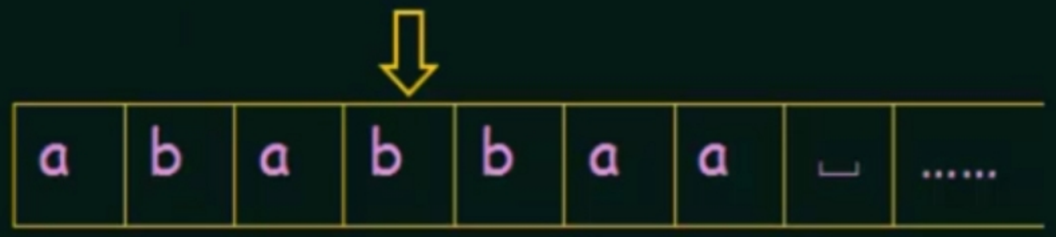


Nondeterministic:



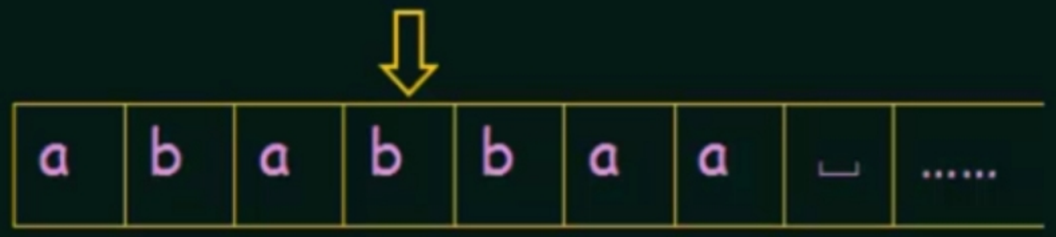
CONFIGURATION

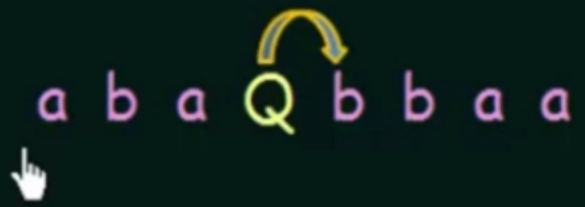
- A way to represent the entire state of a TM at a moment during computation
- A string which captures:
  - The current state
  - The current position of the Head
  - The entire Tape contents



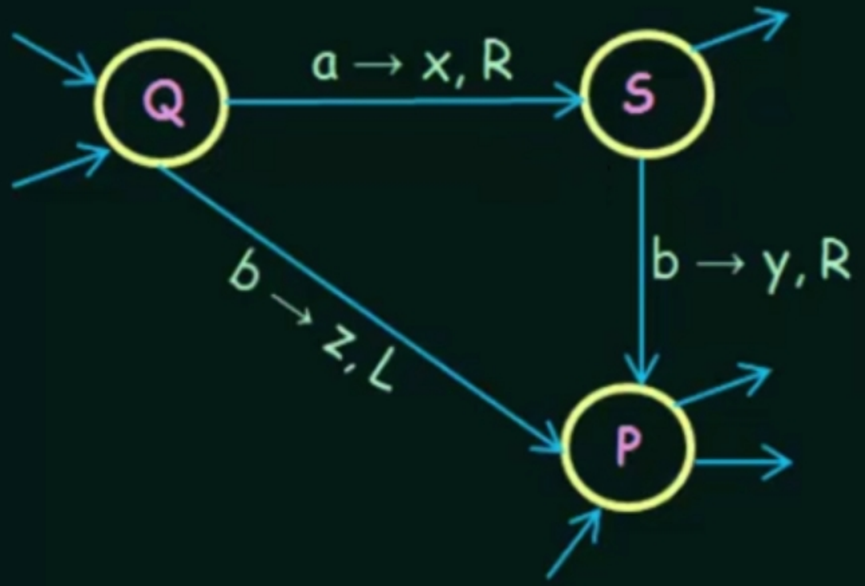
CONFIGURATION

- A way to represent the entire state of a TM at a moment during computation
- A string which captures:
  - The current state
  - The current position of the Head
  - The entire Tape contents

