Theory of Computation

SET I

HandWritten Notes

by

prof. Ajay Pashankar

Theory of Computation

ALPHABETS :-

An alphabet is a finite, non-empty set of symbols.

We use the symbol Σ for an alphabet

e.g. $\Sigma = \{0, 1\}$ is the binary alphabet.

STRING

A string (or sometimes word) is a finite sequence of symbols choosen from some alphabets.

e.g:- 10010 is a string from the binary alphabet $\Sigma = \{0, 1\}$

EMPTY STRING

Empty string is the string with zero occurences of the symbols

Denoted by ϵ.

LENGTH OF STRING.

The no. of symbols in the string

Notation for length of string w is $|w|$ ← cardinality.

$|011| = 3$

$|\epsilon| = 0$

The set of all strings over an alphabet Σ is denoted by Σ^*

$\{0,1\}^* = \{\epsilon, 0, 1, 01, 10, 11, 000, \ldots\}$

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

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots$

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$ ← empty string

$\Sigma^+ = \Sigma^* - \{\epsilon\}$

* zero or more occurence

Σ finite set

Σ^*, Σ^+ — infinite

\+ one or more.

e.g:- $w = abaa$ $v = baa$ a, b, c 0, 1, 2. alphabet

L_1 L_2 u v w → strings

The concatenation of two strings w and v is the string obtained by appending the symbol of v to the right end of w

$wv = abaabaa$

Reverse of String

if $w = a_1 a_2 \ldots a_n$

denoted by w^R

$$w^R = a_n \ldots a_2 a_1$$

Language.

Language is denoted by L

complement of Language L is

$$\bar{L} = \Sigma^* - L$$

Reverse of a language is the set of all string reversals.

$$L^R = \{ w^R : w \in L \}$$

concatenation of two languages.

L_1 and L_2

$$L_1, L_2 = \{ xy : x \in L_1, y \in L_2 \}$$

(belong; xy: word or string)

Star closure

$$L^* = L^0 \cup L^1 \cup L^2 \ldots \quad L^0 = \{\lambda\}$$

(λ: empty string)

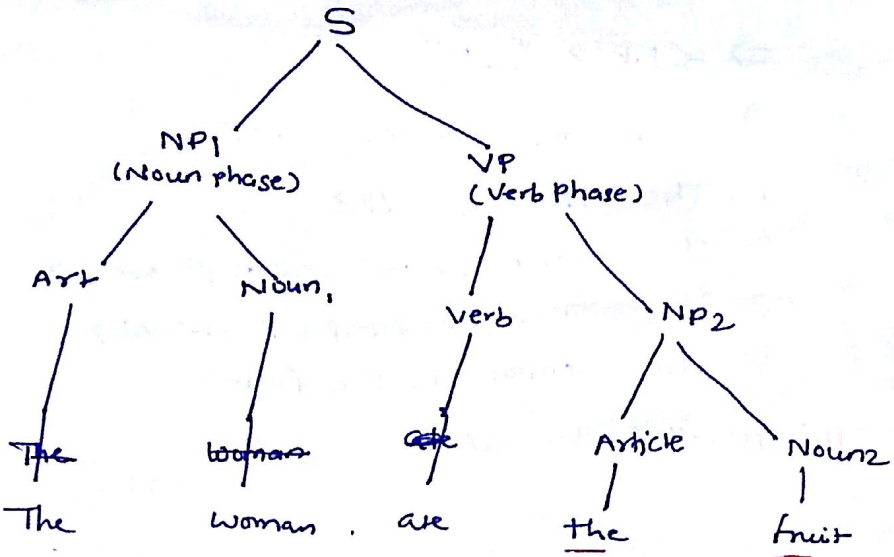
* – zero or more

Positive closure

$$L^+ = L^1 \cup L^2 \ldots$$

\+ – one or more.

Parse Tree or Syntax tree



We do Parsing to check whether the sentence is according to the rules of english grammer.

Set of rules

Production rules Or Rules.

<S> → <NP1><VP>
<NP1> → <Article> <noun1>
<VP> → <Verb> <NP2>
<NP2> → <Article> <Noun2>
<Art> → The
<Noun1> → Woman
<verb> → ate
<noun2> → fruit

N → set of non-terminals
T → set of terminals
S → start symbol
V → Total Alphabet

S is special Non-Terminal

$$V = N \cup T$$

Non-Terminal nodes are those Nodes from which you can derive nodes.

S is special Non-Terminal Node.

Finite State Machine (Finite Automata)

<S> ⇒ <NP1> <VP>

⇒ <Article> <Noun1> <VP>

⇒ The <Noun1> <VP>

⇒ The woman <Verb> <noun phrase 2>

⇒ The woman ate <article> <Noun2>

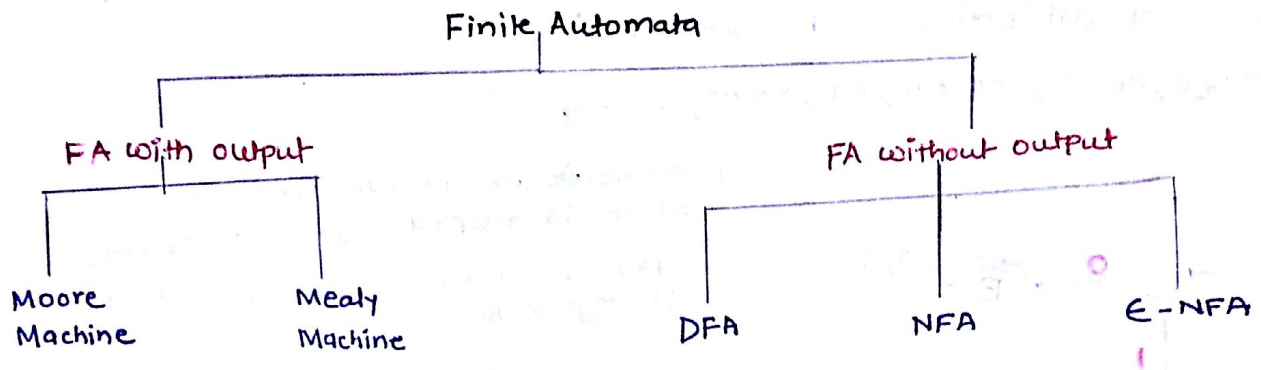
⇒ The woman ate the fruit.

This is called derivation

→ rewritten as

⇒ directly derives

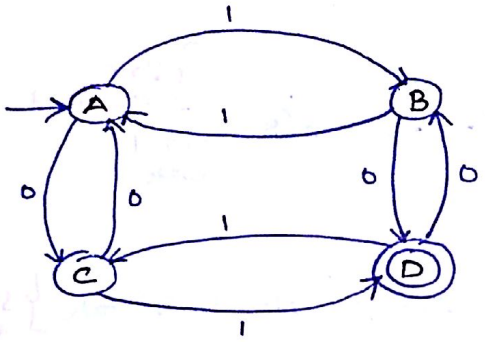
Finite State Machine (Finite Automata)



DFA - Deterministic Finite Automata

FSM - It is the simplest model of computation
- It has a very limited memory

DFA:-



Circle is known as states :- A, B, C, D
edges is transition
labeling of edges is inputs.

A is the initial or starting states of DFA - arrow
D is the final or terminating states of DFA - Double Circle.

Def^{n}:-

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

Q = set of all states
Σ = inputs
q_0 = start state / initial state
F = set of Final states
δ = transition function from $Q \times \Sigma \to Q$

$Q = \{A, B, C, D\}$
$\Sigma = \{0, 1\}$
$q_0 = \{A$
$F = \{D\}$
$\delta =$

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

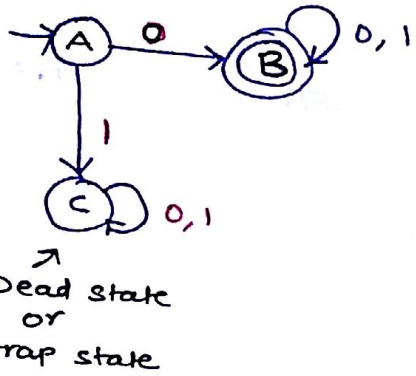
Deterministic Finite Automata (Example-1)

L_1 = set of all strings that start with '01'

= {0, 00, 01, 000, 010, 011, 0000, ...}

[whenever string reaches final state string is accepted, else it is rejected]

(after reaching final state, if we get input it stays in final state only)



E.g. 001 ✓

initial state A → B → B → B — Final state } string accepted

E.g. 101 ✗

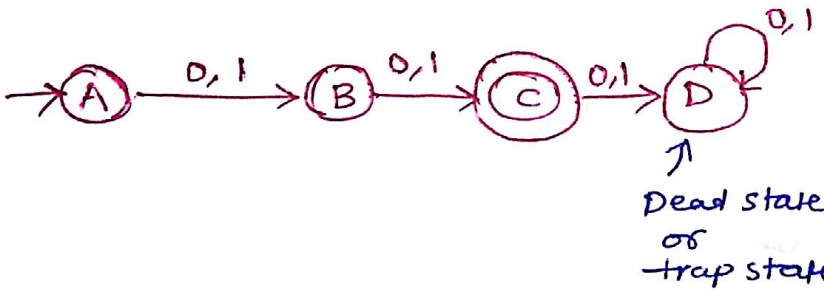
initial state A → C → C → C — Not final state } string rejected

Deterministic Finite Automata (Example-2)

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

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

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



E.g.: 00 ✓

A → B → C — Final state

E.g.: 10 ✓

A → B → C — Final state

E.g 001 ✗

A → B → C → D → Not final state

E.g. 1 ✗

A → B — Not final state

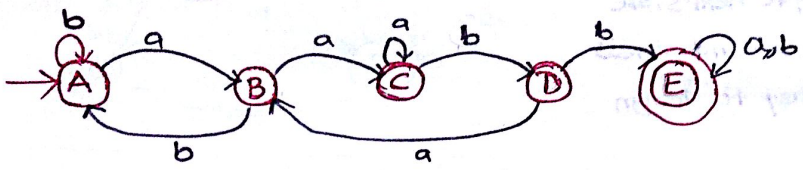
Deterministic Finite Automata (Example-3)

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.

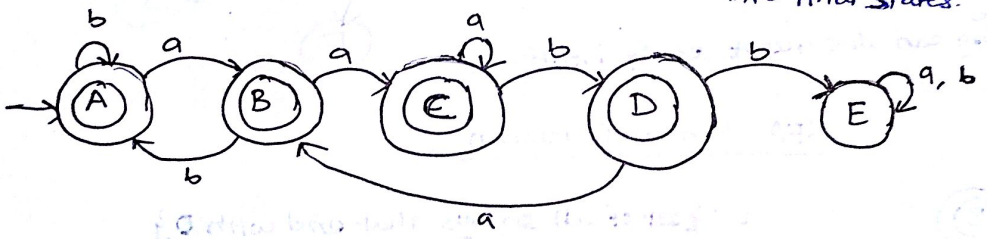
Let us construct a DFA that accepts all strings over {a,b} that contains the string aabb in it.



aaabb

aaba...
↑

- Flip the states
- make the final state into non-final state and
- make the non-final states into final states.



Since E is not a final state string is not accepted.

NFA - Non-Deterministic Finite Automata

Deterministic Finite Automata

↓

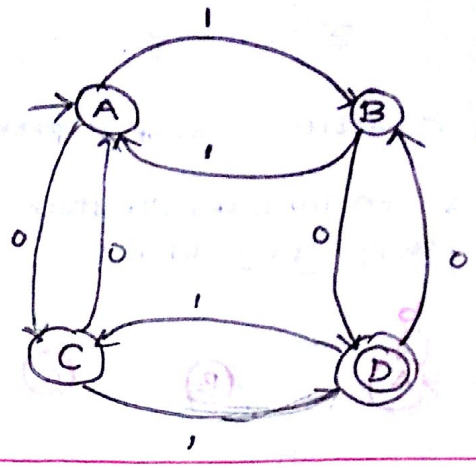
DETERMINISM

» In DFA, given the current state we know ~~that~~ 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

↓

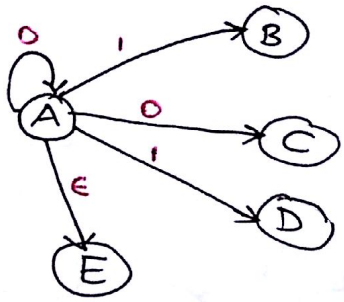
NON-DETERMINISM

» In NFA, given the current state there could multiple next states.

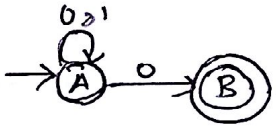
» The next state may be chosen at random

» All the next states may be chosen in parallel.

ϵ → This ~~string~~ state can also accept empty input



NFA - Formal Definition



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

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

Q = set of 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

A × 0 → B

A × 1 → A

B × 0 → ϕ

B × 1 → ϕ

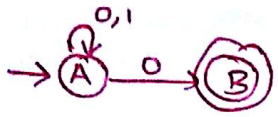
A $\xrightarrow{1}$ A, B, AB, ϕ . $2^{2 \rightarrow 4}$

3 states - A, B, C

A $\xrightarrow{1}$ A, B, C, AB, AC, BC, ABC, ϕ

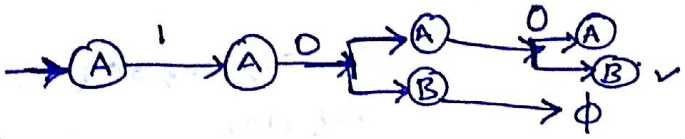
$2^3 \rightarrow 8$

NFA - Example-1



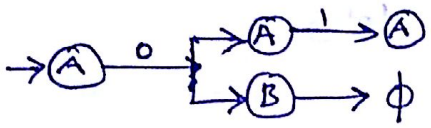
L = { set of all strings that end with 0 }

e.g. 100



✓

e.g. 01



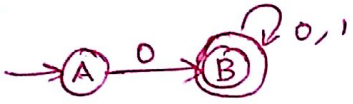
✗

✓ [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.]