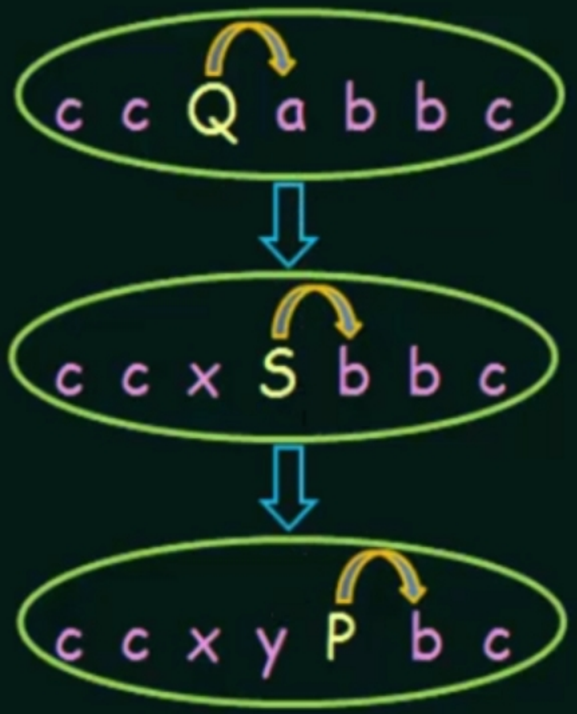




Deterministic TM:



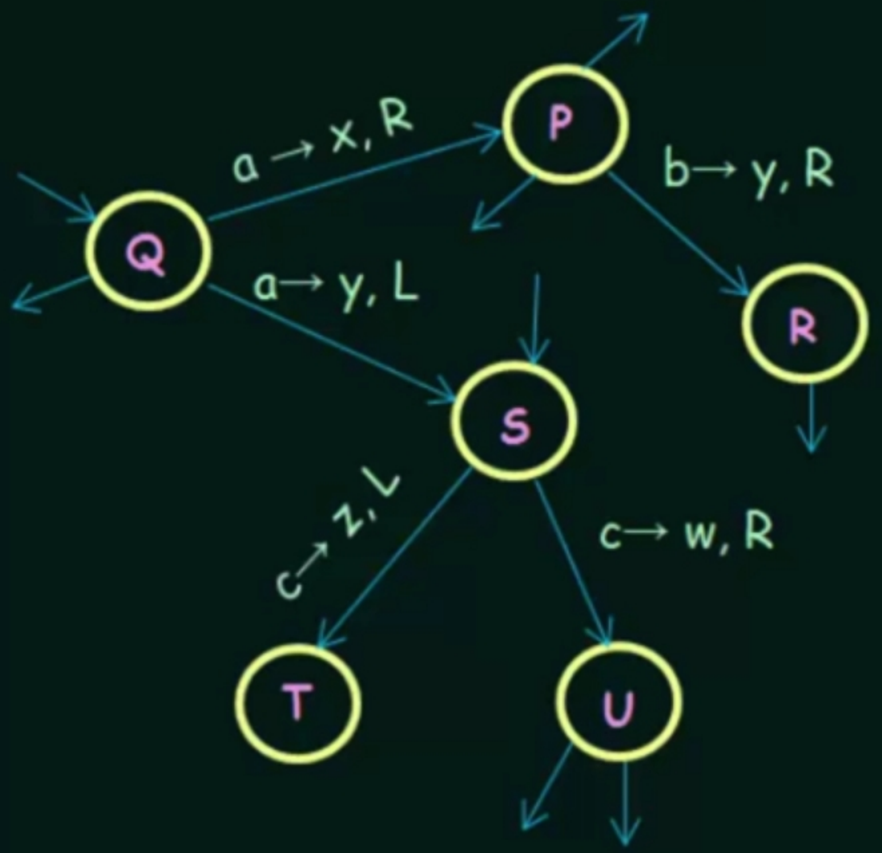
Computation History:



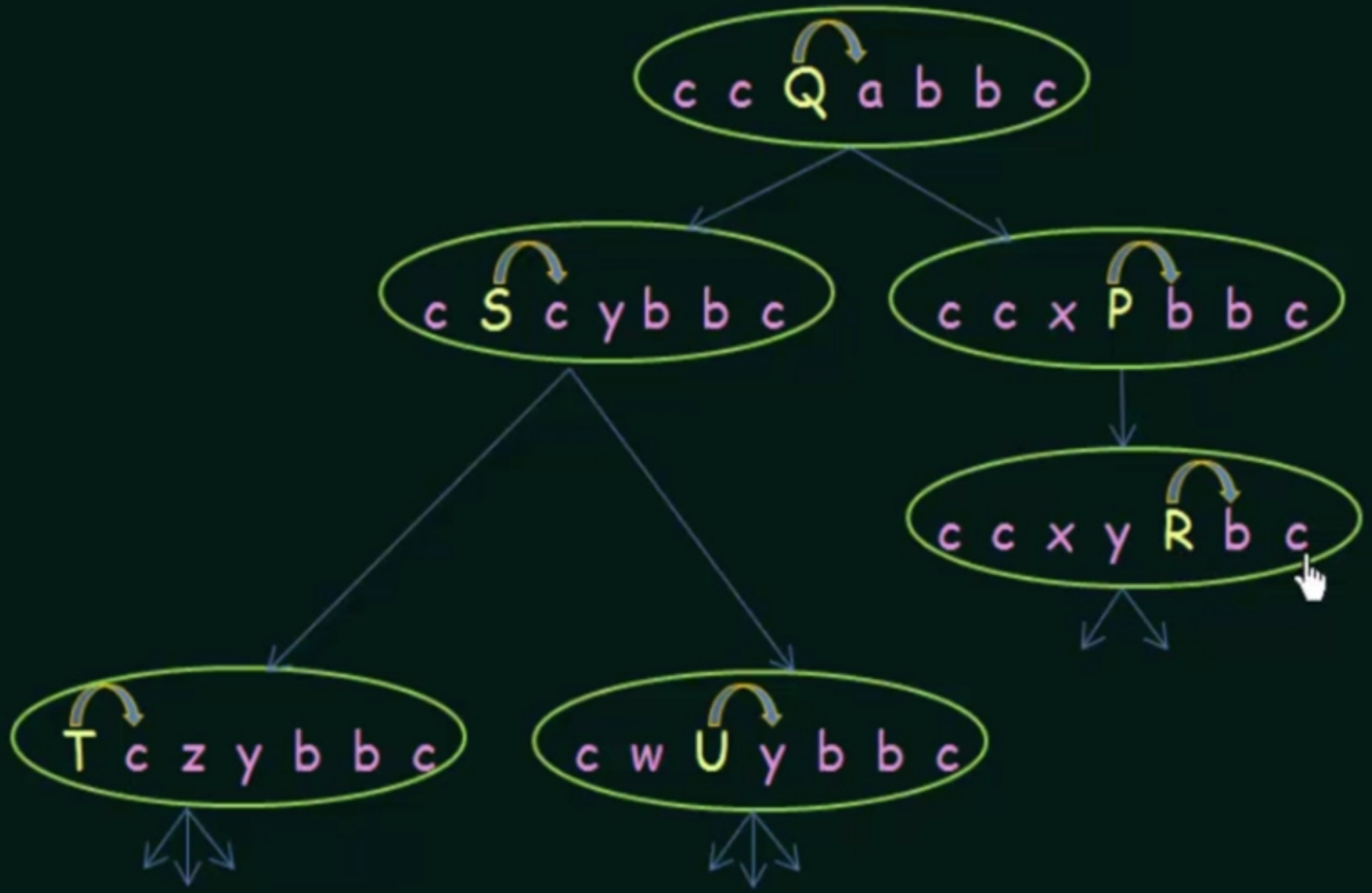
With Nondeterminism:

At each moment in the computation there can be more than one successor configuration

Nondeterministic TM:



Computation History:



Nondeterminism in Turing Machine (Part-2)

Theorem: Every NondeterministicTM has an equivalent Deterministic TM

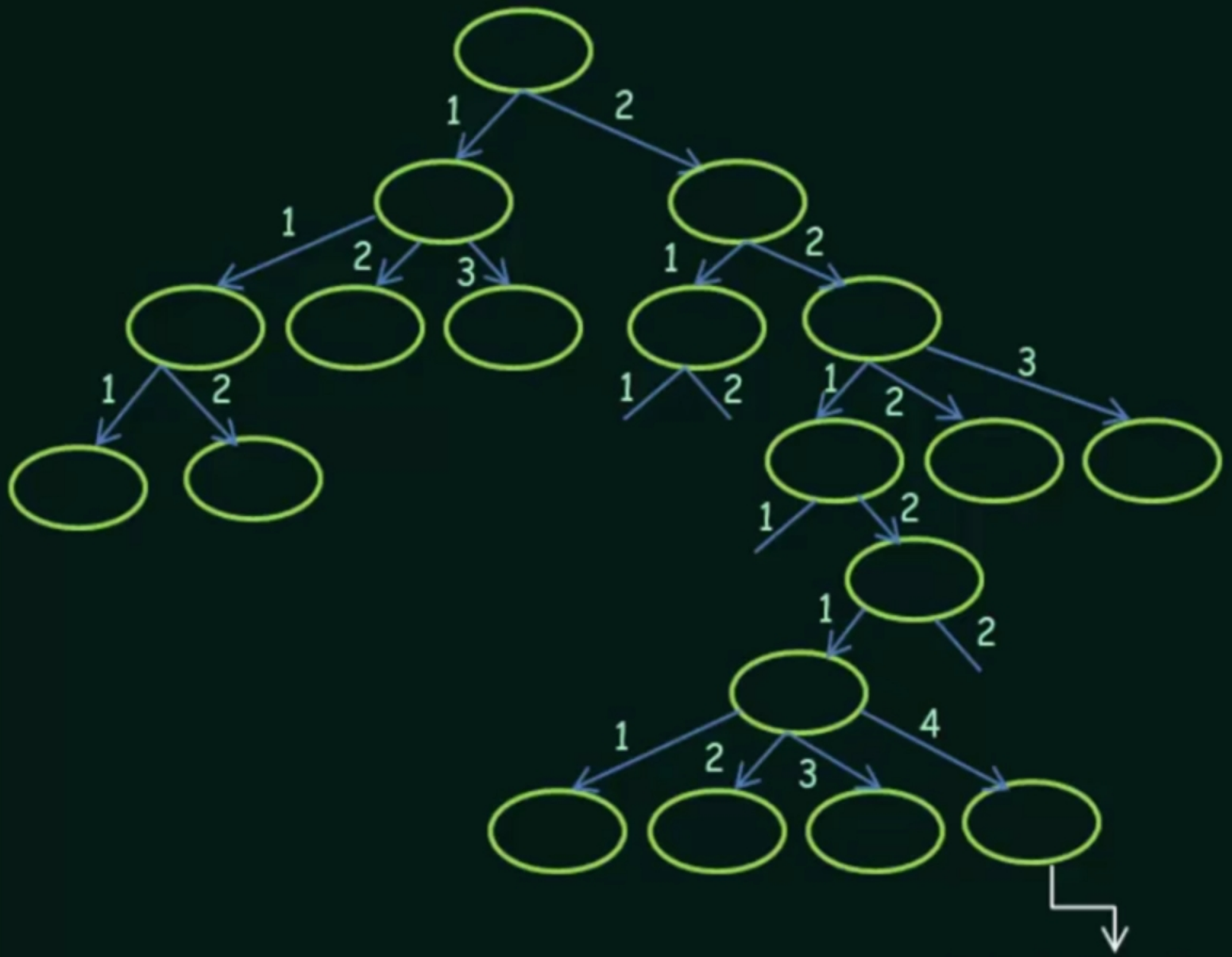
Proof:

- Given a Nondeterministic TM (N) show how to construct an equivalent Deterministic TM (D)
- If N accepts on any branch, the D will Accept
- If N halts on every branch without any ACCEPT, then D will Halt and Reject.