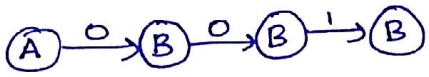
NFA - EXAMPLES. - 2

L = { set of all strings that starts with 0 }

= { 0, 00, 01, 000, - - - }



e.g. 001 ✓



e.g. 101 ✗

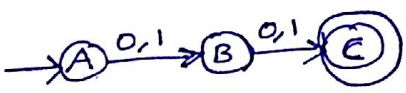
Dead configuration

---

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

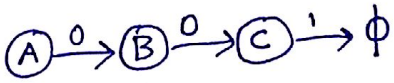
Σ = {0,1}

L = {00, 01, 10, 11}



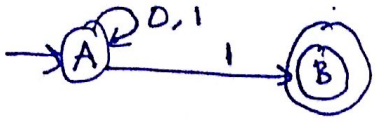
e.g. 00 ✓

e.g. 001 ✗



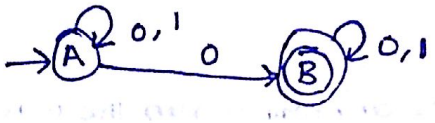
NFA - EXAMPLES 3

Ex 1) L_1 = {set of all strings that ends with '1'}

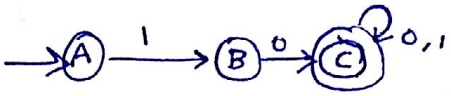


(0,1, 001, 0001, 0#1, 1, 101, 1101,)

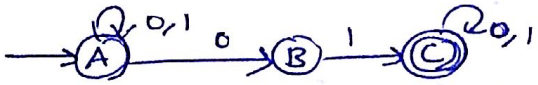
Ex 2) L_2 = {set of all strings that contain '0'}.



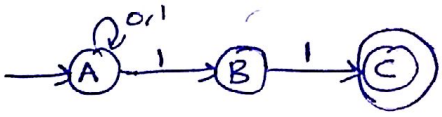
Ex 3) L_3 = {set of all strings that starts with '10'}



Ex 4) L_4 = {set of all strings that contain '01'}



Ex 5) L_5 = {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.

1) 2
2) 2
3) 4
4) 3
5) 3

Conversion of NFA to DFA

Every DFA is an NFA, but not vice versa.

But there is an equivalent DFA for every NFA.

DFA

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

NFA

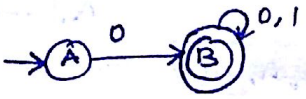
$\delta = Q \times \Sigma \to 2^Q$

NFA $\cong$ DFA

L = { set of all strings over {0,1} that starts with '0' }

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

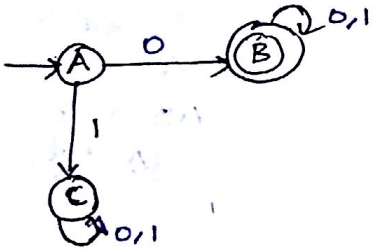
NFA



	0	1
A	B	ϕ
B	B	B

dead configuration

DFA



	0	1
A	B	C
B	B	B
C	C	C

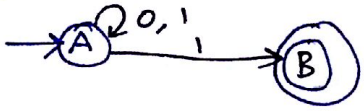
C - Dead state / trap state

Conversion of NFA to DFA Example 1

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

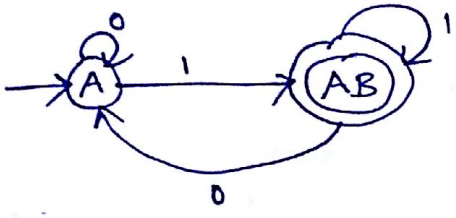
$\Sigma = 0, 1$

NFA



	0	1
A	{A}	{A, B}
B	ϕ	ϕ

DFA



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

AB - single state

for calculating AB just Union set of A and B

A ∪ B

AB on 0

$\{A \cup \phi) = \{A\}$

✓ (Subset construction Method)

Finite Automata With outputs

Mealy Machine

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

where

Q = finite set of sets

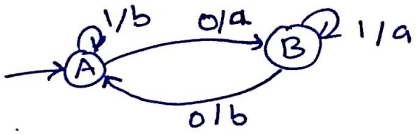
Σ = finite non-empty set of Input Alphabets

Δ = The set of output Alphabets

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

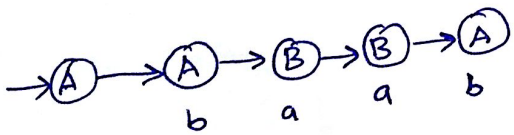
λ = output function: $\Sigma \times Q \to \Delta$

q_0 = Initial state / start state



output is dependent on input and state

E.g. 1010



length of output string
$n \to n$

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

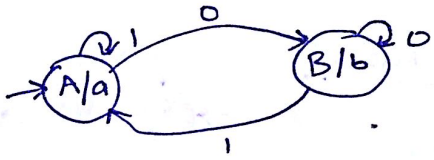
δ = Transition function:

$Q \times \Sigma \to Q$

λ = output function:

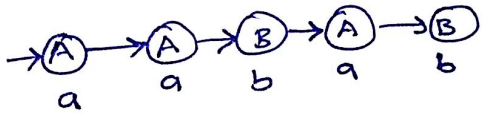
$Q \to \Delta$

q_0 = Initial state / start state



output dependent on State only.

Eg. 1010



length of output string.
$n \to n+1$

Construction of Mealy Machine

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

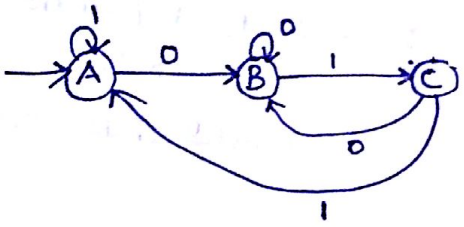
1's complement

10100
01011

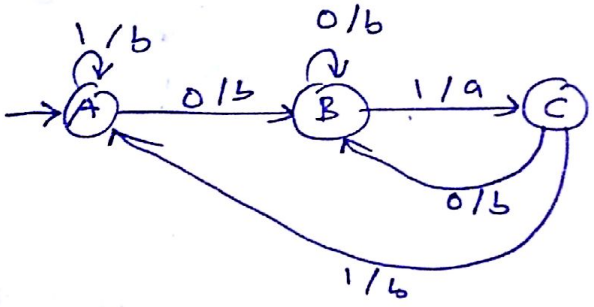
Ex-2:) Construct a Mealy Machine that prints 'a' whenever the sequence '01' is encountered in any input binary string

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

DFA



Mealy Machine



e.g.

0110 ✓
babb

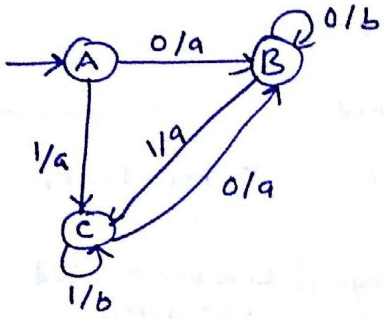
1000 ×
bbbb

Conversion of Mealy Machine to Moore Machine

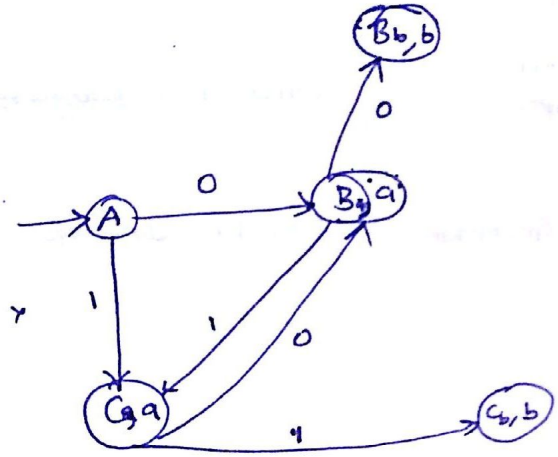
Convert the following Mealy Machine to its equivalent Moore Machine

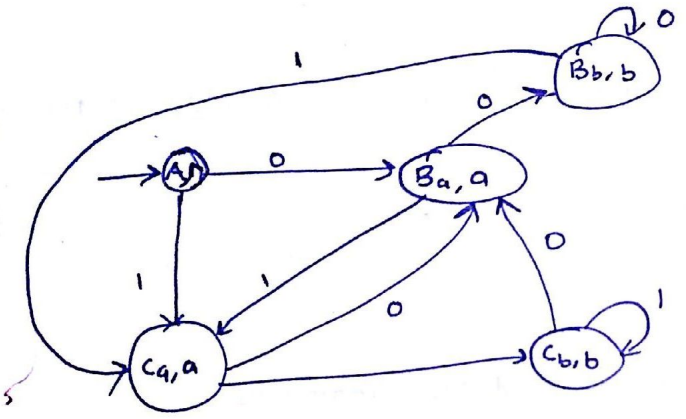
$\Sigma = \{0, 1\}$

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



Moore Machine





Moore → Mealy ⇒ No of states were same

Mealy → Moore ⇒ No. of states increased

x and y ⇒ $(x \times y)$ no. of states at maximum.

↓ states ↓ outputs

CHAPTER 2:- Formal languages

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 $\alpha \rightarrow \beta$ where α and β are strings on $V \cup T$ 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	Left Linear Grammar
A grammar is said to be Right Linear if all productions are of the form $A \to xB$ $A \to x$ where $A, B \in V$ and $x \in T$	A grammar is said to be Left Linear if all productions are of the form $A \to Bx$ $A \to x$ where $A, B \in V$ and $x \in T$.
Eg:- $S \to abS \mid b$ → Right linear $S \to Sbb \mid b$ → left linear	$S \to Sbb \mid b$ → left linear

---

Type 3 : Regular Grammar

Right linear Left linear

Context free Grammar / 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 $A \to \alpha$
> 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 will be defined as.

$$G = \{(S, A), (a, b), (S \to aAb, A \to aAb|\epsilon)\}$$

Soln :-

$S \to aAb$

$\to aaAbb$ (by $A \to aAb$)

$\to aaaAbbb$ (by $A \to aAb$)

$\to aaa\epsilon bbb$ (by $A \to \epsilon$)

$\to aaabbb$

$\to a^3 b^3 \Rightarrow a^n b^n$

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

$S \to aAb$ [by $S \to aAb$]

$\to aaAbb$ [by $aA \to aaAb$]

$\to aaaAbbb$ [by $aA \to aaAb$]

$\to aaa\epsilon bbb$ [by $A \to \epsilon$]

$\to aaabbb$

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

$S \to AB$

$\to ab$ [by $A \to a, B \to b$]

$L(G2) = \{ab\}$ — The only string generated by Grammar

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

$S \to AB$

$\to ab$ [by $A \to a, B \to b$]

$S \to AB$

$\to aBb$

$\to abb$

$S \to AB$

$\to aAbB$

$\to aabb$

$S \to AB$

$\to aAb$

$\to aab$

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

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

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

Rules for Regular Expression

1) Any terminal symbol i.e. Symbols $\in \Sigma$ $\quad a, b, c, \ldots \wedge, \phi$ including $\wedge$ and ϕ are regular expressions.

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

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 $\quad (+ \longrightarrow \text{OR})$

$R = 0 + 1 + 2$

2) $\{\wedge, ab\}$

$R = \wedge \; ab$

3) $\{abb, a, b, bba\}$

abb or a or b or bba

$R = abb + a + b + bba$

4) $\{\wedge, 0, 00, 000, \ldots\}$

Closure of 0

$R = 0^*$

5) $\{1, 11, 111, 1111, \ldots\}$

$R = 1^+$ $\quad$ Closure of 1.

. it looks like a closure, but $\wedge$ is absent.

Identities of Regular Expression

1) $\phi + R = R$

2) $\phi R + R\phi = \phi$

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

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

5) $R + R = R$

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

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

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

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

Ardens 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. $R = QP^*$

$R = Q + RP \longrightarrow (1)$ $\qquad R = QP^*$

$= Q + QP^*P$

$= Q(\epsilon + P^*P)$ $\qquad [\epsilon + R^*R = R^*]$

$= QP^*$

proved

$R = Q + RP$

$= Q + [Q + RP]P$

$= Q + QP + RP^2$

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

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

$\vdots$

$= Q + QP + QP^2 + \cdots QP^n + RP^{n+1}$ $\qquad [R = QP^*]$

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

$R = \epsilon P^*$

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

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

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

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

$= (\epsilon \cdot 1 + 00^*1)(0+10^*1)^*$ --> $\epsilon \cdot R = R$

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

$= 0^*1(0+10^*1)^*$ --> $\epsilon + R^*R = R^*$

$= R.H.S.$

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

$R_1 = aa + ab + ba + bb$

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

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

2) $L_2 = \{aa, ab, ba, bb, aaa, \ldots\}$

$R = (a+b)(a+b)(a+b)^*$ .... two or more than two

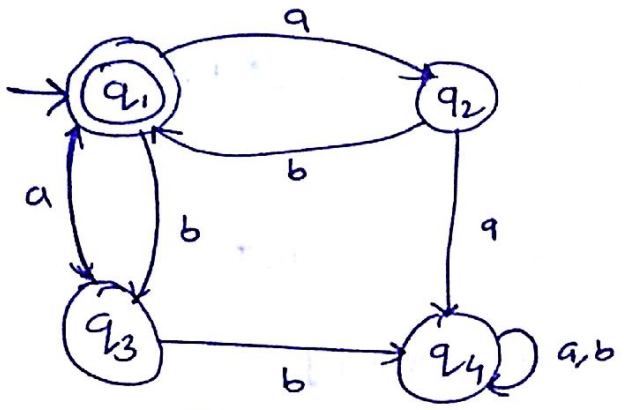
3) $L_1 = \{\epsilon, a, b, aa, ab, ba, bb\}$

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

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

Designing Regular Expression - Examples (3)

Find the Regular Expression for the following DFA



$q_1 = \epsilon + q_2b + q_3a \longrightarrow ①$

$q_2 = q_1a \longrightarrow ②$

$q_3 = q_1b \longrightarrow ③$

$q_4 = q_2a + q_3b + q_4a + q_4b \longrightarrow ④$

① $q_1 = \epsilon + q_2b + q_3a$

Putting values of q_2 and q_3 from ② and ③

$q_1 = \epsilon + q_1ab + q_2ba$

$q_1 = \epsilon + q_1(ab + ba)$ ... $R = Q + RP$

$R \to q_1 = \epsilon + q_1(ab+ba)$ ... $R = QP^*$ Ardens theorem

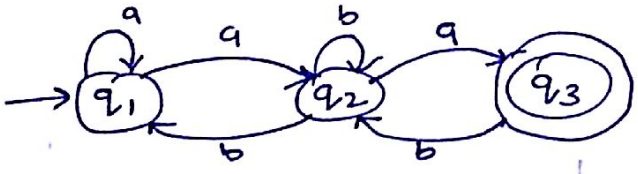
$q_1 = \epsilon(ab+ba)^*$ ... $\epsilon R = R$

$q_1 = (ab+ba)^*$

↳ Regular Expression ..