Approach:

- Simulate N
- Simulate all branches of computation
- Search for any way N can Accept

Computational History:



-A Path to any Node is given by a number

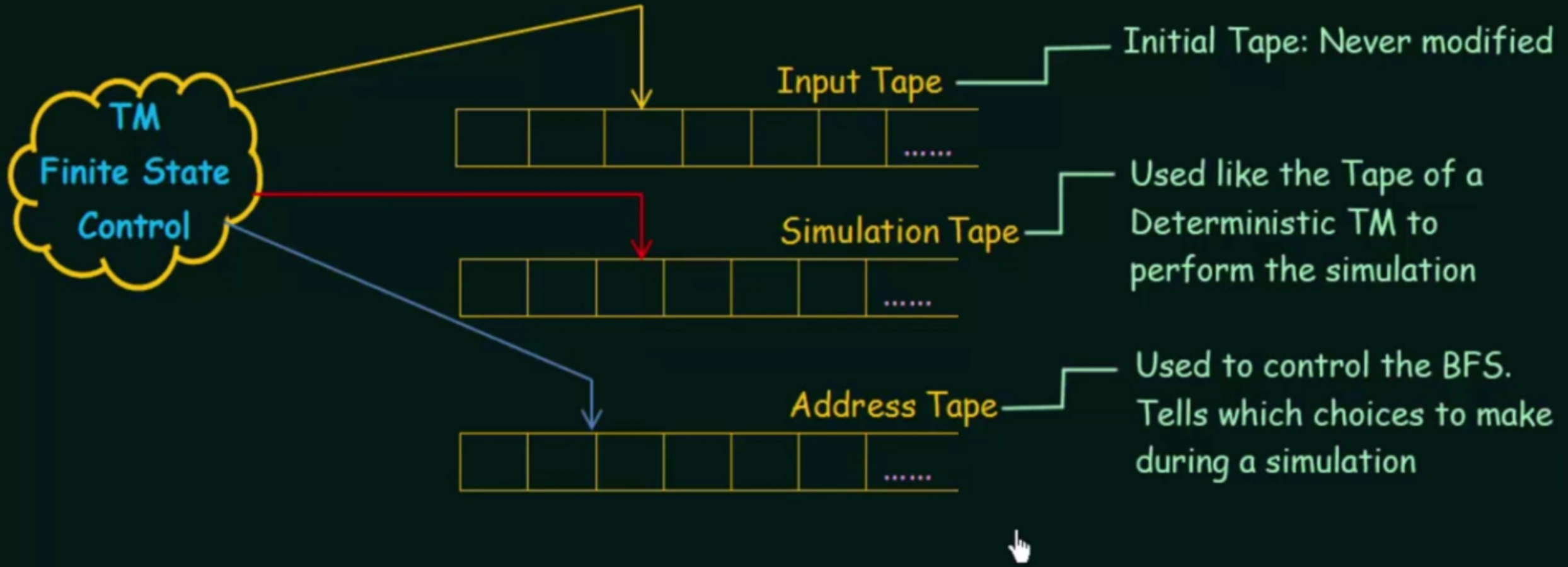
- Search the tree looking for ACCEPT

Search Order:

- DEPTH FIRST SEARCH x
- BREADTH FIRST SEARCH ✓

To examine a node:

- Perform the entire computation from scratch
- The path numbers tells which of the many nondeterministic choices to make


TM
Finite State
Control
Input Tape
.....
Simulation Tape
.....
Address Tape
.....
Initial Tape: Never modified
Used like the Tape of a
Deterministic TM to
perform the simulation
Used to control the BFS.
Tells which choices to make
during a simulation

Algorithm:

Initially: TAPE 1 contains the Input
TAPE 2 and TAPE 3 are empty

- Copy TAPE 1 to TAPE 2
- Run the Simulation
- Use TAPE 2 as "The Tape"
- When choices occur (i.e. when Nondeterministic branch points are encountered) consult TAPE 3
- TAPE 3 contains a Path. Each number tells which choice to make
- Run the Simulation all the way down the branch as far as the address/ path goes (or the computation dies)
- Try the next branch
- Increment the address on TAPE 3
- REPEAT

If ACCEPT is ever encountered,
Halt and Accept
If all branches Reject or die out,
then Halt and Reject