Designing Regular Expression - Examples - (2)

Find the Regular Expression for the following NFA



$q_3 = q_2a \longrightarrow ①$

$q_2 = q_1a + q_2b + q_3b \longrightarrow ②$

$q_1 = \epsilon + q_1a + q_2b \longrightarrow ③$

① → $q_3 = q_2a$

$= (q_1a + q_2b + q_3b)a$ ---- by eqn ②

$= q_1aa + q_2ba + q_3ba$ - → ④

② $q_2 = q_1a + q_2b + q_3b$ putting value of q_3 from ①

$= q_1a + q_2b + (q_2a)b$

$= q_1a + q_2b + q_2ab$

$q_2 = q_1a + q_2(b+ab)$ --- $R = Q + RP$

$R = Q + RP$ --- $R = QP^*$ Ardens Theorem

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

(3) →

$q_1 = \epsilon + q_1 a + q_2 b$

Putting value of q_2 from (5)

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

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

R Q R P

$R = Q + P$

$R = QP^*$

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

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

Final state (q_3)

$q_3 = q_2 a$

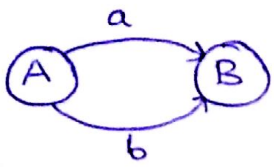
$q_3 = (q_1 a)(b+ab)^* a$ Putting value of q_2 from (5)

$q_3 = (a + a(b+ab)^* b)^* a(b+ab)^* a$ putting value of q_1 from (6)

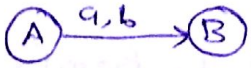
Conversion of Regular Expression to Finite Automata

Rules for designing

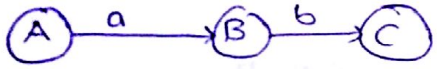
$(a + b)$



or



$(a.b)$



a^*

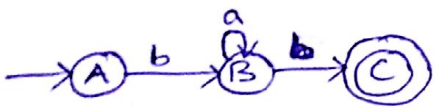
Conversion of Regular Expression to Finite Automata - Examples (Part I)

Convert the following Regular Expressions to their equivalent Finite Automata :

1) ba^*b
2) $(a+b)c$
3) $a(bc)^*$

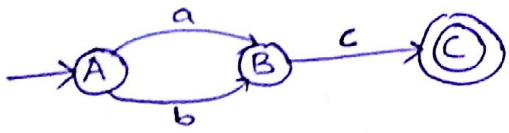
1) ba^*b

bb, bab, baab, ...



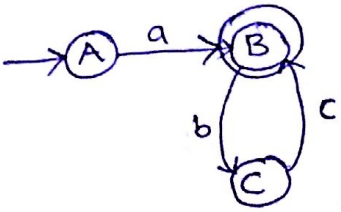
2) $(a+b)c$

ac ✓
bc ✓



3) $a(bc)^*$

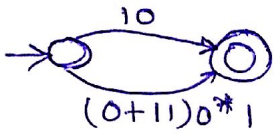
a, abc, abcbc, abcbcbc

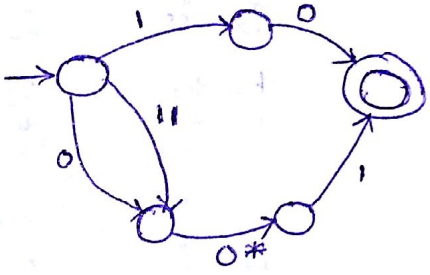


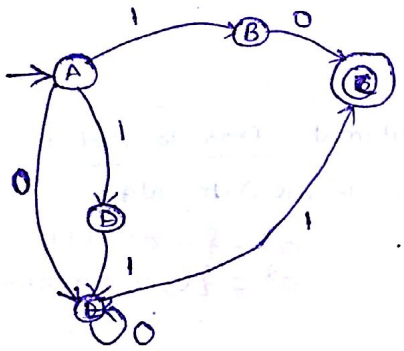
Conversion of Regular Expression to Finite Automata - Example (Part-3)

Convert the following Regular Expression to its equivalent Finite Automata:

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







Equivalence of two Finite Automata

Steps to identify equivalence

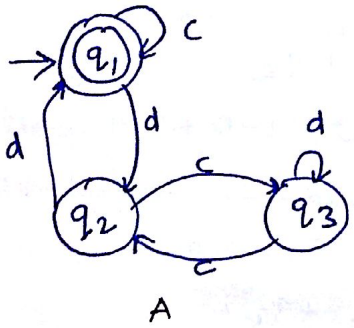
1) For any pair of states $\{q_i, q_j\}$ the transition for input $a \in \Sigma$ is defined by $\{q_a, q_b\}$.

where $\delta\{q_i, a\} = q_a$ and $\delta\{q_j, a\} = q_b$

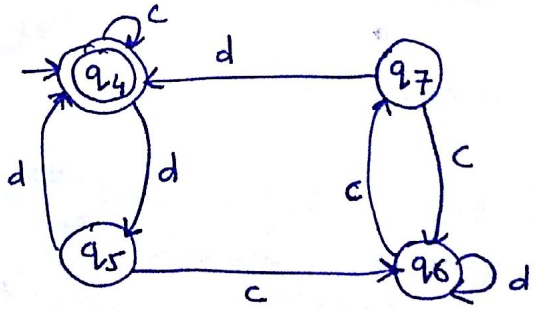
The two automata are not equivalent if for a pair $\{q_a, q_b\}$ one is INTERMEDIATE State and the other is FINAL state.

2) If initial state is Final state of one Automaton, then in second automaton also Initial state must be Final state for them to be equivalent.

Eg.



A



B

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

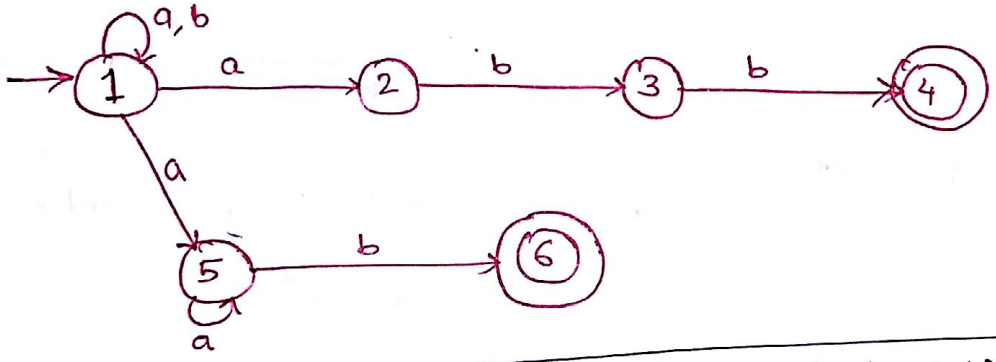
A and B are equivalent

Conversion of Regular Expression to Finite Automata - Examples Part-2)

Convert the following Regular Expression to its equivalent Finite Automata:

$(a|b)^*(abb|a^+b)$ [+ & | are same]

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



Designing Regular Expression - Examples (Part-4)

(When there are Multiple Final states)

← Find the Regular Expression for the following DFA

$q_1 = \epsilon + q_1 0 \longrightarrow (1)$

$q_2 = q_1 1 + q_2 1 \longrightarrow (2)$

$q_3 = q_2 0 + q_3 0 + q_3 1 \longrightarrow (3)$

Final state q_1

(1) →

$q_1 = \epsilon + q_1 0$ --- $R = Q + RP$

R = Q P

$q_1 = (\epsilon 0)^*$ --- $\epsilon R = R^*$

$q_1 = 0^* \longrightarrow (4)$

Final state q_2

(2) →

$q_2 = q_1 1 + q_2 1$

$q_2 = 0^* 1 + q_2 1$ --- by putting value of q_1 from (4)

R = Q P

$R = Q + RP$

$R = QP^*$

$q_2 = 0^* 1 (1)^*$

R = union of both the final states

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

$= 0^* + 0^*11^*$

$= 0^*(\epsilon + 11^*)$ --- $\epsilon + RR^* = R^*$

$R = 0^*1^*$

↳ Required Expression.

Pumping Lemma (For Regular Languages)

>> Pumping Lema 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

$S = xyz$ such that the following conditions must be true

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

(2) $|y| > 0$

(3) $|xy| \leq P$

To prove that a language is not Regular Using PUMPING LEMMA, follow 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 xyz
→ Show that $xy^iz \notin A$ for some i
→ Then Consider all ways that S can be divided into xyz
→ Show that none of these can satisfy all the 3 pumping conditions at the same time
→ S cannot be Pumped == CONTRADICTION.

Pumping Lemma (For Regular Languages) - Example (Part-I)

Using Pumping Lemma Prove that the language $A=\{a^n b^n \mid n \geq 0\}$ is Not Regular

Proof:-

Assume that A is Regular

Pumping length = p

$S = a^p b^p$ $\Rightarrow$ S = aaa aaaabbbbbbb

x y z

P = 7

Case 1: The y part is in the 'a' part

aaaaaaabbbbbbb

x y z

Case 2: The y is in the 'b' part

aaaaaaa bbbbbbb

x y z

Case 3: The y is in the 'a' and 'b' part

aaaaaaa bbbbbbb

x y z

~~For~~ Case 1

$xy^iz \Rightarrow xy^2z$ → This string does not lie in our language because No. of a's and b's are not equal

aaaaaaaaaabbbbbbb

11 ≠ 7

For case 2:-

$xy^iz \Rightarrow xy^2z$

aaaaaaabb bbbb bbbbb → This string does not lie in our language because No. of a's and b's are not equal.

7 ≠ 11

For case 3:-

$xy^iz \Rightarrow xy^2z$ → This string does not lie in our language because No of a's and b's are not equal $\neq a^n b^n$

aaaaaaabb bbbbb aabb bbbbb

$|xy| \leq p \qquad p = 7$

~~Case 1:-~~

The language is Not Regular Using Pumping lemma

Pumping Lemma for Regular Languages (Example Part -II)

Using Pumping Lemma Prove that the language $A = \{yy \mid y \in \{0,1\}^*\}$ is Not Regular

0101

proof:-

Assume that A is Regular

Then it Must have a Pumping length = P

$S = 0^P 1 0^P 1$

x y z

$xy^iz \Rightarrow xy^2z$

$P = 7$

00000000010000000 1

x y z

00000000001000000001

$\notin A$

$|y| > 0$

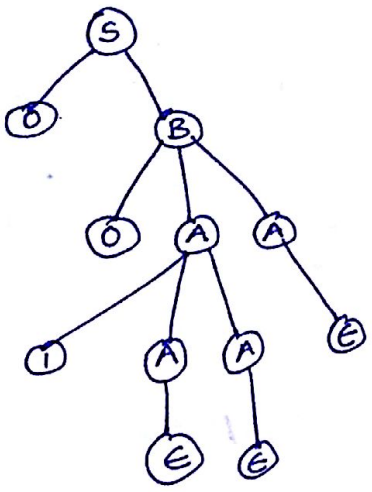
$|xy| \leq P \quad 7$ A is NOT Regular

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 $S \to 0B$, $A \to 1AA \mid \epsilon$

$B \to 0AA$



Root Vertex:- Must be labelled by the start symbol

Vertex:- Labelled by NON-Terminal symbols

Leaves:- Labelled by Terminal symbols or ϵ

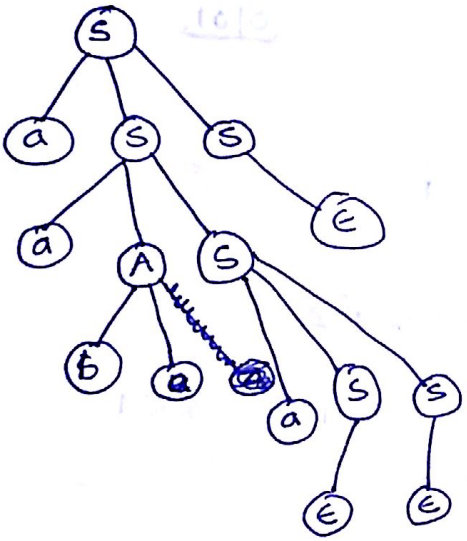
Left Derivation Tree

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

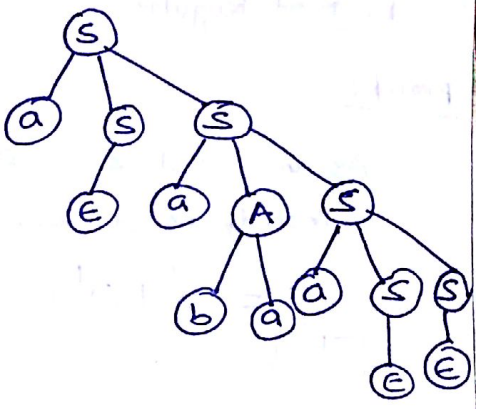
Right Derivation Tree

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

E.g: for generating the string aabaa from the Grammar $S \rightarrow aAS \mid aSS \mid \epsilon$, $A \rightarrow SbA \mid ba$



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

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

phase1:- Derivation of an equivalence grammar 'G', from the CFG, G, such that each variable ~~devices~~ derives some terminal string

Derivation Procedure :-

Step1:- 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 "G1", from the CFG, 'G', such that each symbol appears in a sentential form.

Derivation Procedures:-

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 G1, having production rules

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

Phase 1:

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

$W_1 = \{A, C, E\}$ — set derive terminals

$W_2 = \{A, C, E, S\}$ — set of element of W_1

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

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

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 $A \to B$ where $A, B \in$ Non Terminals is called Unit Production.

Procedure for Removal:-

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

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

Step 3:- Repeat from step1 until all Unit Productions are removed

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

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

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

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

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

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

→ from Start Symbol S we can not reach to Z, M, N.

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 $A \to \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ. (Like $A \to \ldots\ldots \to \epsilon$)

Production for Removal:-

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

Step 2:- 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.

$S \to ABAC,\ A \to aA|\epsilon,\ B \to bB|\epsilon,\ C \to c$

$A \to \epsilon\ ,\ B \to \epsilon$

1) To eliminate $A \to \epsilon$

$S \to ABAC$

$S \to AB\epsilon C$

$\to ABC \mid BAC \mid BC$

$\to$

$A \to aA$

$A \to a$

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

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

2) To eliminate $B \to \epsilon$

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

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

$A \to aA|a$

$B \to bB|b$

$C \to c$

Normal Forms

① Chomsky Normal form
② Greibach Normal Form

① Chomsky Normal Form

In Chomsky Normal Form (CNF) we have a restriction on length of RHS; which is; elements in RHS should either be two variables or a Terminal.

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

$$A \rightarrow a$$

$$A \rightarrow BC$$

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

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. (—"—)

Step 4:- Replace each Production $A \rightarrow B_1 \ldots\ldots B_n$ where $n > 2$, with $A \rightarrow B_1 C$ where $C \rightarrow B_2 \ldots\ldots B_n$ Repeat this step for all Production 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 $A \rightarrow aB$.