Turing Machine as Problem Solvers

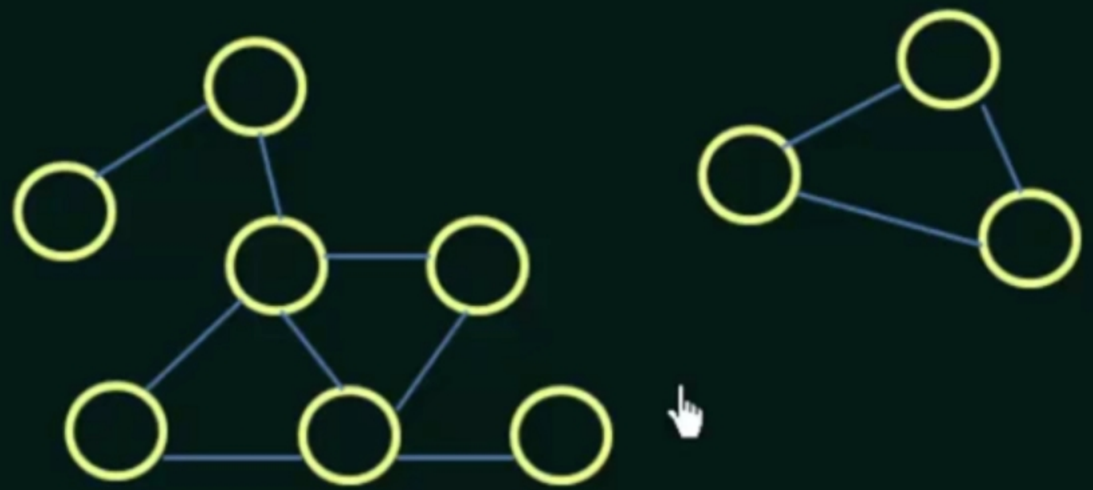
Any arbitrary Problem can be expressed as a language

-Any instance of the problem is encoded into a string

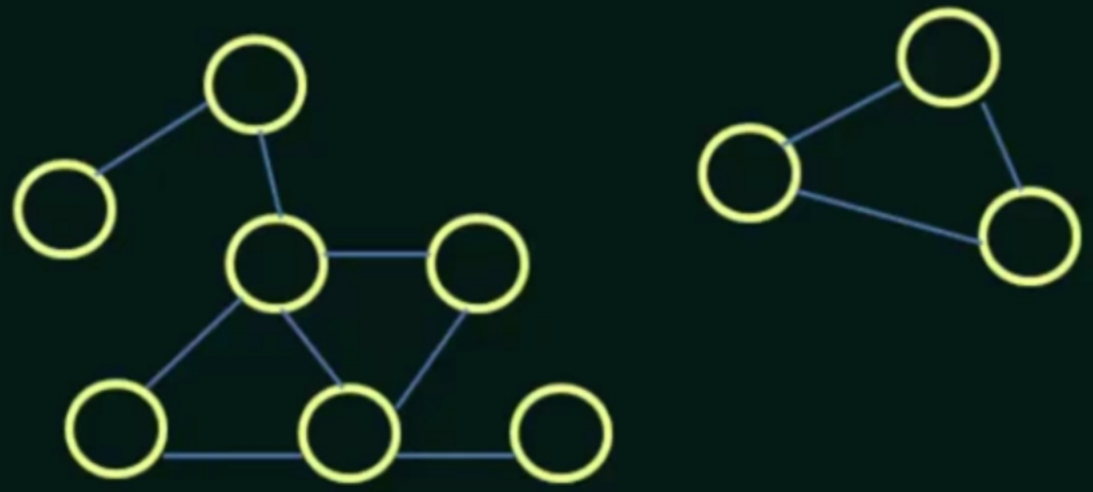
The string is in the language ⇨ The answer is YES

The string is not in the language ⇨ The answer is NO

Example: Is this undirected graph connected?



Example: Is this undirected graph connected?



We must encode the problem into a language.