Conversion of CFG to chomsky Normal form

convert the following CFG to CNF : P: $S \to ASA \mid aB$, $A \to B \mid S$, $B \to b \mid \epsilon$

1) Since S appears in RHS, we add new state S' and $S' \to S$ is added to the production

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

---

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

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

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

---

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

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

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

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

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

---

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

$S' \to ASA$, $S \to ASA$ and $A \to ASA$

After removing these, we get : P: $S' \to AX \mid aB \mid a \mid AS \mid SA$,
$S \to AX \mid aB \mid a \mid AS \mid SA$,
$A \to b \mid AX \mid aB \mid a \mid AS \mid SA$,
$B \to b$,
$X \to SA$

5] Now change the productions $S' \to aB$, $S \to aB$ and $A \to aB$

Finally we get :

P: $S' \to AX \mid YB \mid a \mid AS \mid SA$, (.... $Y \to a$)

$S \to AX \mid YB \mid a \mid AS \mid SA$,

$A \to b \mid AX \mid YB \mid a \mid AS \mid SA$,

$B \to b$,

$X \to SA$,

$Y \to a$

Which is the required chomsky Normal Form for the given CFG.

Theory of Computation

SET-II

Handwritten Notes :

Prepared by: prof. Ajay Pashankar

www.profajaypashankar.com

Chey

Greibach Normal Form

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

$$A \to b$$

$$A \to bC_1C_2 \ldots\ldots C_n$$

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 techniques 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 :- $S \to CA \mid BB$ Replace S with A_1

$B \to b \mid SB$ C with A_2

$C \to b$ A with A_3

$A \to a$ B with A_4

We get:

$$A_1 \to A_2A_3 \mid A_4A_4$$

$$A_4 \to b \mid A_1A_4$$

$$A_2 \to b$$

$$A_3 \to a$$

Step 4:- Alter the rules so that the Non-Terminals are in ascending order, such that If the Production is of the form $A_i \to A_j X$, then,

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

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

$$A_4 \to b \mid \underline{A_2}A_3A_4 \mid A_4A_4A_4$$

$$A_4 \to b \mid bA_3A_4 \mid A_4A_4A_4$$

↓ Left Recursion.

Step 5: Remove Left Recursion

(Greibach Normal Form)

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

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

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

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Left Recursion

Step 5 :- Remove Left Recursion

Introduce a New Variable to remove the Left Recursion

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

$A_4 \rightarrow A_4 A_4 Z \mid A_4 A_4$ ---(new production one with Z and one without Z)

$A_4 \rightarrow b \mid b A_3 A_4 \mid b Z \mid b A_3 A_4 Z$ ...(it is in GNF)

Now the grammar is :

$A_1 \rightarrow A_2 A_3 \mid A_4 A_4$

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

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

$A_2 \rightarrow b$

$A_3 \rightarrow a$

~~$A_1 \rightarrow b A_3 \mid A_4 b \mid b A_3 A_4 \mid b Z \mid b A_3 A_4 Z$~~

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

$A_4 \rightarrow b \mid b A_3 A_4 \mid b Z \mid b A_3 A_4 Z$ ----(Replacing $A_2 \rightarrow b$, $A_4 \rightarrow b$)

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

$b A_4 \mid b A_3 A_4 A_4 Z \mid b Z A_4 Z \mid b A_3 A_4 Z A_4 Z$ ----(Replacing $A_4 \rightarrow b$)

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Pumping Lemma (for Context Free Languages)

Pumping Lemma (for CFL) is used to prove that a language is NOT Context Free.

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 identified by 4 tuples as $G = \{V, \Sigma, S, P\}$ where

V = set of Variable or Non-Terminal Symbols

Σ = set of Terminal Symbols.

S = Start Symbol

P = Production Rule.

Context Free Grammar has Production Rule of the form

$$A \rightarrow a$$

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

If A is a Context Free Language, then, A has a Pumping Length 'p' such that any string 's', where $|s| \geq p$ may be divided into 5 Pieces $s = uvxyz$ such that the following conditions must be true:

1) $u\, v^i x^i y^i z$ is in A for every $i \geq 0$

2) $|vy| > 0$

3) $|vxy| \leq p$

To Prove that a Language is Not context Free using Pumping Lemma (for CFL) follow the steps below: (we prove using CONTRADICTION)

→ Assume that A is context Free

→ 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 uvxyz

→ Show that $uv^i xy^i z \notin A$ for some i

→ Then consider the ways that S can be divided into uvxyz

→ show that none of these can satisfy all the 3 pumping conditions at the same time.

→ S cannot be pumped == CONTRADICTION

Pushdown Automata (Formal Definition)

A pushdown Automata is formally defined by 7 Tuples as show below:

$$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Where,

Q = A finite set of states

Σ = A finite set of Input Symbols

Γ = A finite Stack Alphabet

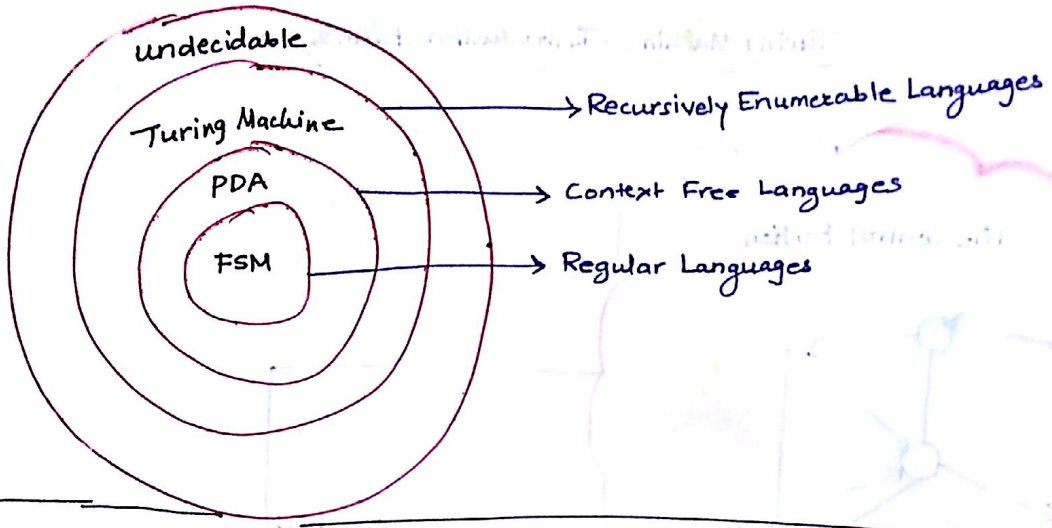
δ = The Transition Function

q_0 = The start state

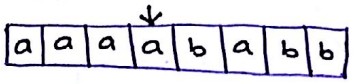
Z_0 = The start stack symbol

F = The set of Final / Accepting States

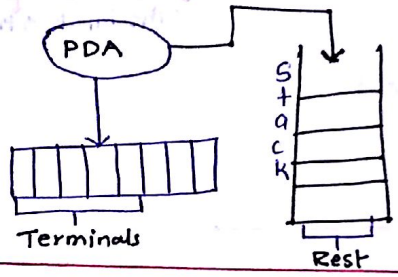
Turing Machine - Introduction (Part-1)



FSM : The Input String

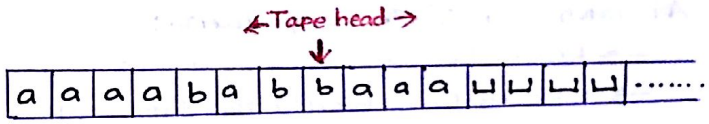


PDA : → The Input String
→ A Stack



TURING MACHINE:

→ A Tape

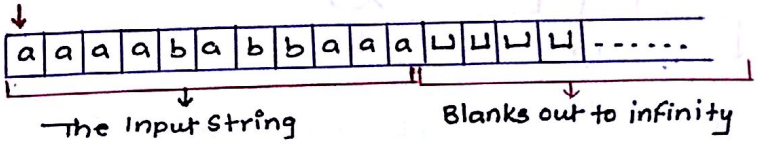


Tape Alphabets: $\Sigma = \{0, 1, a, b, x, Z_0\}$

The Blank ⊔ is a special Symbol $\quad \sqcup \notin \Sigma$

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

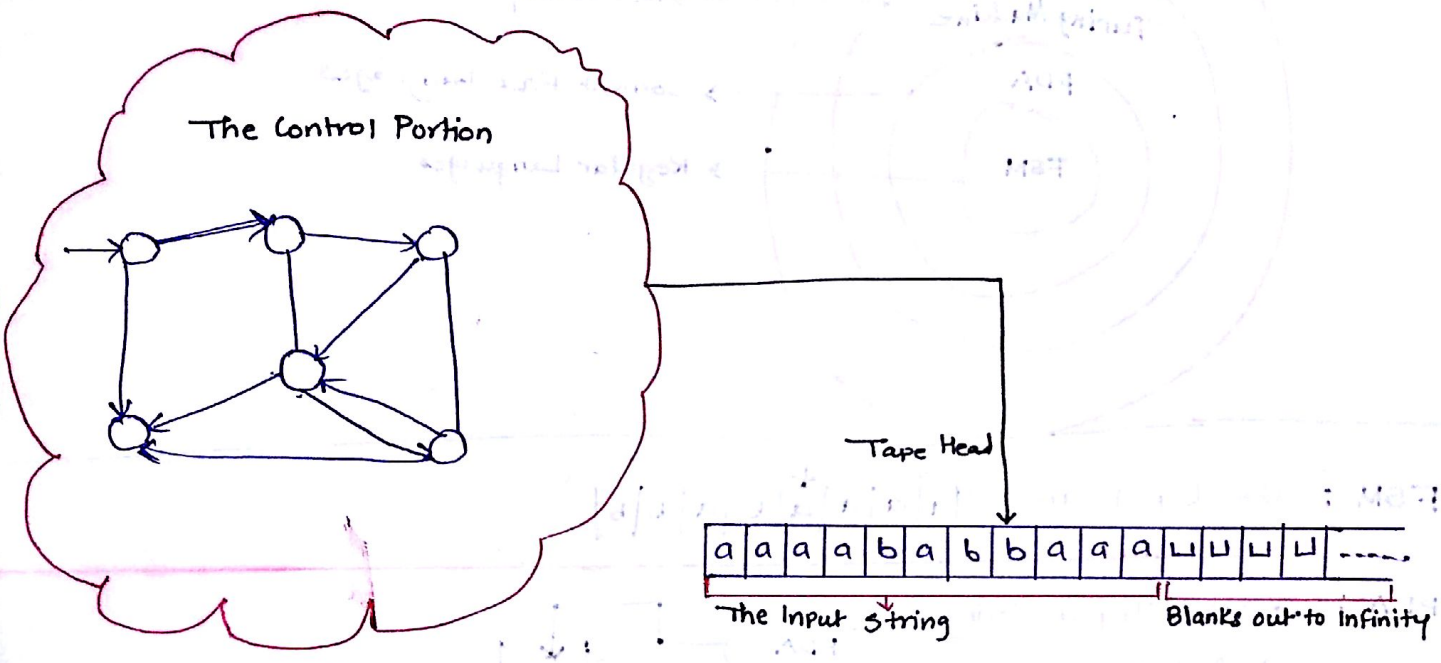
Initial configuration:



Operations on the Tape:

→ Read / Scan symbol below the Tape Head.
→ Update / Write a Symbol below the TapeHead
→ 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:

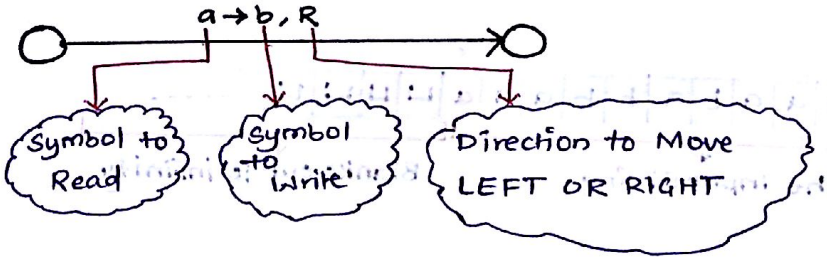
→ Read the current symbol

→ Update (i.e. write) the same cell

→ Move exactly one cell either LEFT OR RIGHT.

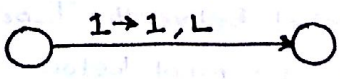
If we are at the left end of the tape, and trying to move LEFT, then do not move.

Stay at the left end.



If you don't 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 → Non empty set of states

Σ → Non empty set of symbols

Γ → Non empty set of Tape symbols ....(Tau)

δ → Transition function defined as

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

q_0 → Initial state

b → Blank symbol

F → 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 an algorithm, for which a Turing Machine Program cannot be written.

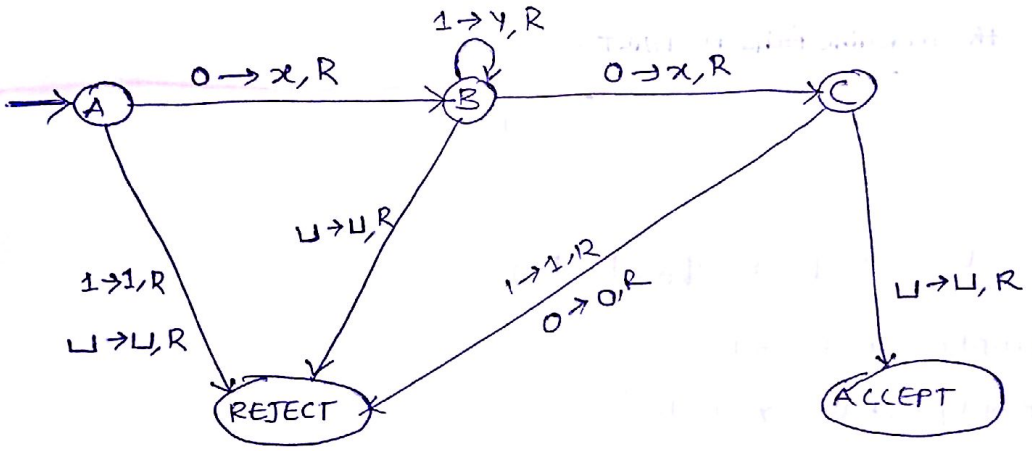
Recursively Enumerable Language:

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

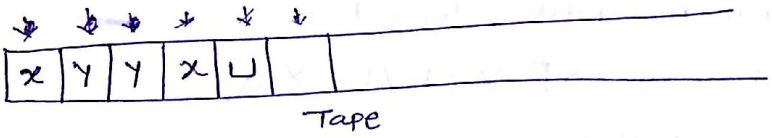
Turing Machine - Example (Part - 1)

Design a Turing Machine which recognizes the language

$$L = 01^*0$$



e.g.:- 0110 ✓



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

$b = U$

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