$A = \{ \langle G \rangle \mid G \text{ is a connected graph} \}$

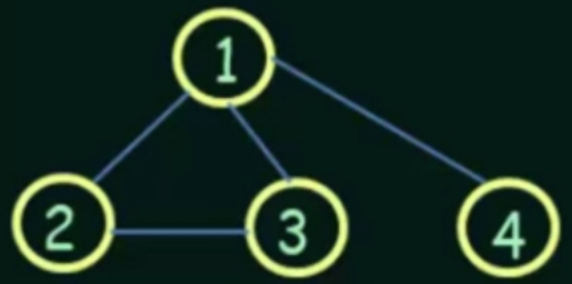
We would like to find a TM to decide this language:

ACCEPT = "YES", This is a connected graph

REJECT = "NO", This is not a connected graph / or this is not a valid Representation of a graph.

LOOP = This problem is decidable. Our TM will always halt

Representation of Graph:



$\langle G \rangle$ = (1,2,3,4) ((1,2), (2,3), (1,3), (1,4))

List of nodes — Edges

Σ = { (,), , , 1, 2, 3, 4, ……., 0 }

(	1	,	2	,	3	,	4	,	……	)	)	␣	……

High Level Algorithm:

```
Select a Node and Mark it
REPEAT
    For each node N
        If N is unmarked and there is an edge from N to an already marked
        node
    Then
        Mark Node N
    End

Until no more nodes can be marked
    For each Node N
        If N is unmarked
        Then REJECT
    End
ACCEPT
```

Implementation Level Algorithm:

- Check that input describes a valid graph
- Check Node List
    - Scan "(" followed by digits ...
    - Check that all nodes are different i.e. no repeats
    - Check edge lists ...
        etc.

- Mark First Node
    - Place a dot under the first node in the node list
    - Scan the node list to find a node that is not marked
        etc.

Decidability and Undecidability

Recursive Language:

- A language 'L' is said to be recursive if there exists a Turing machine which will accept all the strings in 'L' and reject all the strings not in 'L'.
- The Turing machine will halt every time and give an answer (accepted or rejected) for each and every string input.

Recursively Enumerable Language:

- A language 'L' is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in 'L'.
- But may or may not halt for all input strings which are not in 'L'.

Decidable Language:

A language 'L' is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

Partially Decidable Language:

A language 'L' is partially decidable if 'L' is a recursively enumerable language.

Undecidable Language:

- A language is undecidable if it is not decidable.
- An undecidable language may sometimes be partially decidable but not decidable.
- If a language is not even partially decidable, then there exists no Turing machine for that language

Recursive Language	TM will always Halt
Recursively Enumerable Language	TM will halt sometimes & may not halt sometimes
Decidable Language	Recursive Language
Partially Decidable Language	Recursively Enumerable Language
UNDECIDABLE	No TM for that language

The Universal Turing Machine

The Language

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

is Turing Recognizable

Given the description of a TM and some input, can we determine whether the machine accepts it?

- Just Simulate/ Run the TM on the input

M Accepts w: Our Algorithm will Halt & Accept
M Rejects w: Our Algorithm will Halt & Reject.
M Loops on w: Our Algorithm will not Halt.

The Universal Turing Machine

Input: M = the description of some TM
w = an input string for M

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

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

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

The Halting Problem

Given a Program, WILL IT HALT ?

Given a Turing Machine, will it halt when run on some particular given input string?

Given some program written in some language (Java/C/ etc.) will it ever get into an infinite loop or will it always terminate?

Answer:

- In General we can't always know.
- The best we can do is run the program and see whether it halts.
- For many programs we can see that it will always halt or sometimes loop

BUT FOR PROGRAMS IN GENERAL THE QUESTION IS UNDECIDABLE.

Undecidability of the Halting Problem

Given a Program, WILL IT HALT ?

Can we design a machine which if given a program can find out or decide if that program will always halt or not halt on a particular input?

Let us assume that we can:

H (P, I)

- Halt
- Not Halt

This allows us to write another Program:

```
C ( X )
  if { H (X, X) == Halt }
       Loop Forever;
  else
       Return;
```

Let us assume that we can:

H (P, I)

- Halt
- Not Halt

This allows us to write another Program:

```
C ( X )
  if { H (X, X) == Halt }
       Loop Forever;
  else
       Return;
```

If we run 'C' on itself:

C (C)

H (C,C) == Halt → Not Halt

H (C,C) == Not Halt → Halt